Tomás J. Aragón

Applied Epidemiology Using R

October 14, 2013

University of California, Berkeley School of Public Health,
and the San Francisco Department of Public Health

For my wife, Irene, and my children Ángela, Luis, and Tomás, Jr.

Preface

I write this book to introduce R—a language and environment for statistical computing and graphics—to epidemiologists and health data analysts conducting epidemiologic studies. From my experience in public health practice, sometimes even formally trained epidemiologists lack the breadth of analytic skills required at health departments where resources are very limited. Recent graduates come prepared with a solid foundation in epidemiological and statistical concepts and principles and they are ready to run a multivariable analysis (which is not a bad thing we are grateful for highly trained staff). However, what is sometimes lacking is the practical knowledge, skills, and abilities to collect and process data from multiple sources (e.g., Census data; reportable diseases, death and birth registries) and to adequately implement new methods they did not learn in school. One approach to implementing new methods is to look for the "commands" among their favorite statistical packages (or to buy a new software program). If the commands do not exist, then the method may not be implemented. In a sense, they are looking for a custom-made solution that makes their work quick and easy.

In contrast to custom-made tools or software packages, R is a suite of basic tools for statistical programming, analysis, and graphics. One will not find a "command" for a large number of analytic procedures one may want to execute. Instead, R is more like a set of high quality carpentry tools (hammer, saw, nails, and measuring tape) for tackling an infinite number of analytic problems, including those for which custom-made tools are not readily available or affordable. I like to think of R as a set of extensible tools to implement one's analysis plan, regardless of simplicity or complexity. With practice, one not only learns to apply new methods, but one also develops a depth of understanding that sharpens one's intuition and insight. With understanding comes clarity, focused problem-solving, and confidence.

This book is divided into two parts. First, I cover how to process, manipulate, and operate on data in R. Most books cover this material briefly or leave it for an appendix. I decided to dedicate a significant amount of space to this topic with the assumption that the average epidemiologist is not familiar with R and a good grounding in the basics will make the later chapters more understandable. Second, I cover basic epidemiology topics addressed in most books but we infuse R to demonstrate concepts and to exercise your intuition. Readers may notice a heavier emphasis on descriptive epidemiology which is what is more commonly used at health departments, at least as a first step. In this section we do cover regression methods and graphical displays. I have also included "how to" chapters on a diversity of topics that come up in public health. My goal is not to be comprehensive in each topic but to demonstrate how R can be used to implement a diversity of methods relevant to public health epidemiology and evidence-based practice.

To help us spread the word, this book is available on the World Wide Web (`http://www.medepi.com`). I do not want financial or geographical barriers to limit access to this material. I am only presenting what I have learned from the generosity of others. My hope is that more and more epidemiologists will embrace R for epidemiological applications, or at least, include it in their toolbox.

Berkeley, California
October 14, 2013

Tomás J. Aragón

Acknowledgements

I would like to acknowledge the professors at University of California at Berkeley that not only taught me the principles and methods of epidemiology, biostatistics, and demography, but also introduced me to the S language as a tool for exploring and analyzing epidemiologic data. More specifically, I owe special thanks to Professor Steve Selvin, Chair of the Division of Biostatistics at the School of Public Health, Professor Kenneth Wachter, Chair of the Department of Demography, and Professor Arthur Reingold, Chair of the Division of Epidemiology. Professor Selvin convinced me that the investment in learning S would pay off in increased productivity and understanding. From Professor Wachter I learned the fundamentals of demography and how to use S to program functions for demographic analysis. Professor Reingold recruited me to direct the UC Berkeley Center for Infectious Diseases & Emergency Readiness where I have had the opportunity and challenge to think about how to make this material more accessible to public health epidemiologists.

Using the S language is so much fun! It becomes an extension of one's analytic mind. Thanks for getting me started and and for giving me the opportunity to learn and teach!

Berkeley, California *Tomás J. Aragón*

Contents

Part I Working with R

1 Getting Started With R 3
- 1.1 What is R? 3
  - 1.1.1 Who should learn R? 4
  - 1.1.2 Why should I learn R? 4
  - 1.1.3 Where can I get R? 5
- 1.2 How do I use R? 5
  - 1.2.1 Using R on our computer 5
  - 1.2.2 Does R have epidemiology programs? 6
  - 1.2.3 How should I use these notes? 7
- 1.3 Just do it! 8
  - 1.3.1 Using R as your calculator 8
  - 1.3.2 Useful R concepts 9
  - 1.3.3 Useful R functions 11
  - 1.3.4 How do I get help? 14
  - 1.3.5 RStudio—An integrated development environment for R . . . 15
  - 1.3.6 Is there anything else that I need? 15
  - 1.3.7 What's ahead? 17
- Problems 21

2 Working with R data objects 25
- 2.1 Data objects in R 25
  - 2.1.1 Atomic vs. recursive data objects 25
  - 2.1.2 Assessing the structure of data objects 28
- 2.2 A vector is a collection of like elements 29
  - 2.2.1 Understanding vectors 29
  - 2.2.2 Creating vectors 32
  - 2.2.3 Naming vectors 37

2.2.4 Indexing vectors . . . 38
2.2.5 Replacing vector elements (by indexing and assignment) . . . 40
2.2.6 Operations on vectors . . . 41
2.3 A matrix is a 2-dimensional table of like elements . . . 48
2.3.1 Understanding matrices . . . 48
2.3.2 Creating matrices . . . 49
2.3.3 Naming a matrix . . . 55
2.3.4 Indexing a matrix . . . 56
2.3.5 Replacing matrix elements . . . 57
2.3.6 Operations on a matrix . . . 57
2.4 An array is a *n*-dimensional table of like elements . . . 62
2.4.1 Understanding arrays . . . 62
2.4.2 Creating arrays . . . 65
2.4.3 Naming arrays . . . 69
2.4.4 Indexing arrays . . . 69
2.4.5 Replacing array elements . . . 69
2.4.6 Operations on arrays . . . 70
2.5 A list is a collection of like or unlike data objects . . . 76
2.5.1 Understanding lists . . . 76
2.5.2 Creating lists . . . 78
2.5.3 Naming lists . . . 79
2.5.4 Indexing lists . . . 80
2.5.5 Replacing lists components . . . 83
2.5.6 Operations on lists . . . 83
2.6 A data frame is a list in a 2-dimensional tabular form . . . 84
2.6.1 Understanding data frames and factors . . . 84
2.6.2 Creating data frames . . . 87
2.6.3 Naming data frames . . . 88
2.6.4 Indexing data frames . . . 90
2.6.5 Replacing data frame components . . . 92
2.6.6 Operations on data frames . . . 94
2.7 Managing data objects . . . 96
2.8 Managing our workspace . . . 100
Problems . . . 103

3 Managing epidemiologic data in R . . . 107
3.1 Entering and importing data . . . 107
3.1.1 Entering data at the command prompt . . . 107
3.1.2 Importing data from a file . . . 115
3.1.3 Importing data using a URL . . . 118
3.2 Editing data . . . 119
3.2.1 Text editor . . . 119
3.2.2 The `data.entry`, `edit`, or `fix` functions . . . 119
3.2.3 Vectorized approach . . . 121
3.2.4 Text processing . . . 123

3.3 Sorting data 124
3.4 Indexing (subsetting) data 126
  3.4.1 Indexing 127
  3.4.2 Subsetting 128
3.5 Transforming data 129
  3.5.1 Numerical transformation 129
  3.5.2 Creating categorical variables (factors) 130
  3.5.3 “Re-coding” levels of a categorical variable 132
  3.5.4 Use factors instead of dummy variables 135
  3.5.5 Conditionally transforming the elements of a vector 135
3.6 Merging data 136
3.7 Executing commands from, and directing output to, a file 139
  3.7.1 The `source` function 139
  3.7.2 The `sink` and `capture.output` functions 140
3.8 Working with missing and “not available” values 142
  3.8.1 Testing, indexing, replacing, and recoding 144
  3.8.2 Importing missing values with the `read.table` function . 145
  3.8.3 Working with NA values in data frames and factors 146
  3.8.4 Viewing number of missing values in tables 148
  3.8.5 Setting default NA behaviors in statistical models 149
  3.8.6 Working with finite, infinite, and NaN numbers 150
3.9 Working with dates and times 151
  3.9.1 Date functions in the `base` package 152
  3.9.2 Date functions in the `chron` and `survival` packages 158
3.10 Exporting data objects 158
  3.10.1 Exporting to a generic ASCII text file 159
  3.10.2 Exporting to R ASCII text file 161
  3.10.3 Exporting to R binary file 163
  3.10.4 Exporting to non-R ASCII text and binary files 164
3.11 Working with regular expressions 164
  3.11.1 Single characters 165
  3.11.2 Character class 166
  3.11.3 Concatenation 167
  3.11.4 Repetition 168
  3.11.5 Alternation 168
  3.11.6 Repetition > Concatenation > Alternation 170
  3.11.7 Metacharacters 170
  3.11.8 Other regular expression functions 172
Problems 175

Part II Applied Epidemiology

A Available data sets 183
- A.1 Latina Mothers and their Newborn 183
- A.2 Oswego County (outbreak) 184
- A.3 Western Collaborative Group Study (cohort) 184
- A.4 Evans County (cohort) 185
- A.5 Myocardial infarction case-control study 185
- A.6 AIDS surveillance cases 187
- A.7 Hepatitis B surveillance cases 187
- A.8 Measles surveillance cases 187
- A.9 University Group Diabetes Program 187
- A.10 Novel influenza A (H1N1) pandemic 187
  - A.10.1 United States reported cases and deaths as of July 23, 2009 . 187

B Outbreak analysis template in R 189

C Programming and creating R functions 197
- C.1 Basic programming 197
- C.2 Intermediate programming 198
  - C.2.1 Control statements 198
  - C.2.2 Vectorized approach 200
  - C.2.3 Looping 201
- C.3 Writing R functions 204
  - C.3.1 Arguments default values 206
  - C.3.2 Passing optional arguments using the `...` function 208
- C.4 Advanced topics 208
  - C.4.1 Lexical scoping 208

Solutions 211

References 227

Part I
Working with R

CHAPTER 1

Getting Started With R

1.1 What is R?

R is a freely available "computational language and environment for data analysis and graphics." R is indispensable for anyone that uses and interprets data. As medical, public health, and research epidemiologists, we use R in the following ways:

- Full-function calculator
- Extensible statistical package
- High-quality graphics tool
- Multi-use programming language

We use R to explore, analyze, and understand epidemiological data. We analyze data straight out of tables provided in reports or articles as well as analyze usual data sets. The data might be a large, individual-level data set imported from another source (e.g., cancer registry); an imported matrix of group-level data (e.g, population estimates or projections); or some data extracted from a journal article we are reviewing. The ability to quantitatively express, graphically explore, and describe epidemiologic data and processes enables one to work and strengthen one's epidemiologic intuition.

In fact, we only use a very small fraction of the R package. For those who develop an interest or have a need, R also has many of the statistical modeling tools used by epidemiologists and statisticians, including logistic and Poisson regression, and Cox proportional hazard models. However, for many of these routine statistical models, almost any package will suffice (SAS, Stata, SPSS, etc.). The real advantage of R is the ability to easily manipulate, explore, and graphically display data. Repetitive analytic tasks can be automated or streamlined with the creation of simple functions (programs that execute specific tasks). The initial learning curve is steep, but in the long run one is able to conduct analyses that would otherwise require a tremendous amounts of programming and time.

Some may find R challenging to learn if they are not familiar with statistical programming. R was created by statistical programmers and is more often used by analysts comfortable with matrix algebra and programming. However, even for those unfamiliar with matrix algebra, there are many analyses one can accomplish in R without using any advanced mathematics, which would be difficult in other programs. The ability to easily manipulate data in R will allow one to conduct good descriptive epidemiology, life table methods, graphical displays, and exploration of epidemiologic concepts. R allows one to work with data in any way they come.

1.1.1 Who should learn R?

Anyone that uses a calculator or spreadsheet, or analyzes numerical data at least weekly should seriously consider learning and using R. This includes epidemiologists, statisticians, physician researchers, engineers, and faculty and students of mathematics and science courses, to name just a few. We jokingly tell our staff analysts that once they learn R they will never use a spreadsheet program again (well almost never!).

1.1.2 Why should I learn R?

To implement numerical methods we need a computational tool. On one end of the spectrum are calculators and spreadsheets for simple calculations, and on the other end of the spectrum are specialized computer programs for such things as statistical and mathematical modeling. However, many numerical problems are not easily handled by these approaches. Calculators, and even spreadsheets, are too inefficient and cumbersome for numerical calculations whose scope and scale change frequently. Statistical packages are usually tailored for the statistical analysis of data sets and often lack an intuitive, extensible, open source programming language for tackling new problems efficiently.[1] R can do the simplest and the most complex analysis efficiently and effectively.

When we learn and use R regularly we will save significant amounts of time and money. It's powerful and it's free! It's a complete environment for data analysis and graphics. Its straightforward programming language facilitates the development of functions to extend and improve the efficiency of our analyses.

---

[1] Read my recommendations for mostly free and open source software (FOSS) at `medepi.com`.

1.1.3 *Where can I get R?*

R is available for many computer platforms, including Mac OS, Linux, Microsoft Windows, and others. R comes as source code or a binary file. Source code needs to be compiled into an executable program for our computer. Those not familiar with compiling source code (and that's most of us) just install the binary program. We assume most readers will be using R in the Mac OS or MS Windows environment. Listed here are useful R links:

- R Project home page at `http://www.r-project.org/`;
- R download page at `http://cran.r-project.org`;
- Numerous free tutorials are at `http://cran.r-project.org/other-docs.html`;
- R Wikibook at `http://en.wikibooks.org/wiki/R_Programming`; and
- R Journal at `http://journal.r-project.org/`.

To install R for Windows, do the the following:

1. Go to `http://www.r-project.org/`;
2. From the left menu list, click on the "CRAN" (Comprehensive R Archive Network) link;
3. Select a nearby geographic site (e.g., `http://cran.cnr.berkeley.edu/`);
4. Select appropriate operating system;
5. Select on "base" link;
6. For Windows, save `R-X.X.X-win32.exe` to the computer; and for Mac OS, save the `R-X.X.X-mini.dmg` disk image.
7. Run the installation program and accept the default installation options. That's it!

1.2 How do I use R?

1.2.1 *Using R on our computer*

Use R by typing commands at the R command line prompt (>) and pressing Enter on our keyboard. This is how to use R interactively. Alternatively, from the R command line, we can also execute a list of R commands that we have saved in a text file (more on this later). Here is an example of using R as a calculator:

```
> 8*4
[1] 32
> (4 + 6)^3
[1] 1000
```

Use the `scan` function to enter data at the command line. At the end of each line press the Enter key to move to the next line. Pressing the Enter key twice completes the data entry.

```
> quantity <- scan()
1: 34 56 22
4:
Read 3 items
> price <- scan()
1: 19.95 14.95 10.99
4:
Read 3 items
> total <- quantity*price
> cbind(quantity, price, total)
     quantity price  total
[1,]       34 19.95 678.30
[2,]       56 14.95 837.20
[3,]       22 10.99 241.78
```

1.2.2 Does R have epidemiology programs?

The default installation of R does not have packages that specifically implement epidemiologic applications; however, many of the statistical tools that epidemiologists use are readily available, including statistical models such as unconditional logistic regression, conditional logistic regression, Poisson regression, Cox proportional hazards regression, and much more.

To meet the specific needs of public health epidemiologists and health data analysts, I maintain a freely available suite of Epidemiology Tools: the `epitools` R package can be directly installed from within R.

For example, to evaluate the association of consuming jello with a diarrheal illness after attending a church supper in 1940, we can use the `epitab` function from the `epitools` package. In the R code that follows, the `#` symbol precedes comments that are not evaluated by R.

```
> library(epitools)     #load 'epitools' package
> data(oswego)          #load Oswego dataset
> attach(oswego)        #attach dataset
> round(epitab(jello, ill, method = "riskratio")$tab, 3)
         Outcome
Predictor  N    p0  Y    p1 riskratio lower upper p.value
        N 22 0.423 30 0.577     1.000    NA    NA      NA
        Y  7 0.304 16 0.696     1.206 0.844 1.723   0.442
> round(epitab(jello, ill, method = "oddsratio")$tab, 3)
         Outcome
```

```
Predictor  N    p0  Y    p1 oddsratio lower upper p.value
        N 22 0.759 30 0.652     1.000    NA    NA      NA
        Y  7 0.241 16 0.348     1.676  0.59 4.765   0.442
> detach(oswego)        #detach dataset
```

The risk of illness among those that consumed jello was 69.6% compared to 57.7% among those that did not consume jello. Both the risk ratio and the odds ratio were elevated but we cannot exclude random error as an explanation (p value = 0.44). We also notice that the odds ratio is not a good approximation to the risk ratio. This occurs when the risks among the exposed and unexposed is greater than 10%. In this case, the risks of diarheal illness were 69.6% and 57.7% among exposed and nonexposed, respectively.

1.2.3 How should I use these notes?

The best way to learn R is to use it! Use it as our calculator! Use it as our spreadsheet! Finally read these notes sitting at a computer and use R interactively (this works best sitting in a cafe that brews great coffee and plays good music). In this book, when we display R code it appears as if we are typing the code directly at the R command line:

```
> x <- matrix(1:9, nrow = 3, ncol = 3)
> x
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

Sometimes the R code appears as it would in a text editor (e.g., Notepad) before it has been evaluated by R. When a text editor version of R code is displayed it appears without the command line prompt and without output:

```
x <- matrix(1:9, nrow = 3, ncol = 3)
x
```

When the R code displayed exceeds the page width it will continue on the next line but indented. Here's an example:

```
agegrps <- c("Age < 1", "Age 1 to 4", "Age 5 to 14", "Age 15
    to 24", "Age 25 to 44", "Age 45 to 64", "Age 64+")
```

Although we encourage initially to use R interactively by typing expressions at the command line, as a general rule, it is much better to type our code into a text editor. We save our code with a convenient file name such as job01.R[2]. For

---

[2] The .R extension, although not necessary, is useful when searching for R command files. Additionally, this file extension is recognized by some text editors.

Table 1.1 Selected math operators

Operator	Description	Try these examples
$+$	addition	`5+4`
$-$	subtraction	`5-4`
$*$	multiplication	`5*4`
/	division	`5/4`
ˆ	exponentiation	`5^4`
$-$	unary minus (change current sign)	`-5`
abs	absolute value	`abs(-23)`
exp	exponentiation (*e* to a power)	`exp(8)`
log	logarithm (default is natural log)	`log(exp(8))`
sqrt	square root	`sqrt(64)`
%/%	integer divide	`10%/%3`
%%	modulus	`10%%3`

%*%	matrix multiplication	`xx <- matrix(1:4, 2, 2)`
		`xx%*%c(1, 1)`
		`c(1, 1)%*%xx`

convenience, R comes with its own text editor. For Windows, under File, select New script to open an empty text file. For Mac OS, under File, select New Document. As before, save this text file with a `.R` extension. Within R's text editor, we can highlight and run selected commands.

The code in our text editor can be run in the following ways:

- Highlight and run selected command in the R editor;
- Paste the code directly into R at the command line;
- Run the file in batch mode from the R command line using the `source("job01.R")`.

1.3 Just do it!

1.3.1 Using R as your calculator

Open R and start using it as our calculator. The most common math operators are displayed in Table 1.1. From now on make R our default calculator! Study the examples and spend a few minutes experimenting with R as a calculator. Use parentheses as needed to group operations. Use the keyboard Up-arrow to recall what we previously entered at the command line prompt.

Table 1.2 Types of evaluable expressions[a]

Expression type	Try these examples

literal	`"hello, I'm John Snow" #character`
	`3.5 #numeric`
	`TRUE; FALSE #logical`

math operation	`6*7`
assignment	`x <- 4*4`
data object	`x`
function	`log(x)`

[a] lines preceded with `#` are not evaluated

1.3.2 *Useful R concepts*

1.3.2.1 Types of evaluable expressions

Every *expression* that is entered at the R command line is evaluated by R and returns a value. A *literal* is the simplist expression that can be evaluated (number, character string, or logical value). Mathematical operations involve numeric literals. For example, R evaluates the expression `4*4` and returns the value `16`). The exception to this is when an evaluable expression is assigned an object name: `x <- 4*4`. To display the assigned expression, wrap the expression in parentheses: `(x <- 4*4)`, or type the object name.

```
> 4*4
[1] 16
> x <- 4*4
> x
[1] 16
> (x <- 4*4)
[1] 16
```

Finally, evaluable expressions must be separated by either newline breaks or a semicolon.

```
> x <- 4*4; x
[1] 16
```

Table 1.2 summarizes evaluable expressions.

1.3.2.2 Using the assignment operator

Most calculators have a memory function: the ability to assign a number or numerical result to a key for recalling that number or result at a later time. The same is true in R but it is much more flexible. Any evaluable expression can be assigned a name and recalled at a later time. We refer to these variables as data objects. We use the assignment operator (`<-`) to name an evaluable expression and save it as a data object.

```
> xx <- "hello, what's your name"
> xx
[1] "hello, what's your name"
```

Wrapping the assignment expression in parentheses makes the assignment and displays the data object value(s).

```
> yy <- 5^3   #assignment; no display
> (yy <- 5^3) #assignment; displays evaluation
[1] 125
```

In this book, we might use `(yy <- 5^3)` to display the value of `yy` and save space on the page. In practice, this is more common:

```
> yy <- 5^3
> yy
[1] 125
```

Multiple assignments work and are read from right to left:

```
> aa <- bb <- 99
> aa; bb
[1] 99
[1] 99
```

Finally, the equal sign (=) can be used for assignments, although I prefer and the `<-` symbol:

```
> ages = c(34, 45, 67)
> ages
[1] 34 45 67
```

The reason I prefer `<-` for assigning object names in the workspace is because later we use = for assigning values to function arguments. For example,

```
> x <- 20:25                  #object name assignment
> x
[1] 20 21 22 23 24 25
> sample(x = 1:10, size = 5)  #argument assignments
[1] 9 6 3 2 5
> x
[1] 20 21 22 23 24 25
```

The first `x` is an object name assignment in the workspace which persist during the R session. The second `x` is a function argument assignment which is only recognized locally in the function and only for the duration of the function execution. For clarity, it is better to keep these types of assignments separate in our mind by using different assignment symbols.

Study these previous examples and spend a few minutes using the assignment operator to create and call data objects. Try to use descriptive names if possible. For example, suppose we have data on age categories; we might name the data `agecat`, `age.cat`, or `age_cat`[3]. These are all acceptable.

1.3.3 *Useful R functions*

When we start R you have opened a *workspace*. The first time we use R, the workspace is empty. Every time we create a data object, it is in the workspace. If a data object with the same name already exists, the old data object will be overwritten withou warning, so be careful. To list the objects in your workspace use the `ls` or `objects` functions:

```
> ls() ##display workspace objects
character(0)
> x <- 1:5
> ls()
[1] "x"
> x
[1] 1 2 3 4 5
> x <- 10:15 ##overwrites without warning
> x
[1] 10 11 12 13 14 15
```

Data objects can be saved between sessions. We will be prompted with "Save workspace image?" (You can also use `save.image()` at the command line.) The workspace image is saved in a file called `.RData`.[4] Use `getwd()` to display the file path to the `.RData` file. Table 1.3 on the following page has more useful R functions.

1.3.3.1 What are packages?

R has many available functions. When we open R, several packages are attached by default. Each package has its own suite of functions. To display the list of attached packages use the `search` function.

```
> search() # Linux
[1] ".GlobalEnv"        "package:stats"     "package:graphics"
```

---

[3] In older versions of R, the underscore symbol (_) could be used for assignments, but this is no longer permitted. The "_" symbol can be used to make your object name more readable and is consistent with other software.

[4] In some operating systems files names that begin with a period (.) are hidden files and are not displayed by default. You may need to change the viewing option to see the file.

Table 1.3 Useful R functions

Function	Description	Try these examples
q	Quit R	`q()`
ls objects	List objects	`ls()` `objects() #equivalent`

rm
remove
	Remove object(s)	`yy <- 1:5; ls()`
		`rm(yy); ls()`
		`#remove everything: caution!`
		`rm(list = ls()); ls()`

help	Open help instructions; or get help on specific topic.	`help()`
		`help(plot)`
		`?plot #equivalent`

help.search	Search help system given character string	`help.search("print")`
help.start	Start help browser	`help.start()`
apropos	Displays all objects in the search list matching topic	`apropos(plot)`
getwd	Get working directory	`getwd()`
setwd	Set working directory	`setwd("c:\mywork\rproject")`
args	Display arguments of function	`args(sample)`
example	Runs example of function	`example(plot)`
data	Information on available R data sets; load data set	`data() #displays data sets` `data(Titanic) #load data set`
save.image	Saves current workspace to .RData	`save.image()`

```
[4] "package:grDevices" "package:utils"     "package:datasets"
[7] "package:methods"   "Autoloads"         "package:base"
```

To display the file paths to the packages use the `searchpaths` function.

```
> searchpaths() # Linux
[1] ".GlobalEnv"                      "/usr/lib/R/library/stats"
[3] "/usr/lib/R/library/graphics"  "/usr/lib/R/library/grDevices"
[5] "/usr/lib/R/library/utils"     "/usr/lib/R/library/datasets"
[7] "/usr/lib/R/library/methods"   "Autoloads"
[9] "/usr/lib/R/library/base"
```

To learn more about a specific package enter `library(help=`*package_name*`)`. Alternatively, we can get more detailed information by entering `help.start()` which opens the HTML help page. On this page click on the Packages link to see the available packages. If we need to load a package enter `library(`*package_name*`)`. For example, when we cover survival analysis we will need to load the `survival` package.

1.3.3.2 What are function arguments?

We will be using many R functions for data analysis, so we need to know some function basics. Suppose we are interested in taking a random sample of days from the month of June, which has 30 days. We want to use the `{sample}` function but we forgot the syntax. Let's explore:

```
> sample
function (x, size, replace = FALSE, prob = NULL)
{
    if (length(x) == 1 && x >= 1) {
        if (missing(size))
            size <- x
        .Internal(sample(x, size, replace, prob))
    }
    else {
        if (missing(size))
            size <- length(x)
        x[.Internal(sample(length(x), size, replace, prob))]
    }
}
<environment: namespace:base>
```

Whoa! What happened? Whenever we type the function name without any parentheses it usually returns the whole function code. This is useful when we start programming and we need to alter an existing function, borrow code for our own functions, or study the code for learning how to program. If we are already familiar with the `sample` function we may only need to see the syntax of the function arguments. For this we use the `args` function:

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

The terms `x`, `size`, `replace`, and `prob` are the function arguments. First, notice that `replace` and `prob` have default values; that is, we do not need to specify these arguments unless we want to override the default values. Second, notice the order of the arguments. If you enter the argument values in the same order as the argument list we do not need to specify the argument.

```
> dates <- 1:30
> sample(dates, 18)
 [1] 29 12 28  8  5  2 13 24 20 18 11 14 23  1 21  4 22  9
```

Third, if we enter the arguments out of order then we will get either an error message or an undesired result. Arguments entered out of their default order need to be specified.

```
> sample(18, dates) #gives undesired results
[1] 2
> #No! We wanted sample of size = 18
> sample(size = 18, x = dates) #gives desired result
 [1]  6 29  1 27 13  7 24 19 21  5 22 12 14 23 25 15 18 11
```

Fourth, when we specify an argument we only need to type a sufficient number of letters so that R can uniquely identify it from the other arguments.

```
> sample(s = 18, x = dates, r = T) #sampling with replacement
 [1] 23 10 23 27 13 14  1  7 23 26 28  3 23 28  9  6 23  5
```

Fifth, argument values can be any valid R expression (including functions) that yields to an appropriate value. In the following example we see two sample functions that provide random values to the `sample` function arguments.

```
> sample(s = sample(1:100, 1), x = sample(1:10, 5), r=T)
 [1]  3  4  9  3  3  9 10  3 10  3 10  4  9  3  5  9  4  5
```

Finally, if we need more guidance on how to use the `sample` function enter `?sample` or `help(sample)`.

1.3.4 How do I get help?

R has extensive help capabilities. From the main menu select Help to get you started (Figure 1.1 on the next page). The Frequently Asked Questions (FAQ) and R manuals are available from this menu. The R functions (text)..., Html help, Search help..., and Apropos... selections are also available from the command line.

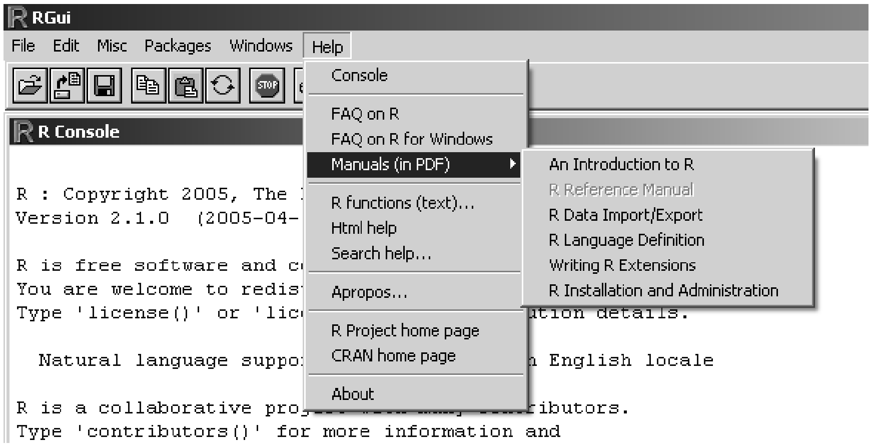
From the command line, we have several options. Suppose you are interested in learning abouting help capabilities.

```
> ?help                #opens help page for 'help' function
> help()               #opens help page for 'help' function
> help(help)           #opens help page for 'help' function
> help.start()         #starts HTML help page
> help.search("help")  #searches help system for "help"
> apropos("help")      #displays 'help' objects in search list
> apropos("help")
[1] "help"         "help.request" "help.search"  "help.start"
```

To learn about about available data sets use the `data` function:

```
> data()                              #displays available data sets
> try(data(package = "survival"))     #lists survival pkg data sets
> help(pbc, package = "survival")     #displays pbc data help page
```

Figure 1.1 on the facing page shows that a R Graphical User Interface (GUI) main menu will have a Help selection.


RGui
File
Edit
Misc
Packages
Windows
Help
R Console
Console
FAQ on R
FAQ on R for Windows
Manuals (in PDF)
R functions (text)...
Html help
Search help...
Apropos...
R Project home page
CRAN home page
About
An Introduction to R
R Reference Manual
R Data Import/Export
R Language Definition
Writing R Extensions
R Installation and Administration

```
R : Copyright 2005, The
Version 2.1.0  (2005-04-

R is free software and c
You are welcome to redis
Type 'license()' or 'lic                 ution details.

  Natural language suppo                 English locale

R is a collaborative pro                 ributors.
Type 'contributors()' for more information and
```

Fig. 1.1 Select Help from the main menu in the MS Windows R GUI.

1.3.5 RStudio—An integrated development environment for R

RStudio[5] is a free and open source integrated development environment (IDE) for R that runs on any desktop (Windows, Mac, or Linux) (Figure 1.2 on the next page). For our purposes, I highly recommend the installation of RStudio: it has all the tools we need to learn and apply R.

1.3.6 Is there anything else that I need?

Maybe.[6] A good text editor will make your programming and data processing easier and more efficient. A text editor is a program for, we guessed it, editing text! The functionality we look for in a text editor are the following:

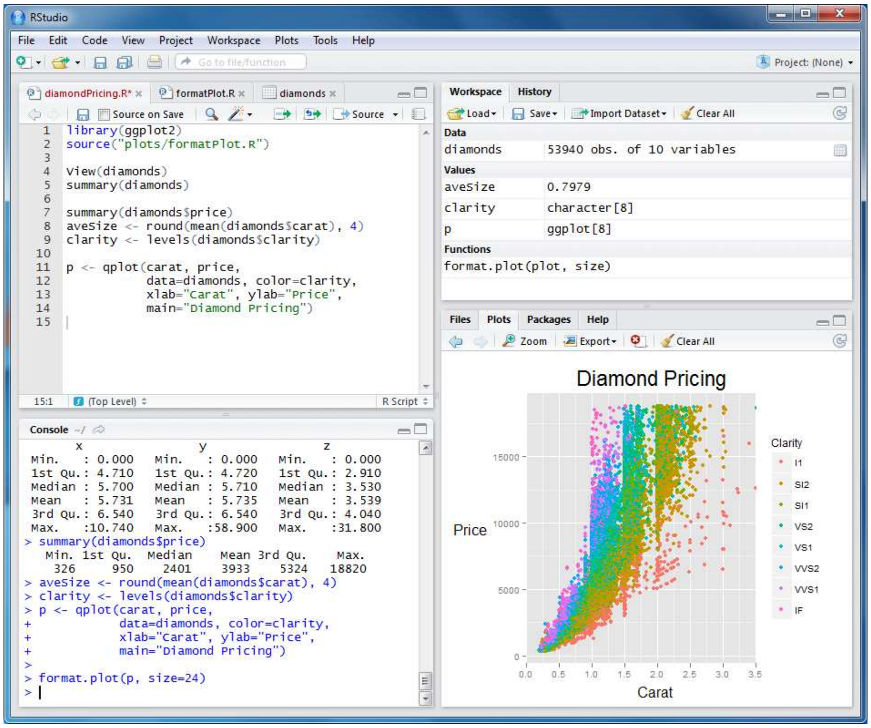
1. Toggle between wrapped and unwrapped text
2. Block cutting and pasting (also called column editing)
3. Easy macro programming
4. Search and replace using regular expressions
5. Ability to import large datasets for editing

When we are programming we want our text to wrap so we can read all of your code. When we import a data set that is wider than the screen, we do not want the data set to wrap: we want it to appear in its tabular format. Column editing allows us to cut and paste columns of text at will. A macro is just a way for the text editor to learn a set of keystrokes (including search and replace) that can be executed

---

[5] `http://www.rstudio.org/`

[6] If your only goal is to learn R, then RStudio is more than sufficient.


RStudio
File Edit Code View Project Workspace Plots Tools Help
Go to file/function
Project: (None)
diamondPricing.R*
formatPlot.R
diamonds
Source on Save
Source
15:1
(Top Level)
R Script
Console ~/
Workspace
History
Load
Save
Import Dataset
Clear All
Files
Plots
Packages
Help
Zoom
Export
Clear All
Diamond Pricing
Price
15000
10000
5000
0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
Carat
Clarity
I1
SI2
SI1
VS2
VS1
VVS2
VVS1
IF

```
library(ggplot2)
source("plots/formatPlot.R")

View(diamonds)
summary(diamonds)

summary(diamonds$price)
aveSize <- round(mean(diamonds$carat), 4)
clarity <- levels(diamonds$clarity)

p <- qplot(carat, price,
           data=diamonds, color=clarity,
           xlab="Carat", ylab="Price",
           main="Diamond Pricing")

```

```
Data
diamonds       53940 obs. of 10 variables
Values
aveSize        0.7979
clarity        character[8]
p              ggplot[8]
Functions
format.plot(plot, size)
```

```
       x               y               z         
 Min.   : 0.000   Min.   : 0.000   Min.   : 0.000  
 1st Qu.: 4.710   1st Qu.: 4.720   1st Qu.: 2.910  
 Median : 5.700   Median : 5.710   Median : 3.530  
 Mean   : 5.731   Mean   : 5.735   Mean   : 3.539  
 3rd Qu.: 6.540   3rd Qu.: 6.540   3rd Qu.: 4.040  
 Max.   :10.740   Max.   :58.900   Max.   :31.800  
> summary(diamonds$price)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
    326     950    2401    3933    5324   18820 
> aveSize <- round(mean(diamonds$carat), 4)
> clarity <- levels(diamonds$clarity)
> p <- qplot(carat, price,
+            data=diamonds, color=clarity,
+            xlab="Carat", ylab="Price",
+            main="Diamond Pricing")
> 
> format.plot(p, size=24)
> 
```

Fig. 1.2 RStudio—An integrated development environment for R that runs in Linux, Mac OS, or MS Windows. In this Figure, RStudio is running in MS Windows.

Fig. 1.3 Emacs and ESS in the Mac OS

as needed. Searching using regular expressions means searching for text based on relative attributes. For example, suppose you want to find all words that begin with "b," end with "g," have any number of letters in between but not "r" and "f." Regular expression searching makes this a trivial task. These are powerful features that once we use regularly, we will wonder how we ever got along without them.

If we do not want to install a text editing program then we can use the default text editor that comes with our computer operating system (gedit in Ubuntu Linux, TextEdit in Mac OS, Notepad in Windows). However, it is much better to install a text editor that works with R. My favorite text editor is the free and open source GNU Emacs. GNU Emacs can be extended with the "Emacs Speaks Statistics" (ESS) package. For more information on Emacs and ESS pre-installed for Windows, visit `http://ess.r-project.org/`. For the Mac OS, I recommend GNU Emacs for Mac OS[7] (Figure 1.3) or Aquamacs.[8]

---

[7] `http://emacsformacosx.com/`

[8] `http://aquamacs.org/`

Table 1.4 Types of actions taken on R data objects and where to find examples

Action	Vector	Matrix	Array	List	Data Frame
Creating	Table 2.4 (p. 34)	Table 2.11 (p. 50)	Table 2.18 (p. 65)	Table 2.24 (p. 79)	Table 2.30 (p. 88)
Naming	Table 2.5 (p. 37)	Table 2.12 (p. 54)	Table 2.19 (p. 68)	Table 2.25 (p. 80)	Table 2.31 (p. 89)
Indexing	Table 2.6 (p. 39)	Table 2.13 (p. 56)	Table 2.20 (p. 69)	Table 2.26 (p. 80)	Table 2.32 (p. 90)
Replacing	Table 2.7 (p. 41)	Table 2.14 (p. 57)	Table 2.21 (p. 69)	Table 2.27 (p. 82)	Table 2.33 (p. 93)
Operating on	Table 2.8 (p. 42) Table 2.9 (p. 46)	Table 2.15 (p. 58)	Table 2.22 (p. 70)	Table 2.28 (p. 83)	Table 2.34 (p. 93)

1.3.7 What's ahead?

To the novice user, R may seem complicated and difficult to learn. In fact, for its immense power and versatility, R is easier to learn and deploy compared to other statistical software (e.g. SAS, Stata, SPSS). This is because R was built from the ground up to be an efficient and intuitive programming language and environment. If one understands the logic and structure of R, then learning proceeds quickly. Just like a spoken language, once we know its rules of grammar, syntax, and pronunciation, and can write legible sentences, we can figure out how to communicate almost anything. Before the next chapter, we want to describe the "forest": the logic and structure of working with R objects and epidemiologic data.

1.3.7.1 Working with R objects

For our purposes, there are only five types of data objects in R[9] and five types of actions we take on these objects (Table 1.4). That's it! No more, no less. You will learn to create, name, index (subset), replace components of, and operate on these data objects using a systematic, comprehensive approach. As you learn about each new data object type, it will reinforce and extend what you learned previously.

A *vector* is a collection of elements (usually numbers):

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

A *matrix* is a 2-dimensional representaton of a vector:

---

[9] The sixth type of R object is a function. Functions can create, manipulate, operate on, and store data; however, we will use functions primarily to execute a series of R "commands" and not as primary data objects.

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

An *array* is a 3 or more dimensional represention of a vector:

```
> z <- array(x, dim = c(2, 3, 2))
> z
, , 1

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

, , 2

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

A *list* is a collection of “bins,” each containing any kind of R object:

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

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

[[3]]
, , 1

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

, , 2

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

A *data frame* is a list in tabular form where each “bin” contains a data vector of the same length. A data frame is the usual tabular data set familiar to epidemiologists. Each row is an record and each column (“bin”) is a field.

```
> kids <- c("Tomasito", "Lusito", "Angelita")
> gender <- c("boy", "boy", "girl")
> age <- c(8, 7, 4)
> mydf <- data.frame(kids, gender, age)
> mydf
      kids gender age
1 Tomasito    boy   8
2   Lusito    boy   7
3 Angelita   girl   4
```

In the next chapter we explore these R data objects in greater detail.

Problems

1.1. If you have not done so already, install R on your personal computer. What is the R workspace file on your operating systems? What is the file path to your R workspace file? What is the name of this workspace file?

1.2. By default, which R packages come already loaded? What are the file paths to the default R packages?

1.3. List all the object in the current workspace. If there are none, create some data objects. Using one expression, remove all the objects in the current workspace.

1.4. One inch equals 2.54 centimeters. Correct the following R code and create a conversion table.

```
inches <- 1:12
centimeters <- inches/2.54
cbind(inches, centimeters)
```

1.5. To convert between temperatures in degrees Celsius (C) and Farenheit (F), we use the following conversion formulas:

$$C = (F - 32)\frac{5}{9}$$

$$F = \frac{9}{5}C + 32$$

At standard temperature and pressure, the freezing and boiling points of water are 0 and 100 degrees Celsius, respectively. What are the freezing and boiling points of water in degrees Fahrenheit?

1.6. For the Celsius temperatures 0, 5, 10, 15, 20, 25, ..., 80, 85, 90, 95, 100, construct a conversion table that displays the corresponding Fahrenheit temperatures. Hint: to create the sequence of Celsius temperatures use `seq(0, 100, 5)`.

1.7. BMI is a reliable indicator of total body fat, which is related to the risk of disease and death. The score is valid for both men and women but it does have some limits. BMI does have some limitations: it may overestimate body fat in athletes and others who have a muscular build, it may underestimate body fat in older persons and others who have lost muscle mass.

Table 1.5 Body mass index classification

BMI	Classification
< 18.5	Underweight
$[18.5, 25)$	Normal weight
$[25, 30)$	Overweight
≥ 30	Obesity

Body Mass Index (BMI) is calculated from your weight in kilograms and height in meters:

$$BMI = \frac{kg}{m^2}$$

$$1\,\text{kg} \approx 2.2\,\text{lb}$$

$$1\,\text{m} \approx 3.3\,\text{ft}$$

Calculate your BMI (don't report it to us).

1.8. Using Table 1.1 on page 8, explain in words, and use R to illustrate, the difference between modulus and integer divide.

1.9. In mathematics, a logarithm (to base b) of a number x is written $\log_b(x)$ and equals the exponent y that satisfies $x = b^y$. In other words,

$$y = \log_b(x)$$

is equivalent to

$$x = b^y$$

In R, the `log` function is to the base e. Implement the following R code and study the graph:

```
curve(log(x), 0, 6)
abline(v = c(1, exp(1)), h = c(0, 1), lty = 2)
```

What kind of generalizations can you make about the natural logarithm and its base—the number e?

1.10. Risk (R) is a probability bounded between 0 and 1. Odds is the following transformation of R:

$$Odds = \frac{R}{1-R}$$

Use the following code to plot the odds:

```
curve(x/(1-x), 0, 1)
```

Now, use the following code to plot the log(odds):

```
curve(log(x/(1-x)), 0, 1)
```

What kind of generalizations can you make about the log(odds) as a transformation of risk?

1.11. Use the data in Table 1.6 on the next page. Assume one is HIV-negative. If the probability of infection per act is p, then the probability of not getting infected per act is $(1-p)$. The probability of not getting infected after 2 consecutive acts is $(1-p)^2$, and after 3 consecutive acts is $(1-p)^3$. Therefore, the probability of not getting

Table 1.6 Estimated per-act risk (transmission probability) for acquisition of HIV, by exposure route to an infected source. Source: CDC [1]

Exposure route	Risk per 10,000 exposures
Blood transfusion (BT)	9,000
Needle-sharing injection-drug use (IDU)	67
Receptive anal intercourse (RAI)	50
Percutaneous needle stick (PNS)	30
Receptive penile-vaginal intercourse (RPVI)	10
Insertive anal intercourse (IAI)	6.5
Insertive penile-vaginal intercourse (IPVI)	5
Receptive oral intercourse on penis (ROI)	1
Insertive oral intercourse with penis (IOI)	0.5

infected infected after n consecutive acts is $(1-p)^n$, and the probability of getting infected after n consecutive acts is $1-(1-p)^n$. For each non-blood transfusion transmission probability (per act risk) in Table 1.6, calculate the cumulative risk of being infected after one year (365 days) if one carries out the same act once daily for one year with an HIV-infected partner. Do these cumulative risks make intuitive sense? Why or why not?

1.12. The `source` function in R is used to "source" (read in) ASCII text files. Take a group of R commands that worked from a previous problem above and paste them into an ASCII text file and save it with the name job01.R. Then from R command line, source the file. Here is how it looked on my Linux computer running R:

```
> source("/home/tja/Documents/courses/ph251d/jobs/job01.R")
```

Describe what happened. Now, set `echo` option to TRUE.

```
> source("/home/tja/Documents/courses/ph251d/jobs/job01.R", echo = TRUE)
```

Describe what happened. To improve your understanding read the help file on the `source` function.

1.13. Now run the `source` again (without and with `echo =` TRUE) but each time create a log file using the `sink` function. Create two log files: job01.log1a and job01.log1b.

```
> sink("/home/tja/Documents/courses/ph251d/jobs/job01.log1a")
> source("/home/tja/Documents/courses/ph251d/jobs/job01.R")
> sink() #closes connection
>
> sink("/home/tja/Documents/courses/ph251d/jobs/job01.log1b")
> source("/home/tja/Documents/courses/ph251d/jobs/job01.R", echo = TRUE)
> sink() #closes connection
```

Examine the log files and describe what happened.

1.14. Create a new job file (job02.R) with the following code:

```
n <- 365
per.act.risk <- c(0.5, 1, 5, 6.5, 10, 30, 50, 67)/10000
risks <- 1-(1-per.act.risk)^n
show(risks)
```

Source this file at the R command line and describes what happens.

CHAPTER 2

Working with R data objects

2.1 Data objects in R

2.1.1 Atomic vs. recursive data objects

The analysis of data in R involves creating, manipulating, and operating on data objects using functions. Data in R are organized as objects and have been assigned a name. We have already been introduced to several R data objects. We will now make some additional distinctions. Every data object has a *mode* and *length*. The mode of an object describes the type of data it contains and is available by using the `mode` function. An object can be of mode character, numeric, logical, list, or function.

```
> fname <- c("Juan", "Miguel"); mode(fname)
[1] "character"
> age <- c(34, 20); mode(age)
[1] "numeric"
> lt25 <- age<25
> lt25
[1] FALSE  TRUE
> mode(lt25)
[1] "logical"
> mylist <- list(fname, age); mode(mylist)
[1] "list"
> mydat <- data.frame(fname, age); mode(mydat)
[1] "list"
> myfun <- function(x) {x^2}
> myfun(5)
```

```
[1] 25
> mode(myfun)
[1] "function"
```

Data objects are further categorized into atomic or recursive objects. An *atomic* data object can only contain elements from one, and only one, of the following modes: character, numeric, or logical. Vectors, matrices, and arrays are atomic data objects. A *recursive* data object can contain data objects of any mode. Lists, data frames, and functions are recursive data objects. We start by reviewing atomic data objects.

