Tomás J. Aragón

Population Health Data Science with R

For my wife, Irene, and my children Ángela Isabel, Luis Miguel and Tomás Martín, whose unconditional love and support make the impossible possible.

For my staff, students, colleagues and community residents who teach, mentor and coach me every day.

Contents

List of Tables

List of Figures

Preface

We are writing this book to introduce R—a programming language and environment for statistical computing and graphics—to public health epidemiologists, health care data analysts, data scientists, statisticans, and others conducting population health analyses. Recent graduates come prepared with a solid foundation in epidemiological and statistical concepts and skills. However, what is sometimes lacking is the ability to implement *new* methods and approaches they did not learn in school. This is more apparent today with the emergence of data science and the new field of **population health data science** (PHDS)—*the art and science of transforming data into actionable knowledge to improve health.*[1]

The key word is **actionable knowledge**—*knowledge that informs, influences, or optmizes decision-making.* PHDS is a transdisciplinary field that integrates the expertise from public health and medicine, probability and statistics, computer science, decision sciences, health and behavioral economics, and human-centered design. PHDS is the future of public health data analysis and synthesis, and **knowledge integration**—*the management, synthesis and translation of knowledge into decision support systems to improve policy, practice, and—ultimately—population health.*

Why R? 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. We like to think of R as a set of extensible tools to implement one's analysis plan whether it is simple or complicated. 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, creativity, innovation, and confidence.

This book is divided into two parts. First, we cover how to process, manipulate, and operate on data in R. Most books cover this material briefly or leave it

[1] **Population health** is *a systems framework for studying and improving the health of populations through collective action and learning.*

for an appendix. We decided to dedicate a appropiate amount of space to this topic with the assumption that the average health analyst is not familiar with R and a good grounding in the basics will make the later chapters more understandable, and enable one to pick up any book on R and implement new methods quickly.

Second, We cover basic PHDS from an public health epidemiologic perspective. We build on the strengths of epidemiology (descriptive and analytic studies). However, in public health practice we need much more:

> To transform population health we need improve decision-making, problem solving, performance improvement, priority-setting, and resource allocation. The decision makers include patients, clients, policy makers, colleagues, and community stakeholders. Beyond "analysis" we need "synthesis" of data, information, and knowledge from diverse sources to promote better decision making in the setting of complex environments, limited information, multiple objectives, competing trade-offs, uncertainty, and time constraints.

How do we do this? Traditionally, epidemiologic methods are described as either *descriptive* (describing needs or generating hypotheses) or *analytic* (testing causal or intervention effects). Building upon this PHDS has five domains of analysis (Table 0.1).

TABLE 0.1: Population health data science analytic levels

Type of analysis	Description
1 Description	Surveillance and early detection of events
	Prevalence and incidence of risks and outcomes
2 Prediction	Early prediction and targeting of interventions
3 Causal inference	Discovery of new causal effects and pathways
	Estimation of intervention effectiveness
4 Simulation	Modeling for epidemiologic or decision insights
5 Optimization	Informing or optimizing decisions or efficiencies[2]

Each one of these analytic domains can "drive" decision-making (often referred

[2]For example, cost-benefit or cost-effectiveness analysis

to as "data-driven" decision-making). For PHDS, we will emphasize decision quality (DQ) in all decisions [1]. DQ is at the core of PHDS!

The field of data science is exploding! And the field of epidemiology—a public health basic science—is learning how to work effectively on transdisciplinary teams with mathematicians, statisticians, computer scientists, informaticians, clinicians, and subject matter experts. No individual has all the required technical expertise for data science. *Data science is a team sport.* What will we bring to the data science table? We hope this book will contribute to this answering this question.

Our 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 PHDS. We hope that more and more epidemiologists will embrace R and become population health data scientists, or at least, include R in their epidemiologic toolbox.

Finally, for population health leaders and data scientists, PHDS sharpens and supports **population health thinking** which is continuous improvement in:

1. probabilistic reasoning,
2. causal inference, and
3. decision quality.

From cognitive neuroscience we know that humans perform poorly at all three, especially in the face of complexity, uncertainty, competing trade-offs, confounding, mediation, or interaction [1–5]. In this book, we introduce **graphical models** (primarily Bayesian networks and variants[3]) as a unifying framework that connects the fields of probability and statistics, epidemiology and medicine, and decision and computer sciences in a profoundly elegant way. Once you experience the visual simplicity, analytic power, and profound insights from graphical models you will never look back. Population health thinking is the heart and soul of PHDS—making PHDS much more than the sum of its parts!

Tomás J. Aragón[4] & Wayne T. Enanoria
School of Public Health, Epidemiology
University of California, Berkeley, California

Department of Epidemiology and Biostatistics
University of California, San Francisco, California

[3] For example, causal graphs (also called directed acyclic graphs) and decision networks (also influence or relevance diagrams, or hybrid Bayesian networks)

[4] `https://taragonmd.github.io/` (blog) and `https://github.com/taragonmd` (GitHub)

About the Authors

Tomás J. Aragón is the health officer of the City and County of San Francisco, and director of the Population Health Division (PHD) at the San Francisco Department of Public Health.[5] Striving to embody and promote the universal values of dignity, equity, compassion and humility, he works to convene, connect and catalyze communities and institutions to transform narratives, policies and systems toward a sustainable culture of equity, healing and health for all people and our planet. As health officer, he exercises leadership and legal authority to protect and promote equity and health. As PHD director, he directs public health services. He teaches population health data science (with R) at the University of California, Berkeley School of Public Health. He maintains a public health blog with a practice-based focus on population health equity, leadership, lean,[6] and data science at https://taragonmd.github.io.

[5] https://www.sfdph.org

[6] Lean is systematically developing people to solve problems and consuming the fewest possible resources *while* continuously improving processes to provide value to community members and prosperity to society. To learn more visit https://www.lean.org/.