A *vector* is a collection of like elements without dimensions.[1] The vector elements are all of the same mode (either character, numeric, or logical). When R returns a vector the `[n]` indicates the position of the element displayed to its immediate right.

```
> y <- c("Pedro", "Paulo", "Maria")
> y
[1] "Pedro" "Paulo" "Maria"
> x <- c(1, 2, 3, 4, 5)
> x
[1] 1 2 3 4 5
> x < 3
[1]  TRUE  TRUE FALSE FALSE FALSE
```

A *matrix* is a collection of like elements organized into a 2-dimensional (tabular) data object. We can think of a matrix as a vector with a 2-dimensional structure. When R returns a matrix the `[n,]` indicates the *n*th row and `[,m]` indicates the *m*th column.

```
> x <- c("a", "b", "c", "d")
> y <- matrix(x, 2, 2)
> y
     [,1] [,2]
[1,] "a"  "c"
[2,] "b"  "d"
```

An *array* is a collection of like elements organized into a *n*-dimensional data object. We can think of an array as a vector with an *n*-dimensional structure. When R returns an array the `[n,,]` indicates the *n*th row and `[,m,]` indicates the *m*th column, and so on.

```
> x <- 1:8
> y <- array(x, dim=c(2, 2, 2))
> y
, , 1
```

---

[1] In other programming languages, vectors are either row vectors or column vectors. R does not make this distinction until it is necessary.

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

, , 2

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

If we try to include elements of different modes in an atomic data object, R will *coerce* the data object into a single mode based on the following hierarchy: character $>$ numeric $>$ logical. In other words, if an atomic data object contains any character element, all the elements are coerced to character.

```
> c("hello", 4.56, FALSE)
[1] "hello" "4.56"  "FALSE"
> c(4.56, FALSE)
[1] 4.56 0.00
```

A *recursive* data object can contain one or more data objects where each object can be of any mode. Lists, data frames, and functions are recursive data objects. Lists and data frames are of mode list, and functions are of mode function (Table 2.1 on the next page).

A *list* is a collection of data objects without any restrictions:

```
> x <- c(1, 2, 3)
> y <- c("Male", "Female", "Male")
> z <- matrix(1:4, 2, 2)
> mylist <- list(x, y, z)
> mylist
[[1]]
[1] 1 2 3

[[2]]
[1] "Male"   "Female" "Male"

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

A *data frame* is a list with a 2-dimensional (tabular) structure. Epidemiologists are very experienced working with data frames where each row usually represents data collected on individual subjects (also called records or observations) and columns represent fields for each type of data collected (also called variables).

```
> subjno <- c(1, 2, 3, 4)
```

Table 2.1 Summary of six types of data objects in R

Data object	Possible mode	Default class
Atomic		
vector	character, numeric, logical	NULL
matrix	character, numeric, logical	NULL
array	character, numeric, logical	NULL
Recursive		
list	list	NULL
data frame	list	data frame
function	function	NULL

```
> age <- c(34, 56, 45, 23)
> sex <- c("Male", "Male", "Female", "Male")
> case <- c("Yes", "No", "No", "Yes")
> mydat <- data.frame(subjno, age, sex, case)
> mydat
  subjno age    sex case
1      1  34   Male  Yes
2      2  56   Male   No
3      3  45 Female   No
4      4  23   Male  Yes
> mode(mydat)
[1] "list"
```

2.1.2 Assessing the structure of data objects

Summarized in Table 2.1 are the key attributes of atomic and recursive data objects. Data objects can also have *class* attributes. Class attributes are just a way of letting R know that an object is "special," allowing R to use specific methods designed for that class of objects (e.g., `print`, `plot`, and `summary` methods). The `class` function displays the class if it exists. For our purposes, we do not need to know any more about classes.

Frequently, we need to assess the structure of data objects. We already know that all data objects have a *mode* and *length* attribute. For example, let's explore the infert data set that comes with R. The `infert` data comes from a matched case-control study evaluating the occurrence of female infertility after spontaneous and induced abortion.

```
> data(infert) #loads data
> mode(infert)
[1] "list"
> length(infert)
```

```
[1] 8
```

At this point we know that the data object named “infert” is a list of length 8. To get more detailed information about the structure of `infert` use the `str` function (`str` comes from “str”ucture).

```
> str(infert)
'data.frame':   248 obs. of  8 variables:
 $ education     : Factor w/ 3 levels "0-5yrs",..: 1 1 1 1 ...
 $ age           : num  26 42 39 34 35 36 23 32 21 28 ...
 $ parity        : num  6 1 6 4 3 4 1 2 1 2 ...
 $ induced       : num  1 1 2 2 1 2 0 0 0 0 ...
 $ case          : num  1 1 1 1 1 1 1 1 1 1 ...
 $ spontaneous   : num  2 0 0 0 1 1 0 0 1 0 ...
 $ stratum       : int  1 2 3 4 5 6 7 8 9 10 ...
```

Great! This is better. We now know that `infert` is a data frame with 248 observations and 8 variables. The variable names and data types are displayed along with their first few values. In this case, we now have sufficient information to start manipulating and analyzing the infert data set.

Additionally, we can extract more detailed structural information that becomes useful when we want to extract data from an object for further manipulation or analysis (Table 2.2 on the next page). We will see extensive use of this when we start programming in R.

To get practice calling data from the command line, enter `data()` to display the available data sets in R. Then enter `data(`*`data_set`*`)` to load a dataset. Study the examples in Table 2.2 on the following page and spend a few minutes exploring the structure of the data sets we have loaded. To display detailed information about a specific data set use `?`*data_set* at the command prompt (e.g., `?infert`).

2.2 A vector is a collection of like elements

2.2.1 Understanding vectors

A vector is a collection of like elements (i.e., the elements all have the same mode). There are many ways to create vectors (see Table 8). The most common way of creating a vector is using the concatenation function `c`:

```
> #numeric
> chol <- c(136, 219, 176, 214, 164)
> chol
[1] 136 219 176 214 164
> #character
> fname <- c("Mateo", "Mark", "Luke", "Juan", "Jaime")
```

Table 2.2 Useful functions to assess R data objects

Function	Description	Try these examples
Returns summary objects		
str	Displays summary of data object structure	`str(infert)`
attributes	Returns list with data object attributes	`attributes(infert)`
Returns specific information		
mode	Returns mode of object	`mode(infert)`
class	Returns class of object, if it exists	`class(infert)`
length	Returns length of object	`length(infert)`
dim	Returns vector with object dimensions, if applicable	`dim(infert)`
nrow	Returns number of rows, if applicable	`nrow(infert)`
ncol	Returns number of columns, if applicable	`ncol(infert)`
dimnames	Returns list containing vectors of names for each dimension, if applicable	`dimnames(infert)`
rownames	Returns vector of row names of a matrix-like object	`rownames(infert)`
colnames	Returns vector of column names of a matrix-like object	`colnames(infert)`
names	Returns vector of names for the list, if applicable (for a data frame it returns the field names)	`names(infert)`
row.names	Returns vector of row names for a data frame	`row.names(infert)`
head	Display first 6 lines of a data frame	`head(infert)` `infert[1:6,] #equivalent`

```
> fname
[1] "Mateo" "Mark"  "Luke"  "Juan"  "Jaime"
> #logical
> z <- c(T, T, F, T, F)
> z
[1]  TRUE  TRUE FALSE  TRUE FALSE
```

A single digit is also a vector; that is, a vector of length = 1. Let’s confirm this.

```
> 5
[1] 5
> is.vector(5)
[1] TRUE
```

2.2.1.1 Boolean operations on vectors

In R, we use *relational* and *logical* operators (Table 2.3 on page 33) to conduct Boolean queries. Boolean operations is a methodological workhorse of data analysis. For example, suppose we have a vector of female movie stars and a corresponding vector of their ages (as of January 16, 2004), and we want to select a subset of actors based on age criteria.

```
> movie.stars
[1] "Rebecca De Mornay"  "Elisabeth Shue"  "Amanda Peet"
[4] "Jennifer Lopez"  "Winona Ryder" "Catherine Zeta Jones"
[7] "Reese Witherspoon"
> ms.ages
[1] 42 40 32 33 32 34 27
```

Let's select the actors who are in their 30s. This is done using logical vectors that are created by using relational operators (<, >, <=, >=, ==, `!=`). Study the following example:

```
> #logical vector for stars with ages >=30
> ms.ages >= 30
[1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
> #logical vector for stars with ages <40
> ms.ages < 40
[1] FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
> #logical vector for stars with ages >=30 and <40
> (ms.ages >= 30) & (ms.ages < 40)
[1] FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE
> thirtysomething <- (ms.ages >= 30) & (ms.ages < 40)
> #indexing vector based on logical vector
> movie.stars[thirtysomething]
[1] "Amanda Peet"       "Jennifer Lopez"     "Winona Ryder"
[4] "Catherine Zeta Jones"
```

We also saw that we can compare logical vectors using logical operators (`&`, `|`, `!`). For more examples see Table 7. The expression `movie.stars[thirtysomething]` is an example of indexing using a logical vector. Now, we can use the `!` function to select the stars that are not "thirtysomething." Study the following:

```
> thirtysomething
[1] FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE
> !thirtysomething
[1]  TRUE  TRUE FALSE FALSE FALSE FALSE  TRUE
> movie.stars[!thirtysomething]
[1] "Rebecca De Mornay"  "Elisabeth Shue"  "Reese Witherspoon"
```

To summarize:

- Logical vectors are created using Boolean comparisons,
- Boolean comparisons are constructed using relational and logical operators
- Logical vectors are commonly used for indexing (subsetting) data objects

Before moving on, we need to be sure we understand the previous examples, then study the examples in Table 2.3 on the facing page. For practice, study the examples and spend a few minutes creating simple numeric vectors, then (1) generate logical vectors using relational operators, (2) use these logical vectors to index the original numerical vector or another vector, (3) generate logical vectors using the combination of relational and logical operators, and (4) use these logical vectors to index the original numerical vector or another vector.

The element-wise exclusive "or" operator (`xor`) returns TRUE if either comparison element is TRUE, but not if both are TRUE. In contrast, the `|` returns TRUE if either or both comparison elements are TRUE.

The `&&` and `||` logical operators are used for control flow in `if` functions. If logical vectors are provided, only the first element of each vector is used. Therefore, for element-wise comparisons of 2 or more vectors, use the `&` and `|` operators but not the `&&` and `||` operators.

2.2.2 Creating vectors

Vectors are created directly, or indirectly as the result of manipulating an R object. The `c` function for concatenating a collection has been covered previously. Another, possibly more convenient, method for collecting elements into a vector is with the `scan` function.

```
> x <- scan()
1:  45 67 23 89
5:
Read 4 items
> x
[1] 45 67 23 89
```

This method is convenient because we do not need to type `c`, parentheses, and commas to create the vector. The vector is created after executing a newline twice.

To generate a sequence of consecutive integers use the `:` function.

```
> -9:8
 [1] -9 -8 -7 -6 -5 -4 -3 -2 -1  0  1  2  3  4  5  6  7  8
```

However, the `seq` function provides more flexibility in generating sequences. Here are some examples:

```
> seq(1, 5, by = 0.5)    ##specify interval
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
> seq(1, 5, length = 8)  ##specify length
```

Table 2.3 Using Boolean relational and logical operators

Operator	Description	Try these examples
Relational operators		

<	Less than	`pos <- c("p1", "p2", "p3", "p4",`
		`"p5")`
		`x <- c(1, 2, 3, 4, 5)`
		`y <- c(5, 4, 3, 2, 1)`
		`x < y`
		`pos[x < y]`

>	Greater than	`x > y` `pos[x > y]`
<=	Less than or equal to	`x <= y` `pos[x <= y]`
>=	Greater than or equal to	`x >= y` `pos[x >= y]`
==	Equal to	`x == y` `pos[x == y]`
!=	Not equal to	`x != y` `pos[x != y]`
Logical operators		

!	NOT	`x > 2`
		`!(x > 2)`
		`pos[!(x > 2)]`

&	Element-wise AND	`(x > 1) & (x < 5)` `pos[(x > 1) & (x < 5)]`
|	Element-wise OR	`(x <= 1) | (x > 4)` `pos[(x <= 1) | (x > 4)]`

xor	Exclusive OR; similar to `|` for comparing two vectors, but only TRUE if one or the other is true, not both	`xx <- x <= 1`
		`yy <- x > 4`
		`xor(xx, yy)`
		`xx | yy`

Logical operators for `if` function		

&&	Similar to & but used with `if` function	`if(T && T) print("Both TRUE")`
		`# nothing printed if(F && T)`
		`print("Both TRUE")`

||	Similar to `|` but used with `if` function	`if(T || F) print("Either TRUE")`
		`# nothing printed if(F || F)`
		`print("Either TRUE")`

```
[1] 1.0000 1.5714 2.1429 2.7143 3.2857 3.8571 4.4286 5.0000
> x <- 1:8
> seq(1, 5, along = x)   ##by length of other object
[1] 1.0000 1.5714 2.1429 2.7143 3.2857 3.8571 4.4286 5.0000
```

Table 2.4 Common ways of creating vectors

Function	Description	Try these examples

c	Concatenate a collection	`x <- c(1, 2, 3, 4, 5)`
		`y <- c(6, 7, 8, 9, 10)`
		`z <- c(x, y)`

scan	Scan a collection (after entering data press **Enter** twice)	`xx <- scan()`
		`1 2 3 4 5`
		
		`yy <- scan(what = "")`
		`"Javier" "Miguel" "Martin"`
		
		`xx; yy`

:	Generate integer sequence	`1:10` `10:(-4)`

seq	Generate sequence of numbers	`seq(1, 5, by = 0.5)`
		`seq(1, 5, length = 3)`
		`zz <- c("a", "b", "c")`
		`seq(along = zz)`

rep	Replicate argument	`rep("Juan Nieve", 3)`
		`rep(1:3, 4)`
		`rep(1:3, 3:1)`

which	Integer vector from Boolean operation	`age <- c(8, NA, 7, 4)` `which(age<5 | age>=8)`
paste	Paste elements creating a character string	`paste(c("A", "B", "C"), 1:3)` `paste(c("A", "B", "C"), 1:3, sep="")`

`[row#, ]`
or
`[ ,col#]`
	Indexing a matrix returns a vector	`xx <- matrix(1:8, nrow = 2, ncol = 4)`
		`xx[2,]`
		`xx[,3]`

sample	Sample from a vector	`sample(c("H","T"), 20, replace = TRUE)`

runif
rnorm
rbinom
rpois
	Generate random numbers from a probability distribution	`rnorm(10, mean = 50, sd = 19)`
		`runif(n = 10, min = 0, max = 1)`
		`rbinom(n = 10, size = 20, p = 0.5)`
		`rpois(n = 10, lambda = 15)`

as.vector	Coerce data objects into a vector	`mx <- matrix(1:4, nrow = 2, ncol = 2)`
		`mx`
		`as.vector(mx)`

vector	Create vector of specified mode and length	`vector("character", 5)`
		`vector("numeric", 5)`
		`vector("logical", 5)`

character
numeric
logical
	Create vector of specified type	`character(5)`
		`numeric(5)`
		`logical(5)`

These types of sequences are convenient for plotting mathematical equations.[2] For example, suppose we wanted to plot the standard normal curve using the normal equation. For a standard normal curve $\mu = 0$ (mean) and $\sigma = 1$ (standard deviation)

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{ \frac{-(x-\mu)^2}{2\sigma^2} \right\}$$

Here is the R code to plot this equation (see Figure 2.1):

```
mu <- 0; sigma <- 1
x <- seq(-4, 4, .01)
fx <- (1/sqrt(2*pi*sigma^2))*exp(-(x-mu)^2/(2*sigma^2))
plot(x, fx, type = "l", lwd = 2)
```

After assigning values to `mu` and `sigma`, we assigned to `x` a sequence of numbers from -4 to 4 by intervals of 0.01. Using the normal curve equation, for every value of x we calculated $f(x)$, represented by the numeric vector `fx`. We then used the `plot` function to plot x vs. $f(x)$. The optional argument `type="l"` produces a "line" and `lwd=2` doubles the line width. For comparison, we also plotted a density histogram of 500 standard normal variates that were simulated using the `rnorm` function (Figure 2.1).[3]

The `rep` function is used to replicate its arguments. Study the examples that follow:

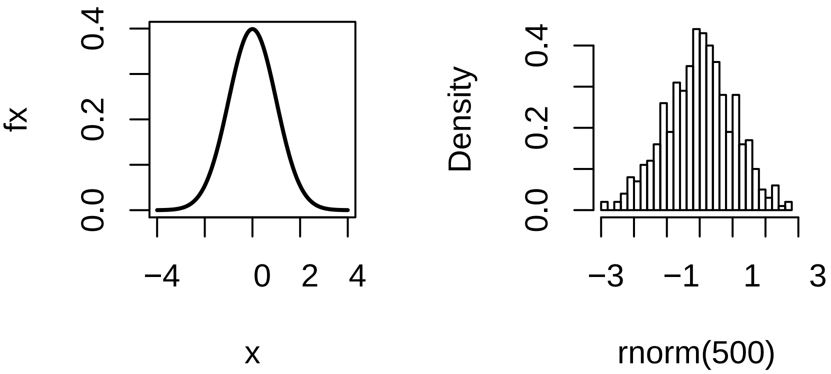


Fig. 2.1 Standard normal curve from equation and simulation

---

[2] See also the `curve` function for graphing mathematical equations.

[3] `hist(rnorm(500), freq = FALSE, breaks = 25, main="")`

```
> rep(5, 2) #repeat 5 2 times
[1] 5 5
> rep(1:2, 5) # repeat 1:2 5 times
 [1] 1 2 1 2 1 2 1 2 1 2
> rep(1:2, c(5, 5)) # repeat 1 5 times; repeat 2 5 times
 [1] 1 1 1 1 1 2 2 2 2 2
> rep(1:2, rep(5, 2)) # equivalent to previous
 [1] 1 1 1 1 1 2 2 2 2 2
> rep(1:5, 5:1) # repeat 1 5 times, repeat 2 4 times, etc
 [1] 1 1 1 1 1 2 2 2 2 3 3 3 4 4 5
```

The `paste` function pastes character strings:

```
> fname <- c("John", "Kenneth", "Sander")
> lname <- c("Snow", "Rothman", "Greenland")
> paste(fname, lname)
[1] "John Snow"      "Kenneth Rothman"  "Sander Greenland"
> paste("var", 1:7, sep="")
[1] "var1" "var2" "var3" "var4" "var5" "var6" "var7"
```

Indexing (subsetting) an object often results in a vector. To preserve the dimensionality of the original object use the `drop` option.

```
> x <- matrix(1:8, 2, 4)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
> x[2,]                    #index 2nd row
[1] 2 4 6 8
> x[2, , drop = FALSE]     #index 2nd row; keep object structure
     [,1] [,2] [,3] [,4]
[1,]    2    4    6    8
```

Up to now we have generated vectors of known numbers or character strings. On occasion we need to *generate random numbers* or *draw a sample* from a collection of elements. First, sampling from a vector returns a vector:

```
> # toss 8 coins
> sample(c("Head","Tail"), size = 8, replace = TRUE)
[1] "Head" "Head" "Tail" "Head" "Tail" "Tail" "Head" "Head"
> # toss 2 die
> sample(1:6, size = 2, replace = TRUE)
[1] 1 4
```

Second, generating random numbers from a probability distribution returns a vector:

```
> # toss 8 coins twice using the binomial distribution
> rbinom(n = 2, size = 8, p = 0.5)
```

Table 2.5 Common ways of naming vectors

Function	Description	Try these examples
c	Name vector elements at time that vector is created	`x <- c(a = 1, b = 2, c = 3, d = 4);` `x`

names	Name vector elements	`y <- 1:4; y`
		`names(y) <- c("a", "b", "c", "d"); y`
		`#return names, if they exist`
		`names(y)`

unname	Remove names	`y <- unname(y)` `y #equivalent: names(y) <- NULL`

```
[1] 5 2
> # generate 6 standard normal distribution values
> rnorm(6)
[1]  1.52826 -0.50631  0.56446  0.10813 -1.96716  2.01802
```

There are additional ways to create vectors. To practice creating vectors study the examples in Table 2.4 on page 34 and spend a few minutes creating simple vectors. If we need help with a function remember enter `?`*`function_name`* or `help(`*`function_name`*`)`.

Finally, notice that we use vectors as arguments to functions:

```
# character vector used in 'sample' function
sample(c("head", "tail"), 100, replace = TRUE)
# numeric vector used in 'rep' function
rep(1:2, rep(5, 2))
# numeric vector used in 'matrix' function
matrix(c(23, 45, 16, 17), nrow = 2, ncol = 2)
```

2.2.3 Naming vectors

The first way of naming vector elements is when the vector is created:

```
> x <- c(chol = 234, sbp = 148, dbp = 78, age = 54)
> x
chol  sbp  dbp  age
 234  148   78   54
```

The second way is to create a character vector of names and then assign that vector to the numeric vector using the `names` function:

```
> z <- c(234, 148, 78, 54)
> z
[1] 234 148  78  54
```

```
> names(z) <- c("chol", "sbp", "dbp", "age")
> z
chol  sbp  dbp  age
 234  148   78   54
```

The `names` function, without an assignment, returns the character vector of names, if it exist. This character vector can be used to name elements of other vectors.

```
> names(z)
[1] "chol" "sbp"  "dbp"  "age"
> z2 <- c(250, 184, 90, 45)
> z2
[1] 250 184  90  45
> names(z2) <- names(z)
> z2
chol  sbp  dbp  age
 250  184   90   45
```

The `unname` function removes the element names from a vector:

```
> unname(z2)
[1] 250 184  90  45
```

For practice study the examples in Table 2.5 on the preceding page and spend a few minutes creating and naming simple vectors.

2.2.4 Indexing vectors

Indexing a vector is subsetting or extracting elements from a vector. A vector is indexed by *position(s)*, by *name(s)*, and by *logical* vector. Positions are specified by positive or negative integers.

```
> x <- c(chol = 234, sbp = 148, dbp = 78, age = 54)
> x[c(2, 4)] #extract 2nd and 4th element
sbp age
148  54
> x[-c(2, 4)] #exclude 2nd and 4th element
chol  dbp
 234   78
```

Although indexing by position is concise, indexing by name (when the names exists) is better practice in terms of documenting our code. Here is an example:

```
> x[c("sbp", "age")] #extract 2nd and 4th element
sbp age
148  54
```

Table 2.6 Common ways of indexing vectors

Indexing	Try these examples

By position	`x <- c(chol = 234, sbp = 148, dbp = 78, age = 54)`
	`x[2] #positions to include`
	`x[c(2, 3)]`
	`x[-c(1, 3, 4)] #positions to exclude`
	`x[-c(1, 4)]`
	`x[which(x<100)]`

By name (if exists)	`x["sbp"]` `x[c("sbp", "dbp")]`

By logical	`x < 100`
	`x[x < 100]`
	`(x < 150) & (x > 70)`
	`bp <- (x < 150) & (x > 70)`
	`x[bp]`

Unique values	`samp <- sample(1:5, 25, replace=T); samp` `unique(samp)`
Duplicated values	`duplicated(samp) #generates logical` `samp[duplicated(samp)]`

A logical vector indexes the positions that corresponds to the TRUEs. Here is an example:

```
> x<=100 | x>200
 chol   sbp   dbp   age
 TRUE FALSE  TRUE  TRUE
> x[x<=100 | x>200]
chol  dbp  age
 234   78   54
```

Any expression that evaluates to a valid vector of integers, names, or logicals can be used to index a vector.

```
> (samp1 <- sample(1:4, 8, replace = TRUE))
[1] 1 3 3 3 1 3 4 1
> x[samp1]
chol  dbp  dbp  dbp chol  dbp  age chol
 234   78   78   78  234   78   54  234
> (samp2 <- sample(names(x), 8, replace = TRUE))
[1] "dbp" "sbp" "sbp" "dbp" "dbp" "age" "dbp" "sbp"
> x[samp2]
dbp sbp sbp dbp dbp age dbp sbp
 78 148 148  78  78  54  78 148
```

Notice that when we indexed by position or name we indexed the same position repeatly. This will not work with logical vectors. In the example that follows NA means “not available.”

```
> (samp3 <- sample(c(TRUE, FALSE), 8, replace = TRUE))
[1] FALSE FALSE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE
> x[samp3]
 dbp <NA> <NA> <NA> <NA>
  78   NA   NA   NA   NA
```

We have already seen that a vector can be indexed based on the characteristics of another vector.

```
> kid <- c("Tomasito", "Irene", "Luisito", "Angelita", "Tomas")
> age <- c(8, NA, 7, 4, NA)
> age<=7 # produces logical vector
[1] FALSE    NA  TRUE  TRUE    NA
> kid[age<=7] # index 'kid' using 'age'
[1] NA         "Luisito"  "Angelita" NA
> kid[!is.na(age)] # remove missing values
[1] "Tomasito" "Luisito"  "Angelita"
> kid[age<=7 & !is.na(age)]
[1] "Luisito"  "Angelita"
```

In this example, `NA` represents missing data. The `is.na` function returns a logical vector with TRUEs at NA positions. To generate a logical vector to index values that are not missing use `!is.na`.

For practice study the examples in Table 2.6 on the previous page and spend a few minutes creating, naming, and indexing simple vectors.

2.2.4.1 The `which` function

A Boolean operation that returns a logical vector contains `TRUE` values where the condition is true. To identify the position of each `TRUE` value we use the `which` function. For example, using the same data above:

```
> which(age<=7) # which positions meet condition
[1] 3 4
> kid[which(age<=7)]
[1] "Luisito"  "Angelita"
```

Notice that is was unnecessary to remove the missing values.

2.2.5 Replacing vector elements (by indexing and assignment)

To replace vector elements we combine indexing and assignment. Any elements of a vector that can be indexed can be replaced. Replacing vector elements is one method of recoding a variable.

Table 2.7 Common ways of replacing vector elements

Replacing	Try these examples

By position	`x <- c(chol = 234, sbp = 148, dbp = 78, age = 54)`
	`x[1]`
	`x[1] <- 250`
	`x`

By name (if exists)	`x["sbp"]`
	`x["sbp"] <- 150`
	`x`

By logical	`x[x<100]`
	`x[x<100] <- NA`
	`x`

```
> # simulate vector with 1000 age values
> age <- sample(0:100, 1000, replace = TRUE)
> mean(age)
[1] 50.378
> sd(age)
[1] 28.25947
> agecat <- age
> agecat[age<15] <- "<15"
> agecat[age>=15 & age<25] <- "15-24"
> agecat[age>=25 & age<45] <- "25-44"
> agecat[age>=45 & age<65] <- "45-64"
> agecat[age>=65] <- "65+"
> table(agecat)
agecat
  <15 15-24 25-44 45-64   65+
  125   107   207   206   355
```

First, we made a copy of the numeric vector `age` and named it `agecat`. Then, we replaced elements of `agecat` with character strings for each age category, creating a character vector.

For practice study the examples in Table 2.7 and spend a few minutes replacing vector elements.

2.2.6 Operations on vectors

Operations on vectors is very common in epidemiology and statistics. In this section we cover common operations on single vectors (Table 2.8 on the next page) and multiple vectors (Table 2.9 on page 46).

Table 2.8 Selected operations on single vectors

Function	Description	Function	Description
sum	summation	range	range
cumsum	cumulative sum	rev	reverse order
diff	x[i+1]-x[i]	order	order
prod	product	sort	sort
cumprod	cumulative product	rank	rank
mean	mean	sample	random sample
median	median	quantile	percentile
min	minimum	var	variance, covariance
max	maximum	sd	standard deviation

2.2.6.1 Operations on single vectors

First, we focus on operating on single numeric vectors (Table 2.8). This also gives us the opportunity to see how common mathematical notation is translated into simple R code.

To sum elements of a numeric vector x of length n, ($\sum_{i=1}^{n} x_i$), use the `sum` function:

```
> # generate and sum 100 random standard normal values
> x <- rnorm(100)
> sum(x)
[1] -0.34744
```

To calculate a cumulative sum of a numeric vector x of length n, ($\sum_{i=1}^{k} x_i$, for $k = 1,\ldots,n$), use the `cumsum` function which returns a vector:

```
# generate sequence of 2's and calculate cumulative sum
> x <- rep(2, 10)
> x
 [1] 2 2 2 2 2 2 2 2 2 2
> cumsum(x)
 [1]  2  4  6  8 10 12 14 16 18 20
```

To multiply elements of a numeric vector x of length n, ($\prod_{i=1}^{n} x_i$), use the `prod` function:

```
> x <- c(1, 2, 3, 4, 5, 6, 7, 8)
> prod(x)
[1] 40320
```

To calculate the cumulative product of a numeric vector x of length n, ($\prod_{i=1}^{k} x_i$, for $k = 1,\ldots,n$), use the `cumprod` function:

```
> x <- c(1, 2, 3, 4, 5, 6, 7, 8)
> cumprod(x)
[1]     1     2     6    24   120   720  5040 40320
```

To calculate the mean of a numeric vector x of length n, ($\frac{1}{n}\sum_{i=1}^{n} x_i$), use the `sum` and `length` functions, or use the `mean` function:

```
> x <- rnorm(100)
> sum(x)/length(x)
[1] 0.05843341
> mean(x)
[1] 0.05843341
```

To calculate the sample variance of a numeric vector x of length n, use the `sum`, `mean`, and `length` functions, or, more directly, use the `var` function.

$$S_X^2 = \frac{1}{n-1}\left[\sum_{i=1}^{n}(x_i - \bar{x})^2\right]$$

```
> x <- rnorm(100)
> sum((x-mean(x))^2)/(length(x)-1)
[1] 0.9073808
> var(x)      # equivalent
[1] 0.9073808
```

This example illustrates how we can implement a formula in R using several functions that operate on single vectors (`sum`, `mean`, and `length`). The `var` function, while available for convenience, is not necessary to calculate the sample variance.

When the `var` function is applied to two numeric vectors, x and y, both of length n, the sample covariance is calculated:

$$S_{XY} = \frac{1}{n-1}\left[\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})\right]$$

```
> x <- rnorm(100); y <- rnorm(100)
> sum((x-mean(x))*(y-mean(y)))/(length(x)-1)
[1] -0.09552851
> var(x, y)     # equivalent
[1] -0.09552851
```

The sample standard deviation, of course, is just the square root of the sample variance (or use the `sd` function):

$$S_X = \sqrt{\frac{1}{n-1}\left[\sum_{i=1}^{n}(x_i - \bar{x})^2\right]}$$

```
> sqrt(var(x))
[1] 0.9525654
> sd(x)
[1] 0.9525654
```

To sort a numeric or character vector use the `sort` function.

```
> ages <- c(8, 4, 7)
> sort(ages)
[1] 4 7 8
```

However, to sort one vector based on the *ordering* of another vector use the `order` function.

```
> ages <- c(8, 4, 7)
> subjects <- c("Tomas", "Angela", "Luis")
> subjects[order(ages)]
[1] "Angela" "Luis"   "Toms"
> # 'order' returns positional integers for sorting
> order(ages)
[1] 2 3 1
```

Notice that the `order` function does not return the data, but rather indexing integers in new positions for sorting the vector age or another vector. For example, `order(ages)` returned the integer vector `c(2, 3, 1)` which means "move the 2nd element ($age = 4$) to the first position, move the 3rd element ($age = 7$) to the second position, and move the 1st element ($age = 8$) to the third position." Verify that `sort(ages)` and `ages[order(ages)]` are equivalent.

To sort a vector in reverse order combine the `rev` and `sort` functions.

```
> x <- c(12, 3, 14, 3, 5, 1)
> sort(x)
[1]  1  3  3  5 12 14
> rev(sort(x))
[1] 14 12  5  3  3  1
```

In contrast to the `sort` function, the `rank` function gives each element of a vector a rank score but does not sort the vector.

```
> x <- c(12, 3, 14, 3, 5, 1)
> rank(x)
[1] 5.0 2.5 6.0 2.5 4.0 1.0
```

The median of a numeric vector is that value which puts 50% of the values below and 50% of the values above, in other words, the 50% percentile (or 0.5 quantile). For example, the median of `c(4, 3, 1, 2, 5)` is 3. For a vector of even length, the middle values are averaged: the median of `c(4, 3, 1, 2)` is 2.5. To get the median value of a numeric vector use the `median` or `quantile` function.

```
> ages <- c(23, 45, 67, 33, 20, 77)
> median(ages)
[1] 39
> quantile(ages, 0.5)
50%
 39
```

To return the minimum value of a vector use the `min` function; for the maximum value use the `max` function. To get both the minimum and maximum values use the `range` function.

```
> ages <- c(23, 45, 67, 33, 20, 77)
> min(ages)
[1] 20
> sort(ages)[1] # equivalent
[1] 20
> max(ages)
[1] 77
> sort(ages)[length(ages)] # equivalent
[1] 77
> range(ages)
[1] 20 77
> c(min(ages), max(ages)) # equivalent
[1] 20 77
```

To sample from a vector of length n, with each element having a default sampling probability of $1/n$, use the `sample` function. Sampling can be with or without replacement (default). If the sample size is greater than the length of the vector, then sampling must occur with replacement.

```
> coin <- c("H", "T")
> sample(coin, size = 10, replace = TRUE)
 [1] "H" "H" "T" "T" "T" "H" "H" "H" "H" "T"
> sample(1:100, 15)
 [1]  9 24 53 11 15 63 52 73 54 84 82 66 65 20 67
```

2.2.6.2 Operations on multiple vectors

Next, we review selected functions that work with one or more vectors. Some of these functions manipulate vectors and others facilitate numerical operations.

In addition to creating vectors, the `c` function can be used to append vectors.

```
> x <- 6:10
> y <- 20:24
> c(x, y)
 [1]  6  7  8  9 10 20 21 22 23 24
```

The `append` function also appends vectors; however, one can specify at which position.

```
> append(x, y)
 [1]  6  7  8  9 10 20 21 22 23 24
> append(x, y, after = 2)
 [1]  6  7 20 21 22 23 24  8  9 10
```

Table 2.9 Selected operations on multiple vectors

Function	Description
c	concatenates vectors
append	Appends a vector to another vector (default is to append at the end of the first vector)
cbind	Column-bind vectors or matrices
rbind	Row-bind vectors or matrices
table	Creates contingency table from 2 or more vectors
xtabs	Creates contingency table from 2 or more factors in a data frame
ftable	Creates flat contingency table from 2 or more vectors
outer	Outer product
tapply	Applies a function to strata of a vector
<, >, <=, >=, ==, !=	Relational operators, See Table 2.3 on page 33
!, &, &&, |, ||, xor	Logical operators, See Table 2.3 on page 33

In contrast, the `cbind` and `rbind` functions concatenate vectors into a matrix. During the outbreak of severe acute respiratory syndrome (SARS) in 2003, a patient with SARS potentially exposed 111 passengers on board an airline flight. Of the 23 passengers that sat “close” to the index case, 8 developed SARS; among the 88 passengers that did not sit “close” to the index case, only 10 developed SARS [2]. Now, we can bind 2 vectors to create a 2×2 table (matrix).

```
> case <- c("exposed" = 8, "unexposed" = 10)
> noncase <- c("exposed" = 15, "unexposed" = 78)
> cbind(case, noncase)
          case noncase
exposed      8      15
unexposed   10      78
> rbind(case, noncase)
        exposed unexposed
case          8        10
noncase      15        78
```

For the example that follows, let’s recreate the SARS data as two character vectors.

```
> outcome <- c(rep("case", 8+10), rep("noncase", 15+78))
> tmp <- c("exposed", "unexposed")
> exposure <- c(rep(tmp, c(8, 10)), rep(tmp, c(15, 78)))
> cbind(exposure,outcome)[1:4,] # display 4 rows
     exposure  outcome
[1,] "exposed" "case"
```

```
[2,] "exposed" "case"
[3,] "exposed" "case"
[4,] "exposed" "case"
```

Now, use the `table` function to cross-tabulate one or more vectors.

```
> table(outcome, exposure)
         exposure
outcome   exposed unexposed
  case      8      10
  noncase 15      78
```

The `ftable` function creates a flat contingency table from one or more vectors.

```
> ftable(outcome, exposure)
        exposure exposed unexposed
outcome
case                   8        10
noncase               15        78
```

This will come in handy later when we want to display a 3 or more dimensional table as a “flat” 2-dimensional table.

The `outer` function applies a function to every combination of elements from two vectors. For example, create a multiplication table for the numbers 1 to 5.

```
> outer(1:5, 1:5, "*")
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    2    4    6    8   10
[3,]    3    6    9   12   15
[4,]    4    8   12   16   20
[5,]    5   10   15   20   25
```

The `tapply` function applies a function to strata of a vector that is defined by one or more “indexing” vectors. For example, to calculate the mean age of females and males:

```
> age <- c(23, 45, 67, 88, 22, 34, 80, 55, 21, 48)
> sex <- c("M", "F", "M", "F", "M", "F", "M", "F", "M", "F")
> tapply(X = age, INDEX = sex, FUN = mean)
   F    M
54.0 42.6
> #  equivalent
> tapply(age, sex, sum)/tapply(age, sex, length)
   F    M
54.0 42.6
```

The `tapply` function is an important and versatile function because it allows us to apply *any* function that can be applied to a vector, to be applied to strata of a vector. Moveover, we can use our user-created functions as well.

Table 2.10 Deaths among subjects who received tolbutamide and placebo in the Unversity Group Diabetes Program (1970)

	Tolbutamide	Placebo
Deaths	30	21
Survivors	174	184

2.3 A matrix is a 2-dimensional table of like elements

2.3.1 Understanding matrices

A matrix is a 2-dimensional table of like elements. Matrix elements can be either numeric, character, or logical. Contingency tables in epidemiology are represented in R as numeric matrices or arrays. An array is the generalization of matrices to 3 or more dimensions (commonly known as stratified tables). We cover arrays later, for now we will focus on 2-dimensional tables.

Consider the 2 × 2 table of crude data in Table 2.10 [3]. In this randomized clinical trial (RCT), diabetic subjects were randomly assigned to receive either tolbutamide, an oral hypoglycemic drug, or placebo. Because this was a prospective study we can calculate risks, odds, a risk ratio, and an odds ratio. We will do this using R as a calculator.

```
> dat <- matrix(c(30, 174, 21, 184), 2, 2)
> rownames(dat) <- c("Deaths", "Survivors")
> colnames(dat) <- c("Tolbutamide", "Placebo")
> coltot <- apply(dat, 2, sum)   #column totals
> risks <- dat["Deaths",]/coltot
> risk.ratio <- risks/risks[2]   #risk ratio
> odds <- risks/(1-risks)
> odds.ratio <- odds/odds[2]     #odds ratio
> # display results
> dat
          Tolbutamide Placebo
Deaths             30      21
Survivors         174     184
> rbind(risks, risk.ratio, odds, odds.ratio)
           Tolbutamide   Placebo
risks        0.1470588 0.1024390
risk.ratio   1.4355742 1.0000000
odds         0.1724138 0.1141304
odds.ratio   1.5106732 1.0000000
```

Now let's review each line briefly to understand the analysis in more detail.

```
dat <- matrix(c(30, 174, 21, 184), 2, 2)
```

We used the `matrix` function to take a vector and convert it into a matrix with 2 rows and 2 columns. Notice the `matrix` function reads in the vector column-wise. To read the vector in row-wise we would add the `byrow=TRUE` option. Try creating a matrix reading in a vector column-wise (default) and row-wise.

```
rownames(dat) <- c("Deaths", "Survivors")
colnames(dat) <- c("Tolbutamide", "Placebo")
```

We used the `rownames` and the `colnames` functions to assign row and column names to the matrix `dat`. The row names and the column names are both character vectors.

```
coltot <- apply(dat, 2, sum)   #column totals
```

We used the `apply` function to sum the columns; it is a versatile function for applying any function to matrices or arrays. The second argument is the `MARGIN` option: in this case, `MARGIN=2`, meaning apply the `sum` function to the columns. To sum the rows, set `MARGIN=1`.

```
risks <- dat["Deaths",]/coltot
risk.ratio <- risks/risks[2]   #risk ratio
```

We calculated the risks of death for each treatment group. We got the numerator by indexing the `dat` matrix using the row name `"Deaths"`. The numerator is a vector containing the deaths for each group and the denominator is the total number of subjects in each group. We calculated the risk ratios using the placebo group as the reference.

```
odds <- risks/(1-risks)
odds.ratio <- odds/odds[2]     #odds ratio
```

Using the definition of the odds, we calculated the odds of death for each treatment group. Then we calculated the odds ratios using the placebo group as the reference.

```
dat
rbind(risks, risk.ratio, odds, odds.ratio)
```

Finally, we display the `dat` table we created. We also created a table of results by row binding the vectors using the `rbind` function.

In the sections that follow we will cover the necessary concepts to make the previous analysis routine.

2.3.2 Creating matrices

There are several ways to create matrices (Table 2.11 on the next page). In general, we create or use matrices in the following ways:

Table 2.11 Common ways of creating a matrix

Function	Description	Try these examples

cbind	Column-bind vectors or matrices	`x <- 1:3`
		`y <- 3:1`
		`z <- cbind(x, y); z`

rbind	Row-bind vectors or matrices	`z2 <- rbind(x, y); z2`
matrix	Generates matrix	`mtx <- matrix(1:4, nrow=2, ncol=2); mtx`
dim	Assign dimensions to a data object	`mtx2 <- 1:4; mtx2` `dim(mtx2) <- c(2, 2); mtx2`
array	Generates matrix when array is 2-dimensional	`mtx <- array(1:4, dim = c(2, 2)); mtx`
table	Creates contingency table	`table(infert$educ, infert$case)`
xtabs	Create a contingency table using a formula interface	`xtabs(~education + case, data = infert)`
ftable	Creates flat contingency table	`ftable(infert$educ, infert$spont, infert$case)`
as.matrix	Coerces object into a matrix	`1:3` `as.matrix(1:3)`
outer	Outer product of two vectors	`outer(1:5, 1:5, "*")`

`x[row, , ]`
`x[ ,col, ]`
`x[ , ,dep]`
	Indexing an array can return a matrix	`x <- array(1:8, c(2, 2, 2))`
		`x[1, , ]`
		`x[ ,1, ]`
		`x[ , ,1]`

- Contingency tables (cross tabulations)
- Spreadsheet calculations and display
- Collecting results into tabular form
- Results of 2-variable equations

2.3.2.1 Contingency tables (cross tabulations)

In the previous section we used the `matrix` function to create the 2×2 table for the UGDP clinical trial:

```
> dat <- matrix(c(30, 174, 21, 184), 2, 2)
> rownames(dat) <- c("Deaths", "Survivors")
> colnames(dat) <- c("Tolbutamide", "Placebo")
> dat
          Tolbutamide Placebo
```

```
Deaths              30      21
Survivors          174     184
```

Alternatively, we can create a 2-way contingency table using the `table` function with fields from a data set;

```
> dat2 <- read.table("http://www.medepi.net/data/ugdp.txt",
+   header = TRUE, sep = ",")
> names(dat2) #display field names
[1] "Status"    "Treatment" "Agegrp"
> table(dat2$Status, dat2$Treatment)

            Placebo Tolbutamide
  Deaths         21          30
  Survivors     184         174
```

Alternatively, the `xtabs` function cross tabulates using a formula interface. An advantage of this function is that the field names are included.

```
> xtabs(~Status + Treatment, data = dat2)
           Treatment
Status      Placebo Tolbutamide
  Deaths         21          30
  Survivors     184         174
```

Finally, a multi-dimensional contingency table can be presented as a 2-dimensional flat contingency table using the `ftable` function. Here we stratify the above table by the variable `Agegrp`.

```
> xtab3way <- xtabs(~Status + Treatment + Agegrp, data=dat2)
> xtab3way
, , Agegrp = <55

           Treatment
Status      Placebo Tolbutamide
  Deaths          5           8
  Survivors     115          98

, , Agegrp = 55+

           Treatment
Status      Placebo Tolbutamide
  Deaths         16          22
  Survivors      69          76

> ftable(xtab3way) #convert to flat table
                      Agegrp <55 55+
Status    Treatment
```

```
Deaths    Placebo             5  16
          Tolbutamide         8  22
Survivors Placebo           115  69
          Tolbutamide        98  76
> #alternative and more consistent with xtab3way
> ftable(xtabs(˜Agegrp + Status + Treatment, data=dat2))
                 Treatment Placebo Tolbutamide
Agegrp Status
<55    Deaths                    5           8
       Survivors               115          98
55+    Deaths                   16          22
       Survivors                69          76
```

2.3.2.2 Spreadsheet calculations and display

Matrices are commonly used to display spreadsheet-like calculations. In fact, a very efficient way to learn R is to use it as our spreadsheet. For example, assuming the rate of seasonal influenza infection is 10 infections per 100 person-years, let's calculate the individual cumulative risk of influenza infection at the end of 1, 5, and 10 years. Assuming no competing risk, we can use the *exponential formula*:

$$R(0,t) = 1 - e^{-\lambda t}$$

where , $\lambda =$ infection rate, and $t =$ time.