Stylistic Conventions

R code that is evaluated

When R expressions are typed at the R console prompt `>` and evaluated they appear like this:

```
> x <- matrix(1:6, nrow = 2, ncol = 3,
+             byrow = TRUE) # This is a comment
> ## This is also a comment
> x
```

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

The `>` symbol is the command prompt. Notice that when R evaluates an expression that continues onto the next line(s), a `+` sign appears to the left. This means that this line is a continuation of the previous line, and the R expression is not complete. The output appears without the prompt symbol.

In this book however, the above expressions will appear like this:

```
x <- matrix(1:6, nrow = 2, ncol = 3,
            byrow = TRUE) # This is a comment
## This is also a comment
x
```

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

The R prompt (`>`) and continuation (`+`) symbols have been removed, and output is preceded by these two characters (`#>`). *Again, R output is preceded by the (`#>`) characters.* Actual comments will be preceded by one or more comment (`#`) characters without `>`.

These stylistic conventions (`#>` characters preceding the output lines) is for

convenience only. This enables readers to copy and test code from PDF or ebook versions of this book. Removing extraneous symbols also improves readability of R expressions.

R code in a script file (not evaluated)

The same R code above can also be saved as a *R script* for later evaluation. A R script is a collection of R expressions and saved as an ASCII text file with a `.R` extension. The R script can be edited using a text editor or RStudio. R script snippets are displayed without evaluation like this:

```
x <- matrix(1:6, nrow = 2, ncol = 3,
            byrow = TRUE) # This is a comment
## This is also a comment
x
```

If the R script is evaluated it will appears as in above examples.

Part I

Learning R

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 public health 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 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.

1.2 What is RStudio?

To get started quickly we need tools that makes the process of writing and compiling R code quick and (mostly) pain free. Fortunately for us there is RStudio[1][2]—it is a free, open source, and powerful integrated development environment (IDE) for R that runs on most operating systems (Windows, Mac, or Linux). After installing R, install RStudio—it has all the tools we need to learn and apply R.

1.3 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 data scientists, epidemiologists, statisticians, physician researchers, engineers, health economists, health systems analysts, business analysts, 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!).

1.4 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

[1] http://www.rstudio.com
[2] https://www.rstudio.com/

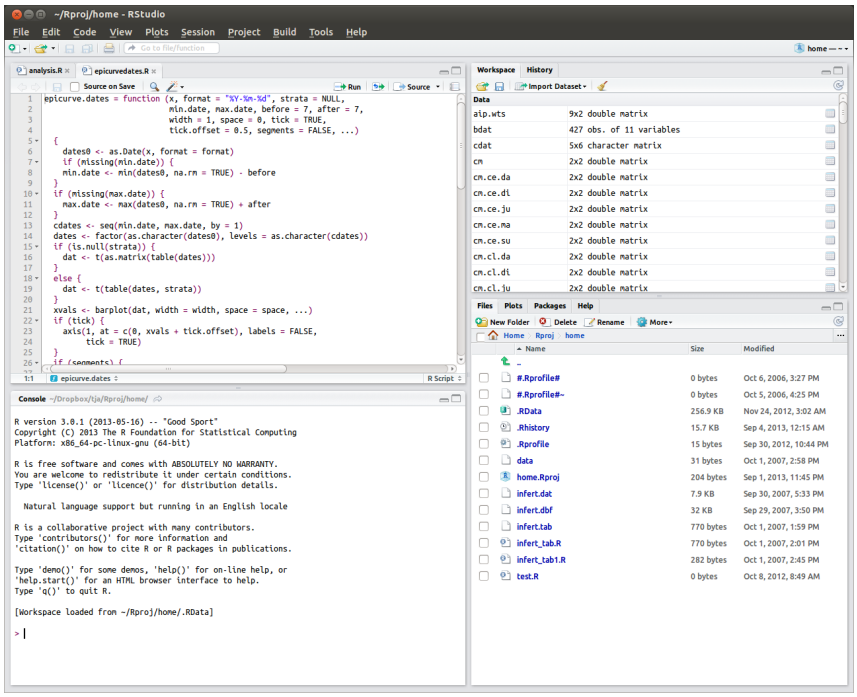


FIGURE 1.1: Screenshot of RStudio in Ubuntu Linux: Top left is for R script editor, or for viewing tabular data objects. Top right has tabs for workspace and history. Bottom right has tabs for files, plots, packages, or help. Bottom left is the console for typing R expressions for evaluation.

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. 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.

1.5 Where can I get R?

R is available for many computer platforms, including Linux, Mac OS, Microsoft (MS) Windows, and others. R comes as source code or a binary file. Source code needs to be compiled into an executable program for your 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`
- R Journal at `http://journal.r-project.org`

To install R for Windows or Mac OS, do the the following:

- Go to `http://www.r-project.org`;
- From the left menu list, click on the "CRAN" (Comprehensive R Archive Network) link;
- Select a nearby geographic site (e.g., `http://cran.cnr.berkeley.edu`);
- Select appropriate operating system;
- Select on "base" link;
- For Windows, save `R-3.6.X-win.exe` to the computer; and for Mac OS, save the `R-3.6.X.pkg` installer package. For Linux, install from the Debian repository, or follow instruction on the CRAN site.
- Run the installation program and accept the default installation options.
- Install RStudio (`https://www.rstudio.com/`). That's it!

An alternative to installing R on a computer is using RStudio Cloud[3]. From a web browser one runs R as if it were on their computer. This resolves occasional quirks of installing and updating R, RStudio, and R packages on a personal computer.

1.6 How do I use R?

1.6.1 Using R on our computer

Use R by typing commands at the R console (`>`) and pressing Enter on our keyboard. This is how to use R interactively. Alternatively, from the R console,

[3] `https://rstudio.cloud/`

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:

```
(4 + 6)^3 - 2*500/4
```

```
#> [1] 750
```

Use the `c` function to collect data entered at the console. Name each collection of data, and then perform a numercal operation. In this example we conduct an analysis that is analogous to working in a spreadsheet.

```
quantity <- c(34, 56, 22)         # quantity data
price <- c(19.95, 14.95, 10.99)   # price data
subtotal <- quantity*price        # subtotal cost
cbind(quantity, price, subtotal) # column bind, like spreadsheet
```

```
#>      quantity price subtotal
#> [1,]       34 19.95   678.30
#> [2,]       56 14.95   837.20
#> [3,]       22 10.99   241.78
```

1.6.2 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. R now has a impressive collection of packages with methods applied to epidemiologic problems. To see more visit https://cran.r-project.org/web/packages/ and search on "epi." The focus of this book is learning how to use R without relying on too heavily on specific packages. Learning the R basics covered in this book will help one take full advantage of these and other R packages, some of which address advanced topics such as network modeling of epidemics.

1.6.3 How should I use these notes?

The best way to learn R is to use it! Use it as your calculator! Use it as your 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). Although we initially encourage you to use R interactively by typing expressions at the console, as a general rule, it is much better to type

your code as a R script. Save your code with a convenient file name such as `job01.R`.[4]

RStudio comes with a text editor for creating and editing R scripts. Our focus will be learning how to use RStudio to edit and run R scripts.

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

- highlight and run selected expressions in the RStudio,
- copy and paste the code directly into R console, or
- run the file in batch mode from the R console using the `source` function (e.g., `source("job01.R")`).

TABLE 1.1: Selected math operators

Expression type	Operator	Example	Value
addition	`+`	`5+4`	9
subtraction	`-`	`5-4`	1
multiplication	`*`	`5*4`	20
division	`/`	`10/3`	3.333333
integer divide	`%/%`	`10%/%3`	3
modulus (remainder)	`%%`	`10%%3`	1
unary minus	`-`	`-(-5)`	5
absolute value	`abs`	`abs(-23)`	23
exponentiation[5]	`^`	`5^4`	625
exponentiation (base e)	`exp`	`exp(8)`	2980.958
logarithm	`log`	`log(exp(8))`	8
square root	`sqrt`	`sqrt(64)`	8

1.7 Just do it!

1.7.1 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 your 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.

[4] The `.R` extension, although not necessary, is useful when searching for R command files. Additionally, this file extension is recognized by RStudio and many text editors.

[5] Python uses `**` instead of `^` for exponentiation.

1.7.2 Useful R concepts

1.7.2.1 Types of evaluable expressions

Every *expression* that is entered at the R console is *evaluated* by R and returns a *value*. A *literal* expression 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. Finally, evaluable expressions must be separated by either newline breaks or a semicolon. Table 1.2 summarizes evaluable R expressions.

```
4 * 4
```

```
#> [1] 16
```

```
x <- 4 * 4    # assign expression to object named 'x'
x             # display 'x' object
```

```
#> [1] 16
```

```
(x <- 4 * 4)  # evaluates expression and displays 'x'
```

```
#> [1] 16
```

```
x <- 4*4; x   # expressions can be separated by semi-colons
```

```
#> [1] 16
```

TABLE 1.2: Types of evaluable R expressions

Expression type	Example	Value returned
literal	`'hello' # character`	`"hello"`
	`3.5 # numeric`	`3.5`
	`TRUE # logical`	`TRUE`
math operation	`6*7`	`42`
assignment	`x <- 4*4`	*none*
	`x = 4*4`	*none*
data object	`x`	`16`
function	`sqrt(x)`	`4`

1.7.2.2 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"
```

Multiple assignments work and are read from right to left:

```
aa <- bb <- 5
aa; bb
```

```
#> [1] 5
```

```
#> [1] 5
```

Data objects can be used in subsequent calculations:

```
cc <- aa * bb; cc
```

```
#> [1] 25
```

However, updating an object on right side of the assignment *does not* automatically update the value of the object on the left side of the assignment. To update the left side we must re-run the assignment expression.

```
bb <- 25
cc # not updated even though bb changed
```

```
#> [1] 25
```

```
cc <- aa * bb; cc # re-run assignment to update cc
```

```
#> [1] 125
```

Finally, similar to Python, the equal sign (`=`) can be used for assignment, although we prefer and the `<-` symbol.

```
ages <- c(34, 45, 67) # equivalent
ages = c(34, 45, 67) # equivalent
```

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

```
x <- 1:10 # assigning object name (x)
sample(x = 1:10, size = 5)  # assigning value to argument x
```

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.[6]

1.7.3 Useful R functions

When we start R we 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 without warning, so be careful! To list the objects in your workspace use the ls or objects functions:

```
ls()[1:4] # lists first four objects in the workspace
```

```
#> [1] "a"            "aa"           "age"          "age.centered"
```

```
## objects()[1:4] # equivalent
```

Data objects can be saved between sessions. We will be prompted with "Save workspace image?" You can also use save.image() at the console prompt. The workspace image is saved in a file called .RData.[7] Use getwd() to display the file path to the .RData file. Table 1.3 has more useful R functions.

[6] To improve readability, a period (.) or underscore (_) symbol can be used in your object name

[7] 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 example	Description
`q()`	Quit R
`ls()`	List objects
`rm(`*object name*`)`	Remove object
`rm(list = ls()); ls()`	Removes all objects—caution!
`help()`	Open help instructions;
`help(`*function.name*`)`	or get help on specific function
`?`*function.name*	Equivalent to get help
`help.search("print")`	Search help system
`help.start()`	Start help browser
`apropos("plot")`	Displays all objects matching topic
`getwd()`	Working directory (location of `.RData`)
`setwd("c:\mywork\rproj")`	Set working directory
`args(sample)`	Display arguments of function
`example(plot)`	Runs example of a function
`data() #displays`	Information on available R data sets
`data(`*data.set.name*`)`	Load data set
`save.image()`	Saves current workspace to .RData

1.7.3.1 What are packages?

R has many available functions, and a *package* is a compiled collection of functions with a shared purpose or common theme. When we open R, several packages are attached by default.[8] Each package has its own suite of functions. To display the list of attached packages use the `search` function. To display the file paths to the packages use the `searchpaths` function.