```
> lamb <- 10/100
> years <- c(1, 5, 10)
> risk <- 1 - exp(-lamb*tim)
> cbind(rate = lamb, years, cumulative.risk = risk)
      rate years cumulative.risk
[1,]  0.1     1      0.09516258
[2,]  0.1     5      0.39346934
[3,]  0.1    10      0.63212056
```

Therefore, the cumulative risk of influenza infection after 1, 5, and 10 years is 9.5%, 39%, and 63%, respectively.

2.3.2.3 Collecting results into tabular form

A 2-way contingency table from the `table` or `xtabs` functions does not have margin totals. However, we can construct a numeric matrix that includes the totals. Using the UGDP data again,

```
> dat2 <- read.table("http://www.medepi.net/data/ugdp.txt",
+   header = TRUE, sep=",")
```

```
> tab2 <- xtabs(~Status + Treatment, data = dat2)
> rowt <- tab2[,1] + tab2[,2]
> tab2a <- cbind(tab2, Total = rowt)
> colt <- tab2a[1,] + tab2a[2,]
> tab2b <- rbind(tab2a, Total = colt)
> tab2b
          Placebo Tolbutamide Total
Deaths         21          30    51
Survivors     184         174   358
Total         205         204   409
```

This table (`tab2b`) is primarily for display purposes.

2.3.2.4 Results of 2-variable equations

When we have an equation with 2 variables, we can use a matrix to display the answers for every combination of values contained in both variables. For example, consider this equation:

$$z = xy$$

And suppose $x = \{1, 2, 3, 4, 5\}$ and $y = \{6, 7, 8, 9, 10\}$. Here's the long way to create a matrix for this equation:

```
> x <- 1:5; y <- 6:10
> z <- matrix(NA, 5, 5) #create empty matrix of missing values
> for(i in 1:5){
+   for(j in 1:5){
+     z[i, j] <- x[i]*y[j]
+   }
+ }
> rownames(z) <- x; colnames(z) <- y
> z
   6  7  8  9 10
1  6  7  8  9 10
2 12 14 16 18 20
3 18 21 24 27 30
4 24 28 32 36 40
5 30 35 40 45 50
```

Okay, but the `outer` function is much better for this task:

```
> x <- 1:5; y <- 6:10
> z <- outer(x, y, "*")
> rownames(z) <- x; colnames(z) <- y
> z
   6  7  8  9 10
1  6  7  8  9 10
```

Table 2.12 Common ways of naming a matrix

Function — Try these examples

matrix

```
#name rows and columns only
dat <- matrix(c(178, 79, 1411, 1486), 2, 2,
  dimnames = list(c("Type A", "Type B"),
  c("Yes", "No")))
dat
#name rows, columns, and fields
dat <- matrix(c(178, 79, 1411, 1486), 2, 2,
  dimnames = list(Behavior = c("Type A", "Type B"),
  "Heart attack" = c("Yes", "No")))
dat
```

rownames

```
#name rows only
dat <- matrix(c(178, 79, 1411, 1486), 2, 2)
rownames(dat) <- c("Type A", "Type B")
dat
```

colnames

```
#name columns only
dat <- matrix(c(178, 79, 1411, 1486), 2, 2)
colnames(dat) <- c("Yes", "No"); dat
```

dimnames

```
#name rows and columns only
dat <- matrix(c(178, 79, 1411, 1486), 2, 2)
dimnames(dat) <- list(c("Type A", "Type B"),
  c("Yes", "No"))
dat
#name rows, columns, and fields
dat <- matrix(c(178, 79, 1411, 1486), 2, 2)
dimnames(dat) <- list(Behavior = c("Type A", "Type B"),
  "Heart attack" = c("Yes", "No"))
dat
```

names

```
#name fields when row and column names already exist
dat <- matrix(c(178, 79, 1411, 1486), 2, 2,
  dimnames = list(c("Type A", "Type B"),
  c("Yes", "No")))
dat #display w/o field names
names(dimnames(dat)) <- c("Behavior", "Heart attack")
dat #display w/ field names
```

```
2 12 14 16 18 20
3 18 21 24 27 30
4 24 28 32 36 40
5 30 35 40 45 50
```

In fact, the `outer` function can be used to calculate the “surface” for any 2-variable equation (more on this later).

2.3.3 *Naming a matrix*

We have already seen several examples of naming components of a matrix. Table 2.12 on the facing page summarizes the common ways of naming matrix components. The components of a matrix can be named at the time the matrix is created, or they can be named later. For a matrix, we can provide the row names, column names, and/or field names. For example, consider the UGDP clinical trial 2 × 2 table:

```
> tab
           Treatment
Outcome     Tolbutamide Placebo
  Deaths             30      21
  Survivors         174     184
```

In the "Treatment" field, the possible values, "Tolbutamide" and "Placebo," are the column names. Similarly, in the "Outcome" field, the possible values, "Deaths" and "Survivors," are the row names.

To review, the components of matrix can be named at the time the matrix is created:

```
> tab <- matrix(c(30, 174, 21, 184), 2, 2,
+     dimnames = list(Outcome = c("Deaths", "Survivors"),
+                   Treatment = c("Tolbutamide", "Placebo")))
> tab
           Treatment
Outcome     Tolbutamide Placebo
  Deaths             30      21
  Survivors         174     184
```

If a matrix does not have field names, we can add them after the fact, but we must use the `names` and `dimnames` functions together. Having field names is necessary if the row and column names are not self-explanatory, as this example illustrates.

```
> y <- matrix(c(30, 174, 21, 184), 2, 2)
> rownames(y) <- c("Yes", "No")
> colnames(y) <- c("Yes", "No")
> y #labels not informative
    Yes  No
Yes  30  21
No  174 184
> #add field names
> names(dimnames(y)) <- c("Death", "Tolbutamide")
> y
     Tolbutamide
Death Yes  No
  Yes  30  21
  No  174 184
```

Study and test the examples in Table 2.12 on the preceding page.

Table 2.13 Common ways of indexing a matrix

Indexing	Try these examples

By position

```
dat <- matrix(c(178, 79, 1411, 1486), 2, 2)
dimnames(dat) <- list(Behavior = c("Type A","Type
B"),
  "Heart attack" = c("Yes","No"))
dat[1, ]
dat[1,2]
dat[2, , drop = FALSE]
```

By name (if exists)

```
dat["Type A", ]
dat["Type A", "Type B"]
dat["Type B", , drop = FALSE]
```

By logical

```
dat[, 1] > 100
dat[dat[, 1] > 100, ]
dat[dat[, 1] > 100, , drop = FALSE]
```

2.3.4 *Indexing a matrix*

Similar to vectors, a matrix can be indexed by position, by name, or by logical. Study and practice the examples in Table 2.13. An important skill to master is indexing rows of a matrix using logical vectors. Consider the following matrix of data, and suppose I want to select the rows for subjects age less than 60 and systolic blood pressure less than 140.

```
> dat
     age chol sbp
[1,]  45  145 124
[2,]  56  168 144
[3,]  73  240 150
[4,]  44  144 134
[5,]  65  210 112
> dat[,"age"]<60
[1]  TRUE  TRUE FALSE  TRUE FALSE
> dat[,"sbp"]<140
[1]  TRUE FALSE FALSE  TRUE  TRUE
> tmp <- dat[,"age"]<60 & dat[,"sbp"]<140
> tmp
[1]  TRUE FALSE FALSE  TRUE FALSE
> dat[tmp,]
     age chol sbp
[1,]  45  145 124
[2,]  44  144 134
```

Notice that the `tmp` logical vector is the intersection of the logical vectors separated by the logical operator `&`.

Table 2.14 Common ways of replacing matrix elements

Replacing	Try these examples

By position	`dat <- matrix(c(178, 79, 1411, 1486), 2, 2)`
	`dimnames(dat) <- list(c("Type A","Type B"),`
	`c("Yes","No"))`
	`dat[1, ] <- 99`
	`dat`

By name (if exists)	`dat["Type A", ] <- c(178, 1411)` `dat`

By logical	`qq <- dat[ ,1]<100`
	`qq`
	`dat[qq, ] <- 99`
	`dat`
	`dat[dat[ ,1]<100, ] <- c(79, 1486)`
	`dat`

2.3.5 Replacing matrix elements

Remember, replacing matrix elements is just indexing plus assignment: anything that can be indexed can be replaced. Study and practice the examples in Table 2.14.

2.3.6 Operations on a matrix

In epidemiology books, authors have preferences for displaying contingency tables. Software packages have default displays for contingency tables. In practice, we may need to manipulate a contingency table to facilitate further analysis. Consider the following 2-way table:

```
> tab
           Treatment
Outcome     Tolbutamide Placebo
  Deaths             30      21
  Survivors         174     184
```

We can transpose the matrix using the `t` function.

```
> t(tab)
             Outcome
Treatment     Deaths Survivors
  Tolbutamide     30       174
  Placebo         21       184
```

We can reverse the order of the rows and/or columns.

```
> tab[2:1,]                      #reverse rows
           Treatment
```

Table 2.15 Common ways of operating on a matrix

Function	Description	Try these examples
t	Transpose matrix	`dat #from Table 2.14 on the preceding page` `t(dat)`

apply	Apply a function to the margins of a matrix	`apply(X = dat, MARGIN = 2, FUN = sum)`
		`apply(dat, 1, FUN=sum)`
		`apply(dat, 1, mean)`
		`apply(dat, 2, cumprod)`

sweep	Return an array obtained from an input array by sweeping out a summary statistic	`rsum <- apply(dat, 1, sum)`
		`rdist <- sweep(dat, 1, rsum, "/")`
		`rdist`
		`csum <- apply(dat, 2, sum)`
		`cdist <- sweep(dat, 2, csum, "/")`
		`cdist`

The following short-cuts use `apply` and/or `sweep` functions.		

margin.table
rowSums
colSums
	For a contingency table in array form, compute the sum of table entries for a given index. These functions are really just the `apply` function using `sum`.	`margin.table(dat)`
		`margin.table(dat, 1)`
		`rowSums(dat) #equivalent`
		`apply(dat, 1, sum) #equivalent`
		`margin.table(dat, 2)`
		`colSums(dat) #equivalent to previous`
		`apply(dat, 2, sum)`

addmargins	Calculate and display marginal totals of a matrix	`addmargins(dat)`

rowMeans
colMeans
	For a contingency table in array form, compute the mean of table entries for a given index. These functions are really just the `apply` function using `mean`.	`rowSums(dat)`
		`apply(dat, 1, mean) #equivalent`
		`colSums(dat)`
		`apply(dat, 2, mean) #equivalent`

prop.table	Short cut that uses the `sweep` and `apply` functions to get margin and joint distributions	`prop.table(dat)`
		`dat/sum(dat)`
		`prop.table(dat, 1)`
		`sweep(dat, 1, apply(y, 1, sum), "/")`
		`prop.table(dat, 2)`
		`sweep(y, 2, apply(y, 2, sum), "/")`

```
Outcome     Tolbutamide Placebo
  Survivors         174     184
  Deaths             30      21
> tab[,2:1]                     #reverse columns
           Treatment
Outcome     Placebo Tolbutamide
  Deaths         21          30
  Survivors     184         174
> tab[2:1,2:1]                  #reverse rows and columns
           Treatment
Outcome     Placebo Tolbutamide
  Survivors     184         174
  Deaths         21          30
```

2.3.6.1 The `apply` function

The `apply` function is an important and versatile function for conducting operations on rows or columns of a matrix, including user-created functions. The same functions that are used to conduct operations on single vectors (Table 2.8 on page 42) can be applied to rows or columns of a matrix.

To calculate the row or column totals use the `apply` with the `sum` function:

```
> tab
           Treatment
Outcome     Tolbutamide Placebo
  Deaths             30      21
  Survivors         174     184
> apply(tab, 1, sum) #row totals
   Deaths Survivors
       51       358
> apply(tab, 2, sum) #column totals
Tolbutamide     Placebo
        204         205
```

These operations can be used to calculate marginal totals and have them combined with the original table into one table.

```
> tab
           Treatment
Outcome     Tolbutamide Placebo
  Deaths             30      21
  Survivors         174     184
> rtot <- apply(tab, 1, sum) #row totals
> tab2 <- cbind(tab, Total = rtot)
> tab2
          Tolbutamide Placebo Total
```

```
Deaths              30      21    51
Survivors          174     184   358
> ctot <- apply(tab2, 2, sum) #column totals
> rbind(tab2, Total = ctot)
          Tolbutamide Placebo Total
Deaths              30      21    51
Survivors          174     184   358
Total              204     205   409
```

For convenience, R provides some functions for calculating marginal totals, and calculating row or column means (`margin.table`, `rowSums`, `colSums`, `rowMeans`, and `colMeans`). However, these functions just use the `apply` function[4].

Here's an alternative method to calculate marginal totals:

```
> tab
           Treatment
Outcome     Tolbutamide Placebo
  Deaths             30      21
  Survivors         174     184
> tab2 <- cbind(tab, Total=rowSums(tab))
> rbind(tab2, Total=colSums(tab2))
          Tolbutamide Placebo Total
Deaths              30      21    51
Survivors          174     184   358
Total              204     205   409
```

For convenience, the `addmargins` function calculates and displays the marginals totals with the original data in one step.

```
> addmargins(tab)
           Treatment
Outcome     Tolbutamide Placebo Sum
  Deaths             30      21  51
  Survivors         174     184 358
  Sum               204     205 409
```

The power of the `apply` function comes from our ability to pass many functions (including our own) to it. For practice, combine the `apply` function with functions from Table 2.8 on page 42 to conduct operations on rows and columns of a matrix.

2.3.6.2 The `sweep` function

The `sweep` function is another important and versatile function for conducting operations across rows or columns of a matrix. This function "sweeps" (operates on) a row or column of a matrix using some function and a value (usually derived from the row or column values). To understand this, we consider an example involving a single vector. For a given integer vector `x`, to convert the values of `x` into proportions involves two steps:

---

[4] More specifically, `rowSums`, `colSums`, `rowMeans`, and `colMeans` are optimized for speed.

```
> x <- c(1, 2, 3, 4, 5)
> sumx <- sum(x)   #Step 1: summation
> propx <- x/sumx  #Step 2: division (the "sweep")
> propx
[1] 0.066667 0.133333 0.200000 0.266667 0.333333
```

To apply this equivalent operation across rows or columns of a matrix requires the `sweep` function.

For example, to calculate the row and column distributions of a 2-way table we combine the `apply` (step 1) and the `sweep` (step 2) functions:

```
> tab
           Treatment
Outcome     Tolbutamide Placebo
  Deaths             30      21
  Survivors         174     184
> rtot <- apply(tab, 1, sum) #row totals
> tab.rowdist <- sweep(tab, 1, rtot, "/")
> tab.rowdist
           Treatment
Outcome     Tolbutamide Placebo
  Deaths        0.58824 0.41176
  Survivors     0.48603 0.51397
> ctot <- apply(tab, 2, sum) #column totals
> tab.coldist <- sweep(tab, 2, ctot, "/")
> tab.coldist
           Treatment
Outcome     Tolbutamide Placebo
  Deaths        0.14706 0.10244
  Survivors     0.85294 0.89756
```

Because R is a true programming language, these can be combined into single steps:

```
> sweep(tab, 1, apply(tab, 1, sum), "/") #row distribution
           Treatment
Outcome     Tolbutamide Placebo
  Deaths        0.58824 0.41176
  Survivors     0.48603 0.51397
> sweep(tab, 2, apply(tab, 2, sum), "/") #column distribution
           Treatment
Outcome     Tolbutamide Placebo
  Deaths        0.14706 0.10244
  Survivors     0.85294 0.89756
```

Table 2.16 Deaths among subjects who received tolbutamide and placebo in the Unversity Group Diabetes Program (1970), stratifying by age

	Age<55		Age≥55		Combined	
	Tolbutamide	Placebo	Tolbutamide	Placebo	Tolbutamide	Placebo
Deaths	8	5	22	16	30	21
Survivors	98	115	76	69	174	184
Total	106	120	98	85	204	205

For convenience, R provides `prop.table`. However, this function just uses the `apply` and `sweep` functions.

2.4 An array is a *n*-dimensional table of like elements

2.4.1 Understanding arrays

While a matrix is a 2-dimensional table of like elements, an array is the generalization of matrices to *n*-dimensions. Stratified contingency tables in epidemiology are represented as array data objects in R. For example, the randomized clinical trial previously shown comparing the number deaths among diabetic subjects that received tolbutamide vs. placebo is now also stratified by age group (Table 2.16):

This is 3-dimensional array: outcome status vs. treatment status vs. age group. Let's see how we can represent this data in R.

```
> tdat <- c(8, 98, 5, 115, 22, 76, 16, 69)
> tdat <- array(tdat, c(2, 2, 2))
> dimnames(tdat) <- list(Outcome=c("Deaths", "Survivors"),
+     Treatment=c("Tolbutamide", "Placebo"),
+     "Age group"=c("Age<55", "Age>=55"))
> tdat
, , Age group = Age<55

           Treatment
Outcome     Tolbutamide Placebo
  Deaths              8       5
  Survivors          98     115

, , Age group = Age>=55

           Treatment
Outcome     Tolbutamide Placebo
  Deaths             22      16
  Survivors          76      69
```

Table 2.17 Example of 4-dimensional array: Year 2000 population estimates by age, ethnicity, sex, and county

		Ethnicity					
County/Sex	Age	White	AfrAmer	AsianPI	Latino	Multirace	AmerInd
Alameda							
Female	<=19	58160	31765	40653	49738	10120	839
	20–44	112326	44437	72923	58553	7658	1401
	45–64	82205	24948	33236	18534	2922	822
	65+	49762	12834	16004	7548	1014	246
Male	<=19	61446	32277	42922	53097	10102	828
	20–44	115745	36976	69053	69233	6795	1263
	45–64	81332	20737	29841	17402	2506	687
	65+	33994	8087	11855	5416	711	156
San Francisco							
Female	<=19	14355	6986	23265	13251	2940	173
	20–44	85766	10284	52479	23458	3656	526
	45–64	35617	6890	31478	9184	1144	282
	65+	27215	5172	23044	5773	554	121
Male	<=19	14881	6959	24541	14480	2851	165
	20–44	105798	11111	48379	31605	3766	782
	45–64	43694	7352	26404	8674	1220	354
	65+	20072	3329	17190	3428	450	76

R displays the first stratum (`tdat[,,1]`) then the second stratum (`tdat[,,2]`). Our goal now is to understand how to generate and operate on these types of arrays. Before we can do this we need to thoroughly understand the structure of arrays.

Let's study a 4-dimensional array. Displayed in Table 2.17 is the year 2000 population estimates for Alameda and San Francisco Counties by age, ethnicity, and sex. The first dimension is age category, the second dimension is ethnicity, the third dimension is sex, and the fourth dimension is county. Learning how to visualize this 4-dimensional sturcture in R will enable us to visualize arrays of any number of dimensions.

Displayed in Figure 2.4.1 on the next page is a schematic representation of the 4-dimensional array of population estimates in Table 2.17. The left cube represents the population estimates by age, race, and sex (dimensions 1, 2, and 3) for Alameda County (first component of dimension 4). The right cube represents the population estimates by age, race, and sex (dimensions 1, 2, and 3) for San Francisco County (second component of dimension 4). We see, then, that it is possible to visualize data arrays in more than three dimensions.

To convince ourselves further, displayed in Figure 2.4.1 on the next page is a theorectical 5-dimensional data array. Suppose this 5-D array contained data on age ("Young", "Old"), ethnicity ("White", "Nonwhite"), sex ("Male", "Female"), party affiliation ("Democrat", "Republican"), and state ("California", "Washington

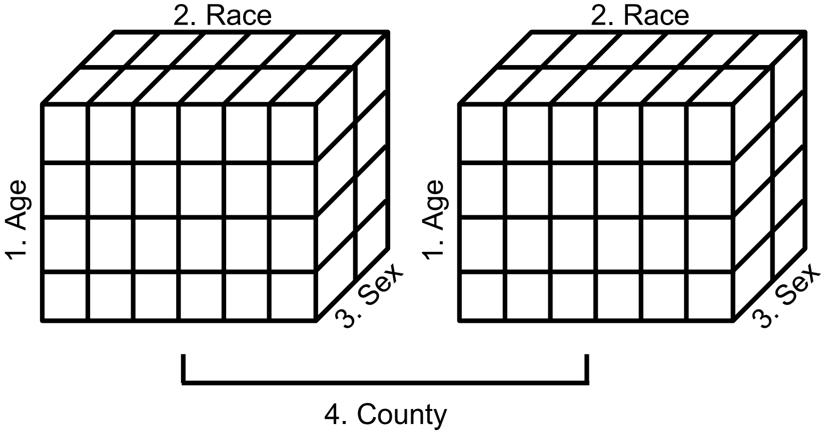


Fig. 2.2 Schematic representation of a 4-dimensional array (Year 2000 population estimates by age, race, sex, and county)

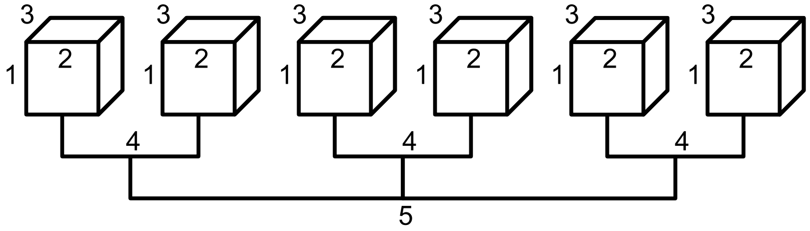


Fig. 2.3 Schematic representation of a theoretical 5-dimensional array (possibly population estimates by age (1), race (2), sex (3), party affiliation (4), and state (5)). From this diagram, we can infer that the field "state" has 3 levels, and the field "party affiliation" has 2 levels; however, it is not apparent how many age levels, race levels, and sex levels have been created. Although not displayed, age levels would be represented by row names (along 1st dimension), race levels would be represented by column names (along 2nd dimension), and sex levels would be represented by depth names (along 3rd dimension).

State", "Florida"). For practice, using fictitious data, try the following R code and study the output:

```
tab5 <- array(1:48, dim = c(2,2,2,2,3))
dn1 <- c("Young", "Old")
dn2 <- c("White", "Nonwhite")
dn3 <- c("Male", "Female")
dn4 <- c("Democrat", "Republican")
dn5 <- c("California", "Washington State", "Florida")
dimnames(tab5) <- list(Age=dn1, Race=dn2, Sex=dn3, Party=dn4,
    State=dn5)
tab5
```

Table 2.18 Common ways of creating arrays

Function	Description	Try these examples
array	Reshapes vector into an array	`aa <- array(1:12, dim = c(2, 3, 2))` `aa`

table	Creates *n*-dimensional contingency table from *n* vectors	`data(infert) # load infert data set`
		`table(infert$educ, infert$spont,`
		`  infert$case)`

xtabs	Creates a contingency table from cross-classifying factors contained in a data frame using a formula interface	`xtabs(˜education + case + parity,` `  data = infert)`

as.table	Creates n-dimensional contingency table from *n*-dimensional ftable	`ft <- ftable(infert$educ,`
		`infert$spont,`
		`  infert$case)`
		`ft`
		`as.table(ft)`

dim	Assign dimensions to a data object	`x <- 1:12`
		`x`
		`dim(x) <- c(2, 3, 2)`
		`x`

2.4.2 Creating arrays

In R, arrays are most often produced with the `array`, `table`, or `xtabs` functions (Table 2.18). As in the previous example, the `array` function works much like the `matrix` function except the `array` function can specify 1 or more dimensions, and the `matrix` function only works with 2 dimensions.

```
> array(1, dim = 1)
[1] 1
> array(1, dim = c(1, 1))
     [,1]
[1,]    1
> array(1, dim = c(1, 1, 1))
, , 1

     [,1]
[1,]    1
```

The `table` function cross tabulates 2 or more categorical vectors: character vectors or factors. In R, categorical data are represented as factors (more on this later). In contrast, using a formula interface, the `xtabs` function cross tabulates 2 or more factors from a data frame. Additionally, the `xtabs` function includes field names (which is highly preferred). For illustration, we will cross tabulate character vectors.

```
> #read in data "as is" (no factors created)
> udat1 <- read.csv("http://www.medepi.net/data/ugdp.txt",
+ as.is = TRUE)
> str(udat1)
'data.frame': 409 obs. of  3 variables:
 $ Status   : chr  "Death" "Death" "Death" "Death" ...
 $ Treatment: chr  "Tolbutamide" "Tolbutamide" "Tolbutamide" ...
 $ Agegrp   : chr  "<55" "<55" "<55" "<55" ...
> table(udat1$Status, udat1$Treatment, udat1$Agegrp)
, ,  = 55+

          Placebo Tolbutamide
 Death         16          22
 Survivor      69          76

, ,  = <55

          Placebo Tolbutamide
 Death          5           8
 Survivor     115          98
```

The `xtabs` function will not work on character vectors.

By default, R converts character fields into factors. With factors, both the `table` and `xtabs` functions cross tabulate the fields.

```
> #read in data and convert character vectors to factors
> udat2 <- read.csv("http://www.medepi.net/data/ugdp.txt")
> str(udat2)
'data.frame': 409 obs. of  3 variables:
 $ Status   : Factor w/ 2 levels "Death","Survivor": 1 1 1 1 ...
 $ Treatment: Factor w/ 2 levels "Placebo","Tolbutamide": 2 2 ...
 $ Agegrp   : Factor w/ 2 levels "55+","<55": 2 2 2 2 2 2 2 ...
> table(udat2$Status, udat1$Treatment, udat1$Agegrp)
, ,  = 55+

          Placebo Tolbutamide
 Death         16          22
 Survivor      69          76

, ,  = <55

          Placebo Tolbutamide
 Death          5           8
 Survivor     115          98

> xtabs(~Status + Treatment + Agegrp, data = udat2)
, , Agegrp = 55+
          Treatment
Status     Placebo Tolbutamide
  Death         16          22
  Survivor      69          76

, , Agegrp = <55

          Treatment
Status     Placebo Tolbutamide
  Death          5           8
  Survivor     115          98
```

Notice that the `xtabs` function above included the field names. Field names can be added manually with the `table` functions:

```
> table(Outcome = udat2$Status, Therapy = udat1$Treatment,
+ Age = udat1$Agegrp)
, , Age = 55+

          Therapy
Outcome    Placebo Tolbutamide
  Death         16          22
  Survivor      69          76

, , Age = <55

          Therapy
Outcome    Placebo Tolbutamide
  Death          5           8
  Survivor     115          98
```

Recall that the `ftable` function creates a flat contingency from categorical vectors. The `as.table` function converts the flat contingency table back into a multidimensional array.

```
> ftab <- ftable(udat2$Agegrp, udat1$Treatment, udat1$Status)
> ftab
                 Death Survivor

<55 Placebo          5      115
    Tolbutamide      8       98
55+ Placebo         16       69
    Tolbutamide     22       76
> as.table(ftab)
, ,  = Death
```

Table 2.19 Common ways of naming arrays

Function — Try these examples

array

```
# name components at time array is created x <- c(140,
11, 280, 56, 207, 9, 275, 32)
# create labels for values of each dimension
rn <- c(">=1 cups per day", "0 cups per day")
cn <- c("Cases", "Controls")
dn <- c("Females", "Males")
x <- array(x, dim = c(2, 2, 2), dimnames = list(Coffee
= rn,
  Outcome = cn, Gender = dn))
x
```

dimnames

```
x <- c(140, 11, 280, 56, 207, 9, 275, 32)
# create labels for values of each dimension
rn <- c(">=1 cups per day", "0 cups per day")
cn <- c("Cases", "Controls")
dn <- c("Females", "Males")
x <- array(x, dim = c(2, 2, 2))
dimnames(x) <- list(Coffee = rn, Outcome = cn, Gender =
dn)
x
```

names

```
x <- c(140, 11, 280, 56, 207, 9, 275, 32)
# create labels for values of each dimension
rn <- c(">=1 cups per day", "0 cups per day")
cn <- c("Cases", "Controls")
dn <- c("Females", "Males")
x <- array(x, dim = c(2, 2, 2))
dimnames(x) <- list(rn, cn, dn)
x # display w/o field names
names(dimnames(x)) <- c("Coffee", "Case status", "Sex")
x # display w/ field names
```

```
      Placebo Tolbutamide
  <55       5           8
  55+      16          22

, ,  = Survivor

      Placebo Tolbutamide
  <55     115          98
  55+      69          76
```

Table 2.20 Common ways of indexing arrays

Indexing	Try these examples

By position	`# use x from Table 2.19`
	`x[1, , ]`
	`x[ ,2, ]`

By name (if exists)	`x[ , ,"Males"]`
	`x[`
	`,"Controls","Females"]`

By logical vector	`zz <- x[ ,1,1]>50`
	`zz`
	`x[zz, , ]`

Table 2.21 Common ways of replacing array elements

Replacing	Try these examples

By position	`# use x from Table 2.19`
	`x[1, 1, 1] <- NA`
	`x`

By name (if exists)	`x[`
	`,"Controls","Females"]`
	`<- 99`
	`x`

By logical	`x>200`
	`x[x>200] <- 999`
	`x`

2.4.3 Naming arrays

Naming components of an array is an extension of naming components of a matrix (Table 2.12 on page 54). Study and implement the examples in Table 2.19 on the facing page.

2.4.4 Indexing arrays

Indexing an array is an extension of indexing a matrix (Table 2.13 on page 56). Study and implement the examples in Table 2.20.

2.4.5 Replacing array elements

Replacing elements of an array is an extension of replacing elements of a matrix (Table 2.14 on page 57). Study and implement the examples in Table 2.21.

Table 2.22 Common ways of operating on an array

Function	Description	Try these examples

aperm	Transpose an array by permuting its dimensions and optionally resizing it	`x <- array(1:24, c(2, 3, 2, 2))`
		`x`
		`aperm(x, c(3, 2, 1, 4))`

apply	Apply a function to the margins of an array	`apply(x, 1, sum)`
		`apply(x, c(2, 3), sum)`
		`apply(x, c(1, 2, 4), sum)`

sweep	Return an array obtained from an input array by sweeping out a summary statistic	`zz <- apply(x, c(1, 2), sum)` `sweep(x, c(1, 2), zz, "/")`
The following short-cuts use `apply` *and/or* `sweep` *functions*		

margin.table	For a contingency table in array form, compute the sum of table entries for a given index	`margin.table(x); sum(x)`
		`margin.table(x, c(1, 2))`
		`apply(x, c(1, 2), sum) #equiv`
		`margin.table(x, c(1, 2, 4))`
		`apply(x, c(1, 2, 4), sum) #equiv`

rowSums	Sum across rows of an array	`rowSums(x) # dims = 1`
		`apply(x, 1, sum) #equiv`
		`rowSums(x, dims = 2)`
		`apply(x, c(1, 2), sum) #equiv`

colSums	Sum down columns of an array	`colSums(x) # dims = 1`
		`apply(x, c(2, 3, 4), sum) #equiv`
		`colSums(x, dims = 2)`
		`apply(x, c(3, 4), sum) #equiv`

addmargins	Calculate and display marginal totals of an array	`addmargins(x)`

rowMeans	Calculate means across rows of an array	`rowMeans(x) # dims = 1`
		`apply(x, 1, mean) #equiv`
		`rowMeans(x, dims = 2)`
		`apply(x, c(1, 2), mean) #equiv`

colMeans	Calculate means down columns of an array	`colMeans(x) # dims = 1`
		`apply(x, c(2, 3, 4), mean) #equiv`
		`colMeans(x, dims = 2)`
		`apply(x, c(3, 4), mean) #equiv`

prop.table	Generates distribution for dimensions that are summed in the `margin.table` function	`prop.table(x)`
		`prop.table(x, margin = 1)`
		`prop.table(x, c(1, 2))`
		`prop.table(x, c(1, 2, 3))`

2.4.6 Operations on arrays

With the exception of the `aperm` function, operating on an array (Table 2.22) is an extension of operating on a matrix (Table 2.15 on page 58). Consider the number of primary and secondary syphilis cases in the United State, 1989, stratified by sex,

Table 2.23 Example of 3-dimensional array with marginal totals: Primary and secondary syphilis morbidity by age, race, and sex, United State, 1989

		Ethnicity			
Age (years)	Sex	White	Black	Other	Total
≤ 14	Male	2	31	7	40
	Female	14	165	11	190
	Total	16	196	18	230
15-19	Male	88	1412	210	1710
	Female	253	2257	158	2668
	Total	341	3669	368	4378
20-24	Male	407	4059	654	5120
	Female	475	4503	307	5285
	Total	882	8562	961	10405
25-29	Male	550	4121	633	5304
	Female	433	3590	283	4306
	Total	983	7711	916	9610
30-34	Male	564	4453	520	5537
	Female	316	2628	167	3111
	Total	880	7081	687	8648
35-44	Male	654	3858	492	5004
	Female	243	1505	149	1897
	Total	897	5363	641	6901
45-54	Male	323	1619	202	2144
	Female	55	392	40	487
	Total	378	2011	242	2631
55+	Male	216	823	108	1147
	Female	24	92	15	131
	Total	240	915	123	1278
Total (all ages)	Male	2804	20376	2826	26006
	Female	1813	15132	1130	18075
Total		4617	35508	3956	44081

ethnicity, and age (Table 2.23). This table contains the marginal and joint distribution of cases. Let's read in the original data and reproduce the table results.

```
> sdat3 <- read.csv("http://www.medepi.net/data/syphilis89c.txt")
> str(sdat3)
'data.frame':   44081 obs. of  3 variables:
 $ Sex : Factor w/ 2 levels "Male","Female": 1 1 1 1 1 1 1 1 ...
 $ Race: Factor w/ 3 levels "White","Black",..: 1 1 1 1 1 1 ...
 $ Age : Factor w/ 8 levels "<=14","15-19",..: 1 1 2 2 2 2 2 ...
> sdat3[1:5,] #display first 5 lines
```

```
   Sex  Race   Age
1 Male White  <=14
2 Male White  <=14
3 Male White 15-19
4 Male White 15-19
5 Male White 15-19
> sdat <- xtabs(~Sex+Race+Age, data=sdat3) #create array
> sdat
, , Age = <=14

        Race
Sex      White Black Other
  Male       2    31     7
  Female    14   165    11

, , Age = 15-19

        Race
Sex      White Black Other
  Male      88  1412   210
  Female   253  2257   158

, , Age = 20-24

        Race
Sex      White Black Other
  Male     407  4059   654
  Female   475  4503   307

, , Age = 25-29

        Race
Sex      White Black Other
  Male     550  4121   633
  Female   433  3590   283

, , Age = 30-34

        Race
Sex      White Black Other
  Male     564  4453   520
  Female   316  2628   167

, , Age = 35-44
```

```
        Race
Sex      White Black Other
  Male     654  3858   492
  Female   243  1505   149

, , Age = 45-54

        Race
Sex      White Black Other
  Male     323  1619   202
  Female    55   392    40

, , Age = 55+

        Race
Sex      White Black Other
  Male     216   823   108
  Female    24    92    15
```

To get marginal totals for one dimension, use the `apply` function and specify the dimension for stratifying the results.

```
> sum(sdat) #total
[1] 44081
> apply(X = sdat, MARGIN = 1, FUN = sum) #by sex
  Male Female
 26006  18075
> apply(sdat, 2, sum) #by race
White Black Other
 4617 35508  3956
> apply(sdat, 3, sum) #by age
 <=14 15-19 20-24 25-29 30-34 35-44 45-54   55+
  230  4378 10405  9610  8648  6901  2631  1278
```

To get the joint marginal totals for 2 or more dimensions, use the `apply` function and specify the dimensions for stratifying the results. This means that the function that is passed to `apply` is applied across the other, non-stratified dimensions.

```
> apply(sdat, c(1, 2), sum) #by sex and race
        Race
Sex      White Black Other
  Male    2804 20376  2826
  Female  1813 15132  1130
> apply(sdat, c(1, 3), sum) #by sex and age
        Age
Sex      <=14 15-19 20-24 25-29 30-34 35-44 45-54  55+
  Male     40  1710  5120  5304  5537  5004  2144 1147
  Female  190  2668  5285  4306  3111  1897   487  131
> apply(sdat, c(3, 2), sum) #by age and race
       Race
Age      White Black Other
  <=14     16   196    18
  15-19   341  3669   368
  20-24   882  8562   961
  25-29   983  7711   916
  30-34   880  7081   687
  35-44   897  5363   641
  45-54   378  2011   242
  55+     240   915   123
```

In R, arrays are displayed by the 1st and 2nd dimensions, stratified by the remaining dimensions. To change the order of the dimensions, and hence the display, use the `aperm` function. For example, the syphilis case data is most efficiently displayed when it is stratified by race, age, and sex:

```
> sdat.ras <- aperm(sdat, c(2, 3, 1))
> sdat.ras
, , Sex = Male

       Age
Race     <=14 15-19 20-24 25-29 30-34 35-44 45-54 55+
  White     2    88   407   550   564   654   323 216
  Black    31  1412  4059  4121  4453  3858  1619 823
  Other     7   210   654   633   520   492   202 108

, , Sex = Female

       Age
Race     <=14 15-19 20-24 25-29 30-34 35-44 45-54 55+
  White    14   253   475   433   316   243    55  24
  Black   165  2257  4503  3590  2628  1505   392  92
  Other    11   158   307   283   167   149    40  15
```

Another method for changing the display of an array is to convert it into a flat contingency table using the `ftable` function. For example, to display Table 2.23 on page 71 as a flat contingency table in R (but without the marginal totals), we use the following code:

```
> sdat.asr <- aperm(sdat, c(3,1,2)) #rearrange to age, sex, race
> ftable(sdat.asr)                  #convert 2-D flat table
             Race White Black Other
Age   Sex
<=14  Male             2    31     7
      Female          14   165    11
```

```
15-19 Male           88  1412   210
      Female        253  2257   158
20-24 Male          407  4059   654
      Female        475  4503   307
25-29 Male          550  4121   633
      Female        433  3590   283
30-34 Male          564  4453   520
      Female        316  2628   167
35-44 Male          654  3858   492
      Female        243  1505   149
45-54 Male          323  1619   202
      Female         55   392    40
55+   Male          216   823   108
      Female         24    92    15
```

This `ftable` object can be treated as a matrix, but it cannot be transposed. Notice that we can combine the `ftable` with `addmargins`:

```
> ftable(addmargins(sdat.asr))
             Race Black Other White   Sum
Age   Sex
15-19 Female       2257   158   253  2668
      Male         1412   210    88  1710
      Sum          3669   368   341  4378
20-24 Female       4503   307   475  5285
      Male         4059   654   407  5120
      Sum          8562   961   882 10405
25-29 Female       3590   283   433  4306
      Male         4121   633   550  5304
      Sum          7711   916   983  9610
30-34 Female       2628   167   316  3111
      Male         4453   520   564  5537
      Sum          7081   687   880  8648
35-44 Female       1505   149   243  1897
      Male         3858   492   654  5004
      Sum          5363   641   897  6901
45-54 Female        392    40    55   487
      Male         1619   202   323  2144
      Sum          2011   242   378  2631
<=14  Female        165    11    14   190
      Male           31     7     2    40
      Sum           196    18    16   230
>55   Female         92    15    24   131
      Male          823   108   216  1147
      Sum           915   123   240  1278
Sum   Female      15132  1130  1813 18075
```

```
Male        20376  2826  2804 26006
Sum         35508  3956  4617 44081
```

To share the U.S. syphilis data in a universal format, we could create a text file with the data in a tabular form. However, the original, individual-level data set has over 40,000 observations. Instead, it would be more convenient to create a group-level, tabular data set using the `as.data.frame` function on the data array object.

```
> sdat.df <- as.data.frame(sdat)
> str(sdat.df)
'data.frame':   48 obs. of  4 variables:
 $ Sex : Factor w/ 2 levels "Male","Female": 1 2 1 2 1 2 1 2 1 2 ...
 $ Race: Factor w/ 3 levels "White","Black",..: 1 1 2 2 3 3 1 1 2 2 ...
 $ Age : Factor w/ 8 levels "<=14","15-19",..: 1 1 1 1 1 1 2 2 2 2 ...
 $ Freq: num    2  14  31 165   7 ...
> sdat.df[1:8,]
     Sex  Race   Age Freq
1   Male White  <=14    2
2 Female White  <=14   14
3   Male Black  <=14   31
4 Female Black  <=14  165
5   Male Other  <=14    7
6 Female Other  <=14   11
7   Male White 15-19   88
8 Female White 15-19  253
```

For additional practice, study and implement the examples in Table 2.22 on page 70.

2.5 A list is a collection of like or unlike data objects

2.5.1 Understanding lists

Up to now, we have been working with atomic data objects (vector, matrix, array). In contrast, lists, data frames, and functions are recursive data objects. Recursive data objects have more flexibility in combining diverse data objects into one object. A list provides the most flexibility. Think of a list object as a collection of “bins” that can contain any R object (see Figure 2.5.1 on the facing page). Lists are very useful for collecting results of an analysis or a function into one data object where all its contents are readily accessible by indexing.

For example, using the UGDP clinical trial data, suppose we perform Fisher’s exact test for testing the null hypothesis of independence of rows and columns in a contingency table with fixed marginals.

```
> udat <- read.csv("http://www.medepi.net/data/ugdp.txt")
```

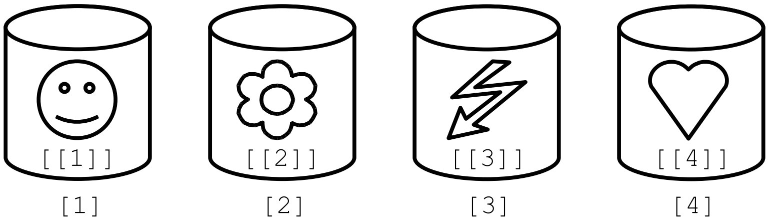


Fig. 2.4 Schematic representation of a list of length four. The first bin `[1]` contains a smiling face `[[1]]`, the second bin `[2]` contains a flower `[[2]]`, the third bin `[3]` contains a lightning bolt `[[3]]`, and the fourth bin `[[4]]` contains a heart `[[4]]`. When indexing a list object, single brackets `[·]` indexes the *bin*, and double brackets `[[·]]` indexes the bin *contents*. If the bin has a name, then $*name* also indexes the contents.

```
> tab <- table(udat$Status, udat$Treatment)[,2:1]
> tab

           Tolbutamide Placebo
  Death             30      21
  Survivor         174     184
> ftab <- fisher.test(tab)
> ftab

        Fisher's Exact Test for Count Data

data:  tab
p-value = 0.1813
alternative hypothesis: true odds ratio is not equal to 1
95 percent confidence interval:
 0.80138 2.88729
sample estimates:
odds ratio
    1.5091
```

The default display only shows partial results. The total results are stored in the object `ftab`. Let's evaluate the structure of `ftab` and extract some results:

```
> str(ftab)
List of 7
 $ p.value    : num 0.181
 $ conf.int   : atomic [1:2] 0.801 2.887
  ..- attr(*, "conf.level")= num 0.95
 $ estimate   : Named num 1.51
  ..- attr(*, "names")= chr "odds ratio"
 $ null.value : Named num 1
  ..- attr(*, "names")= chr "odds ratio"
 $ alternative: chr "two.sided"
 $ method     : chr "Fisher's Exact Test for Count Data"
 $ data.name  : chr "tab"
 - attr(*, "class")= chr "htest"
> ftab$estimate
odds ratio
    1.5091
> ftab$conf.int
[1] 0.80138 2.88729
> ftab$conf.int[2]
[1] 2.887286
attr(,"conf.level")
[1] 0.95
> ftab$p.value
[1] 0.18126
```

Using the `str` function to evaluate the structure of an output object is a common method employed to extract additional results for display or further analysis. In this case, `ftab` was a list with 7 bins, each with a name.

2.5.2 *Creating lists*

To create a list directly, use the `list` function. A list is a convenient method to save results in our customized functions. For example, here's a function to calculate an odds ratio from a 2×2 table:

```
orcalc <- function(x){
  or <- (x[1,1]*x[2,2])/(x[1,2]*x[2,1])
  pval <- fisher.test(x)$p.value
  list(data = x, odds.ratio = or, p.value = pval)
}
```

The `orcalc` function has been loaded in R, and now we run the function on the UGDP data.

```
> tab #display 2x2 table

          Tolbutamide Placebo
  Death            30      21
  Survivor        174     184
> orcalc(tab) #run function
$data

          Tolbutamide Placebo
  Death            30      21
```

Table 2.24 Common ways of creating a list

Function	Description	Try these examples

list	Creates list object	`x <- 1:3`
		`y <- matrix(c("a","c","b","d"), 2,2)`
		`z <- c("Pedro", "Paulo", "Maria")`
		`mm <- list(x, y, z)`
		`mm`

data.frame	List in tabular format where each "bin" has a vector of same length	`x <-`
		`data.frame(id=1:3,sex=c("M","F","T"))`
		`x`
		`mode(x)`
		`class(x)`

as.data.frame	Coerces data object into a data frame	`x <- matrix(1:6, 2, 3)`
		`x`
		`y <- as.data.frame(x)`
		`y`

read.table
read.csv
read.delim
read.fmf
	Reads ASCII text file into data frame object[1]	`wcgs <- read.table(".../wcgs.txt",`
		`header=TRUE, sep=",")`
		`str(wcgs)`

vector	Creates empty list of length *n*	`vector("list", 2)`
as.list	Coercion into list object	`list(1:2) # compare to as.list` `as.list(1:2)`

[1] Try `read.table("http://www.medepi.net/data/wcgs.txt", header=TRUE, sep=",")`

```
  Survivor         174     184

$odds.ratio
[1] 1.5107

$p.value
[1] 0.18126
```

For additional practice, study and implement the examples in Table 2.24.

2.5.3 Naming lists

Components (bins) of a list can be unnamed or named. Components of a list can be named at the time the list is created or later using the `names` function. For practice, try the examples in Table 2.25 on the following page.

Table 2.25 Common ways of naming lists

Function	Try these examples

names	`z <- list(rnorm(20), "Luis", 1:3)`
	`z`
	`# name after creation of list`
	`names(z) <- c("bin1", "bin2", "bin3")`
	`z`
	`# name at creation of list`
	`z <- list(bin1 = rnorm(20), bin2 = "Luis", bin3 = 1:3)`
	`z`
	`# without assignment returns character vector`
	`names(z)`

Table 2.26 Common ways of indexing lists

Indexing	Try these examples

By position	`z <- list(bin1 = rnorm(20), bin2 = "Luis", bin3 = 1:3)`
	`z[1] # indexes "bin" #1`
	`z[[1]] # indexes contents of "bin" #1`

By name (if exists)	`z$bin1` `z$bin2`

Indexing by logical vector	`num <- sapply(z, is.numeric)`
	`num`
	`z[num]`

2.5.4 *Indexing lists*

If list components (bins) are unnamed, we can index the list by bin position with single or double brackets. The single brackets [·] indexes one or more *bins*, and the double brackets indexes *contents* of single bins only.

```
> mylist1 <- list(1:5, matrix(1:4,2,2), c("Juan Nieve", "Guillermo Farro"))
> mylist1[c(1, 3)] #index bins 1 and 3
[[1]]
[1] 1 2 3 4 5

[[2]]
[1] "Juan Nieve"      "Guillermo Farro"

> mylist1[[3]]     #index contents of 3rd bin
[1] "Juan Nieve"      "Guillermo Farro"
```

When list bins are named, we can index the bin contents by name. Using the matched case-control study `infert` data set, we will conduct a conditional logistic regression analysis to determine if spontaneous and induced abortions are independently associated with infertility. For this we'll need to load the `survival` package which contains the `clogit` function.

```
> data(infert)
> library(survival)
> mod1 <- clogit(case ~ spontaneous + induced + strata(stratum),
+ data = infert)
> mod1         #default display
Call:
clogit(case ~ spontaneous + induced + strata(stratum), data = infert)


             coef exp(coef) se(coef)    z       p
spontaneous 1.99      7.29    0.352 5.63 1.8e-08
induced     1.41      4.09    0.361 3.91 9.4e-05

Likelihood ratio test=53.1  on 2 df, p=2.87e-12  n= 248
> str(mod1)    #evaluate structure
List of 17
 $ coefficients     : Named num [1:2] 1.99 1.41
  ..- attr(*, "names")= chr [1:2] "spontaneous" "induced"
 $ var              : num [1:2, 1:2] 0.1242 0.0927 0.0927 0.1301
 $ loglik           : num [1:2] -90.8 -64.2
 $ score            : num 48.4
 $ iter             : int 5
...
> names(mod1)  #names of list components
 [1] "coefficients"      "var"               "loglik"
 [4] "score"             "iter"              "linear.predictors"
 [7] "residuals"         "means"             "method"
[10] "n"                 "terms"             "assign"
[13] "wald.test"         "y"                 "formula"
[16] "call"              "userCall"
> mod1$coeff
spontaneous     induced
     1.9859      1.4090
```

The results from `str(mod1)` are only partially displayed. Sometimes it is more convenience to display the names for the list rather than the complete structure. Additionally, the `summary` function applied to a regression model object creates a list object with more detailed results. This too has a default display, or we can index list components by name.

```
> summod1 <- summary(mod1)
> summod1      #default display of more detailed results
Call:
coxph(formula = Surv(rep(1, 248), case) ~ spontaneous +
```

Table 2.27 Common ways of replacing list components

Replacing	Try these examples

By position	`z <- list(bin1 = rnorm(20), bin2 = "Luis", bin3 = 1:3)`
	`z[1] <- list(c(2, 3, 4)) # replaces "bin" contents`
	`z[[1]] <- c(2, 3, 4) # replaces "bin" contents`

By name (if exists)	`z$bin2 <- c("Tomas", "Luis", "Angela")`
	`z`
	`# replace name of specific "bin"`
	`names(z)[2] <- "mykids"`
	`z`

By logical	`num <- sapply(z, is.numeric)`
	`num`
	`z[num]<- list(rnorm(10), rnorm(10))`
	`z`

```
    induced + strata(stratum), data = infert, method = "exact")

  n= 248
            coef exp(coef) se(coef)    z       p
spontaneous 1.99      7.29    0.352 5.63 1.8e-08
induced     1.41      4.09    0.361 3.91 9.4e-05

            exp(coef) exp(-coef) lower .95 upper .95
spontaneous      7.29      0.137      3.65      14.5
induced          4.09      0.244      2.02       8.3

Rsquare= 0.193   (max possible= 0.519 )
Likelihood ratio test= 53.1  on 2 df,   p=2.87e-12
Wald test            = 31.8  on 2 df,   p=1.22e-07
Score (logrank) test = 48.4  on 2 df,   p=3.03e-11

> names(summod1) #names of list components
 [1] "call"       "fail"       "na.action"  "n"          "icc"
 [6] "coef"       "conf.int"   "logtest"    "sctest"     "rsq"
[11] "waldtest"   "used.robust"
> summod1$coef
              coef exp(coef) se(coef)      z       p
spontaneous 1.9859    7.2854  0.35244 5.6346 1.8e-08
induced     1.4090    4.0919  0.36071 3.9062 9.4e-05
```

Table 2.28 Common ways of operating on a list

Function	Description	Try these examples

lapply	Applies a function to each component of a list and returns a list	`x <- list(1:5, 6:10)`
		`x`
		`lapply(x, mean)`

sapply	Applies a function to each component of a list and simplifies	`sapply(x, mean)`
do.call	Calls and applies a function to the list	`do.call(rbind, x)`

mapply	Applies a function to the first elements of each argument, the second elements, the third elements, and so on.	`x <- list(1:4, 1:4)`
		`x`
		`y <- list(4, rep(4, 4))`
		`y`
		`mapply(rep, x, y,`
		`SIMPLIFY=FALSE)`
		`mapply(rep, x, y)`

2.5.5 *Replacing lists components*

Replacing list components is accomplished by combining indexing with assignment. And of course, we can index by position, name, or logical. Remember, if it can be indexed, it can be replaced. Study and practice the examples in Table 2.27 on the preceding page.

2.5.6 *Operations on lists*

Because lists can have complex structural components, there are not many operations we will want to do on lists. When we want to apply a function to each component (bin) of a list, we use the `lapply` or `sapply` function. These functions are identical except that `sapply` "simplies" the final result, if possible.

The `do.call` function applies a function to the entire list using each each component as an argument. For example, consider a list where each bin contains a vector and we want to `cbind` the vectors.

```
> mylist <- list(vec1=1:5, vec2=6:10, vec3=11:15)
> cbind(mylist)          #will not work
     mylist
vec1 Integer,5
vec2 Integer,5
vec3 Integer,5
> do.call(cbind, mylist) #works
     vec1 vec2 vec3
[1,]    1    6   11
```

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

For additional practice, study and implements the examples in Table 2.28 on the preceding page.

2.6 A data frame is a list in a 2-dimensional tabular form

A data frame is a list in 2-dimensional tabular form. Each list component (bin) is a data field of equal length. A data frame is a list that behaves like a matrix. Anything that can be done with lists can be done with data frames. Many things that can be done with matrices can be done with data frames.

2.6.1 Understanding data frames and factors

Epidemiologists are familiar with tabular data sets where each row is a *record* and each column is a *field*. A record can be data collected on individuals or groups. We usually refer to the field name as a variable (e.g., age, gender, ethnicity). Fields can contain numeric or character data. In R, these types of data sets are handled by data frames. Each column of a data frame is usually either a factor or numeric vector, although it can have complex, character, or logical vectors. Data frames have the functionality of matrices and lists. For example, here is the first 10 rows of the `infert` data set, a matched case-control study published in 1976 that evaluated whether infertility was associated with prior spontaneous or induced abortions.

```
> data(infert)
> str(infert)
'data.frame':   248 obs. of  8 variables:
 $ education     : Factor w/ 3 levels "0-5yrs",..: 1 1 ...
 $ age           : num  NA 45 NA 23 35 36 23 32 21 28 ...
 $ parity        : num  6 1 6 4 3 4 1 2 1 2 ...
 $ induced       : num  1 1 2 2 1 2 0 0 0 0 ...
 $ case          : num  1 1 1 1 1 1 1 1 1 1 ...
 $ spontaneous   : num  2 0 0 0 1 1 0 0 1 0 ...
 $ stratum       : int  1 2 3 4 5 6 7 8 9 10 ...
 $ pooled.stratum: num  3 1 4 2 32 36 6 22 5 19 ...
> infert[1:10, 1:6]
   education age parity induced case spontaneous
1     0-5yrs  NA      6       1    1           2
2     0-5yrs  45      1       1    1           0
```

```
3      0-5yrs  NA      6       2    1          0
4      0-5yrs  23      4       2    1          0
5     6-11yrs  35      3       1    1          1
6     6-11yrs  36      4       2    1          1
7     6-11yrs  23      1       0    1          0
8     6-11yrs  32      2       0    1          0
9     6-11yrs  21      1       0    1          1
10    6-11yrs  28      2       0    1          0
```

The fields are obviously vectors. Let's explore a few of these vectors to see what we can learn about their structure in R.

```
> #age variable
> infert$age
  [1] 26 42 39 34 35 36 23 32 21 28 29 37 31 29 31 27 30 26
...
[235] 25 32 25 31 38 26 31 31 25 31 34 35 29 23
> mode(infert$age)
[1] "numeric"
> class(infert$age)
[1] "numeric"

> #stratum variable
> infert$stratum
  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
...
[235] 70 71 72 73 74 75 76 77 78 79 80 81 82 83
> mode(infert$stratum)
[1] "numeric"
> class(infert$stratum)
[1] "integer"

> #education variable
> infert$education
  [1] 0-5yrs  0-5yrs  0-5yrs  0-5yrs  6-11yrs 6-11yrs
...
[247] 12+ yrs 12+ yrs
Levels: 0-5yrs 6-11yrs 12+ yrs
> mode(infert$education)
[1] "numeric"
> class(infert$education)
[1] "factor"
```

What have we learned so far? In the `infert` data frame, `age` is a vector of mode "numeric" and class "numeric," `stratum` is a vector of mode "numeric" and class "integer," and `education` is a vector of mode "numeric" and class "factor." The numeric vectors are straightforward and easy to understand. However, a factor, R's representation of categorical data, is a bit more complicated.

Contrary to intuition, a factor is a numeric vector, not a character vector, although it may have been created from a character vector (shown later). To see the "true" `education` factor use the `unclass` function:

```
> z <- unclass(infert$education)
> z
  [1] 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
...
[244] 3 3 3 3 3
attr(,"levels")
[1] "0-5yrs"  "6-11yrs" "12+ yrs"
> mode(z)
[1] "numeric"
> class(z)
[1] "integer"
```

Now let's create a factor from a character vector and then unclass it:

```
> cointoss <- sample(c("Head","Tail"), 100, replace = TRUE)
> cointoss
  [1] "Tail" "Head" "Head" "Tail" "Tail" "Tail" "Head"
 ...
 [99] "Tail" "Head"
> fct <- factor(cointoss)
> fct
  [1] Tail Head Head Tail Tail Tail Head Head Head Tail Head
...
[100] Head
Levels: Head Tail
> unclass(fct)
  [1] 2 1 1 2 2 2 1 1 1 2 1 1 1 2 2 2 2 2 1 1 1 1 1 1 1 2 2
 [28] 1 2 2 1 1 2 1 2 2 1 1 1 1 1 2 2 1 1 2 2 2 1 1 2 2 2 1
 [55] 1 1 1 1 2 1 1 2 2 2 1 1 2 2 2 2 1 2 2 1 1 1 2 1 1 2 2
 [82] 1 1 2 1 1 1 2 1 1 1 1 2 1 1 1 2 1 2 1
attr(,"levels")
[1] "Head" "Tail"
```

Notice that we can still recover the original character vector using the `as.character` function:

```
> as.character(cointoss)
  [1] "Tail" "Head" "Head" "Tail" "Tail" "Tail" "Head"
 ...
 [99] "Tail" "Head"
> as.character(fct)
  [1] "Tail" "Head" "Head" "Tail" "Tail" "Tail" "Head"
 ...
 [99] "Tail" "Head"
```

Okay, let's create an *ordered* factor; that is, levels of a categorical variable that have natural ordering. For this we set `ordered=TRUE` in the factor function:

```
> samp <- sample(c("Low","Medium","High"), 100, replace=TRUE)
> ofac1 <- factor(samp, ordered=T)
> ofac1
  [1] Low    Medium High   Medium Medium Medium Medium
 ...
 [99] High   High
Levels: High < Low < Medium
> table(ofac1) #levels and labels not in natural order
ofac1
  High    Low Medium
    43     25     32
```

However, notice that the ordering was done in alphabetical order which is not what we want. To change this, use the `levels` options in the `factor` function:

```
> ofac2 <- factor(samp, levels=c("Low","Medium","High"), ordered=T)
> ofac2
  [1] Low    Medium High   Medium Medium Medium Medium
 ...
 [99] High   High
Levels: Low < Medium < High
> table(ofac2)
ofac2
   Low Medium   High
    28     35     37
```

Great — this is exactly what we want! For review, Table 2.29 on the next page summarizes the variable types in epidemiology and how they are represented in R. Factors (unordered and ordered) are used to represent nominal and ordinal categorical data. The `infert` data set contains nominal factors and the `esoph` data set contains ordinal factors.

2.6.2 *Creating data frames*

In the creation of data frames, character vectors (usually representing categorical data) are converted to factors (mode numeric, class factor), and numeric vectors are converted to numeric vectors of class numeric or class integer.

```
wt <- c(59.5, 61.4, 45.2)
age <- c(11, 9, 6)
```

Table 2.29 Variable types in epidemiologic data and their representations in R data frames

Representations in data		Representations in R		
Variable type	Examples	Mode	Class	Examples[1]
Numeric				
Continuous	3.45, 2/3	numeric	numeric	infert$age
Discrete	1, 2, 3, 4, ...	numeric	integer	infert$stratum
Categorical				
Nominal	male vs. female	numeric	factor	infert$education
Ordinal	low < medium < high	numeric	ordered factor	esoph$agegp

1. First load data: `data(infert); data(esoph)`

Table 2.30 Common ways of creating data frames

Function	Description	Try these examples
data.frame	Data frames are of mode list	`x <- data.frame(id=1:2, sex=c("M","F"))` `mode(x); x`
as.data.frame	Coerces data object into a data frame	`x <- matrix(1:6, 2, 3); x` `as.data.frame(x)`

as.table
ftable
	Combine with as.data.frame to convert a fully labeled array into a data frame	`x <- array(1:8, c(2, 2, 2))`
		`dimnames(x) <- list(Exposure=c("Y",`
		`"N"),`
		`Disease = c("Y", "N"),`
		`Confounder = c("Y", "N"))`
		`as.data.frame(ftable(x))`
		`as.data.frame(as.table(x))`

read.table
read.csv
read.delim
read.fmf
	Reads ASCII text file into data frame object[1]	`wcgs <- read.csv(".../wcgs.txt",`
		`header=T)`
		`str(wcgs)`

1. Try `read.csv("http://www.medepi.net/data/wcgs.txt", header=TRUE)`

```
sex <- c("Male", "Male", "Female")
df <- data.frame(age, sex, wt)
df
str(df)
```

Factors can also be created directly from vectors as described in the previous section.

2.6.3 Naming data frames

Everything that applies to naming list components (Table 2.25 on page 80) also applies to naming data frame components (Table 2.31 on the next page). In general, we may be interested in renaming variables (fields) or row names of a data frame,

Table 2.31 Common ways of naming data frames

Function	Try these examples

names	`x <- data.frame(var1 = 1:3, var2 = c("M", "F", "F"))`
	`x`
	`names(x) <- c("Subjno", "Sex")`
	`x`

row.names	`row.names(x) <- c("Subj 1", "Subj 2", "Subj 3")` `x`

or renaming the levels (possible values) for a given factor (categorical variable). For example, consider the Oswego data set.

```
> odat <- read.table("http://www.medepi.net/data/oswego.txt",
+         sep="", header=TRUE, na.strings=".")
> odat[1:5,1:8]               #Display partial data frame
  id age sex meal.time ill onset.date onset.time baked.ham
1  2  52   F   8:00 PM   Y       4/19   12:30 AM         Y
2  3  65   M   6:30 PM   Y       4/19   12:30 AM         Y
3  4  59   F   6:30 PM   Y       4/19   12:30 AM         Y
4  6  63   F   7:30 PM   Y       4/18   10:30 PM         Y
5  7  70   M   7:30 PM   Y       4/18   10:30 PM         Y
> names(odat)[3] <- "Gender" #Rename 'sex' to 'Gender'
> table(odat$Gender)          #Display 'Gender' distribution

 F  M
44 31
> levels(odat$Gender)         #Display 'Gender' levels
[1] "F" "M"
> #Replace 'Gender' level labels
> levels(odat$Gender) <- c("Female", "Male")
> levels(odat$Gender)         #Display new 'Gender' levels
[1] "Female" "Male"
> table(odat$Gender)          #Confirm distribution is same

Female   Male
    44     31
> odat[1:5,1:8]               #Display partial data frame
  id age Gender meal.time ill onset.date onset.time baked.ham
1  2  52 Female   8:00 PM   Y       4/19   12:30 AM         Y
2  3  65   Male   6:30 PM   Y       4/19   12:30 AM         Y
3  4  59 Female   6:30 PM   Y       4/19   12:30 AM         Y
4  6  63 Female   7:30 PM   Y       4/18   10:30 PM         Y
5  7  70   Male   7:30 PM   Y       4/18   10:30 PM         Y
```

Table 2.32 Common ways of indexing data frames

Indexing	Try these examples
By position	`data(infert)` `infert[1:5, 1:3]`
By name	`infert[1:5, c("education", "age", "parity")]`

By logical	`agelt30 <- infert$age<30; agelt30`
	`infert[agelt30, c("education","induced","parity")]`
	`# can also use 'subset' function`
	`subset(infert, agelt30,`
	`c("education","induced","parity"))`

On occasion, we might be interested in renaming the row names. Currently, the Oswego data set has default integer values from 1 to 75 as the row names.

```
> row.names(odat)
 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "11"
[12] "12" "13" "14" "15" "16" "17" "18" "19" "20" "21" "22"
[23] "23" "24" "25" "26" "27" "28" "29" "30" "31" "32" "33"
[34] "34" "35" "36" "37" "38" "39" "40" "41" "42" "43" "44"
[45] "45" "46" "47" "48" "49" "50" "51" "52" "53" "54" "55"
[56] "56" "57" "58" "59" "60" "61" "62" "63" "64" "65" "66"
[67] "67" "68" "69" "70" "71" "72" "73" "74" "75"
```

We can change the row names by assigning a new character vector.

```
> row.names(odat) <- sample(101:199, size=nrow(odat))
> odat[1:5,1:7]
    id age Gender meal.time ill onset.date onset.time
123  2  52 Female   8:00 PM   Y       4/19   12:30 AM
145  3  65   Male   6:30 PM   Y       4/19   12:30 AM
173  4  59 Female   6:30 PM   Y       4/19   12:30 AM
138  6  63 Female   7:30 PM   Y       4/18   10:30 PM
146  7  70   Male   7:30 PM   Y       4/18   10:30 PM
```

2.6.4 Indexing data frames

Indexing a data frame is similar to indexing a matrix or a list: we can index by position, by name, or by logical vector. Consider, for example, the 2004 California West Nile virus human disease surveillance data. Suppose we are interested in summarizing the Los Angeles cases with neuroinvasive disease ("WNND").

```
> wdat <- read.csv("http://www.medepi.net/data/wnv/wnv2004fin.txt")
> str(wdat)
'data.frame':   779 obs. of  8 variables:
 $ id         : int  1 2 3 4 5 6 7 8 9 10 ...
 $ county     : Factor w/ 23 levels "Butte","Fresno",..: 14 ...
 $ age        : int  40 64 19 12 12 17 61 74 71 26 ...
 $ sex        : Factor w/ 2 levels "F","M": 1 1 2 2 2 2 2 1...
 $ syndrome   : Factor w/ 3 levels "Unknown","WNF",..: 22 3 ...
 $ date.onset : Factor w/ 130 levels "2004-05-14", ..: 3 ...
 $ date.tested: Factor w/ 104 levels "2004-06-02",..: 1 ...
 $ death      : Factor w/ 2 levels "No","Yes": 1 1 1 1 1 1  ...
> levels(wdat$county)  #Review levels of 'county' variable
 [1] "Butte"           "Fresno"          "Glenn"
 [4] "Imperial"        "Kern"            "Lake"
 [7] "Lassen"          "Los Angeles"     "Merced"
[10] "Orange"          "Placer"          "Riverside"
[13] "Sacramento"      "San Bernardino" "San Diego"
[16] "San Joaquin"     "Santa Clara"     "Shasta"
[19] "Sn Luis Obispo" "Tehama"           "Tulare"
[22] "Ventura"         "Yolo"
> levels(wdat$syndrome)  #Review levels of 'syndrome' variable
[1] "Unknown" "WNF"     "WNND"
> myrows <- wdat$county=="Los Angeles" & wdat$syndrome=="WNND"
> mycols <- c("id", "county", "age", "sex", "syndrome", "death")
> wnv.la <- wdat[myrows, mycols]
> wnv.la
     id      county age sex syndrome death
25   25 Los Angeles  70   M     WNND    No
26   26 Los Angeles  59   M     WNND    No
27   27 Los Angeles  59   M     WNND    No
...
734 736 Los Angeles  71   M     WNND   Yes
770 772 Los Angeles  72   M     WNND    No
776 778 Los Angeles  50   F     WNND    No
```

In this example, the data frame rows were indexed by logical vector, and the columns were indexed by names. We emphasize this method because it only requires application of previously learned principles that always work with R objects.

An alternative method is to use the `subset` function. The first argument specifies the data frame, the second argument is a Boolean operation that evaluates to a logical vector, and the third argument specifies what variables (or range of variables) to include or exclude.

```
> wnv.sf2 <- subset(wdat, county=="Los Angeles" & syndrome=="WNND",
+                   select = c(id, county, age, sex, syndrome, death))
> wnv.sf2[1:6,]
   id      county age sex syndrome death
25 25 Los Angeles  70   M     WNND    No
26 26 Los Angeles  59   M     WNND    No
27 27 Los Angeles  59   M     WNND    No
```

```
47 47 Los Angeles  57   M     WNND    No
48 48 Los Angeles  60   M     WNND    No
49 49 Los Angeles  34   M     WNND    No
```

This example is equivalent but specifies range of variables using the `:` function:

```
> wnv.sf3 <- subset(wdat, county=="Los Angeles" & syndrome=="WNND",
+                   select = c(id:syndrome, death))
> wnv.sf3[1:6,]
   id      county age sex syndrome death
25 25 Los Angeles  70   M     WNND    No
26 26 Los Angeles  59   M     WNND    No
27 27 Los Angeles  59   M     WNND    No
47 47 Los Angeles  57   M     WNND    No
48 48 Los Angeles  60   M     WNND    No
49 49 Los Angeles  34   M     WNND    No
```

This example is equivalent but specifies variables to exclude using the − function:

```
> wnv.sf4 <- subset(wdat, county=="Los Angeles" & syndrome=="WNND",
+                   select = -c(date.onset, date.tested))
> wnv.sf4[1:6,]
   id      county age sex syndrome death
25 25 Los Angeles  70   M     WNND    No
26 26 Los Angeles  59   M     WNND    No
27 27 Los Angeles  59   M     WNND    No
47 47 Los Angeles  57   M     WNND    No
48 48 Los Angeles  60   M     WNND    No
49 49 Los Angeles  34   M     WNND    No
```

The `subset` function offers some conveniences such as the ability to specify a range of fields to include using the `:` function, and to specify a group of fields to exclude using the − function.

2.6.5 Replacing data frame components

With data frames, as with all R data objects, anything that can be indexed can be replaced. We already saw some examples of replacing names. For practice, study and implement the examples in Table 2.33 on the facing page.

Table 2.33 Common ways of replacing data frame components

Replacing	Try these examples

By position	`data(infert)`
	`infert[1:4, 1:2]`
	`infert[1:4, 2] <- c(NA, 45, NA, 23)`
	`infert[1:4, 1:2]`

By name	`names(infert)`
	`infert[1:4, c("education", "age")]`
	`infert[1:4, c("age")] <- c(NA, 45, NA, 23)`
	`infert[1:4, c("education", "age")]`

By logical	`table(infert$parity)`
	`# change values of 5 or 6 to missing (NA)`
	`infert$parity[infert$parity==5 | infert$parity==6]`
	`<- NA`
	`table(infert$parity)`
	`table(infert$parity, exclude=NULL)`

Table 2.34 Common ways of operating on a data frame

Function	Description	Try these examples

tapply	Apply a function to strata of a vector that are defined by a unique combination of the levels of selected factors	`data(infert)`
		`args(tapply)`
		`tapply(infert$age, infert$education,`
		`  mean, na.rm = TRACE)`

lapply	Apply a function to each component of the list	`lapply(infert[,1:3], table)`
sapply	Apply a function to each component of a list, and simplify	`sapply(infert[,c("age", "parity")],` `  mean, na.rm = TRUE)`

aggregate	Splits the data into subsets, computes summary statistics for each, and returns the result in a convenient form.	`aggregate(infert[,c("age", "parity")],`
		`  by = list(Education =`
		`infert$education,`
		`  Induced = infert$induced), mean)`

mapply	Apply a function to the first elements of each argument, the second elements, the third elements, and so on.	`df <- data.frame(var1 = 1:4, var2 =`
		`4:1)`
		`mapply("*", df$var1, df$var2)`
		`mapply(c, df$var1, df$var2)`
		`mapply(c, df$var1, df$var2, SIMPLIFY=F)`

2.6.6 Operations on data frames

A data frame is of mode list, and functions that operate on components of a list will work with data frames. For example, consider the California population estimates and projections for the years 2000–2050.

```
> capop <- read.csv("http://www.dof.ca.gov/HTML/DEMOGRAP/
Data/RaceEthnic/Population-00-50/documents/California.txt")
> str(capop)
'data.frame':   10302 obs. of  11 variables:
 $ County          : int  59 59 59 59 59 59 59 59 59 59 ...
 $ Year            : int  2000 2000 2000 2000 2000 2000 ...
 $ Sex             : Factor w/ 2 levels "F","M": 1 1 1 ...
 $ Age             : int  0 1 2 3 4 5 6 7 8 9 ...
 $ White           : int  75619 76211 76701 78551 82314 ...
 $ Hispanic        : int  115911 113706 114177 116733 0 ...
 $ Asian           : int  20879 20424 21044 21920 22760 ...
 $ Pacific.Islander: int  741 765 806 817 884 945 961 ...
 $ Black           : int  14629 15420 15783 16531 17331 ...
 $ American.Indian : int  1022 1149 1169 1318 1344 1363 ...
 $ Multirace       : int  10731 8676 8671 8556 8621 ...
> capop[1:5,1:8]
  County Year Sex Age White Hispanic Asian Pacific.Islander
1     59 2000   F   0 75619   115911 20879              741
2     59 2000   F   1 76211   113706 20424              765
3     59 2000   F   2 76701   114177 21044              806
4     59 2000   F   3 78551   116733 21920              817
5     59 2000   F   4 82314   119995 22760              884
```

Now, suppose we want to assess the range of the numeric fields. If we treat the data frame as a list, both `lapply` or `sapply` works:

```
> sapply(capop[-3], range)
     County Year Age  White Hispanic Asian Pacific.Islander
[1,]     59 2000   0    497      110    76                1
[2,]     59 2050 100 148246   277168 46861             1890
     Black American.Indian Multirace
[1,]    57               0         5
[2,] 26983            8181     17493
```

However, if we treat the data frame as a matrix, `apply` also works:

```
> apply(capop[,-3], 2, range)
     County Year Age  White Hispanic Asian Pacific.Islander
[1,]     59 2000   0    497      110    76                1
[2,]     59 2050 100 148246   277168 46861             1890
     Black American.Indian Multirace
```

```
[1,]    57             0        5
[2,] 26983          8181    17493
```

Some R functions, such as `summary`, will summarize every variable in a data frame without having to use `lapply` or `sapply`.

```
> summary(capop[1:7])
     County        Year       Sex           Age
 Min.   :59   Min.   :2000   F:5151   Min.   :  0
 1st Qu.:59   1st Qu.:2012   M:5151   1st Qu.: 25
 Median :59   Median :2025            Median : 50
 Mean   :59   Mean   :2025            Mean   : 50
 3rd Qu.:59   3rd Qu.:2038            3rd Qu.: 75
 Max.   :59   Max.   :2050            Max.   :100
     White           Hispanic          Asian
 Min.   :   497   Min.   :   110   Min.   :   76
 1st Qu.: 63134   1st Qu.: 29962   1st Qu.:19379
 Median : 75944   Median :115646   Median :30971
 Mean   : 71591   Mean   :101746   Mean   :27769
 3rd Qu.: 88021   3rd Qu.:154119   3rd Qu.:37763
 Max.   :148246   Max.   :277168   Max.   :46861
```

2.6.6.1 The `aggregate` function

The `aggregate` function is almost identical to the `tapply` function. Recall that `tapply` allows us to apply a function to a vector that is stratified by one or more fields; for example, calculating mean age (1 field) stratified by sex and ethnicity (2 fields). In contrast, `aggregate` allows us to apply a function to a group of fields that are stratified by one or more fields; for example, calculating the mean weight and height (2 fields) stratified by sex and ethnicity (2 fields):

```
> sex <- c("M", "M", "M", "M", "F", "F", "F", "F")
> eth <- c("W", "W", "B", "B", "W", "W", "B", "B")
> wgt <- c(140, 150, 150, 160, 120, 130, 130, 140)
> hgt <- c( 60,  70,  70,  80,  40,  50,  50,  60)
> df <- data.frame(sex, eth, wgt, hgt)
> aggregate(df[, 3:4], by = list(Gender = df$sex,
+           Ethnicity = df$eth), FUN = mean)
  Gender Ethnicity wgt hgt
1      F         B 135  55
2      M         B 155  75
3      F         W 125  45
4      M         W 145  65
```

For another example, in the `capop` data frame, we notice that the variable age goes from 0 to 100 by 1-year intervals. It will be useful to aggregate ethnic-specific population estimates into larger age categories. More specifically, we want to calculate the sum of ethnic-specific population estimates (6 fields) stratified by age category, sex, and year (3 fields). We will create a new 7-level age category field commonly used by the National Center for Health Statistics. Naturally, we use the `aggregate` function:

```
> capop <- read.csv("http://www.dof.ca.gov/HTML/DEMOGRAP/Data/
           RaceEthnic/Population-00-50/documents/California.txt")
> to.keep <- c("White", "Hispanic", "Asian", "Pacific.Islander",
+               "Black", "American.Indian", "Multirace")
> age.nchs7 <- c(0, 1, 5, 15, 25, 45, 65, 101)
> capop$agecat7 <- cut(capop$Age, breaks = age.nchs7, right=FALSE)
> capop7 <- aggregate(capop[,to.keep], by = list(Age=capop$agecat7,
+                       Sex=capop$Sex, Year=capop$Year), FUN = sum)
> levels(capop7$Age)[7] <- "65+"
> capop7[1:14, 1:6]
        Age Sex Year   White Hispanic  Asian
1     [0,1)   F 2000   75619   115911  20879
2     [1,5)   F 2000  313777   464611  86148
3    [5,15)   F 2000  924930  1124573 241047
4   [15,25)   F 2000  868767   946948 272846
5   [25,45)   F 2000 2360250  1742366 667956
6   [45,65)   F 2000 2102090   735062 445039
7       65+   F 2000 1471842   279865 208566
8     [0,1)   M 2000   79680   121585  21965
9     [1,5)   M 2000  331193   484068  91373
10   [5,15)   M 2000  979233  1175384 257574
11  [15,25)   M 2000  925355  1080868 279314
12  [25,45)   M 2000 2465194  1921896 614608
13  [45,65)   M 2000 2074833   687549 384011
14      65+   M 2000 1075226   202299 154966
```

2.7 Managing data objects

When we work in R we have a workspace. Think of the workspace as our desktop that contains the “objects” (data and tools) we use to conduct our work. To view the objects in our workspace use the `ls` or `objects` functions (Table 2.35 on the next page):

```
> ls()
 [1] "add.to.x"     "add.to.y"     "age"          "age.nchs7"
 [5] "agecat"       "alt"          "ar.num"       "capop"
 [9] "capop7"       "dat"          "dat.ordered"  "dat.random"
[13] "dat2"         "dat3"         "dat4"         "dd"
```

Table 2.35 Common ways of managing data objects

Function	Description	Try these examples
ls objects	List objects	`ls()` `objects() #equivalent`

rm
remove
	Remove object(s)	`yy <- 1:5; ls()`
		`rm(yy); ls()`
		`# remove in objects in working environment`
		`# Don't do this unless we are really sure`
		`rm(list = ls())`

save.image	Saves current workspace	`save.image()`

save
load
	Writes R objects to the specified external file. The objects can be read back from the file at a later date using 'load'	`x <- runif(20)`
		`y <- list(a = 1, b = TRUE, c = "oops")`
		`save(x, y, file = "c:/temp/xy.Rdata")`
		`rm(x,y); x; y`
		`load(file = "c:/temp/xy.Rdata")`
		`x`
		`y`

```
...
```

We use the `pattern` option to search for object names that contain the pattern we specify.

```
> ls(pattern = "dat")
 [1] "dat"          "dat.ordered" "dat.random"  "dat2"
 [5] "dat3"         "dat4"        "mydat"       "sdat"
 [9] "sdat.asr"     "sdat3"       "st.dates"    "udat1"
[13] "udat2"        "wdat"
```

The `rm` or `remove` functions will remove workspace objects.

```
> rm(dat, dat2, dat3, dat4)
> ls(patt="dat")
 [1] "dat.ordered" "dat.random"  "mydat"       "sdat"
 [5] "sdat.asr"    "sdat3"       "st.dates"    "udat1"
 [9] "udat2"       "wdat"
```

To remove all data objects use the following code with extreme caution:

```
rm(list = ls())
```

However, the object names may not be sufficiently descriptive to know what these objects contain. To assess R objects in our workspace we use the functions summarized in Table 2.2 on page 30. In general, we never go wrong using the `str`, `mode`, and `class` functions.

```
> mode(capop7)
```

```
[1] "list"
> class(capop7)
[1] "data.frame"
> str(capop7)
`data.frame':   714 obs. of  10 variables:
 $ Age             : Factor w/ 7 levels "[0,1)",..: 1 2 6 ...
 $ Sex             : Factor w/ 2 levels "F","M": 1 1 1 1  ...
 $ Year            : Factor w/ 51 levels "2000",..: 1 1 1 ...
 $ White           : int  75619 313777 924930 868767 ...
 $ Hispanic        : int  115911 464611 1124573 946948 ...
 $ Asian           : int  20879 86148 241047 272846 ...
 $ Pacific.Islander: int  741 3272 9741 9629 19085 9898 ...
 $ Black           : int  14629 65065 195533 158923 ...
 $ American.Indian : int  1022 4980 15271 14301 30960 ...
 $ Multirace       : int  10731 34524 78716 56735 82449 ...
>
> mode(orcalc)
[1] "function"
> class(orcalc)
[1] "function"
> str(orcalc)
function (x)
 - attr(*, "source")= chr [1:5] "function(x){" ...
> orcalc
function(x){
  or <- (x[1,1]*x[2,2])/(x[1,2]*x[2,1])
  pval <- fisher.test(x)$p.value
  list(data = x, odds.ratio = or, p.value = pval)
```

Objects created in the workspace are available during the R session. Upon closing the R session, R asks whether to save the workspace. To save the objects without exiting an R session, use the `save.image` function:

```
> save.image()
```

The `save.image` function is actually a special case of the `save` function:

```
save(list = ls(all = TRUE), file = ".RData")
```

The `save` function saves an R object as an external file. This file can be loaded using the `load` function.

```
> x <- 1:5
> x
[1] 1 2 3 4 5
> save(x, file="/home/tja/temp/x")
> rm(x)
> x
```

Table 2.36 Assessing and coercing data objects

Query data type	Coerce to data type
is.vector	as.vector
is.matrix	as.matrix
is.array	as.array
is.list	as.list
is.data.frame	as.data.frame
is.factor	as.factor
is.ordered	as.ordered
is.table	as.table
is.numeric	as.numeric
is.integer	as.integer
is.character	as.character
is.logical	as.logical
is.function	as.function
is.null	as.null
is.na	n/a
is.nan	n/a
is.finite	n/a
is.infinite	n/a

```
Error: object "x" not found
> load(file="/home/tja/temp/x")
> x
[1] 1 2 3 4 5
```

Table 2.36 provides more functions for conducting specific object queries and for coercing one object type into another. For example, a vector is not a matrix.

```
> is.matrix(1:3)
[1] FALSE
```

However, a vector can be coerced into a matrix.

```
> as.matrix(1:3)
     [,1]
[1,]    1
[2,]    2
[3,]    3
> is.matrix(as.matrix(1:3))
[1] TRUE
```

A common use would be to coerce a factor into a character vector.

```
> sex <- factor(c("M", "M", "M", "M", "F", "F", "F", "F"))
> sex
[1] M M M M F F F F
Levels: F M
> unclass(sex)      #does not coerce into character vector
[1] 2 2 2 2 1 1 1 1
attr(,"levels")
[1] "F" "M"
> as.character(sex) #yes, works
[1] "M" "M" "M" "M" "F" "F" "F" "F"
```

In R, missing values are represented by the value `NA` ("not available"). The `is.na` function evaluates an object and returns a logical vector indicating which positions contain `NA`. The `!is.na` version returns positions that do not contain `NA`.

```
> x <- c(12, 34, NA, 56, 89)
> is.na(x)
[1] FALSE FALSE  TRUE FALSE FALSE
> !is.na(x)
[1]  TRUE  TRUE FALSE  TRUE  TRUE
```

We can use `is.na` to replace missing values.

```
> x[is.na(x)] <- 999
> x
[1]  12  34 999  56  89
```

In R, `NaN` represents "not a number" and `Inf` represent an infinite value. Therefore, we can use `is.nan` and `is.infinite` to assess which positions contain `NaN` and `Inf`, respectively.

```
> x <- c(0, 3, 0, -6)
> y <- c(4, 0, 0, 0)
> z <- x/y
> z
[1]    0  Inf  NaN -Inf
> is.nan(z)
[1] FALSE FALSE  TRUE FALSE
> is.infinite(z)
[1] FALSE  TRUE FALSE  TRUE
```

2.8 Managing our workspace

Our workspace is like a desktop that contains the "objects" (data and tools) we use to conduct our work. Use the `getwd` function to list the file path to the workspace file `.RData`.

```
> getwd()
[1] "/home/tja/Data/R/home"
```

Use the `setwd` function to set up a new workspace location. A new `.RData` file will automatically be created there

```
setwd("/home/tja/Data/R/newproject")
```

This is one method to manage multiple workspaces for one's projects.

Use the `search` function to list the packages, environments, or data frames attached and available.

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

The global environment `.GlobalEnv` is our workspace. The `searchpaths` function list the full paths:

```
> searchpaths()
[1] ".GlobalEnv"                     "/usr/lib/R/library/stats"
[3] "/usr/lib/R/library/graphics"    "/usr/lib/R/library/grDevices"
[5] "/usr/lib/R/library/utils"       "/usr/lib/R/library/datasets"
[7] "/usr/lib/R/library/methods"     "Autoloads"
[9] "/usr/lib/R/library/base"
```

Table 2.37 Risk of Death in a 20-year Period Among Women in Whickham, England, According to Smoking Status at the Beginning of the Period

	Smoking	
Vital Status	Yes	No
Dead	139	230
Alive	443	502

Table 2.38 Risk of Death in a 20-year Period Among Women in Whickham, England, According to Smoking Status at the Beginning of the Period

	Smoking		
Vital Status	Yes	No	Total
Dead	139	230	369
Alive	443	502	945
Total	582	732	1314

Table 2.39 Risk Ratio and Odds Ratio of Death in a 20-year Period Among Women in Whickham, England, According to Smoking Status at the Beginning of the Period

	Smoking	
	Yes	No
Risk	0.24	0.31
Risk Ratio	0.76	1.00
Odds	0.31	0.46
Odds Ratio	0.68	1.00

Problems

2.1. Download and install RStudio from `http://www.rstudio.org`. Start a new project. For example, I started a new project in the following directory: /home/tja/Documents/courses/ph251d/Rproj. Inside the Rproj directory I renamed the XXX.Rproj file to ph251d.Rproj. From the main menu select File → New → R Script. Save the R script as ph251d.R in the Rproj directory. Write and run R code from this script. Test R code from the tables in this chapter.

2.2. Names and describe the six type of R objects.

2.3. How many ways can we index an object?

2.4. Finish the sentence: Any R object component(s) that can be indexed, can be ....

2.5. Recreate Table 2.37 using any combination of the `matrix`, `cbind`, `rbind`, `dimnames`, or `names` functions.

2.6. Starting with the 2×2 matrix object we created previously, using only the `apply`, `cbind`, `rbind`, `names`, and `dimnames` functions, recreate Table 2.38.

2.7. Using the 2 × 2 data from Table 2.37 on the preceding page, use the `sweep` and `apply` functions to calculate marginal and joint distributions.

2.8. Using the data from the previous problems, recreate Table 2.39 on the previous page and interpret the results.

2.9. Read in the Whickham, England data using the R code below.

```
wdat = read.table("http://www.medepi.net/data/whickham-engl.txt",
                   sep = ",", header = TRUE)
str(wdat)
xtabs(~Vital.Status + Age + Smoking, data = wdat)
```

Stratified by age category, calculate the risk of death comparing smokers to non-smokers. Show your results. What is your interpretation.

2.10. Use the `read.table` function to read in the syphilis data available at `http://www.medepi.net/data/syphilis89c.txt`. Evaluate structure of data frame. Do not attach `std` data frame (yet). Create a 3-dimensional array using both the `table` or `xtabs` function. Now attach the `std` data frame using the `attach` function. Create the same 3-dimensional array using both the `table` or `xtabs` function.

2.11. Use the `apply` function to get marginal totals for the syphilis 3-dimensional array.

2.12. Use the `sweep` and `apply` functions to get marginal and joint distributions for a 3-D array.

2.13. Review and read in the group-level, tabular data set of primary and secondary syphilis cases in the United States in 1989 available at `http://www.medepi.net/data/syphilis89b.txt`. Use the `rep` function on the data frame fields to recreate the individual-level data frame with over 40,000 observations.

2.14. Working with population estimates can be challenging because of the amount of data manipulation. Study the 2000 population estimates for California Counties: `http://www.medepi.net/data/calpop/CalCounties2000.txt`. Now, study and implement this R code. For each expression or group of expressions, explain in words what the R code is doing. Be sure to display intermediate objects to understand each step.

```
#1 Read county names
cty <- scan("http://www.medepi.net/data/calpop/calcounty.txt",
            what="")

#2 Read county population estimates
calpop =
read.csv("http://www.medepi.net/data/calpop/CalCounties2000.txt",
header = T)
```

```
#2 Replace county number with county name
for(i in 1:length(cty)){
    calpop$County[calpop$County==i] <- cty[i]
}

#3 Discretize age into categories
calpop$Agecat <- cut(calpop$Age, c(0,20,45,65,100),
                     include.lowest = TRUE, right = FALSE)

#4 Create combined API category
calpop$AsianPI <- calpop$Asian + calpop$Pacific.Islander

#5 Shorten selected ethnic labels
names(calpop)[c(6, 9, 10)] = c("Latino", "AfrAmer", "AmerInd")

#6 Index Bay Area Counties
baindex <- calpop$County=="Alameda" | calpop$County=="San Francisco"
bapop <- calpop[baindex,]
bapop

#7 Labels for later use
agelabs <- names(table(bapop$Agecat))
sexlabs <- c("Female", "Male")
racen <- c("White", "AfrAmer", "AsianPI", "Latino", "Multirace",
           "AmerInd")
ctylabs <- names(table(bapop$County))

#8 Aggregate
bapop2 <- aggregate(bapop[,racen],
                    list(Agecat = bapop$Agecat, Sex = bapop$Sex,
                         County = bapop$County), sum)
bapop2

#9 Temp matrix of counts
tmp <- as.matrix(cbind(bapop2[1:4,racen], bapop2[5:8,racen],
            bapop2[9:12,racen], bapop2[13:16,racen]))

#10 Convert into final array
bapop3 <- array(tmp, c(4, 6, 2, 2))
dimnames(bapop3) <- list(agelabs, racen, sexlabs, ctylabs)
bapop3
```

CHAPTER 3

Managing epidemiologic data in R

3.1 Entering and importing data

There are many ways of getting our data into R for analysis. In the section that follows we review how to enter the Unversity Group Diabetes Program data (Table 3.1) as well as the original data from a comma-delimited text file. We will use the following approaches:

- Entering data at the command prompt
- Importing data from a file
- Importing data using an URL

3.1.1 Entering data at the command prompt

We review four methods. For Methods 1 and 2, data are entered directly at the command prompt. Method 3 uses the same R expressions and data as Methods 1 and

Table 3.1 Deaths among subjects who received tolbutamide and placebo in the Unversity Group Diabetes Program (1970), stratifying by age

	Age<55		Age≥55		Combined	
	Tolbutamide	Placebo	Tolbutamide	Placebo	Tolbutamide	Placebo
Deaths	8	5	22	16	30	21
Survivors	98	115	76	69	174	184

2, but they are entered into a text editor, saved as an text file with a .R extension (e.g., job02.R), and then executed from the command prompt using the `source` function. Alternatively, the R expressions and data can be copied and pasted into R.[1] And, for Method 4 we use R's spreadsheet editor (least preferred).

3.1.1.1 Method 1

For review, a convenient way to enter data at the command prompt is to use the `c` function:

```
> #enter data for a vector
> vec1 <- c(8, 98, 5, 115); vec1
[1]   8  98   5 115
> vec2 <- c(22, 76, 16, 69); vec2
[1] 22 76 16 69
>
> #enter data for a matrix
> mtx1 <- matrix(vec1, 2, 2); mtx1
     [,1] [,2]
[1,]    8    5
[2,]   98  115
> mtx2 <- matrix(vec2, 2, 2); mtx2
     [,1] [,2]
[1,]   22   16
[2,]   76   69
>
> #enter data for an array and sum across strata
> udat <- array(c(vec1, vec2), c(2, 2, 2)); udat
, , 1

     [,1] [,2]
[1,]    8    5
[2,]   98  115

, , 2

     [,1] [,2]
[1,]   22   16
[2,]   76   69

> udat.tot <- apply(udat, c(1, 2), sum); udat.tot
     [,1] [,2]
[1,]   30   21
```

---

[1] In RStudio, select from main menu Code → Run Region, etc.

```
[2,]  174  184
>
> #enter a list
> x <- list(crude.data = udat.tot, stratified.data = udat)
> x$crude.data
     [,1] [,2]
[1,]   30   21
[2,]  174  184
> x$stratified
, , 1

     [,1] [,2]
[1,]    8    5
[2,]   98  115

, , 2

     [,1] [,2]
[1,]   22   16
[2,]   76   69

>
> #enter simple data frame
> subjname <- c("Pedro", "Paulo", "Maria")
> subjno <- 1:length(subjname)
> age <- c(34, 56, 56)
> sex <- c("Male", "Male", "Female")
> dat <- data.frame(subjno, subjname, age, sex); dat
  subjno subjname age    sex
1      1    Pedro  34   Male
2      2    Paulo  56   Male
3      3    Maria  56 Female
>
> #enter a simple function
> odds.ratio <- function(aa, bb, cc, dd){ aa*dd / (bb*cc)}
> odds.ratio(30, 174,  21, 184)
[1] 1.510673
```

3.1.1.2 Method 2

Method 2 is identical to Method 1 except one uses the `scan` function. It does not matter if we enter the numbers on different lines, it will still be a vector. Remember that we must press the Enter key twice after we have entered the last number.

```
> udat.tot <- scan()
```

```
1: 30 174
3: 21 184
5:
Read 4 items
> udat.tot
[1]  30 174  21 184
```

To read in a matrix at the command prompt combine the `matrix` and `scan` functions. Again, it does not matter on what lines we enter the data, as long as they are in the correct order because the `matrix` function reads data in column-wise.

```
> udat.tot <- matrix(scan(), 2, 2)
1: 30 174 21 184
5:
Read 4 items
> udat.tot
     [,1] [,2]
[1,]   30   21
[2,]  174  184

> udat.tot <- matrix(scan(), 2, 2, byrow = T) #read row-wise
1: 30 21 174 184
5:
Read 4 items
> udat.tot
     [,1] [,2]
[1,]   30   21
[2,]  174  184
```

To read in an array at the command prompt combine the `array` and `scan` functions. Again, it does not matter on what lines we enter the data, as long as they are in the correct order because the array function reads the numbers column-wise. In this example we include the `dimnames` argument.