```
search() # Mac OS
```

```
#>  [1] ".GlobalEnv"         "package:foreign"
#>  [3] "package:survival"   "package:mosaicData"
#>  [5] "ESSR"               "package:stats"
#>  [7] "package:graphics"   "package:grDevices"
#>  [9] "package:utils"      "package:datasets"
#> [11] "package:methods"    "Autoloads"
#> [13] "package:base"
```

[8] In Python, a package (or "library") is a compiled collection of *modules*. A module is a single Python file. For example, to load the `math` module we type `import math` at the Python console. The list of mathematical functions in the `math` module are available at https://docs.python.org/3/library/math.html. For scientific computing the most common Python packages include NumPy, SciPy, Pandas, and Matplotlib.

```
## searchpaths() # output suppressed
```

To install a package we enter `install.packages("<package name>")`. For example to *install* and *load* the package for survival analysis we enter

```
install.packages("survival") # installs package
library(survival)            # loads and attaches package
```

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.

1.7.3.2 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) == 1L && is.numeric(x) && is.finite(x) && x >=
#>         1) {
#>         if (missing(size))
#>             size <- x
#>         sample.int(x, size, replace, prob)
#>     }
#>     else {
#>         if (missing(size))
#>             size <- length(x)
#>         x[sample.int(length(x), size, replace, prob)]
#>     }
#> }
#> <bytecode: 0x7ff828047968>
#> <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, 16) # sample "size = 16"
```

```
#>  [1] 11  2 18 13 16 23 29 24  1  8 25 22 26  4 17  5
```

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(16, dates) # undesired results; wanted "size = 16"
```

```
#> [1] 2
```

```
sample(size = 16, x = dates) # gives desired result
```

```
#>  [1] 10  7 16 21 28  2  4 17  5 15  1 27 30 25 24 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 = 16, x = dates, r = TRUE) # sampling with replacement
```

```
#>  [1] 13 25 29  5 28  2 18 17 18  7 14 11 12 16 15  6
```

Fifth, argument values can be any valid R expression (including functions) that evaluates 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(1:36, 1), x = sample(1:10, 5), r=T)
```

```
#> [1] 3 3 9
```

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

1.7.4 How do I get help?

RStudio has extensive help capabilities. From the RStudio main menu select `Help` → `R Help` to get you started. The Frequently Asked Questions (FAQ) and R manuals are available from this menu. From the R console, try the following options to learn about the help capabilities:

```
?help               # opens help page for 'help' function
help.start()        # launches HTML help page
help.search("help") # searches help system for "help"
apropos("help")     # displays 'help' objects in search list
```

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

```
data()                           # display avail. data sets
try(data(package = "survival")) # list 'survival' data sets
help(pbc, package = "survival") # display pbc data help page
```

1.7.5 Is there anything else that I need?

Not really. RStudio has everything you will need to use R productively. Some analysts will select to use R with a text editor, rather than RStudio. Like RStudio, a good text editor makes programming and data processing easier and more efficient. If you are considering a text editor, the functionality we look for in a text editor are the following:

- Toggle between wrapped and unwrapped text
- Block cutting and pasting (also called column editing)
- Easy macro programming
- Search and replace using regular expressions
- Ability to import small data sets 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 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**.[9] GNU Emacs can be extended with the "Emacs Speaks Statistics" (ESS) package. For more information on Emacs and ESS pre-installed for Windows or Mac OS, visit http://ess.r-project.org.

1.7.6 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 environment and language. If you understand the logic and structure of R, then learning proceeds quickly. Just like a spoken language, once you know its rules of grammar, syntax, and pronunciation, and can write legible sentences, you can figure out how to communicate almost anything. Before we get into the "trees" (next chapter), we want to describe the "forest": the logic and structure of working with R objects and epidemiologic data.

1.7.6.1 Working with R objects

For our purposes, there are only five types of data objects in R[10] 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.

TABLE 1.4: Tables that summarize types of actions taken on R objects

Action	Vector	Matrix	Array	List	Data frame
Creating	2.6	2.13	2.21	3.1	3.7
Naming	2.7	2.14	2.22	3.2	3.8

[9] https://www.gnu.org/software/emacs/

[10] 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.

Action	Vector	Matrix	Array	List	Data frame
Indexing	2.8	2.15	2.23	3.3	3.9
Replacing	2.9	2.16	2.24	3.4	3.10
Operating on	2.10, 2.11	2.17	2.25	3.5	3.11

A *vector*[11] is a collection of elements (often numbers):[12]

```
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:

```
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 an n dimensional represention of a vector:[13]

```
z <- array(x, dim = c(1, 6, 2)); z
```

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

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

```
mylist <- list(x, y, z); mylist
```

```
#> [[1]]
```

[11] Do not confuse a vector—a collection of elements—with the `vector` function.
[12] In Python, a collection of elements is called an *array*.
[13] In Python, the NumPy package is used for n-dimensional arrays.

```
#>  [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] [,4] [,5] [,6]
#> [1,]    1    2    3    4    5    6
#>
#> , , 2
#>
#>      [,1] [,2] [,3] [,4] [,5] [,6]
#> [1,]    7    8    9   10   11   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.

```
smoker <- c("Y", "Y", "N", "N", "Y", "Y", "Y", "N", "Y")
cancer <- c("Y", "N", "N", "Y", "Y", "Y", "N", "N", "Y")
mydf <- data.frame(smoker, cancer);
mydf[1:4,] # display rows 1 to 4
```

```
#>   smoker cancer
#> 1      Y      Y
#> 2      Y      N
#> 3      N      N
#> 4      N      Y
```

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

1.8 What are graphical models?

Graphical models consist of nodes and edges. Nodes represent data variables and edges represent relationships between nodes. In R, nodes (variables) are represented by vectors. We will focus on **causal graphs** (also called *directed acyclic graphs* or DAGs) that depict the causal relationships between nodes using arrows.

Figure 1.2 is a causal graph encoding the causal effect of smoking on developing cancer. This means that in a population the outcome, cancer, is caused by smoking, even if the effect is small (e.g., only causal in one person). In other words, a causal arrow does indicate the magnitude of the effect.

FIGURE 1.2: Causal graph of exposure (smoker) and outcome (cancer)

A causal graph of two dichotomous variables (Figure 1.2) can also be displayed as a two-way (2×2) table. For example, we can cross-tabulate the smoker-cancer data frame we created above.