```
> udat <- array(scan(), dim = c(2, 2, 2),
+     dimnames = list(Vital.Status = c("Dead","Survived"),
+                     Treatment = c("Tolbutamide", "Placebo"),
+                     Age.Group = c("<55", "55+")))
1: 8 98 5 115 22 76 16 69
9:
Read 8 items
> udat
, , Age.Group = <55

            Treatment
Vital.Status Tolbutamide Placebo
    Dead               8       5
    Survived          98     115

, , Age.Group = 55+

            Treatment
Vital.Status Tolbutamide Placebo
    Dead              22      16
    Survived          76      69
```

To read in a list of vectors of the same length ("fields") at the command prompt combine the `list` and `scan` function. We will need to specify the type of data that will go into each "bin" or "field." This is done by specifying the `what` argument as a list. This list must be values that are either logical, integer, numeric, or character. For example, for a character vector we can use any expression, say *x*, that would evaluate to `TRUE` for `is.character(x)`. For brevity, use `""` for character, `0` for numeric, `1:2` for integer, and `T` or `F` for logical. Look at this example:

```
> dat <- scan("", what = list(1:2, "", 0, "", T))
1: 3 "John Paul" 84.5 Male F
2: 4 "Jane Doe" 34.5 Female T
3:
Read 2 records
> dat
[[1]]
[1] 3 4

[[2]]
[1] "John Paul" "Jane Doe"

[[3]]
[1] 84.5 34.5

[[4]]
[1] "Male"   "Female"

[[5]]
[1] FALSE  TRUE

> str(dat)
List of 5
 $ : int [1:2] 3 4
 $ : chr [1:2] "John Paul" "Jane Doe"
 $ : num [1:2] 84.5 34.5
 $ : chr [1:2] "Male" "Female"
 $ : logi [1:2] FALSE TRUE
```

```
> #same example with field names
> dat <- scan("", what = list(id = 1:2, name = "", age = 0,
+                              sex = "", dead = TRUE))
1: 3 "John Paul" 84.5 Male F
2: 4 "Jane Doe" 34.5 Female T
3:
Read 2 records
> dat
$id
[1] 3 4

$name
[1] "John Paul" "Jane Doe"

$age
[1] 84.5 34.5

$sex
[1] "Male"   "Female"

$dead
[1] FALSE  TRUE

> str(dat)
List of 5
 $ id  : int [1:2] 3 4
 $ name: chr [1:2] "John Paul" "Jane Doe"
 $ age : num [1:2] 84.5 34.5
 $ sex : chr [1:2] "Male" "Female"
 $ dead: logi [1:2] FALSE TRUE
```

To read in a data frame at the command prompt combine the `data.frame`, `scan`, and `list` functions.

```
> dat <- data.frame(scan("", what = list(id=1:2, name="",
+     age=0, sex="", dead=T)) )
1: 3 "John Paul" 84.5 Male F
2: 4 "Jane Doe" 34.5 Female T
3:
Read 2 records
> dat
  id      name  age    sex  dead
1  3 John Paul 84.5   Male FALSE
2  4  Jane Doe 34.5 Female  TRUE
```

3.1.1.3 Method 3

Method 3 uses the same R expressions and data as Methods 1 and 2, but they are entered into a text editor, saved as an ASCII text file with a .R extension (e.g., job01.R), and then executed from the command prompt using the `source` function. Alternatively, the R expressions and data can be copied and pasted into R.[2]

For example, the following expressions are in a text editor and saved to a file named `job01.R`.

```
x <- 1:10
x
```

One can copy and paste this code into R at the commmand prompt.

```
> x <- 1:10
> x
 [1]  1  2  3  4  5  6  7  8  9 10
```

However, if we execute the code using the `source` function, it will only display to the screen those objects that are printed using the `show` or `print` function. Here is the text editor code again, but including `show`.

```
x <- 1:10
show(x)
```

Now, source job01.R using `source` at the command prompt.

```
> source("/home/tja/Documents/Rproj/job01.R")
 [1]  1  2  3  4  5  6  7  8  9 10
```

In general, we highly recommend using a text editor for all our work. The program file (e.g., job01.R) created with the text editor facilitates documenting our code, reviewing our code, debugging our code, replicating our analytic steps, and auditing by external reviewers.

3.1.1.4 Method 4 (optional read)

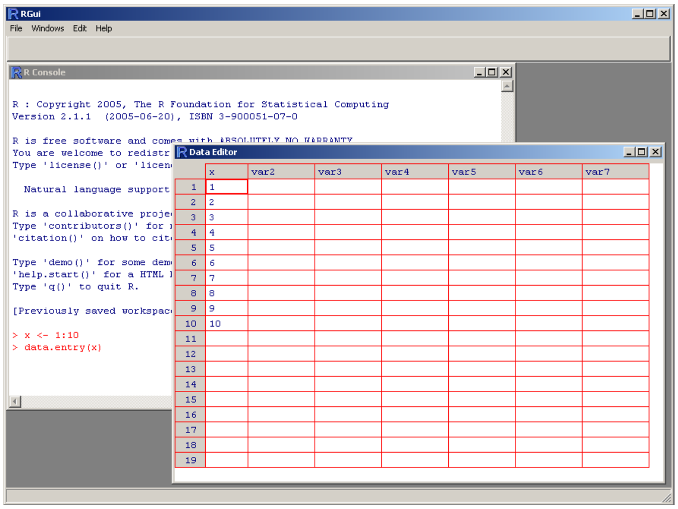
Method 4 uses R's spreadsheet editor.[3] This is not a preferred method because we like the original data to be in a text editor or read in from a data file. We will be using the `data.entry` and `edit` functions. The `data.entry` function allows editing of an existing object and automatically saving the changes to the original object name. In contrast, the `edit` function allows editing of an existing object but it will not save the changes to the original object name; we must explicitly assign it to an object name (event if it is the original name).

To enter a vector we need to initialize a vector and then use the `data.entry` function (Figure 3.1 on the next page).

---

[2] In RStudio, select from main menu Code → Run Region, etc.

[3] Editing of matrix and data frame objects is not currently supported in RStudio.



```
R : Copyright 2005, The R Foundation for Statistical Computing
Version 2.1.1  (2005-06-20), ISBN 3-900051-07-0

R is free software and comes with ABSOLUTELY NO WARRANTY
You are welcome to redistr
Type 'license()' or 'licen

  Natural language support

R is a collaborative proje
Type 'contributors()' for 
'citation()' on how to cit

Type 'demo()' for some dem
'help.start()' for a HTML 
Type 'q()' to quit R.

[Previously saved workspac

> x <- 1:10
> data.entry(x)
```

	x	var2	var3	var4	var5	var6	var7
1	1						
2	2						
3	3						
4	4						
5	5						
6	6						
7	7						
8	8						
9	9						
10	10						
11							
12							
13							
14							
15							
16							
17							
18							
19							

Fig. 3.1 Select Help from the main menu.

```
> x <- numeric(10) #Initialize vector with zeros
> x
 [1] 0 0 0 0 0 0 0 0 0 0
> data.entry(x) #Enter numbers, then close window
> x
 [1]  1  2  3  4  5  6  7  8  9 10
```

However, the `edit` function applied to a vector does not open a spreadsheet. Try the `edit` function and see what happens.

```
xnew <- edit(numeric(10)) #Edit number, then close window
```

To enter data into a spreadsheet matrix, first initialize a matrix and then use the `data.entry` or `edit` function. Notice that the editor added default column names. However, to add our own column names just click on the column heading with our mouse pointer (unfortunately we cannot give row names).

```
> xnew <- matrix(numeric(4),2,2)
> data.entry(xnew)
> xnew <- edit(xnew) #equivalent
>
> #open spreadsheet editor in one step
> xnew <- edit(matrix(numeric(4),2,2))
> xnew
     col1 col2
[1,]   11   33
[2,]   22   44
```

Arrays and nontabular lists cannot be entered using a spreadsheet editor. Hence, we begin to see the limitations of spreadsheet-type approach to data entry. One type of list, the data frame, can be entered using the `edit` function.

To enter a data frame use the `edit` function. However, we do not need to initialize a data frame (unlike with a matrix). Again, click on the column headings to enter column names.

```
> df <- edit(data.frame()) #Spreadsheet screen not shown
> df
    mykids age
1 Tomasito   7
2  Luisito   6
3 Angelita   3
```

When using the `edit` function to create a new data frame we must assign it an object name to save the data frame. Later we will see that when we edit an existing data object we can use the `edit` or `fix` function. The `fix` function differs in that `fix(`*`data_object`*`)` saves our edits directly back to `data_object` without the need to make a new assignment.

```
mypower <- function(x, n){x^n}
fix(mypower)      # Edits saved to 'mypower' object
mypower <- edit(mypower) #equivalent
```

3.1.2 Importing data from a file

3.1.2.1 Reading an ASCII text data file

In this section we review how to read the following types of text data files:

- Comma-separated variable (csv) data file (± headers and ± row names)
- Fixed width formatted data file (± headers and ± row names)

Here is the University Group Diabetes Program randomized clinical trial text data file that is comma-delimited, and includes row names and a header (ugdp.txt).[4] The header is the first line that contains the column (field) names. The row names is the first column that starts on the second line and uniquely identifies each row. Notice that the row names do not have a column name associated with it. A data file can come with either row names or header, neither, or both. Our preference is to work with data files that have a header and data values that are self-explanatory. Even without a data dictionary one can still make sense out of this data set.

```
Status,Treatment,Agegrp
1,Dead,Tolbutamide,<55
2,Dead,Tolbutamide,<55
```

---

[4] Available at `http://www.medepi.net/data/ugdp.txt`

```
...
408,Survived,Placebo,55+
409,Survived,Placebo,55+
```

Notice that the header row has 3 items, and the second row has 4 items. This is because the row names start in the second row and have no column name. This data file can be read in using the `read.table` function, and R figures out that the first column are row names.[5]

```
> ud <- read.table("http://www.medepi.net/data/ugdp.txt",
+     header = TRUE, sep = ",")
> head(ud) #displays 1st 6 lines
  Status   Treatment Agegrp
1   Dead Tolbutamide    <55
2   Dead Tolbutamide    <55
3   Dead Tolbutamide    <55
4   Dead Tolbutamide    <55
5   Dead Tolbutamide    <55
6   Dead Tolbutamide    <55
```

Here is the same data file as it would appear without row names and without a header (ugdp2.txt).

```
Dead,Tolbutamide,<55
Dead,Tolbutamide,<55
...
Survived,Placebo,55+
Survived,Placebo,55+
```

This data file can be read in using the `read.table` function. By default, it adds row names (1, 2, 3, ...).

```
> cnames <- c("Status", "Treatment", "Agegrp")
> udat2 <- read.table("http://www.medepi.net/data/ugdp2.txt",
+     header = FALSE, sep = ",", col.names = cnames)
> head(udat2)
  Status   Treatment Agegrp
1   Dead Tolbutamide    <55
2   Dead Tolbutamide    <55
3   Dead Tolbutamide    <55
4   Dead Tolbutamide    <55
5   Dead Tolbutamide    <55
6   Dead Tolbutamide    <55
```

Here is the same data file as it might appear as a fix formatted file. In this file, columns 1 to 8 are for field #1, columns 9 to 19 are for field #2, and columns 20 to 22 are for field #3. This type of data file is more compact. One needs a data dictionary to know which columns contain which fields.

---

[5] If row names are supplied, they must be unique.

```
Dead    Tolbutamide<55
Dead    Tolbutamide<55
...
SurvivedPlacebo    55+
SurvivedPlacebo    55+
```

This data file would be read in using the `read.fwf` function. Because the field widths are fixed, we must strip the white space using the `strip.white` option.

```
> cnames <- c("Status", "Treatment", "Agegrp")
> udat3 <- read.fwf("http://www.medepi.net/data/ugdp3.txt",
+ width = c(8, 11, 3), col.names = cnames, strip.white = TRUE)
> head(udat3)
  Status   Treatment Agegrp
1   Dead Tolbutamide    <55
2   Dead Tolbutamide    <55
3   Dead Tolbutamide    <55
4   Dead Tolbutamide    <55
5   Dead Tolbutamide    <55
6   Dead Tolbutamide    <55
```

Finally, here is the same data file as it might appear as a fixed width formatted file but with numeric codes (ugdp4.txt). In this file, column 1 is for field #1, column 2 is for field #2, and column 3 is for field #3. This type of text data file is the most compact, however, one needs a data dictionary to make sense of all the 1s and 2s.

```
121
121
...
212
212
```

Here is how this data file would be read in using the `read.fwf` function.

```
> cnames <- c("Status", "Treatment", "Agegrp")
> udat4 <- read.fwf("http://www.medepi.net/data/ugdp4.txt",
+     width = c(1, 1, 1), col.names = cnames)
> head(udat4)
  Status Treatment Agegrp
1      1         2      1
2      1         2      1
3      1         2      1
4      1         2      1
5      1         2      1
6      1         2      1
```

R has other functions for reading text data files (`read.csv`, `read.csv2`, `read.delim`, `read.delim2`). In general, `read.table` is the function used most commonly for reading in data files.

3.1.2.2 Reading data from a binary format (e.g., Stata, Epi Info)

To read data that comes in a binary or proprietary format load the `foreign` package using the `library` function. To review available functions in the the `foreign` package try `help(package = foreign)`. For example, here we read in the 'infert' data set which is also available as a Stata data file.[6]

```
> idat <- read.dta("c:/.../data/infert.dta")
> head(idat)[,1:8]
  id education age parity induced case spontaneous stratum
1  1         0  26      6       1    1           2       1
2  2         0  42      1       1    1           0       2
3  3         0  39      6       2    1           0       3
4  4         0  34      4       2    1           0       4
5  5         1  35      3       1    1           1       5
6  6         1  36      4       2    1           1       6
```

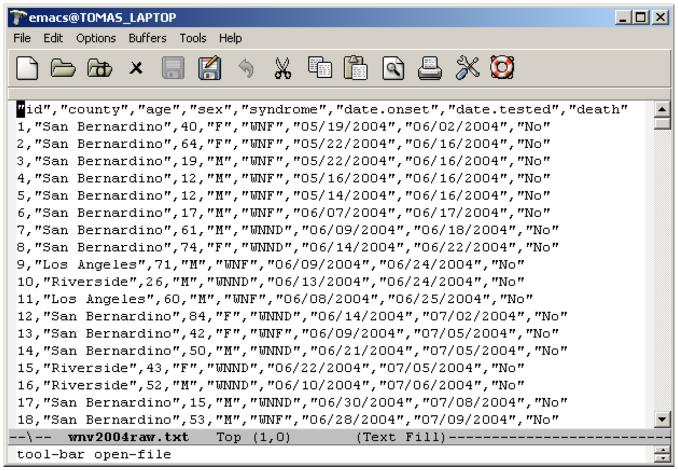
3.1.3 *Importing data using a URL*

As we have already seen, text data files can be read directly off a web server into R using the `read.table` function. Here we load the Western Collaborative Group Study data directly off a web server.

```
> wdat <- read.table("http://www.medepi.net/data/wcgs.txt",
+     header = TRUE, sep = ",")
> str(wdat)
`data.frame':   3154 obs. of  14 variables:
 $ id      : int  2001 2002 2003 2004 2005 2006 2007 2010 ...
 $ age0    : int  49 42 42 41 59 44 44 40 43 42 ...
 $ height0: int  73 70 69 68 70 72 72 71 72 70 ...
 $ weight0: int  150 160 160 152 150 204 164 150 190 175 ...
 $ sbp0    : int  110 154 110 124 144 150 130 138 146 132 ...
 $ dbp0    : int  76 84 78 78 86 90 84 60 76 90 ...
 $ chol0   : int  225 177 181 132 255 182 155 140 149 325 ...
 $ behpat0: int  2 2 3 4 3 4 4 2 3 2 ...
 $ ncigs0 : int  25 20 0 20 20 0 0 0 25 0 ...
 $ dibpat0: int  1 1 0 0 0 0 0 1 0 1 ...
 $ chd69  : int  0 0 0 0 1 0 0 0 0 0 ...
 $ typechd: int  0 0 0 0 1 0 0 0 0 0 ...
 $ time169: int  1664 3071 3071 3064 1885 3102 3074 1032 ...
 $ arcus0 : int  0 1 0 0 1 0 0 0 0 1 ...
```

---

[6] Available at http://www.medepi.net/data/infert.dta



```
"id","county","age","sex","syndrome","date.onset","date.tested","death"
1,"San Bernardino",40,"F","WNF","05/19/2004","06/02/2004","No"
2,"San Bernardino",64,"F","WNF","05/22/2004","06/16/2004","No"
3,"San Bernardino",19,"M","WNF","05/22/2004","06/16/2004","No"
4,"San Bernardino",12,"M","WNF","05/16/2004","06/16/2004","No"
5,"San Bernardino",12,"M","WNF","05/14/2004","06/16/2004","No"
6,"San Bernardino",17,"M","WNF","06/07/2004","06/17/2004","No"
7,"San Bernardino",61,"M","WNND","06/09/2004","06/18/2004","No"
8,"San Bernardino",74,"F","WNND","06/14/2004","06/22/2004","No"
9,"Los Angeles",71,"M","WNF","06/09/2004","06/24/2004","No"
10,"Riverside",26,"M","WNND","06/13/2004","06/24/2004","No"
11,"Los Angeles",60,"M","WNF","06/08/2004","06/25/2004","No"
12,"San Bernardino",84,"F","WNND","06/14/2004","07/02/2004","No"
13,"San Bernardino",42,"F","WNF","06/09/2004","07/05/2004","No"
14,"San Bernardino",50,"M","WNND","06/21/2004","07/05/2004","No"
15,"Riverside",43,"F","WNND","06/22/2004","07/05/2004","No"
16,"Riverside",52,"M","WNND","06/10/2004","07/06/2004","No"
17,"San Bernardino",15,"M","WNND","06/30/2004","07/08/2004","No"
18,"San Bernardino",53,"M","WNF","06/28/2004","07/09/2004","No"
--\--  wnv2004raw.txt   Top (1,0)       (Text Fill)--------------------------
tool-bar open-file
```

Fig. 3.2 Editing West Nile virus human surveillance data in text editor. Source: California Department of Health Services, 2004

3.2 Editing data

In the ideal setting, our data has already been checked, errors corrected, and ready to be analyzed. Post-collection data editing can be minimized by good design and data collection. However, we may still need to make corrections or changes in data values.

3.2.1 *Text editor*

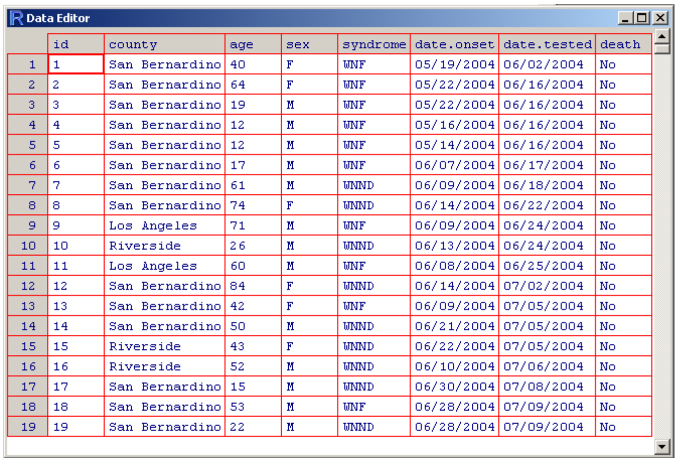
For small data sets, it may be convenience to edit the data in our favorite text editor. Key-recording macros, and search and replace tools can be very useful and efficient. Figure 3.2 displays West Nile virus (WNV) infection surveillance data.[7] This file is a comma-delimited data file with a header.

3.2.2 *The `data.entry`, `edit`, or `fix` functions*

For vector and matrices we can use the `data.entry` function to edit these data object elements. For data frames and functions use the `edit` or `fix` functions. Remember that changes made with the `edit` function are not saved unless we assign it

---

[7] Raw data set available at `http://www.medepi.net/data/wnv2004raw.txt`, and clean data set available at `http://www.medepi.net/data/wnv2004fin.txt`



	id	county	age	sex	syndrome	date.onset	date.tested	death
1	1	San Bernardino	40	F	WNF	05/19/2004	06/02/2004	No
2	2	San Bernardino	64	F	WNF	05/22/2004	06/16/2004	No
3	3	San Bernardino	19	M	WNF	05/22/2004	06/16/2004	No
4	4	San Bernardino	12	M	WNF	05/16/2004	06/16/2004	No
5	5	San Bernardino	12	M	WNF	05/14/2004	06/16/2004	No
6	6	San Bernardino	17	M	WNF	06/07/2004	06/17/2004	No
7	7	San Bernardino	61	M	WNND	06/09/2004	06/18/2004	No
8	8	San Bernardino	74	F	WNND	06/14/2004	06/22/2004	No
9	9	Los Angeles	71	M	WNF	06/09/2004	06/24/2004	No
10	10	Riverside	26	M	WNND	06/13/2004	06/24/2004	No
11	11	Los Angeles	60	M	WNF	06/08/2004	06/25/2004	No
12	12	San Bernardino	84	F	WNND	06/14/2004	07/02/2004	No
13	13	San Bernardino	42	F	WNF	06/09/2004	07/05/2004	No
14	14	San Bernardino	50	M	WNND	06/21/2004	07/05/2004	No
15	15	Riverside	43	F	WNND	06/22/2004	07/05/2004	No
16	16	Riverside	52	M	WNND	06/10/2004	07/06/2004	No
17	17	San Bernardino	15	M	WNND	06/30/2004	07/08/2004	No
18	18	San Bernardino	53	M	WNF	06/28/2004	07/09/2004	No
19	19	San Bernardino	22	M	WNND	06/28/2004	07/09/2004	No

Fig. 3.3 Using the `fix` function to edit the WNV surveillance data frame. Unfortunately, this approach does not facilitate documenting our edits. Source: California Department of Health Services, 2004

to the original or new object name. In contrast, changes made with the `fix` function are saved back to the original data object name. Therefore, be careful when we use the `fix` function because we may unintentionally overwrite data.

Now let's read in the WNV surveillance raw data as a data frame. Then, using the `fix` function, we will edit the first three records where the value for the syndome variable is "Unk" and change it to NA for missing (Figure 3.3). We will also change "." to NA.

```
> wd <- read.table("http://www.medepi.net/data/wnv/wnv2004raw.txt",
+      header = TRUE, sep = ",", as.is = TRUE)
> wd[wd$syndrome=="Unknown",][1:3,]  #before edits (3 records)
     id      county age sex syndrome date.onset date.tested death
128 128 Los Angeles  81   M  Unknown 07/28/2004  08/11/2004     .
129 129   Riverside  44   F  Unknown 07/25/2004  08/11/2004     .
133 133 Los Angeles  36   M  Unknown 08/04/2004  08/11/2004    No
> fix(wd) #open R spreadsheet and make edits (see figure)
> wd[c(128, 129, 133),]  #after edits (3 records)
     id      county age sex syndrome date.onset date.tested death
128 128 Los Angeles  81   M       NA 07/28/2004  08/11/2004    NA
129 129   Riverside  44   F       NA 07/25/2004  08/11/2004    NA
133 133 Los Angeles  36   M       NA 08/04/2004  08/11/2004    No
```

First, notice that in the `read.table` function `as.is=TRUE`. This means the data is read in without R making any changes to it. In other words, character vectors are not automatically converted to factors. We set the option because we knew we were going to edit and make corrections to the data set, and create factors later. In this example, I manually started changing the missing values "Unknown" to NA (R's representation of missing values). However, this manual approach would be very inefficient. A better approach is to specify which values in the data frame should be converted to NA. In the `read.table` function we should have set the option `na.string=c("Unknown", ".")`, converting the character strings "Unknown" and "." into NA. Let's replace the missing values with NAs upon reading the data file.

```
> wd <- read.table("http://www.medepi.net/data/wnv/wnv2004raw.txt",
+     header = TRUE, sep = ",", as.is = TRUE,
+     na.string=c("Unknown", "."))
> wd[c(128, 129, 133),]  #verify change
     id      county age sex syndrome date.onset date.tested death
128 128 Los Angeles  81   M     <NA> 07/28/2004  08/11/2004  <NA>
129 129   Riverside  44   F     <NA> 07/25/2004  08/11/2004  <NA>
133 133 Los Angeles  36   M     <NA> 08/04/2004  08/11/2004    No
```

3.2.3 *Vectorized approach*

How do we make these and other changes after the data set has been read into R? Although using R's spreadsheet function is convenient, we do not recommend it because manual editing is inefficient, our work cannot be replicated and audited, and documentation is poor. Instead use R's vectorized approach. Let's look at the distribution of responses for each variable to assess what needs to be "cleaned up," in addition to converting missing values to NA.

```
> wd <- read.table("http://www.medepi.net/data/wnv/wnv2004raw.txt",
+     header = TRUE, sep = ",", as.is = TRUE)
> str(wd)
'data.frame':   779 obs. of  8 variables:
 $ id         : int  1 2 3 4 5 6 7 8 9 10 ...
 $ county     : chr  "San Bernardino" "San Bernardino" ...
 $ age        : chr  "40" "64" "19" "12" ...
 $ sex        : chr  "F" "F" "M" "M" ...
 $ syndrome   : chr  "WNF" "WNF" "WNF" "WNF" ...
 $ date.onset : chr  "05/19/2004" "05/22/2004" ...
 $ date.tested: chr  "06/02/2004" "06/16/2004" ...
 $ death      : chr  "No" "No" "No" "No" ...
> lapply(wd, table) #apply 'table' function to fields
$id

  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
  1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
...
768 769 770 771 772 773 774 775 776 777 778 779 780 781
```

```
  1   1   1   1   1   1   1   1   1   1   1   1   1   1

$county

         Butte         Fresno          Glenn       Imperial
             7             11              3              1
          Kern           Lake         Lassen    Los Angeles
            59              1              1            306
        Merced         Orange         Placer      Riverside
             1             62              1            109
    Sacramento San Bernardino      San Diego    San Joaquin
             3            187              2              2
   Santa Clara         Shasta Sn Luis Obispo         Tehama
             1              5              1             10
        Tulare        Ventura           Yolo
             3              2              1

$age

 .  1 10 11 12 13 14 15 16 17 18 19  2 20 21 22 23 24 25 26
 6  1  1  1  3  2  3  3  1  4  6  5  1  4  2  3  6  8  3  9
...
82 83 84 85 86 87 88 89  9 91 93 94
10  5  6  4  2  2  1  6  1  4  1  1

$sex

  .   F   M
  2 294 483

$syndrome

Unknown     WNF    WNND
    105     391     283

$date.onset

02/02/2005 05/14/2004 05/16/2004 05/19/2004 05/22/2004
         1          1          1          1          2
...
10/28/2004 10/29/2004 10/30/2004 11/08/2004 11/12/2004
         2          1          1          4          2

$date.tested
```

```
01/21/2005 02/04/2005 02/23/2005 06/02/2004 06/16/2004
         1          1          1          1          4
...
11/29/2004 12/02/2004 12/03/2004 12/07/2004
         8          1          2          1

$death

  .  No Yes
 66 686  27
```

What did we learn? First, there are 779 observations and 781 id's; therefore, 3 observations were removed from the original data set. Second, we see that the variables age, sex, syndrome, and death have missing values that need to be converted to NAs. This can be done one field at a time, or for the whole data frame in one step. Here is the R code.

```
#individually
wd$age[wd$age=="."] <- NA
wd$sex[wd$sex=="."] <- NA
wd$syndrome[wd$syndrome=="Unknown"] <- NA
wd$death[wd$death=="."] <- NA

#or globally
wd[wd=="." | wd=="Unknown"] <- NA
```

After running the above code, let's evaluate one variable to verify the missing values were converted to NAs.

```
> table(wd$death)

 No Yes
686  27
> table(wd$death, exclude=NULL)

  No  Yes <NA>
 686   27   66
```

We also notice that the entry for one of the counties, San Luis Obispo, was misspelled (`Sn Luis Obispo`). We can use replacement to make the corrections:

```
> wd$County[wd$County=="Sn Luis Obispo"] <- "San Luis Obispo"
```

3.2.4 *Text processing*

On occasion, we will need to process and manipulate character vectors using a vectorized approach. For example, suppose we need to convert a character vector of dates from "mm/dd/yy" to "yyyy-mm-dd".[8] We'll start by using the `substr` function. This function extracts characters from a character vector based on position.

```
> bd <- c("07/17/96","12/09/00","11/07/97")
> mon <- substr(bd, start=1, stop=2); mon
[1] "07" "12" "11"
> day <- substr(bd, 4, 5); day
[1] "17" "09" "07"
> yr <- as.numeric(substr(bd, 7, 8)); yr
[1] 96  0 97
> yr2 <- ifelse(yr<=19, yr+2000, yr+1900); yr2
[1] 1996 2000 1997
> bdfin <- paste(yr2, "-", mon, "-", day, sep=""); bdfin
[1] "1996-07-17" "2000-12-09" "1997-11-07"
```

In this example, we needed to convert "00" to "2000", and "96" and "97 to "1996" and "1997", respectively. The trick here was to coerce the character vector into a numeric vector so that 1900 or 2000 could be added to it. Using the `ifelse` function, for values ≤ 19 (arbitrarily chosen), 2000 was added, otherwise 1900 was added. The `paste` function was used to paste back the components into a new vector with the standard date format.

The `substr` function can also be used to replace characters in a character vector. Remember, if it can be indexed, it can be replaced.

```
> bd
[1] "07/17/96" "12/09/00" "11/07/97"
> substr(bd, 3, 3) <- "-"
> substr(bd, 6, 6) <- "-"
> bd
[1] "07-17-96" "12-09-00" "11-07-97"
```

3.3 Sorting data

The `sort` function sorts a vector as expected:

```
> x <- sample(1:10, 10); x
 [1]  4  3  6  1  7  9  5  8  2 10
> sort(x)
 [1]  1  2  3  4  5  6  7  8  9 10
> sort(x, decreasing = TRUE)  #reverse sort
 [1] 10  9  8  7  6  5  4  3  2  1
> rev(sort(x))                #reverse sort
 [1] 10  9  8  7  6  5  4  3  2  1
```

---

[8] ISO 8601 is an international standard for date and time representations issued by the International Organization for Standardization (ISO). See `http://www.iso.org`

Table 3.2 R functions for processing text in character vectors

Function	Description	Try these examples
nchar	Returns the number of characters in each element of a character vector	`x <- c("a", "ab", "abc", "abcd")` `nchar(x)`

substr	Extract or replace substrings in a character vector	`#extraction`
		`mon <- substr(some.dates, 1, 2); mon`
		`day <- substr(some.dates, 4, 5); day`
		`yr <- substr(some.dates, 7, 8); yr`
		`#replacement`
		`mdy <- paste(mon, day, yr); mdy`
		`substr(mdy, 3, 3) <- ’/’`
		`substr(mdy, 6, 6) <- ’/’`
		`mdy`

paste	Concatenate vectors after converting to character	`rd <- paste(mon, "/", day, "/", yr,`
		`sep="")`
		`rd`

strsplit	Split the elements of a character vector into substrings	`some.dates <- c("10/02/70", "02/04/67")`
		`some.dates`
		`strsplit(some.dates, "/")`

However, if we want to sort one vector based on the ordering of elements from another vector, use the `order` function. The `order` function generates an indexing/repositioning vector. Study the following example:

```
> x <- sample(1:20, 5); x
[1] 18 10  6 13 11
> sort(x)     #sorts as expected
[1]  6 10 11 13 18
> y <- sample(1:20, 5); y
[1] 11 10 13  3  9
> order(y)    #4th element to 1st position, 5th to 2nd, etc.
[1] 4 5 2 1 3
> x[order(y)] #use order(y) to sort elements of x
[1] 13 11 10 18  6
```

Based on this we can see that `sort(x)` is just `x[order(x)]`.

Now let us see how to use the order function for data frames. First, we create a small data set.

```
> sex <- rep(c("Male", "Female"), c(4, 4))
> ethnicity <- rep(c("White", "African American", "Latino",
+                    "Asian"), 2)
> age <- sample(1:100, 8)
> dat <- data.frame(age, sex, ethnicity)
> dat <- dat[sample(1:8, 8),] #randomly order rows
> dat
```

```
  age    sex        ethnicity
5  57 Female            White
8  93 Female            Asian
1   7   Male            White
4  65   Male            Asian
6  38 Female African American
3  27   Male           Latino
2  66   Male African American
7  72 Female           Latino
```

Okay, now we will sort the data frame based on the ordering of one field, and then the ordering of two fields:

```
> dat[order(dat$age),] #sort based on 1 variable
  age    sex        ethnicity
1   7   Male            White
3  27   Male           Latino
6  38 Female African American
5  57 Female            White
4  65   Male            Asian
2  66   Male African American
7  72 Female           Latino
8  93 Female            Asian
> dat[order(dat$sex, dat$age),] #sort based on 2 variables
  age    sex        ethnicity
6  38 Female African American
5  57 Female            White
7  72 Female           Latino
8  93 Female            Asian
1   7   Male            White
3  27   Male           Latino
4  65   Male            Asian
2  66   Male African American
```

3.4 Indexing (subsetting) data

For this section, please load the well known Oswego foodborne illness dataset:

```
> odat <- read.table("http://www.medepi.net/data/oswego.txt",
+     header = TRUE, as.is = TRUE, sep = "")
> str(odat)
'data.frame':   75 obs. of  21 variables:
 $ id                : int  2 3 4 6 7 8 9 10 14 16 ...
 $ age               : int  52 65 59 63 70 40 15 33 10 32 ...
 $ sex               : chr  "F" "M" "F" "F" ...
$ meal.time          : chr  "8:00 PM" "6:30 PM" "6:30 PM" ...
$ ill                : chr  "Y" "Y" "Y" "Y" ...
$ onset.date         : chr  "4/19" "4/19" "4/19" "4/18" ...
$ onset.time         : chr  "12:30 AM" "12:30 AM" ...
$ baked.ham          : chr  "Y" "Y" "Y" "Y" ...
...
$ vanilla.ice.cream  : chr  "Y" "Y" "Y" "Y" ...
$ chocolate.ice.cream: chr  "N" "Y" "Y" "N" ...
$ fruit.salad        : chr  "N" "N" "N" "N" ...
```

3.4.1 Indexing

Now, we will practice indexing rows from this data frame. First, we create a new data set that contains only cases. To index the rows with cases we need to generate a logical vector that is `TRUE` for every value of `odat$ill` that “is equivalent to” `"Y"`. For “is equivalent to” we use the == relational operator.

```
> cases <- odat$ill=="Y"
> cases
 [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
...
[73] FALSE FALSE FALSE
> odat.ca <- odat[cases, ]
> odat.ca[, 1:8]
   id age sex meal.time ill onset.date onset.time baked.ham
1   2  52   F   8:00 PM   Y       4/19   12:30 AM         Y
2   3  65   M   6:30 PM   Y       4/19   12:30 AM         Y
3   4  59   F   6:30 PM   Y       4/19   12:30 AM         Y
4   6  63   F   7:30 PM   Y       4/18   10:30 PM         Y
...
43 71  60   M   7:30 PM   Y       4/19    1:00 AM         N
44 72  18   F   7:30 PM   Y       4/19   12:00 AM         Y
45 74  52   M      <NA>   Y       4/19    2:15 AM         Y
46 75  45   F      <NA>   Y       4/18   11:00 PM         Y
```

It is very important to understand what we just did: we extracted the rows with cases by indexing the data frame with a logical vector.

Now, we combine relational operators with logical operators to extract rows based on multiple criteria. Let’s create a data set with female cases, age less than the median age, and consumed vanilla ice cream.

```
> fem.cases.vic <- odat$ill=="Y" & odat$sex=="F" &
+     odat$vanilla.ice.cream=="Y" & odat$age < median(odat$age)
> odat.fcv <- odat[fem.cases.vic, ]
> odat.fcv[ , c(1:6, 19)]
```

```
   id age sex meal.time ill onset.date vanilla.ice.cream
8  10  33   F   7:00 PM   Y       4/18                 Y
10 16  32   F      <NA>   Y       4/19                 Y
13 20  33   F      <NA>   Y       4/18                 Y
14 21  13   F  10:00 PM   Y       4/19                 Y
18 27  15   F  10:00 PM   Y       4/19                 Y
23 36  35   F      <NA>   Y       4/18                 Y
31 48  20   F   7:00 PM   Y       4/19                 Y
37 58  12   F  10:00 PM   Y       4/19                 Y
40 65  17   F  10:00 PM   Y       4/19                 Y
41 66   8   F      <NA>   Y       4/19                 Y
42 70  21   F      <NA>   Y       4/19                 Y
44 72  18   F   7:30 PM   Y       4/19                 Y
```

In summary, we see that indexing rows of a data frame consists of using relational operators (`<, >, <=, >=, ==, !=`) and logical operators (`&, |, !`) to generate a logical vector for indexing the appropriate rows.

3.4.2 Subsetting

Subsetting a data frame using the `subset` function is equivalent to using logical vectors to index the data frame. In general, we prefer indexing because it is generalizable to indexing any R data object. However, the `subset` function is a convenient alternative for data frames. Again, let's create data set with female cases, age $<$ median, and ate vanilla ice cream.

```
> odat.fcv <- subset(odat, subset = {ill=="Y" & sex=="F" &
+                    vanilla.ice.cream=="Y" & age < median(odat$age)},
+                    select = c(id:onset.date, vanilla.ice.cream))
> odat.fcv
   id age sex meal.time ill onset.date vanilla.ice.cream
8  10  33   F   7:00 PM   Y       4/18                 Y
10 16  32   F         .   Y       4/19                 Y
13 20  33   F         .   Y       4/18                 Y
14 21  13   F  10:00 PM   Y       4/19                 Y
18 27  15   F  10:00 PM   Y       4/19                 Y
23 36  35   F         .   Y       4/18                 Y
31 48  20   F   7:00 PM   Y       4/19                 Y
37 58  12   F  10:00 PM   Y       4/19                 Y
40 65  17   F  10:00 PM   Y       4/19                 Y
41 66   8   F         .   Y       4/19                 Y
42 70  21   F         .   Y       4/19                 Y
44 72  18   F   7:30 PM   Y       4/19                 Y
```

In the `subset` function, the first argument is the data frame object name, the second argument (also called `subset`) evaluates to a logical vector, and third argument (called `select`) specifies the fields to keep. In the second argument,

```
subset = {...}
```

the curly brackets are included for convenience to group the logical and relational operations. In the `select` argument, using the `:` operator, we can specify a range of fields to keep.

3.5 Transforming data

Transforming fields in a data frame is very common. The most common transformations include the following:

- Numerical transformation of a numeric vector
- Discretizing a numeric vector into categories or levels ("categorical variable")
- Re-coding integers that represent levels of a categorical variable

For each of these, we must decide whether the newly created vector should be a new field in the data frame, overwrite the original field in the data frame, or not be a field in the data frame (but rather a vector object in the workspace). For the examples that follow load the well known Oswego foodborne illness dataset:

```
> odat <- read.table("http://www.medepi.net/data/oswego.txt",
+     header = TRUE, as.is = TRUE, sep = "")
```

3.5.1 Numerical transformation

```
> # transform age variable centering it
> # create new field in same data frame
> odat$age
 [1] 52 65 59 63 70 40 15 33 10 32 62 36 33 13  7  3 59 15
...
[73] 17 36 14
> odat$age.centered <- odat$age - mean(odat$age)
> odat$age.centered
 [1]  15.1866667  28.1866667  22.1866667  26.1866667
...
[73] -19.8133333  -0.8133333 -22.8133333
>
> # overwrite original field in same data frame (not recommended!!!)
> # odat$age <- odat$age - mean(odat$age)
>
```

```
> # create new vector in workspace; data frame remains unchanged
> age.centered <- odat$age - mean(odat$age)
> age.centered
 [1]  15.1866667  28.1866667  22.1866667  26.1866667
...
[73] -19.8133333  -0.8133333 -22.8133333
```

For convenience, the `transform` function facilitates the transformation of numeric vectors in a data frame. The `transform` function comes in handy when we plan on transforming many fields: we do not need to specify the data frame each time we refer to a field name. For example, the following lines are are equivalent. Both add a new transformed field to the data frame.

```
odat$age.centered <- odat$age - mean(odat$age)
odat <- transform(odat, age.centered = age - mean(age))
```

3.5.2 *Creating categorical variables (factors)*

Now, reload the Oswego data set to recover the original `odat$age` field. We are going to create a new field with the following seven age categories (in years): < 1, 1 to 4, 5 to 14, 15 to 24, 25 to 44, 45 to 64, and 65+. We will demonstrate this using several methods:

3.5.2.1 Using `cut` function (preferred method)

```
> agecat <- cut(odat$age, breaks = c(0, 1, 5, 15, 25, 45,
+     65, 100))
> agecat
 [1] (45,65]  (45,65]  (45,65]  (45,65]  (65,100] (25,45]
...
[73] (15,25]  (25,45]  (5,15]
Levels: (0,1] (1,5] (5,15] (15,25] (25,45] ... (65,100]
```

Note that the `cut` function generated a factor with 7 levels for each interval. The notation `(15, 25]` means that the interval is open on the left boundary (>15) and closed on the right boundary (≤ 25). However, for age categories, it makes more sense to have age boundaries closed on the left and open on the right: `[a, b)`. To change this we set the option `right = FALSE`

```
> agecat <- cut(odat$age, breaks = c(0, 1, 5, 15, 25, 45,
+     65, 100), right = FALSE)
> agecat
 [1] [45,65)  [65,100) [45,65)  [45,65)  [65,100) [25,45)
...
```

```
[73] [15,25)  [25,45)  [5,15)
Levels: [0,1) [1,5) [5,15) [15,25) [25,45) ... [65,100)
> table(agecat)
agecat
   [0,1)  [1,5)  [5,15)  [15,25)  [25,45)  [45,65)  [65,100)
       0      1      14       13       18       20         9
```

Okay, this looks good, but we can add labels since our readers may not be familiar with open and closed interval notation $[a,b)$.

```
> agelabs <- c("<1", "1-4", "5-14", "15-24", "25-44", "45-64",
+     "65+")
> agecat <- cut(odat$age, breaks = c(0, 1, 5, 15, 25, 45,
+     65, 100), right = FALSE, labels = agelabs)
> agecat
 [1] 45-64 65+   45-64 45-64 65+   25-44 15-24
...
[71] 5-14  5-14  15-24 25-44 5-14
Levels: <1 1-4 5-14 15-24 25-44 45-64 65+
> table(agecat, case = odat$ill)
       case
agecat   N  Y
  <1     0  0
  1-4    0  1
  5-14   8  6
  15-24  5  8
  25-44  8 10
  45-64  5 15
  65+    3  6
```

3.5.2.2 Using indexing and assignment (replacement)

The `cut` function is the preferred method to create a categorical variable. However, suppose one does not know about the `cut` function. Applying basic R concepts always works!

```
> agegroup <- odat$age
> agegroup[odat$age<1] <- 1
> agegroup[odat$age>=1 & odat$age<5] <- 2
> agegroup[odat$age>=5 & odat$age<15] <- 3
> agegroup[odat$age>=15 & odat$age<25] <- 4
> agegroup[odat$age>=25 & odat$age<45] <- 5
> agegroup[odat$age>=45 & odat$age<65] <- 6
> agegroup[odat$age>=65] <- 7
> #create factor
> agelabs <-  c("<1", "1 to 4", "5 to 14", "15 to 24",
+     "25 to 44", "45 to 64", "65+")
> agegroup <- factor(agegroup, levels = 1:7, labels = agelabs)
> agegroup
 [1] 45 to 64 65+      45 to 64 45 to 64 65+      25 to 44
...
[73] 15 to 24 25 to 44 5 to 14
7 Levels: <1 1 to 4 5 to 14 15 to 24 25 to 44 ... 65+
> table(case = odat$ill, agegroup)
    agegroup
case <1 1 to 4 5 to 14 15 to 24 25 to 44 45 to 64 65+
   N  0      0       8        5        8        5   3
   Y  0      1       6        8       10       15   6
```

In these previous examples, notice that `agegroup` is a factor object that is not a field in the `odat` data frame.

3.5.3 *"Re-coding" levels of a categorical variable*

In the previous example the categorical variable was a numeric vector (1, 2, 3, 4, 5, 6, 7) that was converted to a factor and provided labels ("<1", "1 to 4", "5 to 14", . . .). In fact, categorical variables are often represented by integers (for example, 0 = no, 1 = yes; or 0 = non-case, 1 = case) and provided labels. Often, ASCII text data files are integer codes that require a data dictionary to convert these integers into categorical variables in a statistical package. In R, keeping track of integer codes for categorical variables is unnecessary. Therefore, re-coding the underlying integer codes is also unnecessary; however, if we feel the need to do so, here's how.

```
> # Create categorical variable
> ethlabs <- c("White", "Black", "Latino", "Asian")
> ethnicity <- sample(ethlabs, 100, replace = T)
> ethnicity <- factor(ethnicity, levels = ethlabs)
> ethnicity
  [1] Black  Asian  Latino White  Black  Asian  White  Black
...
 [97] Black  Black  Asian  Latino
Levels: White Black Latino Asian
```

The `levels` option allowed us to determine the display order, and the first level becomes the reference level in statistical models. To display the underlying numeric code use `unclass` function which preserves the levels attribute.[9]

```
> x <- unclass(ethnicity)
> x
  [1] 2 4 3 1 2 4 1 2 1 3 4 3 2 1 3 2 1 1 1 3 1 2 2 1 3 4 2 3
```

---

[9] The `as.integer` function also works but does not preserve the levels attribute.

```
...
 [85] 2 2 3 3 3 3 1 3 4 4 1 1 2 2 4 3
attr(,"levels")
[1] "White"  "Black"  "Latino" "Asian"
```

To recover the original factor,

```
> factor(x,labels=levels(x))
  [1] Black  Asian  Latino White  Black  Asian  White
...
 [92] Latino Asian  Asian  White  White  Black  Black
 [99] Asian  Latino
Levels: White Black Latino Asian
```

Although one can extract the integer code, why would one need to do so? One is tempted to use the integer codes as a way to share data sets. However, we recommend not using the integer codes, but rather just provide the data in its native format.[10] This way, the raw data is more interpretable and eliminates the intermediate step of needing to label the integer code. Also, if the data dictionary is lost or not provided, the raw data is still interpretable.

In R, we can re-label the levels using the `levels` function and assigning to it a character vector of new labels. Make sure the order of the new labels corresponds to order of the factor levels.

```
> levels(ethnicity2) <- c("Caucasion", "African American",
+     "Hispanic", "Asian")
> table(ethnicity2)
ethnicity2
       Caucasion   African American   Hispanic   Asian
              23                 31         28      18
```

In R, we can re-order and re-label at the same time using the `levels` function and assigning to it a list.