```
xtabs(~smoker + cancer, data = mydf)
```

```
#>        cancer
#> smoker N Y
#>      N 2 1
#>      Y 2 4
```

Notice that the two-way table and the causal graph provide different but complementary information. The causal graph declares that a causal effect exists *and* it is directed from smoker to cancer. A two-way table only enables us to test for a statistical association (correlation) which has no directionality. The "story behind the data" is missing from data tables (and even visual plots); however, *causal graphs encode the story behind the data* and it's known as the **data generating process**.

KEY IDEA: The absence of a causal arrow between two nodes is the *strongest* assertion in a causal graph. This assertion can often be made after interviewing knowledge experts or through common logic. For example, carrying matches does not cause lung cancer.

1.9 Precision and number types?

Integers are numbers like $\{... -2, -1, 0, 1, 2, ...\}$. *Real* numbers have decimal representations like 3.145 or 3.000. R converts all numbers into double-precision floating decimals. We can test the object using the `typeof` function.[14] For example, see how R handles the integer `3` below:

```
typeof(3) # looks like an integer but it's double-precision
```

```
#> [1] "double"
```

If we want R to treat an integer as an integer then add `L` to the integer or use the `as.integer` function.

```
typeof(3L) # convert 3 to integer and test
```

```
#> [1] "integer"
```

```
typeof(as.integer(3)) # convert 3 to integer and test
```

```
#> [1] "integer"
```

Notice that if we divide an integer by an integer R converts the answer to double precision (unless we coerce it back to integer using the `as.integer` function).

```
typeof(4L/2L)
```

```
#> [1] "double"
```

```
typeof(as.integer(4L/2L))
```

```
#> [1] "integer"
```

For the most part we do not have to worry about precision and integer versus floating point numbers. However, when we start working with or mixing very large or very small numbers then we need to pay attention. For a concise summary read "Data Types" chapter in [6].

[14] In Python, use the `type` function to test.

1.10 Exercises

Exercise 1.1 (Get started with R.)**.** If you have not done so already,

(a) Install R on your computer (`https://cran.rstudio.com/`),

(b) Install RStudio on your computer (`https://www.rstudio.com/`), and

(c) Register for a RPubs account (`http://www.rpubs.com/`), and open RStudio.

(d) Consider using RStudio Cloud instead.

(e) Install the `knitr` and `rmarkdown` packages

(f) Open a new Rmarkdown template file (`.Rmd` extension).

(g) Learn Rmarkdown and use it to answer the exercises in this chapter.

(h) Submit the exercises as a HTML link to your Rpubs.com page, Word document, or PDF document (first install the `tinytex` package).

Exercise 1.2 (Get to know your workspace.)**.** Answer the following questions:

(a) What is the R workspace file on your operating system?

(b) What is the file path to your R workspace file?

(c) What is the name of this workspace file?

(d) When you launched, which R packages loaded?

(e) What are the file paths to the loaded R packages?

(f) 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.

Exercise 1.3 (Get to know math operators.)**.** Using Table 1.1, explain in words, and use R to illustrate, the difference between modulus and integer divide.

Exercise 1.4 (Calculating body mass index.)**.** BMI is an 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 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 the BMI for a male with weight of 155 lb and height of 5 ft 7 in.

Exercise 1.5 (Using logarithms)**.** 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?

Exercise 1.6 (Risk and risk odds)**.** 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?

Exercise 1.7 (HIV transmission probabilities)**.** Review Table 1.6 and answer the questions.

TABLE 1.6: Estimated per-act risk (transmission probability) for acquisition of HIV, by exposure route to an infected source. Source: CDC [7]

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

Use the data in Table 1.6. 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 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?

Exercise 1.8 (Sourcing files and sinking log files)**.** 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("~/Documents/courses/ph251d/jobs/job01.R")
```

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

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

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

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

Examine the log files and describe what happened.

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 describe what happened.

2

Working with vectors, matrices, and arrays

2.1 Data objects in R

2.1.1 Atomic vs. recursive data objects

The analysis of data in R involves creating, naming, manipulating, and operating on data objects using functions. Data in R are organized as objects and have been assigned names. We have already been introduced to several R data objects. Now we make further 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 # character
```

```
#> [1] "Pedro" "Paulo" "Maria"
```

```
x <- c(1, 2, 3, 4, 5); x             # numeric
```

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

```
x < 3                                # logical
```

```
#> [1]  TRUE  TRUE FALSE FALSE FALSE
```

Remember that a *numeric* vector can be of type *double* precision or *integer*. By default R does not use *single* precision numeric vectors. The vector type is not apparent by inspection (also see Table 2.1).

[1] In other programming languages, vectors are either row vectors or column vectors. R does not make this distinction until it is necessary. Do not confuse a vector, as discussed in this section, with the `vector` function which can create atomic or recursive objects.

```
(x <- c(3, 4, 5))
```

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

```
typeof(x)            # look like integers but are not
```

```
#> [1] "double"
```

```
(x <- c(3L, 4L, 5L))
```

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

```
typeof(x)            # look like integers and are
```

```
#> [1] "integer"
```

TABLE 2.1: Summary of numeric vectors

vector example	mode(x)	typeof(x)	class(x)
x <- c(3, 4, 5)	numeric	double	numeric
x <- c(3L, 4L, 5L)	numeric	integer	integer

A *matrix* is a collection of like elements organized into a 2-dimensional (tabular) data object with r rows and c columns. We can think of a matrix as a vector re-shaped into a $r \times c$ table. When R returns a matrix the `[`i`,]` indicates the ith row and `[,`j`]` indicates the jth column.

```
x <- c('a', 'b', 'c', 'd') # character vector
y <- matrix(x, 2, 2); y    # shape x into matrix
```

```
#>      [,1] [,2]
#> [1,] "a"  "c"
#> [2,] "b"  "d"
```

An *array* is a collection of like elements organized into a z-dimensional data object. We can think of an array as a vector re-shaped into a $n_1 \times n_2 \times n_3 \ldots n_z$ table. For example, when R returns a 3-dimensional array the `[`i`,,]` indicates the ith row and `[,`j`,]` indicates the jth column, and "`, ,` k" indicates the kth depth.