```
> table(ethnicity)
ethnicity
 White  Black Latino  Asian
    23     31     28     18
> ethnicity3 <- ethnicity
> levels(ethnicity3) <- list(Hispanic = "Latino", Asian = "Asian",
+     Caucasion = "White", "African American" = "Black")
> table(ethnicity3)
ethnicity3
        Hispanic   Asian   Caucasion   African American
              28      18          23                 31
```

The `list` function is necessary to assure the re-ordering. To re-order without re-labeling just do the following:

---

[10] For example, `http://www.medepi.net/data/oswego.txt`

```
> table(ethnicity)
ethnicity
 White  Black Latino  Asian
    23     31     28     18
> ethnicity4 <- ethnicity
> levels(ethnicity4) <- list(Latino = "Latino", Asian = "Asian",
+     White = "White", Black = "Black")
> table(ethnicity4)
ethnicity4
Latino  Asian  White  Black
    28     18     23     31
```

In R, we can sort the factor levels by using the `factor` function in one of two ways:

```
> table(ethnicity)
ethnicity
 White  Black Latino  Asian
    23     31     28     18
> ethnicity5a <- factor(ethnicity, sort(levels(ethnicity)))
> table(ethnicity5a)
ethnicity5a
 Asian  Black Latino  White
    18     31     28     23
> ethnicity5b <- factor(as.character(ethnicity))
> table(ethnicity5b)
ethnicity5b
 Asian  Black Latino  White
    18     31     28     23
```

In the first example, we assigned to the `levels` argument the sorted level names. In the second example, we started from scratch by coercing the original factor into a character vector which is then ordered alphabetically by default.

3.5.3.1 Setting factor reference level

The first level of a factor is the reference level for some statistical models (e.g., logistic regression). To set a different reference level use the `relevel` function.

```
> levels(ethnicity)
[1] "White"  "Black"  "Latino" "Asian"
> ethnicity6 <- relevel(ethnicity, ref = "Asian")
> levels(ethnicity6)
[1] "Asian"  "White"  "Black"  "Latino"
```

Table 3.3 Categorical variable represented as a factor or a set of dummy variables

Factor	Dummy variables		
Ethnicity	Asian	Black	Latino
White	0	0	0
Asian	1	0	0
Black	0	1	0
Latino	0	0	1

As we can see, there is tremendous flexibility in dealing with factors without the need to "re-code" categorical variables. This approach facilitates reviewing our work and minimizes errors.

3.5.4 *Use factors instead of dummy variables*

A nonordered factor (nominal categorical variable) with k levels can also be represented with $k-1$ dummy variables. For example, the `ethnicity` factor has four levels: white, Asian, black, and Latino. Ethnicity can also be represented using 3 dichotomous variables, each coded 0 or 1. For example, using white as the reference group, the dummy variables would be `asian`, `black`, and `latino` (see Table 3.3). The values of those three dummy variables (0 or 1) are sufficient to represents one of four possible ethnic categories. Dummy variables can be used in statistical models. However, in R, it is unnecessary to create dummy variables, just create a factor with the desired number of levels and set the reference level.

3.5.5 *Conditionally transforming the elements of a vector*

We can conditionally transform the elements of a vector using the `ifelse` function. This function works as follows: `ifelse(`*`test, if test = TRUE do this, else do this`*`)`.

```
> x <- sample(c("M", "F"), 10, replace = T); x
 [1] "M" "F" "M" "F" "M" "F" "M" "F" "M" "F"
> y <- ifelse(x=="M", "Male", "Female")
> y
 [1] "Male"   "Female" "Male"   "Female" "Male"   "Female"
 [7] "Male"   "Female" "Male"   "Female"
```

Table 3.4 R functions for transforming variables in data frames

Function	Description	Try these examples

`<-`	Transforming a vector and assigning it to a new data frame variable name	`dat <- data.frame(id=1:3, x=c(0.5,1,2));`
		`dat`
		`dat$logx <- log(x) #creates new field`
		`dat`

transform	Transform one or more variables from a data frame	`dat <- data.frame(id=1:3, x=c(0.5,1,2));`
		`dat`
		`dat <- transform(dat, logx = log(x))`
		`dat`

cut	Creates a factor by dividing the range of a numeric vector into intervals	
levels	Gives access to the levels attribute of a factor	
relevel	Set the reference level for a factor	`sex2 <- relevel(sex, ref = "Women")` `table(sex2)`

ifelse	Conditionally operate on elements of a vector based on a test	`age <- sample(1:100, 1000, replace =`
		`TRUE)`
		`agecat <- ifelse(age<=50, "<=50", ">50")`
		`table(agecat)`

cut

```
age <- sample(1:100, 500, replace = TRUE)
# cut into 2 intervals agecut <- cut(age,
2, right = FALSE)
table(agecut)
#cut using specified intervals agecut2 <-
cut(age, c(0, 50 100),
  right = FALSE, include.lowest = TRUE)
table(agecut2)
```

levels

```
sex <- sample(c("M","F","T"),500,
replace=T)
sex <- factor(sex)
table(sex)
# relabel each level; use same order
levels(sex) <- c("Female", "Male",
  "Transgender")
table(sex)
# relabel/recombine
levels(sex) <- c("Female", "Male",
"Male")
table(sex)
# reorder and/or relabel
levels(sex) <- list ("Men" = "Male",
  "Women" = "Female")
table(sex)
```

3.6 Merging data

In general, R's strength is not data management but rather data analysis. Because R can access and operate on multiple objects in the workspace it is generally not necessary to merge data objects into one data object in order to conduct analyses. On occasion, it may be necessary to merge two data frames into one data frames

Data frames that contain data on individual subjects are generally of two types: (1) each row contains data collected on one and only one individual, or (2) multiple rows contain repeated measurements on individuals. The latter approach is more efficient at storing data. For example, here are two approaches to collecting multiple telephone numbers for two individuals.

```
> tab1
            name   wphone   fphone   mphone
1   Tomas Aragon 643-4935 643-2926 847-9139
2 Wayne Enanoria 643-4934     <NA>     <NA>
>
> tab2
            name telphone teletype
1   Tomas Aragon 643-4935     Work
2   Tomas Aragon 643-2926      Fax
3   Tomas Aragon 847-9139   Mobile
4 Wayne Enanoria 643-4934     Work
```

The first approach is represented by `tab1`, and the second approach by `tab2`.[11] Data is more efficiently stored in `tab2`, and adding new types of telephone numbers only requires assigning a new value (e.g., Pager) to the `teletype` field.

```
> tab2
            name telphone teletype
1   Tomas Aragon 643-4935     Work
2   Tomas Aragon 643-2926      Fax
3   Tomas Aragon 847-9139   Mobile
4 Wayne Enanoria 643-4934     Work
5   Tomas Aragon 719-1234    Pager
```

In both these data frames, an indexing field identifies an unique individual that is associated with each row. In this case, the `name` column is the indexing field for both data frames.

Now, let's look at an example of two related data frames that are linked by an indexing field. The first data frame contains telephone numbers for 5 employees and `fname` is the indexing field. The second data frame contains email addresses for 3 employees and `name` is the indexing field.

```
> phone
  fname phonenum phonetype
1 Tomas 643-4935      work
2 Tomas 847-9139    mobile
3 Tomas 643-4926       fax
4 Chris 643-3932      work
5 Chris 643-4926       fax
```

---

[11] This approach is the basis for designing and implementing relational databases. A relational database consists of multiple tables linked by an indexing field.

```
6  Wayne 643-4934      work
7  Wayne 643-4926       fax
8    Ray 643-4933      work
9    Ray 643-4926       fax
10 Diana 643-3931      work
> email
   name                  mail mailtype
1 Tomas   aragon@berkeley.edu     Work
2 Tomas     aragon@medepi.net Personal
3 Wayne enanoria@berkeley.edu     Work
4 Wayne  enanoria@idready.org     Work
5 Chris  csiador@berkeley.edu     Work
6 Chris     csiador@yahoo.com Personal
```

To merge these two data frames use the `merge` function.

```
> dat <- merge(email, phone, by.x="name", by.y="fname")
> dat
    name                  mail mailtype phonenum phonetype
1  Chris  csiador@berkeley.edu     Work 643-3932      work
2  Chris     csiador@yahoo.com Personal 643-3932      work
3  Chris  csiador@berkeley.edu     Work 643-4926       fax
4  Chris     csiador@yahoo.com Personal 643-4926       fax
5  Tomas   aragon@berkeley.edu     Work 643-4935      work
6  Tomas     aragon@medepi.net Personal 643-4935      work
7  Tomas   aragon@berkeley.edu     Work 847-9139    mobile
8  Tomas     aragon@medepi.net Personal 847-9139    mobile
9  Tomas   aragon@berkeley.edu     Work 643-4926       fax
10 Tomas     aragon@medepi.net Personal 643-4926       fax
11 Wayne enanoria@berkeley.edu     Work 643-4934      work
12 Wayne  enanoria@idready.org     Work 643-4934      work
13 Wayne enanoria@berkeley.edu     Work 643-4926       fax
14 Wayne  enanoria@idready.org     Work 643-4926       fax
> dat <- merge(phone, email, by.x="fname", by.y="name")
> dat
   fname phonenum phonetype                  mail mailtype
1  Chris 643-3932      work  csiador@berkeley.edu     Work
2  Chris 643-4926       fax  csiador@berkeley.edu     Work
3  Chris 643-3932      work    cvsiador@yahoo.com Personal
4  Chris 643-4926       fax    cvsiador@yahoo.com Personal
5  Tomas 643-4935      work   aragon@berkeley.edu     Work
6  Tomas 847-9139    mobile   aragon@berkeley.edu     Work
7  Tomas 643-4926       fax   aragon@berkeley.edu     Work
8  Tomas 643-4935      work     aragon@medepi.net Personal
9  Tomas 847-9139    mobile     aragon@medepi.net Personal
10 Tomas 643-4926       fax     aragon@medepi.net Personal
```

```
11 Wayne 643-4934      work enanoria@berkeley.edu     Work
12 Wayne 643-4926       fax enanoria@berkeley.edu     Work
13 Wayne 643-4934      work  enanoria@idready.org     Work
14 Wayne 643-4926       fax  enanoria@idready.org     Work
```

The `by.x` and `by.y` options identify the indexing fields. By default, R selects the rows from the two data frames that is based on the *intersection* of the indexing fields (`by.x`, `by.y`). To merge the *union* of the indexing fields, set `all=TRUE`:

```
> dat <- merge(phone, email, by.x="fname", by.y="name",
+     all=TRUE)
> dat
   fname phonenum phonetype                  mail mailtype
1  Chris 643-3932      work  csiador@berkeley.edu     Work
2  Chris 643-4926       fax  csiador@berkeley.edu     Work
3  Chris 643-3932      work     csiador@yahoo.com Personal
4  Chris 643-4926       fax     csiador@yahoo.com Personal
5  Diana 643-3931      work                  <NA>     <NA>
6    Ray 643-4933      work                  <NA>     <NA>
7    Ray 643-4926       fax                  <NA>     <NA>
8  Tomas 643-4935      work   aragon@berkeley.edu     Work
9  Tomas 847-9139    mobile   aragon@berkeley.edu     Work
10 Tomas 643-4926       fax   aragon@berkeley.edu     Work
11 Tomas 643-4935      work     aragon@medepi.net Personal
12 Tomas 847-9139    mobile     aragon@medepi.net Personal
13 Tomas 643-4926       fax     aragon@medepi.net Personal
14 Wayne 643-4934      work enanoria@berkeley.edu     Work
15 Wayne 643-4926       fax enanoria@berkeley.edu     Work
16 Wayne 643-4934      work  enanoria@idready.org     Work
17 Wayne 643-4926       fax  enanoria@idready.org     Work
```

To “reshape” tabular data look up and study the `reshape` and `stack` functions.

3.7 Executing commands from, and directing output to, a file

3.7.1 The `source` function

We use the `source` function to execute R commands are contained in an ASCII text file. For example, consider the contents this source file (chap03.R):

```
i <- 1:5
x <- outer(i, i, "*")
show(x)
```

Here we run `source` from the R command prompt:

```
> source("/home/tja/Documents/wip/epir/r/chap03.R")
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    2    4    6    8   10
[3,]    3    6    9   12   15
[4,]    4    8   12   16   20
[5,]    5   10   15   20   25
```

Nothing is printed to the console unless we explicitly use the the `show` (or `print`) function. This enables us to view only the results we want to review.

An alternative approach is to print everything to the console as if the R commands were being enter directly at the command prompt. For this we do not need to use the `show` function in the source file; however, we must set the `echo` option to TRUE in the `source` function. Here is the edited source file (chap03.R)

```
i <- 1:5
x <- outer(i, i, "*")
x
```

Here we run `source` from the R command prompt:

```
> source("/home/tja/Documents/wip/epir/r/chap03.R", echo=TRUE)
> i <- 1:5
> x <- outer(i, i, "*")
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    2    4    6    8   10
[3,]    3    6    9   12   15
[4,]    4    8   12   16   20
[5,]    5   10   15   20   25
```

3.7.2 The `sink` *and* `capture.output` *functions*

We can add code to the source file to "sink" selected results to an output file using the `sink` or `capture.output` functions. Consider our edited source file:

```
i <- 1:5
x <- outer(i, i, "*")
sink("/home/tja/Documents/wip/epir/r/chap03.log")
cat("Here are the results of the outer function", fill=TRUE)
show(x)
sink()
```

Here we run `source` from the R command prompt:

```
> source("/home/tja/Documents/wip/epir/r/chap03.R")
>
```

Nothing was printed to the console because `sink` sent it to the output file (chap03.log). Here are the contents of chap03.log:

```
Here are the results of the outer function
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    2    4    6    8   10
[3,]    3    6    9   12   15
[4,]    4    8   12   16   20
[5,]    5   10   15   20   25
```

The first `sink` opened a connection and created the output file (chap03.log). The `cat` and `show` functions printed results to output file. The second `sink` close the connection.

Alternatively, as before, if we use the `echo = TRUE` option in the `source` function, everything is either printed to the console or output file. The `sink` connection determines what is printed to the output file. Here is the edited source file (chap03.R):

```
i <- 1:5
x <- outer(i, i, "*")
sink("/home/tja/Documents/wip/epir/r/chap03.log")
# Here are the results of the outer function
x
sink()
```

Here we run `source` from the R command prompt:

```
> source("/home/tja/Documents/wip/epir/r/chap03.R", echo=T)
> i <- 1:5
> x <- outer(i, i, "*")
> sink("/home/tja/Documents/wip/epir/r/chap03.log")
>
```

Nothing was printed to the console after the first `sink` because it was printed to output file (chap03.Rout). Here are the contents of chap03.Rout:

```
> # Here are the results of the outer function
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    2    4    6    8   10
[3,]    3    6    9   12   15
[4,]    4    8   12   16   20
[5,]    5   10   15   20   25
> sink()
```

The `sink` and `capture.output` functions accomplish the same task: sending results to an output file. The `sink` function works in pairs: open and closing the connection to the output file. In contrast, `capture.output` function appears once and only prints the last object to the output file. Here is the edited source file (chap03.R) using `capture.output` instead of `sink`:

```
i <- 1:5
x <- outer(i, i, "*")
capture.output(
{
# Here are the results of the outer function
x
}, file = "/home/tja/Documents/wip/epir/r/chap03.log")
```

Here we run `source` from the R command prompt:

```
> source("/home/tja/Documents/wip/epir/r/chap03.R", echo=TRUE)
> i <- 1:5
> x <- outer(i, i, "*")
> capture.output(
+ {
+ # Here are the results of the outer function
+ x
+ }, file = "/home/tja/Documents/wip/epir/r/chap03.Rout")
```

And, Here are the contents of chap03.log:

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    2    4    6    8   10
[3,]    3    6    9   12   15
[4,]    4    8   12   16   20
[5,]    5   10   15   20   25
```

Even though the `capture.output` function can run several R commands (between curly brackets), only the final object (`x`) was printed to the output file. This would be true even if the `echo` option was not set to TRUE in the `source` function.

In summary, the `source` function runs R commands contained in an external source file. Setting the `echo` option to TRUE prints commands and results to the console as if the commands were directly entered at the command prompt. The `sink` and `capture.output` functions print results to an output file.

3.8 Working with missing and “not available” values

In R, missing values are represented by NA, but not all NAs represent missing values — some are just “not available.” NAs can appear in any data object. The NA can represent a true missing value, or it can result from an operation to which a value is "not available." Here are three vectors that contain true missing values.

```
x <- c(2, 4, NA, 5); x
y <- c("M", NA, "M", "F"); y
y <- c(F, NA, F, T); z
```

However, elementary numerical operations on objects that contain NA return a single NA ("not available"). In this instance, R is saying "An answer is 'not available' until you tell R what to do with the NAs in the data object." To remove NAs for a calculation specify the `na.rm` ("NA remove") option.

```
> sum(x) # answer not available
[1] NA
> mean(x) # answer not available
[1] NA
> sum(x, na.rm = TRUE) # better
[1] 11
> mean(x, na.rm = TRUE) # better
[1] 3.666667
```

Here are more examples where NA means an answer is not available:

```
> # Inappropriate coercion
> as.numeric(c("4", "six"))
[1]  4 NA
Warning message:
NAs introduced by coercion
> # Indexing out of range
> c(1:5)[7]
[1] NA
> df <- data.frame(v1 = 1:3, v2 = 4:6)
> df[4, ] # There is no 4th row
   v1 v2
NA NA NA
> # Indexing with non-existing name
> df["4th",]
   v1 v2
NA NA NA
> # Operations with NAs
> NA + 8
[1] NA
> # Variance of a single number
> var(55)
[1] NA
```

In general, these NAs indicate a problem that we must or should be addressed.

3.8.1 Testing, indexing, replacing, and recoding

Regardless of the source of the NAs — missing or not available values — using the `is.na` function, we can generate a logical vector to identify which positions contain or do not contain NAs. This logical vector can be used index the original or another vector. Values that can be indexed can be replaced

```
> x <- c(10, NA, 33, NA, 57)
> y <- c(NA, 24, NA, 47, NA)
> is.na(x) #generate logical vector
[1] FALSE  TRUE FALSE  TRUE FALSE
> which(is.na(x)) #which positions are NA
[1] 2 4
> x[!is.na(x)] #index original vector
[1] 10 33 57
> y[is.na(x)]  #index other vector
[1] 24 47
> x[is.na(x)] <- 999 #replacement
> x
[1]  10 999  33 999  57
```

For a vector, recoding missing values to NA is accomplished using replacement.

```
> x <- c(1, -99, 3, -88, 5)
> x[x==-99 | x==-88] <- NA
> x
[1]  1 NA  3 NA  5
```

For a matrix, we can recode missing values to NA by using replacement one column at a time, or globablly like a vector.

```
> m <- m2 <- matrix (c(1, -99, 3, 4, -88, 5), 2, 3)
> m
     [,1] [,2] [,3]
[1,]    1    3  -88
[2,]  -99    4    5
> # Replacement one column at a time
> m[m[,1]==-99, 1] <- NA
> m
     [,1] [,2] [,3]
[1,]    1    3  -88
[2,]   NA    4    5
> m[m[,3]==-88, 3] <- NA
> m
     [,1] [,2] [,3]
[1,]    1    3   NA
[2,]   NA    4    5
```

```
> # Global replacement
> m2[m2==-99 | m2==-88] <- NA
> m2
     [,1] [,2] [,3]
[1,]    1    3   NA
[2,]   NA    4    5
```

Likewise, for a data frame, we can recode missing values to NA by using replacement one field at a time, or globablly like a vector.

```
> fname <- c("Tom", "Unknown", "Jerry")
> age <- c(56, 34, -999)
> z1 <- z2 <- data.frame(fname, age)
> z1
    fname  age
1     Tom   56
2 Unknown   34
3   Jerry -999
> # Replacement one column at a time
> z1$fname[z1$fname=="Unknown"] <- NA
> z1$age[z1$age==-999] <- NA
> z1
  fname age
1   Tom  56
2  <NA>  34
3 Jerry  NA
> # Global replacement
> z2[z2=="Unknown" | z2==-999] <- NA
> z2
  fname age
1   Tom  56
2  <NA>  34
3 Jerry  NA
```

3.8.2 *Importing missing values with the* `read.table` *function*

When importing ASCII data files using the `read.table` function, use the `na.strings` option to specify what characters are to be converted to NA. The default setting is `na.strings="NA"`. Blank fields are also considered to be missing values in logical, integer, numeric, and complex fields. For example, suppose the data set contains 999, 888, and `.` to represent missing values, then import the data like this:

```
mydat <- read.table("dataset.txt", na.strings = c(999, 888, "."))
```

If a number, say 999, represents a missing value in one field but a valid value in another field, then import the data using the `as.is=TRUE` option. Then replace the missing values in the data frame one field at a time, and convert categorical field to factors.

3.8.3 *Working with NA values in data frames and factors*

There are several function for working with NA values in data frames. First, the `na.fail` function tests whether a data frame contains any NA values, returning an error message if it contains NAs.

```
> name <- c("Jose", "Ana", "Roberto", "Isabel", "Jen")
> gender <- c("M", "F", "M", NA, "F")
> age <- c(34, NA, 22, 18, 34)
> df <- data.frame(name, gender, age)
> df
     name gender age
1    Jose      M  34
2     Ana      F  NA
3 Roberto      M  22
4  Isabel   <NA>  18
5     Jen      F  34
> na.fail(df) # NAs in data frame
Error in na.fail.default(df) : missing values in object
> na.fail(df[c(1, 3, 5),]) # no NAs in data frame
     name gender age
1    Jose      M  34
3 Roberto      M  22
5     Jen      F  34
```

Both `na.omit` and `na.exclude` remove observations for any field that contain NAs. `na.exclude` differs from `na.omit` only in the class of the “na.action” attribute of the result, which is “exclude” (see help for details).

```
> na.omit(df)
     name gender age
1    Jose      M  34
3 Roberto      M  22
5     Jen      F  34
> na.exclude(df)
     name gender age
1    Jose      M  34
3 Roberto      M  22
5     Jen      F  34
```

The `complete.cases` function returns a logical vector for observations that are “complete” (i.e., do not contain NAs).

```
> complete.cases(df)
[1]  TRUE FALSE  TRUE FALSE  TRUE
> df[complete.cases(df),] # equivalent to na.omit
     name gender age
1    Jose      M  34
3 Roberto      M  22
5     Jen      F  34
```

3.8.3.1 NA values in factors

By default, factor levels do not include NA. To include NA as a factor level, use the `factor` function, setting the `exclude` option to NULL. Including NA as a factor level enables counting and displaying the number of NAs in tables, and analyzing NA values in statistical models.

```
> df$gender
[1] M    F    M    <NA> F
Levels: F M
> xtabs(~gender, data = df)
gender
F M
2 2
> df$gender.na <- factor(df$gender, exclude = NULL)
> xtabs(~gender.na, data = df)
gender.na
   F    M <NA>
   2    2    1
```

3.8.3.2 Indexing data frames that contain NAs

Using the original data frame (that can contain NAs), we can index sujects with ages less than 25.

```
> df$age              # age field
[1] 34 NA 22 18 34
> df[df$age<25, ]     # index ages < 25
      name gender age
NA    <NA>   <NA>  NA
3  Roberto      M  22
4   Isabel   <NA>  18
```

The row that corresponds to the age that is missing (NA) has been converted to NAs ("not available") by R. To remove this uninformative row we use the `is.na` function.

```
> df[df$age<25 & !is.na(df$age), ]
     name gender age
3 Roberto      M  22
4  Isabel   <NA>  18
```

This differs from the `na.omit`, `na.exclude`, and `complete.cases` functions that remove all missing values from the data frame first.

3.8.4 Viewing number of missing values in tables

By default, NAs are not tabulated in tables produced by the `table` and `xtabs` functions. The `table` function can tabulate character vectors and factors. The `xtabs` function only works with fields in a data frame. To tabulate NAs in character vectors using the `table` function, set the `exclude` function to NULL in the `table` function.

```
> df$gender.chr <- as.character(df$gender)
> df$gender.chr
[1] "M" "F" "M" NA  "F"
> table(df$gender.chr)

F M
2 2
> table(df$gender.chr, exclude = NULL)

   F    M <NA>
   2    2    1
```

However, this will not work with factors: we must change the factor levels first.

```
> table(df$gender) #does not tabulate NAs

F M
2 2
> table(df$gender, exclude = NULL) #does not work

F M
2 2
> df$gender.na <- factor(df$gender, exclude = NULL) #works
> table(df$gender.na)

   F    M <NA>
   2    2    1
```

Finally, whereas the `exclude` option works on character vectors tabulated with `table` function, it does not work on character vectors or factors tabulated with the `xtabs` function. In a data frame, we must convert the character vector to a factor (setting the `exclude` option to NULL), then the `xtabs` functions tabulates the NA values.

```
> xtabs(~gender, data=df, exclude=NULL) # does not work
gender
F M
2 2
> xtabs(~gender.chr, data=df, exclude=NULL) # still does not work
gender.chr
F M
2 2
> df$gender.na <- factor(df$gender, exclude = NULL) #works
> xtabs(~gender.na, data = df)
gender.na
   F    M <NA>
   2    2    1
```

3.8.5 Setting default NA behaviors in statistical models

Statistical models, for example the `glm` function for generalized linear models, have default NA behaviors that can be reset locally using the `na.action` option in the `glm` function, or reset globally using the `na.action` option setting in the `options` function.

```
> options("na.action")          # display global setting
$na.action
[1] "na.omit"

> options(na.action="na.fail") # reset global setting
> options("na.action")
$na.action
[1] "na.fail"
```

By default, `na.action` is set to "na.omit" in the `options` function. Globally (inside the `options` function) or locally (inside a statistical function), `na.action` can be set to the following:

- "na.fail"
- "na.omit"
- "na.exclude"
- "na.pass"

With "na.fail", a function that calls `na.action` will return an error if the data object contains NAs. Both "na.omit" and "na.exclude" will remove row observations from a data frame that contain NAs.[12] With `na.pass`, the data object is returned unchanged.

3.8.6 *Working with finite, infinite, and NaN numbers*

In R, some numerical operations result in negative infinity, positive infinity, or an indeterminate value (NAN for "not a number"). To assess whether values are finite or infinite, use the `is.finite` or `is.infinite` functions, respectively. To assess where a value is NAN, use the `is.nan` function. While `is.na` can identify NANs, `is.nan` cannot identify NAs.

```
> x <- c(-2:2)/c(2, 0, 0, 0, 2)
> x
[1]   -1 -Inf  NaN  Inf    1
> is.infinite(x)
[1] FALSE  TRUE FALSE  TRUE FALSE
> x[is.infinite(x)]
[1] -Inf  Inf
> is.finite(x)
[1]  TRUE FALSE FALSE FALSE  TRUE
> x[is.finite(x)]
[1] -1  1
> is.nan(x)
[1] FALSE FALSE  TRUE FALSE FALSE
> x[is.nan(x)]
[1] NaN
> is.na(x) # does index NAN
[1] FALSE FALSE  TRUE FALSE FALSE
> x[is.na(x)]
[1] NaN
> x[is.nan(x)] <- NA
> x
[1]   -1 -Inf   NA  Inf    1
> is.nan(x) # does not index NA
[1] FALSE FALSE FALSE FALSE FALSE
```

---

[12] If `na.omit` removes cases, the row numbers of the cases form the "na.action" attribute of the result, of class "omit". `na.exclude` differs from `na.omit` only in the class of the "na.action" attribute of the result, which is "exclude". See help for more details.

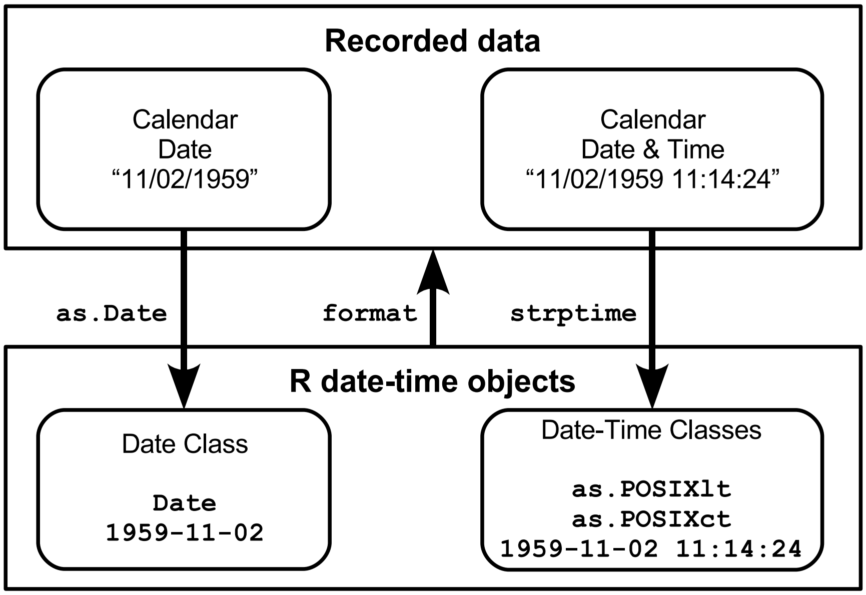

Recorded data
Calendar
Date
"11/02/1959"
Calendar
Date & Time
"11/02/1959 11:14:24"
as.Date
format
strptime
R date-time objects
Date Class
Date
1959-11-02
Date-Time Classes
as.POSIXlt
as.POSIXct
1959-11-02 11:14:24

Fig. 3.4 Displayed are functions to convert calendar date and time data into R date-time classes (`as.Date`, `strptime`, `as.POSIXlt`, `as.POSIXct`), and the `format` function converts date-time objects into character dates, days, weeks, months, times, etc.

3.9 Working with dates and times

There are 60 seconds in 1 minute, 60 minutes in 1 hour, 24 hours in 1 day, 7 days in 1 week, and 365 days in 1 year (except every 4th year we have a leap year with 366 days). Although this seems straightforward, doing numerical calculations with these time measures is not. Fortunately, computers make this much easier. Functions to deal with dates are available in the `base`, `chron`, and `survival` packages.

Summarized in Figure 3.4 is the relationship between recorded data (calendar dates and times) and their representation in R as date-time class objects (`Date`, `POSIXlt`, `POSIXct`). The `as.Date` function converts a calendar date into a Date class object. The `strptime` function converts a calendar date and time into a date-time class object (POSIXlt, POSIXct). The `as.POSIXlt` and `as.POSIXct` functions convert date-time class objects into `POSIXlt` and `POSIXct`, respectively.

The `format` function converts date-time objects into human legible character data such as dates, days, weeks, months, times, etc. These functions are discussed in more detail in the paragraphs that follow.

3.9.1 Date functions in the `base` package

3.9.1.1 The `as.Date` function

Let's start with simple date calculations. The `as.Date` function in R converts calendar dates (e.g., 11/2/1949) into a `Date` objects—a numeric vector of class `Date`. The numeric information is the number of days since January 1, 1970—also called Julian dates. However, because calendar date data can come in a variety of formats, we need to specify the format so that `as.Date` does the correct conversion. Study the following analysis carefully.

```
> bdays <- c("11/2/1959", "1/1/1970")
> bdays
[1] "11/2/1959" "1/1/1970"
> #convert to Julian dates
> bdays.julian <- as.Date(bdays, format = "%m/%d/%Y")
> bdays.julian
[1] "1959-11-02" "1970-01-01"
```

Although this looks like a character vectors, it is not: it is class "Date" and mode "numeric".

```
> #display Julian dates
> as.numeric(bdays.julian)
[1] -3713     0
> #calculate age as of today's date
> date.today <- Sys.Date()
> date.today
[1] "2005-09-25"
> age <- (date.today - bdays.julian)/365.25
> age
Time differences of 45.89733, 35.73169 days
> #the display of 'days' is not correct
> #truncate number to get "age"
> age2 <- trunc(as.numeric(age))
> age2
[1] 45 35
> #create date frame
> bd <- data.frame(Birthday = bdays, Standard = bdays.julian,
+     Julian = as.numeric(bdays.julian), Age = age2)
> bd
   Birthday   Standard Julian Age
1 11/2/1959 1959-11-02  -3713  45
2  1/1/1970 1970-01-01      0  35
```

To summarize, `as.Date` converted the character vector of calendar dates into Julian dates (days since 1970-01-01) are displayed in a standard format (yyyy-mm-dd). The Julian dates can be used in numerical calculations. To see the Julian dates use `as.numeric` or `julian` function. Because the calendar dates to be converted can come in a diversity of formats (e.g., November 2, 1959; 11-02-59; 11-02-1959; 02Nov59), one must specify the `format` option in `as.Date`. Below are selected format options; for a complete list see `help(strptime)`.

```
"%a" Abbreviated weekday name.
"%A" Full weekday name.
"%b" Abbreviated month name.
"%B" Full month name.
"%d" Day of the month as decimal number (01-31)
"%j" Day of year as decimal number (001-366).
"%m" Month as decimal number (01-12).
"%U" Week of the year as decimal number (00-53) using the
     first Sunday as day 1 of week 1.
"%w" Weekday as decimal number (0-6, Sunday is 0).
"%W" Week of the year as decimal number (00-53) using the
     first Monday as day 1 of week 1.
"%y" Year without century (00-99). If you use this on input,
     which century you get is system-specific. So don't!
     Often values up to 69 (or 68) are prefixed by 20 and
     70-99 by 19.
"%Y" Year with century.
```

Here are some examples of converting dates with different formats:

```
> as.Date("November 2, 1959", format = "%B %d, %Y")
[1] "1959-11-02"
> as.Date("11/2/1959", format = "%m/%d/%Y")
[1] "1959-11-02"
> #caution using 2-digit year
> as.Date("11/2/59", format = "%m/%d/%y")
[1] "2059-11-02"
> as.Date("02Nov1959", format = "%d%b%Y")
[1] "1959-11-02"
> #caution using 2-digit year
> as.Date("02Nov59", format = "%d%b%y")
[1] "2059-11-02"
> #standard format does not require format option
> as.Date("1959-11-02")
[1] "1959-11-02"
```

Notice how Julian dates can be used like any integer:

```
> as.Date("2004-01-15"):as.Date("2004-01-23")
[1] 12432 12433 12434 12435 12436 12437 12438 12439 12440
> seq(as.Date("2004-01-15"), as.Date("2004-01-18"), by = 1)
[1] "2004-01-15" "2004-01-16" "2004-01-17" "2004-01-18"
```

3.9.1.2 The `weekdays`, `months`, `quarters`, `julian` functions

Use the `weekdays`, `months`, `quarters`, or `julian` functions to extract information from `Date` and other date-time objects in R.

```
> mydates <- c("2004-01-15","2004-04-15","2004-10-15")
> mydates <- as.Date(mydates)
> weekdays(mydates)
[1] "Thursday" "Thursday" "Friday"
> months(mydates)
[1] "January" "April"   "October"
> quarters(mydates)
[1] "Q1" "Q2" "Q4"
> julian(mydates)
[1] 12432 12523 12706
attr(,"origin")
[1] "1970-01-01"
```

3.9.1.3 The `strptime` function

So far we have worked with calendar dates; however, we also need to be able to work with times of the day. Whereas `as.Date` only works with calendar dates, the `strptime` function will accept data in the form of calendar dates and times of the day (HH:MM:SS, where H = hour, M = minutes, S = seconds). For example, let's look at the Oswego foodborne ill outbreak that occurred in 1940. The source of the outbreak was attributed to the church supper that was served on April 18, 1940. The food was available for consumption from 6 pm to 11 pm. The onset of symptoms occurred on April 18th and 19th. The meal consumption times and the illness onset times were recorded.

```
> odat <- read.table("http://www.medepi.net/data/oswego.txt",
+     sep = "", header = TRUE, as.is = TRUE)
> str(odat)
`data.frame':   75 obs. of  21 variables:
 $ id                 : int  2 3 4 6 7 8 9 10 14 16 ...
 $ age                : int  52 65 59 63 70 40 15 33 10 32 ...
 $ sex                : chr  "F" "M" "F" "F" ...
 $ meal.time          : chr  "8:00 PM" "6:30 PM" "7:30 PM" ...
 $ ill                : chr  "Y" "Y" "Y" "Y" ...
 $ onset.date         : chr  "4/19" "4/19" "4/19" "4/18" ...
 $ onset.time         : chr  "12:30 AM" "10:30 PM" ...
 ...
 $ vanilla.ice.cream  : chr  "Y" "Y" "Y" "Y" ...
 $ chocolate.ice.cream: chr  "N" "Y" "Y" "N" ...
 $ fruit.salad        : chr  "N" "N" "N" "N" ...
```

To calculate the incubation period, for ill individuals, we need to subtract the meal consumption times (occurring on 4/18) from the illness onset times (occurring on 4/18 and 4/19). Therefore, we need two date-time objects to do this arithmetic. First, let's create a date-time object for the meal times:

```
> # look at existing data for meals
> odat$meal.time[1:5]
[1] "8:00 PM" "6:30 PM" "6:30 PM" "7:30 PM" "7:30 PM"
> # create character vector with meal date and time
> mdt <- paste("4/18/1940", odat$meal.time)
> mdt[1:4]
[1] "4/18/1940 8:00 PM" "4/18/1940 6:30 PM"
[3] "4/18/1940 6:30 PM" "4/18/1940 7:30 PM"
> # convert into standard date and time
> meal.dt <- strptime(mdt, format = "%m/%d/%Y %I:%M %p")
> meal.dt[1:4]
[1] "1940-04-18 20:00:00" "1940-04-18 18:30:00"
[3] "1940-04-18 18:30:00" "1940-04-18 19:30:00"
> # look at existing data for illness onset
> odat$onset.date[1:4]
[1] "4/19" "4/19" "4/19" "4/18"
> odat$onset.time[1:4]
[1] "12:30 AM" "12:30 AM" "12:30 AM" "10:30 PM"
> # create vector with onset date and time
> odt <- paste(paste(odat$onset.date, "/1940", sep=""),
+     odat$onset.time)
> odt[1:4]
[1] "4/19/1940 12:30 AM" "4/19/1940 12:30 AM"
[3] "4/19/1940 12:30 AM" "4/18/1940 10:30 PM"
> # convert into standard date and time
> onset.dt <- strptime(odt, "%m/%d/%Y %I:%M %p")
> onset.dt[1:4]
[1] "1940-04-19 00:30:00" "1940-04-19 00:30:00"
[3] "1940-04-19 00:30:00" "1940-04-18 22:30:00"
> # calculate incubation period
> incub.period <- onset.dt - meal.dt
> incub.period
Time differences of 4.5, 6.0, 6.0, 3.0, 3.0, 6.5, 3.0, 4.0,
6.5,  NA,  NA,  NA,  NA, 3.0,  NA,  NA,  NA, 3.0,  NA,  NA,
...
NA,  NA,  NA,  NA,  NA,  NA,  NA hours
> mean(incub.period, na.rm = T)
Time difference of 4.295455 hours
> median(incub.period, na.rm = T)
Error in Summary.difftime(..., na.rm = na.rm) :
        sum not defined for "difftime" objects
```

```
> # try 'as.numeric' on 'incub.period'
> median(as.numeric(incub.period), na.rm = T)
[1] 4
```

To summarize, we used `strptime` to convert the meal consumption date and times and illness onset dates and times into date-time objects (`meal.dt` and `onset.dt`) that can be used to calculate the incubation periods by simple subtraction (and assigned name `incub.period`).

Notice that `incub.period` is an atomic object of class `difftime`:

```
> str(incub.period)
Class 'difftime'  atomic [1:75] 4.5 6 6 3 3 6.5 3 4 NA ...
  ..- attr(*, "tzone")= chr ""
  ..- attr(*, "units")= chr "hours"
```

This is why we had trouble calculating the median (which should not be the case). We got around this problem by coercion using `as.numeric`:

```
> as.numeric(incub.period)
 [1] 4.5 6.0 6.0 3.0 3.0 6.5 3.0 4.0 6.5  NA  NA  NA  NA 3.0
...
```

Now, what kind of objects were created by the `strptime` function?

```
> str(meal.dt)
'POSIXlt', format: chr [1:75] "1940-04-18 18:30:00" ...
> str(onset.dt)
'POSIXlt', format: chr [1:75] "1940-04-19 00:30:00" ...
```

The `strptime` function produces a named list of class `POSIXlt`. POSIX stands for “Portable Operating System Interface,” and “lt” stands for “legible time”.[13]

3.9.1.4 The `POSIXlt` and `POSIXct` functions

The POSIXlt list contains the date-time data in human readable forms. The named list contains the following vectors:

```
'sec'     0-61: seconds
'min'     0-59: minutes
'hour'    0-23: hours
'mday'    1-31: day of the month
'mon'     0-11: months after the first of the year.
'year'    Years since 1900.
'wday'    0-6 day of the week, starting on Sunday.
'yday'    0-365: day of the year.
```

---

[13] For more information visit the Portable Application Standards Committee site at http://www.pasc.org/

```
'isdst'  Daylight savings time flag. Positive if in force,
         zero if not, negative if unknown.
```

Let's examine the `onset.dt` object we created from the Oswego data.

```
> is.list(onset.dt)
[1] TRUE
> names(onset.dt)
[1] "sec"   "min"   "hour"  "mday"  "mon"   "year"  "wday"
[8] "yday"  "isdst"
> onset.dt$min
 [1] 30 30 30 30 30  0  0  0  0 30 30 15  0  0  0 45 45  0
...
> onset.dt$hour
 [1]  0  0  0 22 22  2  1 23  2 10  0 22 22  1 23 21 21  1
...
> onset.dt$mday
 [1] 19 19 19 18 18 19 19 18 19 19 19 18 18 19 18 18 18 19
...
> onset.dt$mon
 [1]  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3
...
> onset.dt$year
 [1] 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40 40
...
> onset.dt$wday
 [1]  5  5  5  4  4  5  5  4  5  5  5  4  4  5  4  4  4  5
...
> onset.dt$yday
 [1] 109 109 109 108 108 109 109 108 109 109 109 108 108 109
...
```

The POSIXlt list contains useful date-time information; however, it is not in a convenient form for storing in a data frame. Using `as.POSIXct` we can convert it to a "continuous time" object that contains the number of seconds since 1970-01-01 00:00:00. `as.POSIXlt` coerces a date-time object to POSIXlt.

```
> onset.dt.ct <- as.POSIXct(onset.dt)
> onset.dt.ct[1:5]
[1] "1940-04-19 00:30:00 Pacific Daylight Time"
[2] "1940-04-19 00:30:00 Pacific Daylight Time"
[3] "1940-04-19 00:30:00 Pacific Daylight Time"
[4] "1940-04-18 22:30:00 Pacific Daylight Time"
[5] "1940-04-18 22:30:00 Pacific Daylight Time"
> as.numeric(onset.dt.ct[1:5])
[1] -937326600 -937326600 -937326600 -937333800 -937333800
```

3.9.1.5 The `format` function

Whereas the `strptime` function converts a character vector of date-time information into a date-time object, the `format` function converts a date-time object into a character vector. The `format` function gives us great flexibility in converting date-time objects into numerous outputs (e.g., day of the week, week of the year, day of the year, month of the year, year). Selected date-time format options are listed on page 153, for a complete list see `help(strptime)`.

For example, in public health, reportable communicable diseases are often reported by "disease week" (this could be week of reporting or week of symptom onset). This information is easily extracted from R date-time objects. For weeks starting on Sunday use the "%U" option in the `format` function, and for weeks starting on Monday use the "%W" option.

```
> decjan <- seq(as.Date("2003-12-15"), as.Date("2004-01-15"),
+     by =1)
> decjan
 [1] "2003-12-15" "2003-12-16" "2003-12-17" "2003-12-18"
...
[29] "2004-01-12" "2004-01-13" "2004-01-14" "2004-01-15"
> disease.week <- format(decjan, "%U")
> disease.week
 [1] "50" "50" "50" "50" "50" "50" "51" "51" "51" "51" "51"
[12] "51" "51" "52" "52" "52" "52" "00" "00" "00" "01" "01"
[23] "01" "01" "01" "01" "01" "02" "02" "02" "02" "02"
```

3.9.2 Date functions in the `chron` *and* `survival` *packages*

The `chron` and `survival` packages have customized functions for dealing with dates. Both packages come with the default R installation. To learn more about date and time classes read R News, Volume 4/1, June 2004.[14]

3.10 Exporting data objects

On occassion, we need to export R data objects. This can be done in several ways, depending on our needs:

- Generic ASCII text file
- R ASCII text file
- R binary file

---

[14] `http://cran.r-project.org/doc/Rnews`

Table 3.5 R functions for exporting data objects

Function	Description	Try these examples
Export to Generic ASCII text file		
write.table write.csv	Write tabular data as a data frame to an ASCII text file; read file back in using `read.table` function	`write.table(infert,"infert.dat")` `write.csv(infert,"infert.csv")`
write	Write matrix elements to an ASCII text file	`x <- matrix(1:4, 2, 2)` `write(t(x), "x.txt")`
Export to R ASCII text file		
dump	"Dumps" list of R objects as R code to an ASCII text file; read back in using `source` function	`dump("Titanic", "titanic.R")`
dput	Writes an R object as R code (but without the object name) to the console, or an ASCII text file; read file back in using `dget` function	`dput(Titanic, "titanic.R")`
Export to R binary file		
save	"Saves" list of R objects as binary *filename*`.Rdata` file; read back in using `load` function	`save(Titanic, "titanic.Rdata")`
Export to non-R ASCII text file		
write.foreign	From `foreign` package: writes text files (SPSS, Stata, SAS) and code to read them	`write.foreign(infert, datafile="infert.dat", codefile="infert.txt", package = "SPSS")`
Export to non-R binary file		
write.dbf	From `foreign` package: writes DBF files	`write.dbf(infert, "infert.dbf")`
write.dta	From `foreign` package: writes files in Stata binary format	`write.dta(infert, "infert.dta")`

- Non-R ASCII text files
- Non-R binary file

3.10.1 Exporting to a generic ASCII text file

3.10.1.1 The `write.table` function

We use the `write.table` function to exports a data frame as a tabular ASCII text file which can be read by most statistical packages. If the object is not a data frame, it converts to a data frame before exporting it. Therefore, this function only make sense with tabular data objects. Here are the default arguments:

```
> args(write.table)
function (x, file = "", append = FALSE, quote = TRUE,
    sep = " ",  eol = "\n", na = "NA", dec = ".",
    row.names = TRUE, col.names = TRUE,
    qmethod = c("escape", "double"))
```

The 1st argument will be the data frame name (e.g., infert), the 2nd will be the name for the output file (e.g., infert.dat), the `sep` argument is set to be space-delimited, and the `row.names` argument is set to `TRUE`.

The following code:

```
write.table(infert,"infert.dat")
```

produces this ASCII text file:

```
"education" "age" "parity" "induced" "case" ...
"1" "0-5yrs" 26 6 1 1 2 1 3
"2" "0-5yrs" 42 1 1 1 0 2 1
"3" "0-5yrs" 39 6 2 1 0 3 4
"4" "0-5yrs" 34 4 2 1 0 4 2
"5" "6-11yrs" 35 3 1 1 1 5 32
...
```

Because `row.names=TRUE`, the number field names in the header (row 1) will one less that the number of columns (starting with row 2). The default row names is a character vector of integers. The following code:

```
write.table(infert,"infert.dat", sep=",", row.names=FALSE)
```

produces a commna-delimited ASCII text file without row names:

```
"education","age","parity","induced","case", ...
"0-5yrs",26,6,1,1,2,1,3
"0-5yrs",42,1,1,1,0,2,1
"0-5yrs",39,6,2,1,0,3,4
"0-5yrs",34,4,2,1,0,4,2
"6-11yrs",35,3,1,1,1,5,32
...
```

Note that the `write.csv` function produces a comma-delimited data file by default.

3.10.1.2 The `write` function

The `write` function writes the contents of a matrix in a columnwise order to an ASCII text file. To get the same appearance as the matrix, we must transpose the matrix and specify the number of columns (the default is 5). If we set `file=""`, then the output is written to the screen:

```
> infert.tab1 <- xtabs(~case+parity,data=infert)
> infert.tab1
    parity
case  1  2  3  4  5  6
   0 66 54 24 12  4  5
   1 33 27 12  6  2  3
> write(infert.tab1, file="") #not what we want
66 33 54 27 24
12 12 6 4 2
5 3
> #much better
> write(t(infert.tab1), file="", ncol=ncol(infert.tab1))
66 54 24 12 4 5
33 27 12 6 2 3
```

To read the raw data back into R, we would use the `scan` function. For example, if the data had been written to `data.txt`, then the following code reads the data back into R:

```
> matrix(scan("data.txt"), ncol=6, byrow=TRUE)
Read 12 items
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]   66   54   24   12    4    5
[2,]   33   27   12    6    2    3
```

Of course, all the labeling was lost.

3.10.2 Exporting to R ASCII text file

Data objects can also be exported as R code in an ASCII text file using the `dump` and `dput` functions. This has advantages for complex R objects (e.g., arrays, lists) that do not have simple tabular structures, and the R code makes the raw data human legible.

3.10.2.1 The `dump` function

The `dump` function exports multiple objects as R code as the next example illustrates:

```
> infert.tab1 <- xtabs(~case+parity,data=infert)
> infert.tab2 <- xtabs(~education+parity+case,data=infert)
> infert.tab1 #display matrix
```

```
    parity
case  1  2  3  4  5  6
   0 66 54 24 12  4  5
   1 33 27 12  6  2  3
> infert.tab2 #display array
, , case = 0

         parity
education  1  2  3  4  5  6
  0-5yrs   2  0  0  2  0  4
  6-11yrs 28 28 14  8  2  0
  12+ yrs 36 26 10  2  2  1

, , case = 1

         parity
education  1  2  3  4  5  6
  0-5yrs   1  0  0  1  0  2
  6-11yrs 14 14  7  4  1  0
  12+ yrs 18 13  5  1  1  1

> dump(c("infert.tab1", "infert.tab2"),"infert_tab.R") #export
```

The `dump` function produced the following R code in the `infert_tab.R` text file.

```
`infert.tab1` <-
structure(c(66, 33, 54, 27, 24, 12, 12, 6, 4, 2, 5, 3),
.Dim = c(2L, 6L), .Dimnames = structure(list(case = c("0", "1"),
parity = c("1", "2", "3", "4", "5", "6")), .Names = c("case",
"parity")), class = c("xtabs", "table"), call = quote(xtabs(
formula = ~case + parity, data = infert)))
`infert.tab2` <-
structure(c(2, 28, 36, 0, 28, 26, 0, 14, 10, 2, 8, 2, 0, 2,
2, 4, 0, 1, 1, 14, 18, 0, 14, 13, 0, 7, 5, 1, 4, 1, 0, 1, 1,
2, 0, 1), .Dim = c(3L, 6L, 2L), .Dimnames = structure(list(
education = c("0-5yrs", "6-11yrs", "12+ yrs"), parity = c("1",
"2", "3", "4", "5", "6"), case = c("0", "1")), .Names =
c("education", "parity", "case")), class = c("xtabs", "table"),
call = quote(xtabs(formula = ~education + parity + case,
data = infert)))
```

Notice that the 1st argument was a character vector of object names. The `infert_tab.R` file can be run in R using the `source` function to recreate all the objects in the workspace.

3.10.2.2 The `dput` function

The `dput` function is similar to the `dump` function except that the object name is not written. By default, the `dput` function prints to the screen:

```
> dput(infert.tab1)
structure(c(66, 33, 54, 27, 24, 12, 12, 6, 4, 2, 5, 3),
.Dim = c(2L, 6L), .Dimnames = structure(list(case = c("0", "1"),
parity = c("1", "2", "3", "4", "5", "6")), .Names = c("case",
"parity")), class = c("xtabs", "table"), call = quote(xtabs(
formula = ˜case + parity, data = infert)))
```

To export to an ASCII text file, give a new file name as the second argument, similar to `dump`. To get back the R code use the `dget` function:

```
> dput(infert.tab1, "infert_tab1.R")
> dget("infert_tab1.R")
    parity
case  1  2  3  4  5  6
   0 66 54 24 12  4  5
   1 33 27 12  6  2  3
```

3.10.3 Exporting to R binary file

3.10.3.1 The `save` function

The `save` function exports R data objects to binary file (*filename*`.RData`) which is the most effient, compact method to export objects. The first argument(s) can be the names of the objects to save followed by the output file name, or list with a character vector of object names followed by the output file name. Here is an example of the first option:

```
> x <- 1:5; y <- xˆ3
> save(x, y, file="xy.RData")
> rm(x, y)
> ls()
character(0)
> load(file="xy.RData")
> ls()
[1] "x" "y"
```

Notice that we used the `load` function to load the binary file back into the workspace.

Now here is an example of the second option using a list:

```
> x <- 1:5; y <- xˆ3
> save(list=c("x", "y"), file="xy.RData")
> rm(x, y)
> ls()
character(0)
> load(file="xy.RData")
> ls()
[1] "x" "y"
```

In fact, the `save.image` function we use to save the entire workspace is just the following:

```
save(list = ls(all=TRUE), file = ".RData")
```

3.10.4 Exporting to non-R ASCII text and binary files

The `foreign` package contains functions for exporting R data frames to non-R ASCII text and binary files. The `write.foreign` function write two ASCII text files: the first file is the data file, and the second file is the code file for reading the data file. The code file contains either SPSS, Stata, or SAS programming code. The `write.dbf` function writes a data frame to a binary DBF file, which can be read back into R using the `read.dbf` function. Finally, the `write.dta` function writes a data frame to a binary Stata file, which can be read back into R using the `read.dta` function.

3.11 Working with regular expressions

A *regular expression* is a special text string for describing a search pattern which can be used for searching text strings, indexing data objects, and replacing object elements. For example, we applied Global Burden of Disease methods to evaluate causes of premature deaths in San Francisco [4]. Using regular expressions we were able to efficiently code over 14,000 death records, with over 900 ICD-10 cause of death codes, into 117 mutually exclusive cause of death categories. Without regular expressions, this local study would have been prohibitively tedious.

A regular expression is built up from specifying one character at a time. Using this approach, we cover the following:

- Single character: matching a single character;
- Character class: matching a single character from among a list of characters;
- Concatenation: combining single characters into a new match pattern;
- Repetition: specifying how many times a single character or match pattern might be repeated;
- Alternation: a regular expression may be matched from among two or more regular expressions; and

- Metacharacters: special characters that require special treatment.

3.11.1 Single characters

The search pattern is built up from specifying one character at a time. For example, the pattern `"x"` looks for the letter *x* in a text string. Next, consider a character vector of text strings. We can use the `grep` function to search for a pattern in this data vector.

```
> vec1 <- c("x", "xa bc", "abc", "ax bc", "ab xc", "ab cx")
> grep("x", vec1)
[1] 1 2 4 5 6
> vec1[grep("x", vec1)] #index by position
[1] "x"     "xa bc" "ax bc" "ab xc" "ab cx"
```

The `grep` function returned an integer vector indicating the positions in the data vector that contain a match. We used this integer vector to index by position.

The caret ˆ matches the empty string at the beginning of a line. Therefore, to match this pattern at the beginning of a line we add the ˆ character to the regular expression:

```
> grep("ˆx", vec1)
[1] 1 2
> vec1[grep("ˆx", vec1)] #index by position
[1] "x"     "xa bc"
```

The `$` character matches the empty string at the end of a line. Therefore, to match this pattern at the end of a line we add the `$` character to the regular expression:

```
> vec1[grep("x$", vec1)] #index by position
[1] "x"     "ab cx"
```

The ˆ and `$` characters are examples of metacharacters (more on these later).

To match this pattern at the beginning of a word, but not the beginning of a line, we add a space to the regular expression:

```
> vec1[grep(" x", vec1)] #index by position
[1] "ab xc"
```

To match this pattern at the end of a word, but not the end of a line, we add a space to the regular expression:

```
> vec1[grep("x ", vec1)] #index by position
[1] "ax bc"
```

The period "." matches any single character, including a space.

```
> vec1[grep(".bc", vec1)]
[1] "xa bc" "abc"   "ax bc"
```

Table 3.6 Predefined character classes for regular expressions

Predefined	Description	Alternative
`[[:lower:]]`	Lower-case letters in the current locale	`[a-z]`
`[[:upper:]]`	Upper-case letters in the current locale	`[A-Z]`
`[[:alpha:]]`	Alphabetic characters	`[A-Za-z]`
`[[:digit:]]`	Digits	`[0-9]`
`[[:alnum:]]`	Alphanumeric characters	`[A-Za-z0-9]`
`[[:punct:]]`	Punctuation characters: ! " # $ % & ′ ( ) * + , - . / : ; < = > ? @ [\ ] ^ _ ` { | } ~	`"[]...^...-]"` (`]`, `^`, and `-` require special placement in character classes. See p. 171)
`[[:space:]]`	Space characters: tab, newline, vertical tab, form feed, carriage return, and space	
`[[:graph:]]`	Graphical characters	`[[:alnum:][:punct:]]`
`[[:print:]]`	Printable characters	`[[:alnum:][:punct:][:space:]]`
`[[:xdigit:]]`	Hexadecimal digits:	`[0-9A-Fa-f]`

3.11.2 Character class

A *character class* is a list of characters enclosed by square brackets `[` and `]` which matches any single character in that list. For example, `"[fhr]"` will match the single character `f`, `h`, or `r`. This can be combined with metacharacters for more specificity; for example, `"^[fhr]"` will match the single character `f`, `h`, or `r` at the beginning of a line.

```
> vec2 <- c("fat", "bar", "rat", "elf", "mach", "hat")
> grep("^[fhr]", vec2)
[1] 1 3 6
> vec2[grep("^[fhr]", vec2)] #index by position
[1] "fat" "rat" "hat"
```

As already shown, ^ character matches the empty string at the beginning of a line. However, when ^ is the first character in a character class list, it matches any character *not* in the list. For example, `"^[^fhr]"` will match any single character at the beginning of a line *except* `f`, `h`, or `r`.

```
> vec2 <- c("fat", "bar", "rat", "elf", "mach", "hat")
> vec2[grep("^[^fhr]", vec2)] #index by position
[1] "bar"  "elf"  "mach"
```

Character classes can be specified as a *range* of possible characters. For example, `[0-9]` matches a single digit with possible values from 0 to 9, `[A-Z]` matches a single letter with possible values from *A* to *Z*, and `[a-z]` matches a single letter from *a* to *z*. The pattern `[0-9A-Za-z]` matches any single alphanumeric character.

For convenience, certain character classes are predefined and their interpretation depend on locale.[15] For example, to match a single lower case letter we place `[:lower:]` inside square brackets like this: `"[[:lower:]]"`, which is equivalent to `"[a-z]"`. Table 3.6 on the facing page lists predefined character classes. This is very convenient for matching punctuation characters and multiple types of spaces (e.g., tab, newline, carriage return).

In this final example, `"^.[^a].`” will match any first character, followed by any character except *a*, and followed by any character one or more times:

```
> vec2[grep("^.[^a].+", vec2)] #index by position
[1] "elf"
```

Combining single character matches is call concatenation.

3.11.3 Concatenation

Single characters (including character classes) can be *concatenated*; for example, the pattern `"^[fhr]at$"` will match the single, isolated words *fat*, *hat*, or *rat*.

```
> vec3 <- c("fat", "bar", "rat", "fat boy", "elf", "mach",
+           "hat")
> vec3[grep("^[fhr]at$", vec3)] #index by position
[1] "fat" "rat" "hat"
```

The concatenation `"[ct]a[br]"` will match the pattern that starts with *c* or *t*, followed by *a*, and followed by *b* or *r*.

```
> vec4 <- c("cab", "carat", "tar", "bar", "tab", "batboy",
+           "care")
> vec4[grep("[ct]a[br]", vec4)] #index by position
[1] "cab"   "carat" "tar"   "tab"   "care"
```

To match single, 3-letter words use `"^[ct]a[br]$"`.

```
> vec4[grep("^[ct]a[br]$", vec4)] #index by position
[1] "cab" "tar" "tab"
```

The period (`.`) is another metacharacter: it matches any single character. For example, `"f.t"` matches the pattern $f +$ any character $+ t$.

```
> vec5 <- c("fate", "rat", "fit", "bat", "futbol")
> vec5[grep("f.t", vec5)] #index by position
[1] "fate"   "fit"    "futbol"
```

---

[15] The locale describes aspects of the internationalization of a program. Initially most aspects of the locale of R are set to “C” (which is the default for the C language and reflects North-American usage).

Table 3.7 Regular expressions may be followed by a repetition quantifier

Repetition quantifier	Description
?	Preceding pattern is optional and will be matched at most once
*	Preceding pattern will be matched zero or more times
+	Preceding pattern will be matched one or more times
$\{n\}$	Preceding pattern is matched exactly n times
$\{n,\}$	Preceding pattern is matched n or more times
$\{n, m\}$	Preceding pattern is matched at least *n* times, but not more than *m* times

3.11.4 Repetition

Regular expressions (so far: single characters, character classes, and concatenations) can be qualified by whether a pattern can repeat (Table 3.7). For example, the pattern `"^f.+t$"` matches single, isolated words that start with *f* or *F*, followed by 1 or more of any character, and ending with *t*.

```
> vec6 <- c("fat", "fate", "feat", "bat", "Fahrenheit", "bat",
+            "foot")
> vec6[grep("^[fF].+t$", vec6)] #index by position
[1] "fat"        "feat"       "Fahrenheit" "foot"
```

Repetition quantifiers gives us great flexibility to specify how often preceding patterns can repeat.

3.11.5 Alternation

Two or more regular expressions (so far: single characters, character classes, concatenations, and repetitions) may be joined by the infix operator |. The resulting regular expression can match the pattern of any subexpression. For example, the World Health Organization (WHO) Global Burden of Disease (GBD) Study used International Classification of Diseases, 10th Revision (ICD-10) codes (ref). The GBD Study ICD-10 codes for hepatitis B are the following:

> B16, B16.0, B16.1, B16.2, B16.3, B16.4, B16.5, B16.7, B16.8, B16.9, B17, B17.0, B17.2, B17.8, B18, B18.0, B18.1, B18.8, B18.9

Notice that B16 and B16.0 are *not* the same ICD-10 code! The GBD Study methods were used to study causes of death in San Francisco, California (ref). Underlying causes of death were obtained from the State of California, Center for Health Statistics. The ICD-10 code field did not have periods so that the hepatitis B codes were the following.

> B16, B160, B161, B162, B163, B164, B165, B167, B168, B169, B17, B170, B172, B178, B18, B180, B181, B188, B189

To match the pattern of ICD-10 codes representing hepatitis B, the following regular expression was used (without spaces):

```
"^B16[0-9]?$|^B17[0,2,8]?$|^B18[0,1,8,9]?$"
```

This regular expression matches `^B16[0-9]?$` *or* `^B17[0,2,8]?$` *or* `^B18[0,1,8,9]?$`. Similar to the first and third pattern, the second regular expression, `^B17[0,2,8]?$`, matches B17, B170, B172, or B178 as isolated text strings.

To see how this works, we can match each subexpression individually and then as an alternation:

```
> hepb <- c("B16", "B160", "B161", "B162", "B163", "B164",
+           "B165", "B167", "B168", "B169", "B17", "B170",
+           "B172", "B178", "B18", "B180", "B181", "B188",
+           "B189")
> grep("^B16[0-9]?$", hepb) #match 1st subexpression
 [1]  1  2  3  4  5  6  7  8  9 10
> grep("^B17[0,2,8]?$", hepb) #match 2nd subexpression
[1] 11 12 13 14
> grep("^B18[0,1,8,9]?$", hepb) #match 3rd subexpression
[1] 15 16 17 18 19
> #match any subexpression
> grep("^B16[0-9]?$|^B17[0,2,8]?$|^B18[0,1,8,9]?$", hepb)
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19
```

A natural use for these pattern matches is for indexing and replacement. We illustrate this using the 2nd subexpression.

```
> #indexing
> hepb[grep("^B17[0,2,8]?$", hepb)]
[1] "B17"  "B170" "B172" "B178"
> #replacement
> hepb[grep("^B17[0,2,8]?$", hepb)] <- "HBV"
> hepb
 [1] "B16"  "B160" "B161" "B162" "B163" "B164" "B165" "B167"
 [9] "B168" "B169" "HBV"  "HBV"  "HBV"  "HBV"  "B18"  "B180"
[17] "B181" "B188" "B189"
```

Using regular expression alternations allowed us to efficiently code over 14,000 death records, with over 900 ICD-10 cause of death codes, into 117 mutually exclusive cause of death categories for our San Francisco study. Suppose `sfdat` was the data frame with San Francisco deaths for 2003–2004. Then the following code would tabulate the deaths caused by hepatatis B:

```
> sfdat$hepb <- rep("No", nrow(sfdat)) #new field
> get.hepb <- grep("^B16[0-9]?$|^B17[0,2,8]?$|^B18[0,1,8,9]?$",
+                  sfdat$icd10)
> sfdat$hepb[get.hepb] <- "Yes"
> table(sfdat$hepb)
```

```
   No   Yes
14125    23
```

Therefore, in San Francisco, during the period 2003-2004, there were 23 deaths caused by hepatitis B. Without regular expressions, this mortality analysis would have been prohibitively tedious.

In this next example we use regular expressions to correct misspelled data. Suppose we have a data vector containing my first name (“Tomas”), but sometimes misspelled. We want to locate the most common misspellings and correct them:

```
 tdat <- c("Tom", "Thomas", "Tomas", "Tommy", "tomas")
> misspelled <- grep("^[Tt]omm?y?$|^[Tt]homas$|^tomas$", tdat)
> misspelled
[1] 1 2 4 5
> tdat[misspelled] <- "Tomas"
> tdat
[1] "Tomas" "Tomas" "Tomas" "Tomas" "Tomas"
```

3.11.6 Repetition > Concatenation > Alternation

Repetition takes precedence over concatenation, which in turn takes precedence over alternation. A whole subexpression may be enclosed in parentheses to override these precedence rules. For example, consider how the following regular expression changes when parentheses are used give concatenation precedence over repetition:

```
> vec7 <- c("Tommy", "Tomas", "Tomtom")
> #repetition takes precedence
> vec7[grep("[Tt]om{2,}", vec7)]
[1] "Tommy"
> #concatenation takes precedence
> vec7[grep("([Tt]om){2,}", vec7)]
[1] "Tomtom"
```

Recall that `{2,}` means repeat the previous 2 or more times.

3.11.7 Metacharacters

Any character, or combination of characters, can be use to specify a pattern except for these *metacharacters*:

```
. \ | ( ) [ { ^ $ * + ?
```

Metacharacters have special meaning in regular expressions, and these have already been presented and are summarized in Table 3.8. However, inside a character class,

Table 3.8 Metacharacters used by regular expressions

Char.	Description	Example	Literal search
`^`	Matches empty string at beginning of line	`"^my car"`	`See p. 171`
	When 1st character in character class, matches any single character not in the list	`"[^abc]"`	`See p. 171`
`$`	Matches empty string at end of line	`"my car$"`	`"[$]"`
`[`	Character class		`"[[]"`
`.`	Matches any single character	`"p.t"`	`"[.]"`
`?`	Repetition quantifier (Table 3.7)	`".?"`	`"[?]"`
`*`	Repetition quantifier (Table 3.7)	`".*"`	`"[*]"`
`+`	Repetition quantifier (Table 3.7)	`".+"`	`"[+]"`
`( )`	Grouping subexpressions	`"([Tt]om){2,}"`	`"[(]"` or `"[)]"`
|	Join subexpresions, any of which can be matched	"Tomas|Luis"	"[|]"
{	Not used in R regular expressions	`n/a`	`"[{]"`

metacharacters have their literal interpretation. For example, to search for data vector elements that contain one or more periods use:

```
> vec8 <- c("oswego.dat", "oswego", "infert.dta", "infert")
> vec8[grep("[.]", vec8)]
[1] "oswego.dat" "infert.dta"
```

If we want to include the following characters inside a character class, they require special placement:

```
] - ^
```

To include a literal `]`, place it first in the list. Similarly, to include a literal `-`, place it first or last in the list.[16] Finally, to include a literal `^`, place it anywhere but first.

```
> ages <- c("<1^1", "[1,15)", "15-34", "[35,65)", "[65,110]")
> ages[grep("[]^-]", ages)]
[1] "<1^1"     "15-34"    "[65,110]"
```

To search for a literal `^` as a single character is tricky because it must be placed inside a character class but preceded by another character (`"^"` will not work, and `"[^]"` returns an error). Because we are only interested in finding `^`, then the first character in the list should be any character we expect not to find in the data vector (`"[/^]"` should work). Study the example that follows:

```
> vec9 <- c("8^2", "89", "y^x", "time")
> grep("/", vec9)     #test that / is not in data
integer(0)
> vec9[grep("[/^]", vec9)]
[1] "8^2" "y^x"
```

---

[16] Although the − sign is not a metacharacter, it does have special meaning inside a character class because it is used to specify a range of characters; e.g., `[A-Za-z0-9]`.

Table 3.9 Commonly used functions that use regular expressions

Function	Description
grep	Searches for pattern matches within a character vector; returns integer vector indicating vector positions containing pattern
regexpr	Similar to `grep` but returns integer vectors with detailed information for the first occurrence of a pattern match within text string elements of a character vector
gregexpr	Similar to `regexpr` but returns a list with detailed information for the multiple occurrences of a pattern match within text string elements of a character vector
sub	Searches and replaces the first occurrence of a pattern match within text string elements of a character vector
gsub	Searches and replaces multiple occurrences of a pattern match within text string elements of a character vector

The first character in the list (`/`) was selected because there was no match in the data vector.

3.11.8 Other regular expression functions

For most epidemiologic applications, the `grep` function will meet our regular expression needs. Table 3.9 summarizes other functions that use regular expressions. Whereas the `grep` function enables indexing and replacing elements of a character vector, the `sub` and `gsub` functions searches and replaces single or multiple pattern matches *within* text string elements of a character vector. Review the following example:

```
> vec10 <- c("California", "MiSSISSIppi")
> grep("SSI", vec10) #can be used for replacement
[1] 2
> sub("SSI", replacement="ssi", vec10)  #replace 1st occurrence
[1] "California"  "MissiSSIppi"
> gsub("SSI", replacement="ssi", vec10) #replace all occurrences
[1] "California"  "Mississippi"
```

The `regexpr` function provides detailed information on the first pattern match within text string elements of a character vector. It returns two integer vectors. In the first vector, `-1` indicates no match, and nonzero positive integers indicate the character position where the first match begins within a text string. In the second vector, the nonzero positive integers indicate the match length. In contrast, the `gregexpr` function provides detailed information on multiple pattern matches within text string elements of a character vector. It returns a list where each bin contains detailed information (similar to `regexpr`) for each text string element of a character vector. Study the following examples:

```
> regexpr("SSI", vec10)
```

```
[1] -1  3
attr(,"match.length")
[1] -1  3
> gregexpr("SSI", vec10)
[[1]]
[1] -1
attr(,"match.length")
[1] -1

[[2]]
[1] 3 6
attr(,"match.length")
[1] 3 3
```

Problems

3.1. Using RStudio and the data from Table 3.1 on page 107 Create the following data frame:

```
> dat
    Status   Treatment Agegrp Freq
1     Dead Tolbutamide    <55    8
2 Survived Tolbutamide    <55   98
3     Dead     Placebo    <55    5
4 Survived     Placebo    <55  115
5     Dead Tolbutamide    55+   22
6 Survived Tolbutamide    55+   76
7     Dead     Placebo    55+   16
8 Survived     Placebo    55+   69
```

3.2. Select 3 to 5 classmates and collect data on first name, last name, affiliation, two email addresses, and today's date. Using a text editor, create a data frame with this data.

3.3. Review the United States data on AIDS cases by year available at `http://www.medepi.net/data/aids.txt`. Read this data into a data frame. Graph a calendar time series of AIDS cases.

```
# Hint
plot(x, y, type = "l", xlab = "x axis label", lwd = 2,
     ylab = "y axis label", main = "main title")
```

3.4. Review the United States data on measles cases by year available at `http://www.medepi.net/data/measles.txt`. Read this data into a data frame. Graph a calendar time series of measle cases using an arithmetic and semi-logarithmic scale.

```
# Hint
plot(x, y, type = "l", lwd = 2, xlab = "x axis label",
     ylab="y axis label", main = "main title")
plot(x, y, type = "l", lwd = 2, xlab = "x axis label", log = "y",
     ylab="y axis label", main = "main title")
```

3.5. Review the United States data on hepatitis B cases by year available at `http://www.medepi.net/data/hepb.txt`. Read this data into a data frame. Using the R code below, plot a times series of AIDS and hepatitis B cases.

```
matplot(hepb$year, cbind(hepb$cases,aids$cases),
    type = "l", lwd = 2, xlab = "Year", ylab = "Cases",
    main = "Reported cases of Hepatitis B and AIDS,
United States, 1980-2003")
legend(1980, 100000, legend = c("Hepatitis B", "AIDS"),
    lwd = 2, lty = 1:2, col = 1:2)
```

Table 3.10 Data dictionary for Evans data set

Variable	Variable name	Variable type	Possible values
id	Subject identifier	Integer	
chd	Coronary heart disease	Categorical-nominal	0 = no 1 = yes
cat	Catecholamine level	Categorical-nominal	0 = normal 1 = high
age	Age	Continuous	years
chl	Cholesterol	Continuous	> 0
smk	Smoking status	Categorical-nominal	0 = never smoked 1 = ever smoked
ecg	Electrocardiogram	Categorical-nominal	0 = no abnormality 1 = abnormality
dbp	Diastolic blood pressure	Continuous	mm Hg
sbp	Systolic blood pressure	Continuous	mm Hg
hpt	High blood pressure	Categorical-nominal	0 = no 1 = yes (dbp $\geq$ 95 or sbp $\geq$ 160)
ch	cat × hpt	Categorical	product term
cc	cat × chl	Continuous	product term

3.6. Review data from the Evans cohort study in which 609 white males were followed for 7 years, with coronary heart disease as the outcome of interest (`http://www.medepi.net/data/evans.txt`). The data dictionary is provided in Table 3.10.

a Recode the binary variables (0, 1) into factors with 2 levels.
b Discretized age into a factor with more than 2 levels.
c Create a new hyptertension categorical variable based on the current classification scheme[17]:
Normal: SBP$<$ 120 and DBP$<$ 80;
Prehypertension: SBP=[120, 140) or DBP=[80, 90);
Hypertension-Stage 1: SBP=[140, 160) or DBP=[90, 100); and
Hypertension-Stage 2: SBP$\geq$ 160 or DBP$\geq$ 100.
d Using R, construct a contigency table comparing the old and new hypertension variables.

3.7. Review the California 2004 surveillance data on human West Nile virus cases available at `http://www.medepi.net/data/wnv/wnv2004raw.txt`. Read in the data, taking into account missing values. Convert the calendar dates into the international standard format. Using the `write.table` function export the data as an ASCII text file.

---

[17] `http://www.nhlbi.nih.gov/guidelines/hypertension/phycard.pdf`

3.8. On April 19, 1940, the local health officer in the village of Lycoming, Oswego County, New York, reported the occurrence of an outbreak of acute gastrointestinal illness to the District Health Officer in Syracuse. Dr. A. M. Rubin, epidemiologist-in-training, was assigned to conduct an investigation. (See Appendix A.2 on page 184 for data dictionary.)

When Dr. Rubin arrived in the field, he learned from the health officer that all persons known to be ill had attended a church supper held on the previous evening, April 18. Family members who did not attend the church supper did not become ill. Accordingly, Dr. Rubin focused the investigation on the supper. He completed Interviews with 75 of the 80 persons known to have attended, collecting information about the occurrence and time of onset of symptoms, and foods consumed. Of the 75 persons interviewed, 46 persons reported gastrointestinal illness.

The onset of illness in all cases was acute, characterized chiefly by nausea, vomiting, diarrhea, and abdominal pain. None of the ill persons reported having an elevated temperature; all recovered within 24 to 30 hours. Approximately 20% of the ill persons visited physicians. No fecal specimens were obtained for bacteriologic examination. The investigators suspected that this was a vehicle-borne outbreak, with food as the vehicle. Dr. Rubin put his data into a line listing.[18]

The supper was held in the basement of the village church. Foods were contributed by numerous members of the congregation. The supper began at 6:00 p.m. and continued until 11:00 p.m. Food was spread out on a table and consumed over a period of several hours. Data regarding onset of illness and food eaten or water drunk by each of the 75 persons interviewed are provided in the line listing. The approximate time of eating supper was collected for only about half the persons who had gastrointestinal illness.

a. Using RStudio plot the cases by time of onset of illness (include appropriate labels and title). What does this graph tell you? (Hint: Process the text data and then use the `hist` function.)
b. Are there any cases for which the times of onset are inconsistent with the general experience? How might they be explained?
c. How could the data be sorted by illness status and illness onset times?
d. Where possible, calculate incubation periods and illustrate their distribution with an appropriate graph. Use the `truehist` function in the MASS package. Determine the mean, median, and range of the incubation period.

---

[18] See data set at `http://www.medepi.net/data/oswego/oswego.txt`.

Part II
Applied Epidemiology

Appendixes

APPENDIX A

Available data sets

A.1 Latina Mothers and their Newborn

From 1980 to 1990 data was collected on 427 Latino mothers that gave birth at the University of California, San Francisco [12, 13]. Data was collected on the characteristics of the mothers and their newborn infants (Table A.1). Mothers were weighed at each prenatal visit. Rate of weight gain during each trimester was based on a linear regression interpolation. The data set can be viewed and downloaded from `http://www.medepi.net/data/birthwt9.txt`.

Table A.1 Data dictionary for Latina mothers and their newborn infants

Variable	Description	Possible values
`age`	Maternal age	In years (self-reported)
`parity`	Parity	Count of previous live births
`gest`	Gestation	Reported in days
`sex`	Gender	Male = 1, Female = 2
`bwt`	Birth weight	Grams
`cigs`	Smoking	Number of cigarettes per day (self-reported)
`ht`	Maternal height	Measured in centimeters
`wt`	Maternal weight	Pre-pregnancy weight (self-reported)
`r1`	Rate of weight gain (1st trimester)	Kilograms per day (estimated)
`r2`	Rate of weight gain (2nd trimester)	Kilograms per day (estimated)
`r2`	Rate of weight gain (3rd trimester)	Kilograms per day (estimated)

A.2 Oswego County (outbreak)

On April 19, 1940, the local health officer in the village of Lycoming, Oswego County, New York, reported the occurrence of an outbreak of acute gastrointestinal illness to the District Health Officer in Syracuse. Dr. A. M. Rubin, epidemiologist-in-training, was assigned to conduct an investigation.

When Dr. Rubin arrived in the field, he learned from the health officer that all persons known to be ill had attended a church supper held on the previous evening, April 18. Family members who did not attend the church supper did not become ill. Accordingly, Dr. Rubin focused the investigation on the supper. He completed Interviews with 75 of the 80 persons known to have attended, collecting information about the occurrence and time of onset of symptoms, and foods consumed. Of the 75 persons interviewed, 46 persons reported gastrointestinal illness.

The onset of illness in all cases was acute, characterized chiefly by nausea, vomiting, diarrhea, and abdominal pain. None of the ill persons reported having an elevated temperature; all recovered within 24 to 30 hours. Approximately 20 physicians. No fecal specimens were obtained for bacteriologic examination.

The supper was held in the basement of the village church. Foods were contributed by numerous members of the congregation. The supper began at 6:00 p.m. and continued until 11:00 p.m. Food was spread out on table and consumed over a period of several hours. Data regarding onset of illness and food eaten or water drunk by each of the 75 persons interviewed are provided in the attached line listing (Oswego dataset). The approximate time of eating supper was collected for only about half the persons who had gastrointestinal illness.

The data set can be viewed and downloaded from `http://www.medepi.net/data/oswego.txt`. The data dictionary is provided in Table A.2 on the facing page.

A.3 Western Collaborative Group Study (cohort)

The Western Collaborative Group Study (WCGS), a prospective cohort studye, recruited middle-aged men (ages 39 to 59) who were employees of 10 California companies and collected data on 3154 individuals during the years 1960–1961. These subjects were primarily selected to study the relationship between behavior pattern and the risk of coronary hearth disease (CHD). A number of other risk factors were also measured to provide the best possible assessment of the CHD risk associated with behavior type. Additional variables collected include age, height, weight, systolic blood pressure, diastolic blood pressure, cholesterol, smoking, and corneal arcus. The median follow up time was 8.05 years.

The data set can be viewed and downloaded from `http://www.medepi.net/data/wcgs.txt`. The data dictionary is provided in Table A.3 on page 186.

Table A.2 Data dictionary for Oswego County data set

Variable	Possible values
id	Subject identificaton number
age	Age in years
sex	Sex: F = Female, M = Male
meal.time	Meal time on April 18th
ill	Developed illness: Y = Yes N = No
onset.date	Onset date: ”4/18” = April 18th, ”4/19” = April 19th
onset.time	Onset time: HH:MM AM/PM
baked.ham	Consumed item: Y = Yes; N = No
spinach	Consumed item: Y = Yes; N = No
mashed.potato	Consumed item: Y = Yes; N = No
cabbage.salad	Consumed item: Y = Yes; N = No
jello rolls	Consumed item: Y = Yes; N = No
brown.bread	Consumed item: Y = Yes; N = No
milk	Consumed item: Y = Yes; N = No
coffee	Consumed item: Y = Yes; N = No
water	Consumed item: Y = Yes; N = No
cakes	Consumed item: Y = Yes; N = No
vanilla.ice.cream	Consumed item: Y = Yes; N = No
chocolate.ice.cream	Consumed item: Y = Yes; N = No
fruit.salad	Consumed item: Y = Yes; N = No

A.4 Evans County (cohort)

The Evans County data set is used to demonstrate a standard logistic regression (unconditional) [15]. The data are from a cohort study in which 609 white males were followed for 7 years, with coronary heart disease as the outcome of interest.

The data set can be viewed and downloaded from `http://www.medepi.net/data/evans.txt`. The data dictionary is provided in Table A.4 on the following page.

A.5 Myocardial infarction case-control study

The myocardial infarction (MI) data set [15] is used to demonstrate conditional logistic regression. The study is a case-control study that involves 117 subjects in 39 matched strata (matched by age, race, and sex). Each stratum contains three subjects, one of whom is a case diagnosed with myocardial infarction and the other two are matched controls.

The data set can be viewed and downloaded from `http://www.medepi.net/data/mi.txt`. The data dictionary is provided in Table A.5 on page 187.

Table A.3 Data dictionary for Western Collaborative Group Study data set

Variable	Variable name	Variable type	Possible values
id	Subject ID	Integer	2001–22101
age0	Age	Continuous	39–59 years
height0	Height	Continuous	60–78 in
weight0	Weight	Continuous	78–320 lb
sbp0	Systolic blood pressure	Continuous	98–230 mm Hg
dbp0	Diastolic blood pressure	Continuous	58–150 mm Hg
chol0	Cholesterol	Continuous	103–645 mg/100 ml
behpat0	Behavior pattern	Categorical	1 = Type A1 2 = Type A2 3 = Type B1 4 = Type B2
ncigs0	Smoking	Integer	Cigarettes/day
dibpat0	Behavior pattern	Categorical	0 = Type B 1 = Type A
chd69	Coronary heart disease event	Categorical	0 = None 1 = Yes
typechd	Coronary heart disease event	Categorical	0 = CHD event 1 = Symptomatic MI 2 = Silent MI 3 = Classical angina
time169	Observation (follow up) time	Continuous	18–3430 days
arcus0	Corneal arcus	Categorical	0 = None 1 = Yes

Table A.4 Data dictionary for Evans data set

Variable	Variable name	Variable type	Possible values
id	Subject identifier	Integer	
chd	Coronary heart disease	Categorical-nominal	0 = no 1 = yes
cat	Catecholamine level	Categorical-nominal	0 = normal 1 = high
age	Age	Continuous	years
chl	Cholesterol	Continuous	> 0
smk	Smoking status	Categorical-nominal	0 = never smoked 1 = ever smoked
ecg	Electrocardiogram	Categorical-nominal	0 = no abnormality 1 = abnormality
dbp	Diastolic blood pressure	Continuous	mm Hg
sbp	Systolic blood pressure	Continuous	mm Hg
hpt	High blood pressure	Categorical-nominal	0 = no 1 = yes (dbp$\geq$ 95 or sbp$\geq$ 160)
ch	cat × hpt	Categorical	product term
cc	cat × chl	Continuous	product term

Table A.5 Data dictionary for myocardial infarction (MI) case-control data set

Variable	Variable name	Variable type	Possible values
match	Matching strata	Integer	1–39
person	Subject identifier	Integer	1–117
mi	Myocardial infarction	Categorical-nominal	0 = No 1 = Yes
smk	Smoking status	Categorical-nominal	0 = Not current smoker 1 = Current smoker
sbp	Systolic blood pressure	Categorical-ordinal	120, 140, or 160
ecg	Electrocardiogram	Categorical-nominal	0 = No abnormality 1 = abnormality

A.6 AIDS surveillance cases

`http://www.medepi.net/data/aids.txt`

A.7 Hepatitis B surveillance cases

`http://www.medepi.net/data/hepb.txt`

A.8 Measles surveillance cases

`http://www.medepi.net/data/measles.txt`

A.9 University Group Diabetes Program

`http://www.medepi.net/data/ugdp.txt`

A.10 Novel influenza A (H1N1) pandemic

A.10.1 United States reported cases and deaths as of July 23, 2009

`http://www.medepi.net/data/h1n1panflu23jul09usa.txt`

APPENDIX B

Outbreak analysis template in R

We provide an analysis template using R (and the 'epitools' package). Examples involve human West Nile virus surveillance, other data sets (AIDS, measles, hepatitis B, etc.).

Read data

Human West Nile virus disease surveillance, California, 2004.

```
wnv <-read.table("http://www.medepi.net/data/wnv/wnv2004raw.txt",
                  sep = ",", header = TRUE, na.strings = ".")
str(wnv)  #display data set structure
head(wnv) #display first 6 lines
edit(wnv) #browse data frame
fix(wnv)  #browse with ability to edit (be careful!!!)
```

Convert non-standard dates to Julian dates

```
wnv$date.onset2 <- as.Date(wnv$date.onset, format="%m/%d/%Y")
wnv$date.tested2 <- as.Date(wnv$date.tested, format="%m/%d/%Y")
```

Display histogram of onset dates (epidemic curve)

```
hist(wnv$date.onset2, breaks= 26, freq=TRUE, col="slategray1")
```

Describe a continuous variable (e.g., age)

```
summary(wnv$age) # no standard deviation provided
range(wnv$age, na.rm=TRUE); mean(wnv$age, na.rm=TRUE)
median(wnv$age, na.rm=TRUE); sd(wnv$age, na.rm=TRUE)
```

Describe continuous variable, stratified by a categorical variable

```
tapply(wnv$age, wnv$sex,  mean, na.rm = TRUE)
tapply(wnv$age, wnv$county,  mean, na.rm = TRUE)
```

Display a continuous variable

```
hist(wnv$age, xlab="x", ylab="y", main="title", col="skyblue")
```

Describe a categorical variable (e.g., sex)

```
sex.tab <- xtabs(˜sex, data = wnv)
sex.dist <- prop.table(sex.tab)
cbind(sex.tab, sex.dist)
```

Display a categorical variable (e.g. sex)

```
barplot(sex.tab, col="pink", ylab="Frequency", main="title")
```

Re-code continuous variable to categorical (e.g., age)

```
wnv$age3 <- cut(wnv$age, breaks=c(0,45,65,100), right=FALSE)
age3.tab <- xtabs(˜age3, data = wnv)
age3.dist <- prop.table(age3.tab)
cbind(age3.tab, age3.dist)
```

Describe two categorical variables (e.g. sex and age)

```
sexage <- xtabs(˜sex + age3, data = wnv)
sexage
prop.table(sexage)    #joint distribution
prop.table(sexage, 1) #row distribution
prop.table(sexage, 2) #column distribution
```

Plot age vs sex distribution

```
barplot(sexage, legend.text=TRUE,
        xlab="Age", ylab="Frequency", main="title")
barplot(sexage, legend.text=TRUE, beside=TRUE,
        xlab="Age", ylab="Frequency", main="title")
barplot(t(sexage), legend.text=TRUE, ylim=c(0, 650),
        xlab="Sex", ylab="Frequency", main="title")
barplot(t(sexage), legend.text=TRUE, beside=TRUE, ylim=c(0, 300),
        xlab="Sex", ylab="Frequency", main="title")
```

Hypothesis testing using 2-way contingency tables

From the main menu select Packages > Install Package(s). Select CRAN mirror near you. Select `epitools` package.

```
library(epitools) #load 'epitools'; only needed once per session
tab.age3 <- xtabs(~age3 + death, data = wnv)
epitab(tab.age3)          #default is odds ratio
epitab(tab.age3, method = "riskratio")
prop.table(tab.age3, 1)   #display row distribution (2=column)
prop.test(tab.age3[,2:1]) #remember to reverse columns
chisq.test(tab.age3)      #Chi-square test
fisher.test(tab.age3)     #Fisher exact test
```

Graphical display of epidemiologic data

Histogram (continuous numbers or date objects)

```
hist(wnv$age, xlab="x", ylab="y", main="title", col="skyblue")
hist(wnv$date.onset2, breaks= 26, freq=TRUE, col="slategray1")
```

Bar chart (categorical variable)

```
barplot(table(wnv$sex), col="skyblue", xlab="Sex", ylab="Freq",
        main="title", legend = TRUE, ylim=c(0,600))
```

Stacked bar chart (2 or more categorical variables)

```
barplot(table(wnv$sex, wnv$age3), col=c("blue","green"),
        xlab="Sex", ylab="Freq", main="WNV Disease, Sex by Age",
        legend = TRUE, ylim=c(0,400))
```

Group bar chart (2 or more categorical variables):

```
barplot(table(wnv$sex, wnv$age3), beside=TRUE, xlab="Sex",
        ylab="Freq", main="Sex by Age", col=c("blue","green"),
        legend = TRUE, ylim=c(0,250))
```

Proportion bar chart (2 or more categorical variables)

```
sexage <- xtabs(˜sex + age3, data = wnv)
barplot(prop.table(sexage, 2), xlab="Sex", ylab="Proportion",
        main="WNV Disease, Sex by Age", col=c("blue","green"),
        legend = TRUE, ylim=c(0,1.2))
```

Time series (single *x* values vs. single *y* values)

United States measles surveillance data

```
measles <- read.table("http://www.medepi.net/data/measles.txt",
           sep="", header=TRUE)
str(measles); head(measles)
plot(measles$year, measles$cases, type="l", lwd=2, col="navy")
plot(measles$year, measles$cases, type="l", lwd=2, log="y")
```

Time series (multiple *x* values vs multiple *y* values)

United State AIDS and hepatitis B surveillance data

```
aids <- read.table("http://www.medepi.net/data/aids.txt",
        sep="", header=TRUE, na.strings=".")
hepb <- read.table("http://www.medepi.net/data/hepb.txt",
        sep="", header=TRUE)
years <- cbind(aids$year, hepb$year)
cases <- cbind(aids$cases, hepb$cases)
matplot(years, cases, type="l", lwd=2, col=1:2, main="title")
legend(x=1980, y=80000, legend= c("AIDS","Hepatitis B"),
       lty=1:2, col=1:2, lwd=2)
```

Working with dates and times

Convert non-standard dates to standard Julian dates

```
dates <- c("11-02-1959","1959Nov02","November 2, 1959")
jdates<-as.Date(dates, format=c("%m-%d-%Y","%Y%b%d","%B %d, %Y"))
jdates; julian(jdates)
```

Converting non-standard dates and times to R date-time object:

```
dtim <- c("4/19/1940 12:30 AM", "4/18/1940 9:45 PM")
std.dt <- strptime(dtim, format="%m/%d/%Y %I:%M %p")
std.dt
```

Try `?strptime` to see all format options.

Manually creating an epidemic curve

Single variable

```
labs <- c("Sun", "Mon","Tue","Wed","Thu","Fri", "Sat")
cases <- c(0, 25, 15, 5, 10, 20, 0)
names(cases) <- labs
barplot(cases, space=0, col="skyblue", xlab="Day", ylab="Cases",
        main="Title")
```

Single variable—Change *x*-axis labels to perpendicular

```
xv <- barplot(cases, space=0, col="red", xlab="Day",
             ylab="Cases", main="Title", axisnames=FALSE)
axis(side=1, at=xv, labels=labs, las=2)
```

Stratified by second variable

```
male.cases <- c(0, 15, 10, 3, 5, 5, 0)
female.cases <- c(0, 10, 5, 2, 5, 15, 0)
cases2 <- rbind(Male = male.cases, Female = female.cases)
colnames(cases2) <- labs
xv <- barplot(cases2, space=0, col=c("blue", "green"),
              xlab="Day", ylab="Cases", main="Title",
              axisnames=FALSE, legend.text=TRUE, ylim=c(0, 30))
axis(side=1, at=xv, labels=labs, las=2)
```

Running batch jobs

```
source("c:/myoutbreak/job01.R") #run program file called job01.R
```

Creating output log files

From within job01.R program file

```
x <- 1:5
y <- x^2
```

Sink printed objects to log file

```
sink("c:/temp/job.log")
print(x)
sink()
```

Capture output without requiring print command

```
capture.output(cbind(x, y),
               file="c:/temp/job.log", append=TRUE)
```

Multivariable analysis

Logistic regression (binomial data: cohort, case-control)

Using WNV data with `age3` variable created previously:

```
mod1 <- glm(death ~ age3, family=binomial, data=wnv)
summary(mod1) #full results
exp(mod1$coef) #calculate odds ratio
mod2 <- glm(death ~ age3 + sex, family=binomial, data=wnv)
summary(mod2) #full results
exp(mod2$coef) #calculate odds ratio
```

Conditional logistic regression (matched case-control)

Here is a case-control study of myocardial infarction (Kleinbaum 2002): one case was matched to two controls on age, race, and sex.

```
library(survival) #load survival package
chd <- read.table("http://www.medepi.net/data/chd.txt", sep=",",
header=TRUE)
head(chd)
chd$mi2 <- ifelse(chd$mi=="Yes", 1, 0) #re-code case status
mod1 <- clogit(mi2~smk+strata(match), data=chd)
summary(mod1)
mod2 <- clogit(mi2~smk+sbp+strata(match), data=chd)
summary(mod2)
mod3 <- clogit(mi2~smk+sbp+ecg+strata(match), data=chd)
summary(mod3)
anova(mod1,mod2,mod3, test="Chisq") #compare nested models
```

APPENDIX C

Programming and creating R functions

"Good programmers write good code, great programmers borrow good code."