```
x <- 1:12
y <- array(x, dim = c(2, 3, 2)); y
```

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

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 2.2).

A *list* is a collection of data objects without any restrictions. Think of a list of length k as k "bins" in which each bin can contain any type of data object.

```
x <- c(1, 2, 3)                    # numeric vector
y <- c('Male', 'Female', 'Male')   # character vector
z <- matrix(1:4, 2, 2)             # numemric matrix
mylist <- list(x, y, z); mylist    # list
```

```
#> [[1]]
#> [1] 1 2 3
#>
```

```
#> [[2]]
#> [1] "Male"   "Female" "Male"
#>
#> [[3]]
#>      [,1] [,2]
#> [1,]    1    3
#> [2,]    2    4
```

A *data frame* is a special 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)
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"
```

A *tibble* "is a modern reimagining of the data.frame, keeping what time has proven to be effective, and throwing out what is not. Tibbles are data.frames that are lazy and surly: they do less (i.e. they don't change variable names or types, and don't do partial matching) and complain more (e.g. when a variable does not exist). This forces you to confront problems earlier, typically leading to cleaner, more expressive code. Tibbles also have an enhanced `print()` method which makes them easier to use with large datasets containing complex objects." To learn more see https://tibble.tidyverse.org.

2.1.2 Assessing the structure of data objects

TABLE 2.2: 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

Summarized in Table 2.2 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","6-11yrs",..: 1 1 1 1 2 2 2 2 2 2 ...
#>  $ 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 ...
#>  $ pooled.stratum: num  3 1 4 2 32 36 6 22 5 19 ...
```

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 2.3). We will see extensive use of this when we start programming in R.

To get practice calling data from the console, enter `data()` to display the available data sets in R. Then enter `data(`*data_set*`)` to load a dataset. Study the examples in Table 2.3 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`).

TABLE 2.3: Useful functions for assessing R data objects

Function	Returns …
`str`	summary of data object structure
`attributes`	list with data object attributes
`mode`	mode of object
`typeof`	type of object; similar to mode but includes `double` and `integer`, if applicable
`length`	length of object
`class`	class of object, if it exists
`dim`	vector with object dimensions, if applicable
`nrow`	number of rows, if applicable
`ncol`	number of columns, if applicable
`dimnames`	list containing vectors of names for each dimension, if applicable
`rownames`	vector of row names of a matrix-like object
`colnames`	vector of column names of a matrix-like object
`names`	vector of names for the list (for a data frame returns field names)
`row.names`	vector of row names for a data frame

2.2 A vector is a collection of like elements

2.2.1 Understanding vectors

A *vector*[2] is a collection of like elements (i.e., the elements all have the same mode). There are many ways to create vectors (see Table 2.6). The most common way of creating a vector is using the `c` function:

```
chol <- c(136, 219, 176, 214, 164); chol     # numeric
```

```
#> [1] 136 219 176 214 164
```

```
fn <- c('Mateo', 'Mark', 'Luke', 'Juan'); fn # character
```

```
#> [1] "Mateo" "Mark"  "Luke"  "Juan"
```

```
z <- c(T, T, F, T, F); z                     # logical
```

```
#> [1]  TRUE  TRUE FALSE  TRUE FALSE
```

A single element is also a vector; that is, a vector of length = 1. Try, for example, `is.vector('Zoe')`. In mathematics, a single number is called a *scalar*; in R, it is a numeric vector of length = 1. When we execute a math operation between a scalar and a vector, in the background, the scalar expands to the same length as the vector in order to execute a vectorized operation. For example, these two calculation are equivalent:

```
x1 <- 5; y1 <- c(10, 20, 30)
x2 <- c(5, 5, 5); y2 <- c(10, 20, 30)
x1 * y1
```

```
#> [1]  50 100 150
```

```
x2 * y2
```

```
#> [1]  50 100 150
```

A note of caution: when vector lengths differ, the shorter vector will recyle its elements to enable a vectorized operation. For example,

[2] In other programming languages, vectors are either row vectors or column vectors. R does not make this distinction until it is necessary. Do not confuse a vector, as discussed in this section, with the `vector` function which can create atomic or recursive objects.

```
x <- c(3, 5); y = c(10, 20, 30, 40)
z <- x * y     # multipy vectors with different lengths
cbind(x, y, z) # display results
```

```
#>      x  y   z
#> [1,] 3 10  30
#> [2,] 5 20 100
#> [3,] 3 30  90
#> [4,] 5 40 200
```

In effect, `x` expanded from `c(3, 5)` to `c(3, 5, 3, 5)` in order to execute a vectorized operation. This expansion of `x` is not apparent from the math operation; that is, no warning was produced. Therefore, be aware: math operations with vectors of different lengths may not produce a warning.

2.2.1.1 Boolean queries and subsetting of vectors

In R, we use *relational* and *logical* operators (Table 2.4) and (Table 2.5) 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. 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:

```
movie.stars <- c('Rebecca', 'Elisabeth', 'Amanda', 'Jennifer',
                 'Winona', 'Catherine', 'Reese')
ms.ages <- c(42, 40, 32, 33, 32, 34, 27)
ms.ages >= 30 # logical vector for stars with ages >=30
```

```
#> [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
```

```
ms.ages < 40 # logical vector for stars with ages <40
```

```
#> [1] FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```
(ms.ages >= 30) & (ms.ages < 40) # intersection of vectors
```

```
#> [1] FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE
```

```
thirtysomething <- (ms.ages >= 30) & (ms.ages < 40)
movie.stars[thirtysomething] # indexing using logical
```