R is a comprehensive and powerful programming language. In this section we briefly summarize how to use R for introductory programming, including writing and executing functions.

C.1 Basic programming

Basic epidemiologic programming in R is just a list of R expressions, that are collected and executed as a batch job. The list of R expressions in an ASCII text file with a .R extension. In RStudio, from the main menu, select File → New → R Script. This script file will be saved with an .R extension. This script file can be edited and executed within RStudio. Alternatively, we can edit this file using our favorite text editor (e.g., GNU Emacs).

What are the characteristics of good R programming?

- Use a good text editor (or RStudio) for programming
- Organize batch jobs into numbered sequential files (e.g., job01.R)
- Avoid graphical menu-driven approaches

First, use a good text editor (or RStudio) for programming. Each R expression will span one or more lines. Although one could write and submit each line at the R console, this approach is inefficient and not recommended. Instead, type the expressions into your favorite text editor and save with a *.R* extension. Then, selected expressions or the whole file can be executed in R. Use the text editor that comes with R, or text editors customized to work with R (e.g., RStudio, Emacs with ESS).

Second, we organize batch jobs into sequential files. Data analysis is a series of tasks involving data entry, checking, cleaning, analysis, and reporting. Although data analysts are primarily involved in analysis and reporting, they may be involved in earlier phases of data preparation. Regardless of stage of involvement, data analysts should organize, conduct, and document their analytics tasks and batch jobs in chronological order. For example, batch jobs might be named as follows: job01-cleaning.R, job02-recoding.R, job03-descriptive.R, job04-logistic.R, etc. Naming the program file has two components: 'jobs' represent major tasks and are always numbered in chronological order (job01-*.R, job02-*.R, etc.); and a brief descriptor can be appended to the first component of the file name (job01-recode-data.R, job02-bivariate-analysis.R).

If one needs to repeat parts of a previous job, then add new jobs, not edit old jobs. This way our analysis can always be reviewed, replicated, and audited exactly in order it was conducted. We avoid editing earlier jobs. If we edit previous jobs, then we must rerun all subsequent jobs in chronological order.

Third, we avoid graphical, menu-driven approaches. While this is a tempting approach, our work cannot be documented, replicated, and audited. The best approach is to collect R expressions into batch jobs and run them using the `source` function.

C.2 Intermediate programming

The next level of R programming involves (1) implementing control flow (decision points); (2) implementing dependencies in calculations or data manipulation; and (3) improving execution efficiency,

C.2.1 Control statements

Control flow involves one or more decision points. A simplest decision point goes like this: if a condition is `TRUE`, do {this} and then continue; if it is `FALSE`, do not do {this} and then continue. When R continues, the next R expression can be any valid expression, including another decision point.

C.2.1.1 The `if` function

We use the `if` function to implement single decision points.

`if(TRUE)` {*`execute these R expressions`*}

If the condition if false, R skips the bracketed expression and continues executing subsequent lines. Study this example:

```
> x <- c(1, 2, NA, 4, 5)
> y <- c(1, 2, 3, 4, 5)
> if(any(is.na(x))) {x[is.na(x)] <- 999}
> x
[1]   1   2 999   4   5
> if(any(is.na(y))) {y[is.na(y)] <- 999}
> y
[1] 1 2 3 4 5
```

The first `if` condition evaluated to `TRUE` and the missing value was replaced. The second `if` condition evaluated to `FALSE` and the bracketed expressions were not evaluated.

C.2.1.2 The `else` functions

Up to now the `if` condition had only one possible response. If there are two, mutually exclusive possible responses, add one `else` statement:

```
if(TRUE) {
  execute these R expressions
} else {
  execute these R expressions
}
```

Here is an example:

```
> x <- c(1, 2, NA, 4, 5);
> y <- c(1, 2, 3, 4, 5)
> if(any(is.na(x))) {
+   x[is.na(x)] <- 999; cat("NAs replaced\n")
+ } else {cat("No missing values; no replacement\n")}
NAs replaced
> if(any(is.na(y))) {
+   y[is.na(y)] <- 999; cat("NAs replaced\n")
+ } else {cat("No missing values; no replacement\n")}
No missing values; no replacement
> x
[1]   1   2 999   4   5
> y
[1] 1 2 3 4 5
> y
[1] 1 2 3 4 5
```

Therefore, use the `if` and `else` combination if one needs to evaluat of one of two possible collection of R expressions.

If one needs to evaluate *possibly* one of two possible collection of R expressions then use the following pattern:

```
if(TRUE) {
  execute these R expressions
} else if(TRUE) {
  execute these R expressions
}
```

The `if` and `else` functions can be combined to achieve any desired control flow.

C.2.1.3 The “short circuit” logical operators

The “short circuit” `&&` and `||` logical operators are used for control flow in `if` functions. If logical vectors are provided, only the first element of each vector is used. Therefore, for element-wise comparisons of 2 or more vectors, use the `&` and `|` operators but not the `&&` and `||` operators (discussed in Chapter 2). For `if` function comparisons use the `&&` and `||` operators.

Suppose we want to square the elements of a numeric vector but not if it is a matrix.

```
> x <- 1:5
> y <- matrix(1:4, 2, 2)
> if(is.numeric(x) && !is.matrix(x)) {
+   x^2
+ } else cat("Either not numeric or is a matrix\n")
[1]  1  4  9 16 25
> if(!is.matrix(y) && is.numeric(y)) {
+   y^2
+ } else cat("Either not numeric or is a matrix\n")
Either not numeric or is a matrix
```

The `&&` and `||` operators are called “short circuit” operators because not all its arguments may be evaluated: moving from left to right, only sufficient arguments are evaluated to determine if the `if` function should return TRUE or FALSE. This can save considerable time if some the arguments are complex functions that require significant computing time to evaluate to either TRUE or FALSE. In the previous example, because `!is.matrix(y)` evaluates to FALSE, it was not necessary to evaluate `is.numeric(y).`

C.2.2 Vectorized approach

An important advantage of R is the availability of functions that perform vectorized calculations. For example, suppose we wish to add to columns of a matrix. Here is one approach:

```
> tab <- matrix(1:12, 3, 4)
> tab
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> colsum <- tab[,1]+tab[,2]+tab[,3]+tab[,4]
> colsum
[1] 22 26 30
```

However, this can be accomplished more efficiently using the `apply` function:

```
> colsum2 <- apply(tab, 1, sum)
> colsum2
[1] 22 26 30
```

In general, we want to use these types of functions (e.g., `tapply`, `sweep`, `outer`, `mean`, etc.) because they have been optimized to performed vectorized calculations.

C.2.2.1 The `ifelse` function

The `ifelse` function is a vectorized element-wise implementation of the `if` and `else` functions. We demonstrate using the practical example of recoding a 2-level variable.

```
> sex <- c("M", NA, "F", "F", NA, "M", "F", "M")
> sex2 <- ifelse(sex=="M", "Male", "Female")
> sex2
[1] "Male"  NA  "Female" "Female"  NA  "Male"  "Female"  "Male"
```

If an element of `sex` contains ”M” (TRUE), it is recoded to ”Male” (in `sex2`), and otherwise (FALSE) it is recoded to ”Female”. This assumes that there are only ”M”s and ”F”s in the data vector.

C.2.3 Looping

Looping is a common programming approach that is discouraged in R because it is inefficient. It is much better to conduct vectorized calculations using existing functions. For example, suppose we want to sum a numeric vector.

```
> x <- 1:10
> xsum <- 0
> for (i in 1:10){
+   xsum <- xsum + x[i]
+ }
```

```
> xsum
[1] 55
```

A much better approach is to use the `sum` function:

```
> sum(x)
[1] 55
```

Unless it is absolutely necessary, we avoid looping.

Looping is necessary when (1) there is no R function to conduct a vectorized calculation, and (2) when the result of an element depends on the result of a preceding element which may not be known beforehand (e.g., when it is the result of a random process).

C.2.3.1 The `for` function

The previous example was a `for` loop. Here is the syntax:

```
  for (i in somevector{
  do some calcuation with ith element of somevector
}
```

In the `for` function R loops and uses the ith element of *somevector* either directly or indirectly (e.g., indexing another vector). Here is using the vector directly:

```
> for(i in 1:3){
+   cat(i^2,"\n")
+ }
1
4
9
```

The `letters` contain the American English alphabet. Here we use an integer vector for indexing `letters`:

```
> for(i in 1:3){
+   cat(letters[i],"\n")
+ }
a
b
c
```

Somevector can be any vector:

```
> kids <- c("Tomasito", "Luisito", "Angela")
> for (i in kids) {print(i)}
[1] "Tomasito"
[1] "Luisito"
[1] "Angela"
```

C.2.3.2 The `while` function

The `while` function will continue to evaluate a collection of R expressions while a condition is true. Here is the syntax:

```
while(TRUE) {
  execute these R expressions
}
```

Here is a trivial example:

```
> x <- 1; z <- 0
> while(z < 5){
+   show(z)
+   z <- z + x
+ }
[1] 0
[1] 1
[1] 2
[1] 3
[1] 4
```

The `while` function is used for optimization functions that are converging to a numerical value.

C.2.3.3 The `break` and `next` functions

The `break` expression will break out of a `for` or `while` loop if a condition is met, and transfers control to the first statement outside of the inner-most loop. Here is the general syntax:

```
  for (i in somevector{
   do some calcuation with ith element of somevector
  if(TRUE) break
}
```

The `next` expression halts the processing of the current iteration and advances the looping index. Here is the general syntax:

```
  for (i in somevector{
   do some calcuation with ith element of somevector
  if(TRUE) next
}
```

Both `break` and `next` apply only to the innermost of nested loops.

C.2.3.4 double `for`

In the next example we nest two `for` loop to generate a multiplication table for the integers 6 to 10:

```
> x <- 6:10
> mtab <- matrix(NA, 5, 5)
> rownames(mtab) <- x
> colnames(mtab) <- x
> for(i in 1:5){
+   for(j in 1:5){
+     mtab[i, j] <- x[i]*x[j]
+   }
+ }
> mtab
    6  7  8  9  10
6  36 42 48 54  60
7  42 49 56 63  70
8  48 56 64 72  80
9  54 63 72 81  90
10 60 70 80 90 100
```

C.3 Writing R functions

Writing R functions involves three steps:

- Prepare inputs
- Do calculations
- Collect results

The best way to learn these steps is to incorporate them into our regular R programming. For example, suppose we are writing R code to calculate the odds ratio from a 2×2 table with the appropriate format. For this we will use the Oswego data set available from the `epitools` package.

```
> ## Prepare inputs
> library(epitools)
> data(oswego)
> tab1 = xtabs(~ ill + spinach, data = oswego)
> tab1
   spinach
ill  N  Y
  N 12 17
  Y 20 26
> aa = tab1[1, 1]
> bb = tab1[1, 2]
> cc = tab1[2, 1]
> dd = tab1[2, 2]
>
> ## Do calculations
> crossprod.OR = (aa*dd)/(bb*cc)
>
> ## Collect results
> list(data = tab1, odds.ratio = crossprod.OR)
$data
   spinach
ill  N  Y
  N 12 17
  Y 20 26

$odds.ratio
[1] 0.9176471
```

Now that we are familiar of what it takes to calculate an odds ratio from a 2-way table we can convert the code into a function and load it at the R console. Here is new function:

```
myOR = function(x){
  ## Prepare input
  ## x = 2x2 table amenable to cross-product
  aa = x[1, 1]
  bb = x[1, 2]
  cc = x[2, 1]
  dd = x[2, 2]

  ## Do calculations
  crossprod.OR = (aa*dd)/(bb*cc)

  ## Collect results
  list(data = x, odds.ratio = crossprod.OR)
}
```

Now we can test the function:

```
> tab.test = xtabs(~ ill + spinach, data = oswego)
> myOR(tab.test)
$data
   spinach
ill  N  Y
  N 12 17
  Y 20 26
```

```
$odds.ratio
[1] 0.9176471
```

C.3.1 Arguments default values

Now suppose we wish to add calculating a 95% confidence interval to this function. We will use the following normal approximation standard error formula for an odds ratio:

$$SE[\log(OR)] = \sqrt{\frac{1}{A_1} + \frac{1}{B_1} + \frac{1}{A_0} + \frac{1}{B_0}}$$

And here is the $(1-\alpha)\%$ confidence interval:

$$OR_L, OR_U = \exp\{log(OR) \pm Z_{\alpha/2} SE[\log(OR)]\}$$

Here is the improved function:

```
myOR2 = function(x, conf.level){
  ## Prepare input
  ## x = 2x2 table amenable to cross-product
  aa = x[1, 1]
  bb = x[1, 2]
  cc = x[2, 1]
  dd = x[2, 2]
  if(missing(conf.level)) stop("Must specify confidence level")
  Z <- qnorm((1 + conf.level)/2)

  ## Do calculations
  logOR <- log((aa*dd)/(bb*cc))
  SE.logOR <- sqrt(1/aa + 1/bb + 1/cc + 1/dd)
  OR <- exp(logOR)
  CI <- exp(logOR + c(-1, 1)*Z*SE.logOR)

  ## Collect results
  list(data = x, odds.ratio = OR, conf.int = CI)
}
```

Notice that `conf.level` is a new argument, but with no default value. If a user forgets to specify a default value, the following line handles this possibility:

```
if(missing(conf.level)) stop("Must specify confidence level")
```

Now we test this function:

```
> tab.test = xtabs(~ ill + spinach, data = oswego)
> myOR2(tab.test)
Error in myOR2(tab.test) : Must specify confidence level
```

```
> myOR2(tab.test, 0.95)
$data
   spinach
ill  N  Y
  N 12 17
  Y 20 26

$odds.ratio
[1] 0.9176471

$conf.int
[1] 0.3580184 2.3520471
```

If an argument has a usual value, then specify this as an argument default value:

```
myOR3 = function(x, conf.level = 0.95){
  ## Prepare input
  ## x = 2x2 table amenable to cross-product
  aa = x[1, 1]
  bb = x[1, 2]
  cc = x[2, 1]
  dd = x[2, 2]
  Z <- qnorm((1 + conf.level)/2)

  ## Do calculations
  logOR <- log((aa*dd)/(bb*cc))
  SE.logOR <- sqrt(1/aa + 1/bb + 1/cc + 1/dd)
  OR <- exp(logOR)
  CI <- exp(logOR + c(-1, 1)*Z*SE.logOR)

  ## Collect results
  list(data = x, odds.ratio = OR, conf.int = CI)
}
```

We test our new function:

```
> tab.test = xtabs(~ ill + spinach, data = oswego)
> myOR3(tab.test)
$data
   spinach
ill  N  Y
  N 12 17
  Y 20 26

$odds.ratio
[1] 0.9176471
```

```
$conf.int
[1] 0.3580184 2.3520471

> myOR3(tab.test, 0.90)
$data
   spinach
ill  N  Y
  N 12 17
  Y 20 26

$odds.ratio
[1] 0.9176471

$conf.int
[1] 0.4165094 2.0217459
```

C.3.2 Passing optional arguments using the `...` *function*

On occasion we will have a function nested inside one of our functions and we need to be able to pass optional arguments to this nested function. This commonly occurs when we write functions for customized graphics but only wish to specify some arguments for the nested function and leave the remaining arguments optional. For example, consider this function:

```
myplot = function(x, y, type = "b", ...){
  plot(x, y, type = type, ...)
}
```

When using `myplot` one only needs to provide `x` and `y` arguments. The `type` option has been set to a default value of ”b”. The `...` function will pass any optional arguments to the nested `plot` function. Of course, they optional arguments must be valid options for `plot` function.

C.4 Advanced topics

C.4.1 Lexical scoping

The variables which occur in the body of a function can be divided into three classes; formal parameters, local variables and free variables. The *formal parameters* of a function are those occurring in the argument list of the function. Their values are determined by the process of binding the actual function arguments to the formal parameters. *Local variables* are those whose values are determined by the evaluation of expressions in the body of the functions. Variables which are not formal parameters or local variables are called *free variables*. Free variables become local variables if they are assigned to. Consider the following function definition.

```
f <- function(x){
  y <- 2*x
  print(x)
  print(y)
  print(z)
}
```

In this function, `x` is a formal parameter, `y` is a local variable and `z` is a free variable. In R the free variable bindings are resolved by first looking in the environment in which the function was created. This is called `lexical scope`. If the free variable is not defined there, R looks in the enclosing environment. For the function `f` this would be the global environment (workspace).

To understand the implications of lexical scope consider the following:

```
> rm(list = ls())
> ls()
character(0)
> f <- function(x){
+     y <- 2*x
+     print(x)
+     print(y)
+     print(z)
+ }
> f(5)
[1] 5
[1] 10
Error in print(z) : object 'z' not found
> z = 99
> f(5)
[1] 5
[1] 10
[1] 99
```

In the `f` function `z` is a free variable. The first time `f` is executed `z` is not defined in the function. R looks in the enclosing environment and does not find a value for `z` and reports an error. However, when a object `z` is created in the global environment, R is able to find it and uses it.

Lexical scoping is convenient because it allows nested functions with free variables to run provided the variable has been defined in an enclosing environment. This convenience becomes obvious when one writes many programs. However, there is a danger: an unintended free variable many find an unintended value in an enclosing environment. This may go undetected because no error is reported. This can happen when there are many objects in the workspace from previous sessions. A good habit is to clear the workspace of all objects at the beginning of every session.

Here is another example from the R introductory manual[1]. Consider a function called `cube`.

```
cube <- function(n) {
  sq <- function() n*n
  n*sq()
}
```

The variable `n` in the function `sq` is not an argument to that function. Therefore it is a free variable and the scoping rules must be used to ascertain the value that is to be associated with it. Under static scope (S-Plus) the value is that associated with a global variable named `n`. Under lexical scope (R) it is the parameter to the function cube since that is the active binding for the variable `n` at the time the function `sq` was defined. The difference between evaluation in R and evaluation in S-Plus is that S-Plus looks for a global variable called `n` while R first looks for a variable called `n` in the environment created when cube was invoked.

```
## first evaluation in S
S> cube(2)
Error in sq(): Object "n" not found
Dumped
S> n <- 3
S> cube(2)
[1] 18

## then the same function evaluated in R
R> cube(2)
[1] 8
```

---

[1] `http://cran.r-project.org/doc/manuals/r-release/R-intro.pdf`

Solutions

Problems of Chapter 1

1.1 To download R go to `http://cran.r-project.org/` and follow the instructions for your operating system. Once installed, start R. The computer file path to the workspace file, `.RData`, is obtained using the `getwd` function (see Table 1.3 on page 12 for more useful functions).

```
> getwd()
[1] "/home/tja/Data/R/home"
```

This displayed in R the file path on the computer. To see the actual `.RData` file, we must enable our computer system to view hidden files and then use the computer's program for viewing files. This is useful to know in case we want to physically move the file to another location.

1.2 To list the R packages currently loaded, use the `search` function.

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

Alternatively, use the `searchpaths` function to see the file paths.

```
> searchpaths()
[1] ".GlobalEnv"                       "/usr/lib64/R/library/stats"
[3] "/usr/lib64/R/library/graphics"  "/usr/lib64/R/library/grDevices"
[5] "/usr/lib64/R/library/utils"     "/usr/lib64/R/library/datasets"
[7] "/usr/lib64/R/library/methods"   "Autoloads"
[9] "/usr/lib64/R/library/base"
```

1.3

```
ls()
rm(list=ls())
```

The `ls` function returns a character vector of object names. This character vector can be used as the `list` argument in the `rm` function to remove all the objects. Instead of `ls` we could have used the `objects` function.

1.4

```
inches <- 1:12
centimeters <- inches*2.54
cbind(inches, centimeters)
```

1.5

```
celsius <- c(0, 100)
fahrenheit <- ((9/5)*celsius) + 32
fahrenheit
```

Notice that we used a numeric vector so that the calculation requires fewer steps. This is an example of a vectorized (or spreadsheat-like) operation.

1.6

```
celsius <- seq(0, 100, 5)
celsius
fahrenheit <- ((9/5)*celsius) + 32
fahrenheit
```

Notice that we used a numeric vector so that the calculation requires fewer steps. This is an example of a vectorized (or spreadsheat-like) operation.

1.7 Suppose you weigh 150 lbs, what is your weight in kilograms? Hint: Remember dimensional analysis?

$$150\,\mathrm{lb} \times \frac{1\,\mathrm{kg}}{2.2\,\mathrm{lb}} = \frac{150\,\mathrm{kg}}{2.2}$$

Suppose your height is 5 feet 8 inches, What is your height in meters? (5’ 8” = 5’ + 8/12” = 5.75 feet)

$$5.75\,\mathrm{ft} \times \frac{1\,\mathrm{m}}{3.3\,\mathrm{ft}} = \frac{5.75\,\mathrm{m}}{3.3}$$

```
mywt.lb <- 150
myht.ft <- 5.75
mywt.kg <- mywt.lb/2.2
myht.m <- myht.ft/3.3
bmi <- mywt.kg/myht.m^2
bmi
```

1.8

```
> 7/2   #divide
[1] 3.5
```

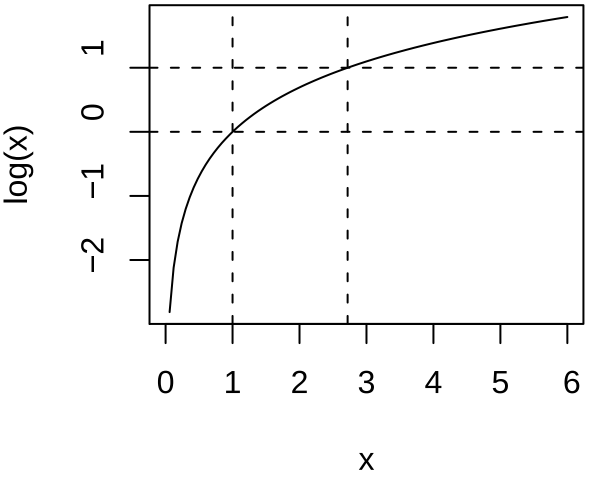


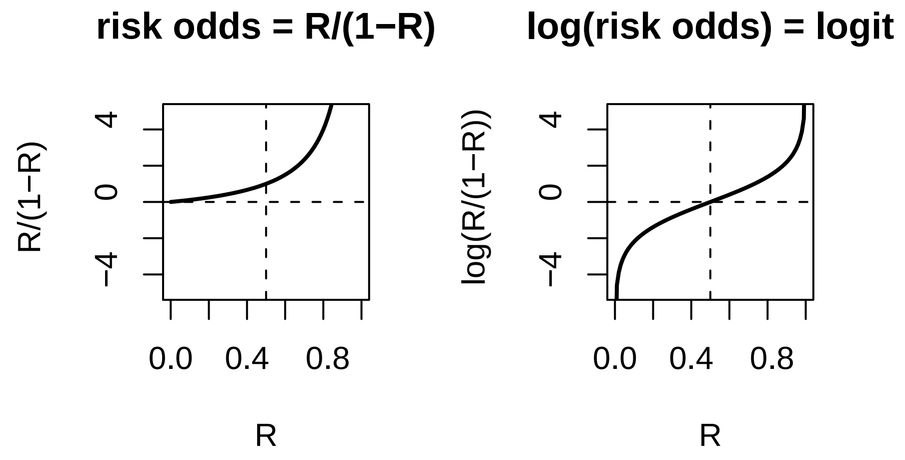
Fig. C.1 Plot of $y = \log_e(x)$

```
> 7%/%2 #integer divide
[1] 3
> 7%%2  #modulus = remainder
[1] 1
```

1.9 See Figure C.1. The number *e* is a very special number. When we take the logarithm of a number using base *e* we map the values [0, $+\infty$) into ($-\infty$, $+\infty$), where the $\log_e(1) = 0$. More specifically, the number range [0,1] maps into ($-\infty$, 0], and [1, $+\infty$) maps into [0, $+\infty$). Of note, the $\log_e(e) = 1$. In epidemiology, disease counts and physical measurements (e.g., weight) have the asymmetric range $[0, +\infty)$. The natural logarithm transformation allows us to work with values between 0 and 1, in a sense unbounding the left tail distribution.

1.10 See Figure C.2. The logit transformation is a *double* transformation. First, the odds tranformation ($R/(1-R)$) unbounds the probabilities near 1; second, the natural logarithm of the log-odds, or the *logit transformation*, ($\log(odds)$) unbounds the probabilities near 0. In other words, the logit transformation maps the numeric range $[0,1]$ into ($-\infty$, $+\infty$), where the $\log(0.5/(1-0.5)) = 0$. The logit transformation allows us to work with probabilities that, of course, have the range $[0,1]$.

1.11



risk odds = R/(1−R)

log(risk odds) = logit

Fig. C.2 The logit transformation is a double transformation. First, the odds tranformation ($R/(1-R)$) unbounds the probabilities near 1; second, the logit transformation ($\log(odds)$) additionally unbounds the probabilities near 0.

```
n <- 365
per.act.risk <- c(0.5, 1, 5, 6.5, 10, 30, 50, 67)/10000
risks <- 1-(1-per.act.risk)ˆn
risks
##label risks (optional)
act <- c("IOI", "ROI", "IPVI", "IAI", "RPVI", "PNS", "RAI", "IDU")
names(risks) <- act
risks
```

1.12 For this problem I put the following code into an ASCII text file named job01.R:

```
n <- 365
per.act.risk <- c(0.5, 1, 5, 6.5, 10, 30, 50, 67)/10000
risks <- 1-(1-per.act.risk)ˆn
risks
##label risks (optional)
act <- c("IOI", "ROI", "IPVI", "IAI", "RPVI", "PNS", "RAI", "IDU")
names(risks) <- act
risks
```

Here is what happened when I sourced it:

```
> source("/home/tja/Documents/courses/ph251d/jobs/job01.R")
> source("/home/tja/Documents/courses/ph251d/jobs/ph251d-chp1-job01.R", echo =
> n <- 365
> per.act.risk <- c(0.5, 1, 5, 6.5, 10, 30, 50, 67)/10000
> risks <- 1-(1-per.act.risk)^n
> risks
[1] 0.01808493 0.03584367 0.16685338 0.21126678 0.30593011
[6] 0.66601052 0.83951869 0.91402762

> ##label risks (optional)
> act <- c("IOI", "ROI", "IPVI", "IAI", "RPVI", "PNS", "RAI", "IDU")
> names(risks) <- act
> risks
       IOI        ROI       IPVI        IAI       RPVI        PNS
0.01808493 0.03584367 0.16685338 0.21126678 0.30593011 0.66601052
       RAI        IDU
0.83951869 0.91402762
```

Conclusion: running `source` alone runs R commands in a source file but does not echo the input and output to the screen unless `echo = TRUE`.

1.13 I ran this code in R (Linux):

```
sink("/home/tja/Documents/courses/ph251d/jobs/job01.log1a")
source("/home/tja/Documents/courses/ph251d/jobs/job01.R")
sink() #closes connection

sink("/home/tja/Documents/courses/ph251d/jobs/job01.log1b")
source("/home/tja/Documents/courses/ph251d/jobs/job01.R", echo = TRUE)
sink() #closes connection
```

The job01.log1a is empty. Here are the contents of job01.log1b:

```
> n <- 365
> per.act.risk <- c(0.5, 1, 5, 6.5, 10, 30, 50, 67)/10000
> risks <- 1-(1-per.act.risk)^n
> risks
[1] 0.01808493 0.03584367 0.16685338 0.21126678 0.30593011
[6] 0.66601052 0.83951869 0.91402762

> ##label risks (optional)
> act <- c("IOI", "ROI", "IPVI", "IAI", "RPVI", "PNS", "RAI", "IDU")
> names(risks) <- act
> risks
       IOI        ROI       IPVI        IAI       RPVI        PNS
0.01808493 0.03584367 0.16685338 0.21126678 0.30593011 0.66601052
       RAI        IDU
0.83951869 0.91402762
```

Conclusion: running the `sink` function sends what would normally go to the screen to a log file.

1.14 Sourcing job02.R at the R command line looks like this:

```
> source("/home/tja/Documents/courses/ph251d/jobs/job02.R")
[1] 0.01808493 0.03584367 0.16685338 0.21126678 0.30593011
[6] 0.66601052 0.83951869 0.91402762
```

Conclusion: The `source` function, without `echo = TRUE`, will not return anything to the screen unless the `show` (or `print`) function is used to "show" an R object. This make complete sense. If one is sourcing a file with thousands of R expressions, we do not want to see all those expressions, we only want to selected data objects with relevant results. Sinking a file only directs anything that would appear on the screen to a log file.

Problems of Chapter 2

2.1 n/a

2.2 See Table 2.1 on page 28.

2.3 We can index by position, by logical, and by name—if it exists.

2.4 Any R object component(s) that can be indexed, can be replaced.

2.5 Study and practice the following R code.

```
tab <- matrix(c(139, 443, 230, 502), nrow = 2, ncol = 2,
  dimnames = list("Vital Status" = c("Dead", "Alive"),
                  Smoking = c("Yes", "No")))
tab

# equivalent
tab <- matrix(c(139, 443, 230, 502), 2, 2)
dimnames(tab) <- list("Vital Status" = c("Dead", "Alive"),
                  Smoking = c("Yes", "No"))
tab

# equivalent
tab <- matrix(c(139, 443, 230, 502), 2, 2)
rownames(tab) <- c("Dead", "Alive")
colnames(tab) <- c("Yes", "No")
names(dimnames(tab)) <- c("Vital Status", "Smoking")
tab
```

2.6 Using the `tab` object from Solution 2.5, study and practice the following R code to recreate Table 2.38 on page 103.

```
rowt <- apply(tab, 1, sum)
tab2 <- cbind(tab, Total = rowt)
colt <- apply(tab2, 2, sum)
tab2 <- rbind(tab2, Total = colt)
names(dimnames(tab2)) <- c("Vital Status", "Smoking")
tab2
```

2.7 Using the `tab` object from Solution 2.5, study and practice the following R code to calculate row, column, and joint distributions.

```
# row distrib
rowt <- apply(tab, 1, sum)
rowd <- sweep(tab, 1, rowt, "/")
rowd

# col distrib
```

```
colt <- apply(tab, 2, sum)
cold <- sweep(tab, 2, colt, "/")
cold

# joint distrib
jtd <- tab/sum(tab); jtd
distr <- list(row.distribution = rowd,
              col.distribution = cold,
              joint.distribution = jtd)
distr
```

2.8 Using the `tab2` object from Solution 2.6, study and practice the following R code to recreate Table 2.39 on page 103. Note that the column distributions from Solution 2.7 can also be used.

```
risk = tab2[1,1:2]/tab2[3,1:2]
risk.ratio <- risk/risk[2]
odds <- risk/(1-risk)
odds.ratio <- odds/odds[2]
ratios <- rbind(risk, risk.ratio, odds, odds.ratio)
ratios
```

Interpretation: The risk of death among non-smokers is higher than the risk of death among smokers, suggesting that there may be some confounding.

2.9 Implement analysis below.

```
wdat = read.table("http://www.medepi.net/data/whickham-engl.txt",
                  sep = ",", header = TRUE)
str(wdat)
wdat.vas = xtabs(˜Vital.Status + Age + Smoking, data = wdat)
wdat.vas
wdat.tol.vas = apply(wdat.vas, c(2, 3), sum)
wdat.risk.vas = sweep(wdat.vas, c(2, 3), wdat.tot.vas, "/")
round(wdat.risk.vas, 2)
```

Here are the final results:

```
> round(wdat.risk.vas, 2)
, , Smoking = No

            Age
Vital.Status 18-24 25-34 35-44 45-54 55-64 65-74  75+
       Alive  0.98  0.97  0.94  0.85  0.67  0.22 0.00
       Dead   0.02  0.03  0.06  0.15  0.33  0.78 1.00

, , Smoking = Yes

            Age
```

```
Vital.Status 18-24 25-34 35-44 45-54 55-64 65-74  75+
       Alive  0.96  0.98  0.87  0.79  0.56  0.19 0.00
       Dead   0.04  0.02  0.13  0.21  0.44  0.81 1.00
```

Interpretation: The risk of death is not larger in non-smokers, in fact it is larger among smokers in older age groups..

2.10 First, look at the data set at `http://www.medepi.net/data/syphilis89c.txt`. Then read in.

```
std <- read.table("http://www.medepi.net/data/syphilis89c.txt",
                  head = TRUE, sep = ",")
str(std)
head(std)
lapply(std, table)
```

Creating 3-D array without attaching `std` data frame.

```
table(std$Race, std$Age, std$Sex)
xtabs(~ Race + Age + Sex, data = std)
```

Now repeat attaching `std` data frame using `attach` function. Study the differences.

```
attach(std)
table(Race, Age, Sex)
xtabs(~ Race + Age + Sex)
detach(std)
```

2.11

```
tab.ars <- table(std$Age, std$Race, std$Sex)
# 2-D tables
tab.ar <- apply(tab.ars, c(1, 2), sum); tab.ar
tab.as <- apply(tab.ars, c(1, 3), sum); tab.as
tab.rs <- apply(tab.ars, c(2, 3), sum); tab.rs

# 1-D tables
tab.a <- apply(tab.ars, 1, sum); tab.a
tab.r <- apply(tab.ars, 2, sum); tab.r
tab.s <- apply(tab.ars, 3, sum); tab.s
```

2.12 For this example, we'll choose one 3-D array.

```
tab.ars <- table(std$Age, std$Race, std$Sex)
# row distrib
rowt <- apply(tab.ars, c(1, 3), sum)
rowd <- sweep(tab.ars, c(1, 3), rowt, "/"); rowd
#confirm
apply(rowd, c(1, 3), sum)
```

```
# col distrib
colt <- apply(tab.ars, c(2, 3), sum)
cold <- sweep(tab.ars, c(2, 3), colt, "/"); cold
#confirm
apply(cold, c(2, 3), sum)

# joint distrib
jtt <- apply(tab.ars, 3, sum)
jtd <- sweep(tab.ars, 3, jtt, "/"); jtd
#confirm
apply(jtd, 3, sum)

distr <- list(row.distribution = rowd,
              col.distribution = cold,
              joint.distribution = jtd)
distr
```

2.13 It is a good idea to understand how the `rep` function works with two vectors:

```
> rep(4:6, 1:3)
[1] 4 5 5 6 6 6
```

We can see that the second vector determines the frequency of the first vector elements. Now use this understanding with the syphilis data.

```
sdat89b <- read.csv("http://www.medepi.net/data/syphilis89b.txt")
str(sdat89b)
Sex <- rep(sdat89b$Sex, sdat89b$Freq)
Race <- rep(sdat89b$Race, sdat89b$Freq)
Age <- rep(sdat89b$Age, sdat89b$Freq)
sdat89.df <- data.frame(Sex, Race, Age)
str(sdat89.df)
```

Discuss in class

2.14 Pending

Problems of Chapter 3

3.1 First, we recognize that this data frame contains aggregrate-level data, not individual-level data. Each row represents a unique covariate pattern, and the last field is the frequency of that pattern. Because the data frame only has a few rows here is one way:

```
Status <- rep(c("Dead", "Survived"), 4)
Treatment <- rep(c("Tobutamide", "Tobutamide",
                   "Placebo", "Placebo"), 2)
Agegrp <- c(rep("<55", 4), rep("55+", 4))
Freq <- c(8, 98, 5, 115, 22, 76, 16, 69)
dat <- data.frame(Status, Treatment, Agegrp, Freq)
dat
```

An alternative, and better way, is to create an array that reproduce the core data from Table 3.1 on page 107. Then we use the `data.frame` and `as.table` functions. Here we show a few ways to create this array object.

```
#answer 1
udat <- array(c(8, 98, 5, 115, 22, 76, 16, 69), dim =
  c(2, 2, 2),
  dimnames = list(Status = c("Dead", "Survived"),
                  Treatment = c("Tolbutamide", "Placebo"),
                  Agegrp = c("<55", "55+")))
dat <- data.frame(as.table(udat))
dat

#answer 2
Status <- rep(c("Dead", "Survived"), 4)
Treatment <- rep(rep(c("Tolbutamide", "Placebo"),
  c(2, 2)), 2)
Agegrp <- rep(c("<55", "55+"), c(4, 4))
Freq <- c(8, 98, 5, 115, 22, 76, 16, 69)
dat <- data.frame(Status, Treatment, Agegrp, Freq)
dat

#answer 2b, equivalent to 2a
dat <- data.frame(
  Status = rep(c("Dead", "Survived"), 4),
  Treatment = rep(rep(c("Tolbutamide", "Placebo"),
  c(2, 2)), 2),
  Agegrp = rep(c("<55", "55+"), c(4, 4)),
  Freq = c(8, 98, 5, 115, 22, 76, 16, 69)
)
dat
```

3.2 See Chapter.

3.3

```
adat <- read.table("http://www.medepi.net/data/aids.txt", header=TRUE,
sep="", na.strings=".")
head(adat)
plot(adat$year, adat$cases, type = "l", xlab = "Year", lwd = 2,
     ylab = "Cases", main = "Reported AIDS Cases in United States, 1980--2003"
```

3.4

```
mdat <- read.table("http://www.medepi.net/data/measles.txt",
  header=TRUE, sep="")
head(mdat)
plot(mdat$year, mdat$cases, type = "l", xlab = "Year", lwd = 2,
  ylab = "Cases",
  main = "Reported Measles Cases in United States, 1980--2003")
plot(mdat$year, mdat$cases, type = "l", xlab = "Year", lwd = 2,
  log = "y", ylab = "Cases",
  main = "Reported Measles Cases in United States, 1980--2003")
```

3.5

```
aids <- read.table("http://www.medepi.net/data/aids.txt", sep="",
                   header = TRUE, na.strings=".")
hepb <- read.table("http://www.medepi.net/data/hepb.txt", sep="",
                   header = TRUE)
matplot(hepb$year, cbind(hepb$cases,aids$cases),
    type = "l", lwd = 2, xlab = "Year", ylab = "Cases",
    main = "Reported cases of Hepatitis B and AIDS,
United States, 1980-2003")
legend(1980, 100000, legend = c("Hepatitis B", "AIDS"),
    lwd = 2, lty = 1:2, col = 1:2)
```

3.6 Answer to (a):

```
edat <- read.table("http://www.medepi.net/data/evans.txt",
  header = TRUE, sep="")
str(edat)
#
table(edat$chd)
edat$chd2 <- factor(edat$chd, levels = 0:1,
  labels = c("No", "Yes"))
table(edat$chd2)
#
table(edat$cat)
edat$cat2 <- factor(edat$cat, levels = 0:1,
  labels = c("Normal", "High"))
```

```
table(edat$cat2)
#
table(edat$smk)
edat$smk2 <- factor(edat$smk, levels = 0:1,
  labels = c("Never", "Ever"))
table(edat$smk2)
#
table(edat$ecg)
edat$ecg2 <- factor(edat$ecg, levels = 0:1,
  labels = c("Normal", "Abnormal"))
table(edat$ecg2)
#
table(edat$hpt)
edat$hpt2 <- factor(edat$hpt, levels = 0:1,
  labels = c("No", "Yes"))
table(edat$hpt2)
```

Answer to (b):

```
quantile(edat$age)
edat$age4 <- cut(edat$age, quantile(edat$age),
  right = FALSE, include.lowest = TRUE)
table(edat$age4)
```

Answer to (c):

```
hptnew <- rep(NA, nrow(edat))
normal <- edat$sbp<120 & edat$dbp<80
hptnew[normal] <- 1
prehyp <- (edat$sbp>=120 & edat$sbp<140) |
          (edat$dbp>=80 & edat$dbp<90)
hptnew[prehyp] <- 2
stage1 <- (edat$sbp>=140 & edat$sbp<160) |
          (edat$dbp>=90 & edat$dbp<100)
hptnew[stage1] <- 3
stage2 <- edat$sbp>=160 | edat$dbp>=100
hptnew[stage2] <- 4
edat$hpt4 <- factor(hptnew, levels=1:4,
  labels=c("Normal", "PreHTN", "HTN.Stage1", "HTN.Stage2"))
table(edat$hpt4)
```

Answer to (d):

```
table("Old HTN"=edat$hpt2, "New HTN"=edat$hpt4)
```

3.7

```
wdat <- read.table("http://www.medepi.net/data/wnv/wnv2004raw.txt",
  header=TRUE, sep=",", as.is=TRUE, na.strings=c(".","Unknown"))
```

```
str(wdat)
wdat$date.onset2 <- as.Date(wdat$date.onset, format="%m/%d/%Y")
wdat$date.tested2 <- as.Date(wdat$date.tested, format="%m/%d/%Y")
write.table(wdat, "c:/temp/wnvdat.txt", sep=",", row.names=FALSE)
```

3.8 See Appendix A.2 on page 184 for Oswego data dictionary.

a. Using RStudio plot the cases by time of onset of illness (include appropriate labels and title). What does this graph tell you? (Hint: Process the text data and then use the `hist` function.)

Plotting an epidemic curve with this data has special challenges because we have dates and times to process. To do this in R, we will create date objects that contain both the date and time for each primary event of interest: meal time, and onset time of illness. From this we can plot the distribution of onset times (epidemic curve). An epidemic curve is the distribution of illness onset times and can be displayed with a histogram. First, carefully study the Oswego data set at `http://www.medepi.net/data/oswego.txt`. We need to do some data preparation in order to work with dates and times. Our initial goal is to get the date/time data to a form that can be passed to R's `strptime` function for conversion in a date-time R object. To construct the following curve, study, and implement the R code that follows:

```
odat <- read.table("http://www.medepi.net/data/oswego.txt",
                   sep = "", header = TRUE, na.strings = ".")
str(odat)
head(odat)
## create vector with meal date and time
mdt <- paste("4/18/1940", odat$meal.time)
## convert into standard date and time
meal.dt <- strptime(mdt, "%m/%d/%Y %I:%M %p")
## create vector with onset date and time
odt <- paste(paste(odat$onset.date,"/1940",sep = ""), odat$onset.time)
## convert into standard date and time
onset.dt <- strptime(odt, "%m/%d/%Y %I:%M %p")
hist(onset.dt, breaks = 30, freq = TRUE)
```

b. Are there any cases for which the times of onset are inconsistent with the general experience? How might they be explained?

Now that we have our data frame in R, we can identify those subjects that correspond to minimum and maximum onset times. We will implement R code that can be interpreted as "which positions in vector Y correspond to the minimum values in Y?" We then use these position numbers to indexing the corresponding rows in the data frame.

```
##Generate logical vectors and identify 'which' position
min.obs.pos <- which(onset.dt==min(onset.dt,na.rm=T))
min.obs.pos
max.obs.pos <- which(onset.dt==max(onset.dt,na.rm=T))
max.obs.pos
##index data frame to display outliers
odat[min.obs.pos,]
odat[max.obs.pos,]
```

c. How could the data be sorted by illness onset times?

We can sort the data frame based values of one or more fields. Suppose we want to sort on illness status and illness onset times. We will use our onset.times vector we created earlier; however, we will need to convert it to "continuous time" in seconds to sort this vector. Study and implement the R code below.

```
onset.ct <- as.POSIXct(onset.dt)
odat2 <- odat[order(odat$ill, onset.ct), ]
odat2
```

d. Where possible, calculate incubation periods and illustrate their distribution with an appropriate graph. Use the `truehist` function in the MASS package. Determine the mean, median, and range of the incubation period.

```
##Calculate incubation periods
incub.dt <- onset.dt - meal.dt
library(MASS)  #load MASS package
truehist(as.numeric(incub.dt), nbins = 7, prob = FALSE,
         col = "skyblue", xlab = "Incubation Period (hours)")

##Calculate mean, median, range; remember to remove NAs
mean(incub.dt, na.rm = TRUE)
median(incub.dt, na.rm = TRUE)
range(incub.dt, na.rm = TRUE)
```

References

1. Centers for Disease Control and Prevention. Antiretroviral postexposure prophylaxis after sexual, injection-drug use, or other nonoccupational exposure to HIV in the United States: recommendations from the U.S. Department of Health and Human Services. MMWR Recomm Rep. 2005 Jan;54(RR-2):1–20. Available from: `http://www.cdc.gov/mmwr/preview/mmwrhtml/rr5402a1.htm`.
2. Olsen SJ, Chang HL, Cheung TYY, Tang AFY, Fisk TL, Ooi SPL, et al. Transmission of the severe acute respiratory syndrome on aircraft. N Engl J Med. 2003 Dec;349(25):2416–2422. Available from: `http://dx.doi.org/10.1056/NEJMoa031349`.
3. Rothman KJ. Epidemiology: An Introduction. 1st ed. Oxford University Press, USA; 2002.
4. Aragón TJ, Lichtensztajn DY, Katcher BS, Reiter R, Katz MH. Calculating expected years of life lost for assessing local ethnic disparities in causes of premature death. BMC Public Health. 2008;8:116. Available from: `http://dx.doi.org/10.1186/1471-2458-8-116`.
5. Glass RI, Svennerholm AM, Stoll BJ, Khan MR, Hossain KM, Huq MI, et al. Protection against cholera in breast-fed children by antibodies in breast milk. N Engl J Med. 1983 Jun;308(23):1389–1392. Available from: `http://dx.doi.org/10.1056/NEJM198306093082304`.
6. Jewell NP. Statistics for Epidemiology. 1st ed. Chapman and Hall/CRC; 2003.
7. Rothman KJ, Greenland S, Lash TL. Modern Epidemiology. Third edition ed. Lippincott Williams & Wilkins; 2008.
8. Centers for Disease Control & Prevention. Prevention and control of meningococcal disease. Recommendations of the Advisory Committee on Immunization Practices (ACIP). MMWR Recomm Rep. 2005 May;54(RR-7):1–21. Available from: `http://www.cdc.gov/mmwr/PDF/rr/rr5407.pdf`.
9. Selvin S. Statistical Tools for Epidemiologic Research. 1st ed. Oxford University Press, USA; 2011.
10. Selvin S. Survival Analysis for Epidemiologic and Medical Research (Practical Guides to Biostatistics and Epidemiology). 1st ed. Cambridge University Press; 2008.
11. Selvin S. Statistical Analysis of Epidemiologic Data (Monographs in Epidemiology and Biostatistics). 3rd ed. Oxford University Press, USA; 2004.
12. Selvin S. Epidemiologic Analysis: A Case-Oriented Approach. 1st ed. Oxford University Press, USA; 2001.
13. Abrams B, Carmichael S, Selvin S. Factors associated with the pattern of maternal weight gain during pregnancy. Obstet Gynecol. 1995 Aug;86(2):170–176.
14. Dalgaard P. Introductory Statistics with R (Statistics and Computing). 2nd ed. Springer; 2008.

15. Kleinbaum DG, Klein M, Pryor ER. Logistic Regression: A self-learning text. 2nd ed. Springer; 2002.