```
#> [1] "Amanda"    "Jennifer"  "Winona"    "Catherine"
```

TABLE 2.4: Using relational operators with vectors

Operator	Description	Try these examples
`<`	Less than	`pos <- c('p1', 'p2', 'p3', 'p4')` `x <- c(1, 2, 3, 4)` `y <- c(5, 4, 3, 2)` `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]`

TABLE 2.5: Using logical operators with vectors

Operator	Description	Try these examples
`!`	Element-wise NOT	`x <- c(1, 2, 3, 4)` `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: only TRUE if one or other element is true, not both	`xx <- x <= 1` `yy <- x > 4` `xor(xx, yy)`

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 the Table 2.4 and Table 2.5. 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.

Notice that 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`.

2.2.1.2 Integer queries and subsetting of vectors

TODO - which function

2.2.2 Creating vectors

TABLE 2.6: Common ways of creating vectors *de novo*

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 (press enter twice after last entry)	`xx <- scan()` `1 2 3 4 5` `yy <- scan(what = '')` `'Javier' 'Miguel' 'Martin'` `xx; yy # Display vectors`
`:`	Make integer sequence	`1:10` `10:(-4)`
`seq`	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)`
`paste`	Paste character strings	`paste(c('A','B','C'),1:3,sep='')`

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
```

```
#> [1] 1.000000 1.571429 2.142857 2.714286 3.285714 3.857143
#> [7] 4.428571 5.000000
```

```
x <- 1:8
seq(1, 5, along = x)   # by length of other object
```

```
#> [1] 1.000000 1.571429 2.142857 2.714286 3.285714 3.857143
#> [7] 4.428571 5.000000
```

These types of sequences are convenient for plotting mathematical equations.[3] For example, suppose we wanted to plot the standard normal curve using the

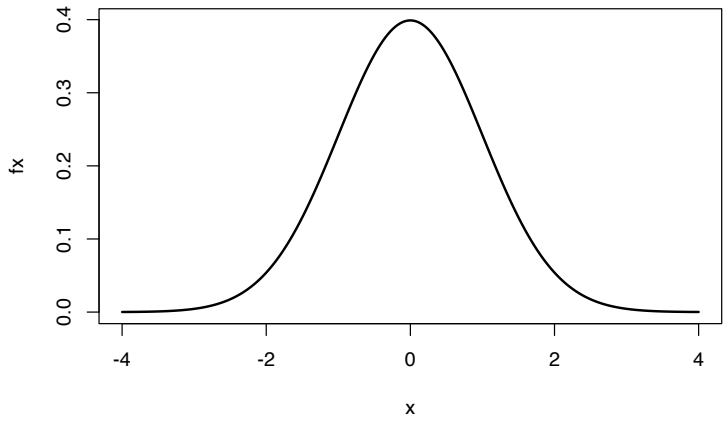
[3] See also the `curve` function for graphing mathematical equations.

normal equation. For a standard normal curve $\mu = 0$ (mean) and $\sigma = 1$ (standard deviation)

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{\frac{-(x-\mu)^2}{2\sigma^2}\right\} \tag{2.1}$$

Here is the R code to plot this equation:

```
mu <- 0; sigma <- 1  # set constants
x <- seq(-4, 4, .01) # generate x values to plot
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.

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

```
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,...
```

```
#>  [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 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:

```
sample(c('H', 'T'), size = 8, replace = T) # toss coin 8 times
```

```
#> [1] "H" "H" "H" "H" "H" "H" "T" "T"
```

```
sample(1:6, size = 10, replace = TRUE) # toss die 10 times
```

```
#>  [1] 6 6 3 6 6 5 6 6 5 3
```

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

```
rbinom(n = 10, size = 50, p = 0.5)  # toss 50 coins 10 times
```

```
#>  [1] 24 20 33 19 18 22 25 28 21 28
```

```
rnorm(5)  # generate 5 standard normal distribution values
```

```
#> [1] -0.80761411  0.05180713 -0.44901020  0.38182407  0.09212866
```

There are additional ways to create vectors. To practice creating vectors study the examples in Table 2.6 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 can use vectors as arguments to functions:

```
sample(c('head', 'tail'), 100, replace = TRUE)
rep(1:2, rep(5, 2))
matrix(c(23, 45, 16, 17), nrow = 2, ncol = 2)
```

2.2.3 Naming vectors

Each element of a vector can be labeled with a name at the time the vector is created or later. 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 2.7 and spend a few minutes creating and naming simple vectors.

TABLE 2.7: Common ways of naming vectors

Function	Description	Try these examples
`c`	Name elements at creation	`x <- c(a = 1, b = 2, c = 3)`
`names`	Name elements by assignment;	`y <- 1:3`
	Returns names if they exist	`names(y) <- c('a', 'b', 'c')`
		`names(y) # return names`
`unname`	Remove names (if they exist)	`y <- unname(y)`
		`names(y) <- NULL #equivalent`

2.2.4 Indexing (subsetting) vectors

Indexing a vector is subsetting or extracting elements from a vector. A vector is indexed by *position*, by *name* (if it exist), or by *logical* value (TRUE vs FALSE). 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
```

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, 7, replace = TRUE))
```

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

```
x[samp1]
```

```
#>  age chol  sbp  dbp  dbp  sbp  dbp
#>   54  234  148   78   78  148   78
```

```
(samp2 <- sample(names(x), 7, replace = TRUE))
```

```
#> [1] "sbp"  "dbp"  "sbp"  "age"  "dbp"  "chol" "sbp"
```

```
x[samp2]
```

```
#>  sbp  dbp  sbp  age  dbp chol  sbp
#>  148   78  148   54   78  234  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), 7, replace = TRUE))
```

```
#> [1]  TRUE FALSE FALSE FALSE  TRUE FALSE  TRUE
```

```
x[samp3]
```

```
#> chol <NA> <NA>
#>  234   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 2.8 and spend a few minutes creating, naming, and indexing simple vectors.

TABLE 2.8: Common ways of indexing vectors

Indexing	Try these examples
By logical	`x < 100` `x[x < 100]` `(x < 150) & (x > 70)` `bp <- (x < 150) & (x > 70)` `x[bp]`
By position	`x <- c(chol = 234, sbp = 148, dbp = 78)` `x[2]  # positions to include` `x[c(2, 3)]` `x[-c(1, 3)]  # positions to exclude` `x[which(x<100)]`
By name (if exists)	`x['sbp']` `x[c('sbp', 'dbp')]`
Unique values	`samp <- sample(1:5, 25, replace = TRUE)` `unique(samp)`
Duplicated values	`duplicated(samp) # generates logical`

Indexing	Try these examples
	`samp[duplicated(samp)]`

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)

If a vector element can be indexed, it can be replaced. Therefore, to replace vector elements we combine indexing *and* assignment. Replacing vector elements is one method of recoding a variable. In this example, we recode a vector of ages into age categories:

```
age <- sample(15:100, 1000, replace = TRUE) # create vector
agecat <- age # copy vector
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
#>  >=65 15-24 25-44 45-64
#>   394   131   251   224
```

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 (`agecat`) from a numeric vector (`age`).

Review the results of `table(agecat)`. What's wrong? Age categories have a

natural ordering which was loss in our recoding. This would be fine if this were a nominal variable where there is no ordering. (Later we will learn how to use the `cut` function for discretizing continous data into a factor.) Another problem is that the category "<15" is not included in our table with the value of 0.

Let's try replacement with integers to preserve the natural ordering, and convert it to a factor *and* set the possible *levels* to preserve the "<15" category for tracking and reporting.

```
agecat2 <- age # copy age vector from above
agecat2[age<15] <- 1
agecat2[age>=15 & age<25] <- 2
agecat2[age>=25 & age<45] <- 3
agecat2[age>=45 & age<65] <- 4
agecat2[age>=65] <- 5
table(agecat2)    # table of numeric vector - not sufficient
```

```
#> agecat2
#>   2   3   4   5
#> 131 251 224 394
```

```
agecat2 <- factor(agecat2)
table(agecat2)    # table of factor - not sufficient
```

```
#> agecat2
#>   2   3   4   5
#> 131 251 224 394
```

```
levels(agecat2) <- c("<15", "15-24", "25-44", "45-64", ">=65")
table(agecat2)    # table of factor with levels - works!
```

```
#> agecat2
#>   <15 15-24 25-44 45-64  >=65
#>   131   251   224   394     0
```

Notice that transforming the numeric vector `agecat2` into a factor is *not sufficient*, we must also specify the possible levels with the `levels` function. *Do not assume that all possible levels of responses appear in the data.* Note that setting levels can occur as an option to the `factor` function.

```
agelabs <- c("<15", "15-24", "25-44", "45-64", ">=65")
agecat2 <- factor(agecat2, levels = agelabs) # set levels option
```

For practice study the examples in Table 2.9 and spend a few minutes replacing vector elements.

TABLE 2.9: Common ways of replacing vector elements

Replacing	Try these examples
By logical	`x[x<100]`
	`x[x<100] <- NA; x`
By position	`x <- c(chol = 234, sbp = 148, dbp = 78)`
	`x[1]`
	`x[1] <- 250; x`
By name (if exists)	`x['sbp']`
	`x['sbp'] <- 150; x`

2.2.6 Operating on vectors

Operating on vectors is very common in epidemiology and statistics. In this section we cover common operations on single vectors (Table 2.10) and multiple vectors (Table 2.11).

2.2.6.1 Operating on single vectors

TABLE 2.10: Selected operations on single vectors

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

First, we focus on operating on single numeric vectors (Table 2.10). 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:

```
x <- rnorm(100) # generate random standard normal values
sum(x)
```

```
#> [1] -5.226359
```

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:

```
x <- rep(2, 10); x # generate sequence of 2's; calculate cumulative sum
```

```
#>  [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] 0.1499375
```

```
mean(x)
```

```
#> [1] 0.1499375
```

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.

$$S_X^2 = \frac{1}{n-1}\left[\sum_{i=1}^{n}(x_i - \bar{x})^2\right] \tag{2.2}$$

```
x <- rnorm(100)
sum((x-mean(x))^2)/(length(x)-1)
```

```
#> [1] 0.9855883
```

```
var(x)  # equivalent
```

```
#> [1] 0.9855883
```

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:

$$S_{XY} = \frac{1}{n-1}\left[\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})\right] \tag{2.3}$$

```
x <- rnorm(100); y <- rnorm(100)
sum((x - mean(x)) * (y - mean(y)))/(length(x) - 1)
```

```
#> [1] 0.09364036
```

```
var(x, y)  # equivalent
```

```
#> [1] 0.09364036
```

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

$$S_X = \sqrt{\frac{1}{n-1}\left[\sum_{i=1}^{n}(x_i - \bar{x})^2\right]} \tag{2.4}$$

```
sqrt(var(x))
```

```
#> [1] 0.9130861
```

```
sd(x)
```

```
#> [1] 0.9130861
```

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"   "Tomas"
```

```
order(ages) # 'order' returns positional integers for sorting
```

```
#> [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" "H" "T" "T" "T" "H" "T" "H"
```

```
sample(1:100, 15)
```

```
#>  [1] 88 85 22 45 10  1 71 15 73 83 67 16 35 51 19
```

2.2.6.2 Operating on multiple vectors

TABLE 2.11: Selected operations on multiple vectors

Function	Description
`c`	Concatenates into vectors
`append`	Appends a vector to another 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

Function	Description
`ftable`	Creates flat contingency table from 2 or more vectors
`outer`	Outer product
`tapply`	Applies a function to strata of a vector
`<`, `>`	Relational operators
`<=`, `>=`	
`==`, `!=`	
`!`, `|`, `xor`	Logical operators

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
```

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 [8]. 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
```

```
tapply(age, sex, sum)/tapply(age, sex, length)  # equivalent
```

```
#>    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.

2.2.7 Converting vectors into factors (categorical variables)

The Williams Institute recommends a two-step approach for asking survey questions for gender identity [9]:

1. [`sex`] What sex were you assigned at birth, on your original birth certificate? (select one)
   ○ Male
   ○ Female

2. [gender] How do you describe yourself? (select one)
   ○ Male
   ○ Female
   ○ Transgender
   ○ Do not identify as female, male, or transgender

Suppose you conduct a survey of 100 subjects and the `gender` variable is represented by a character vector. Here is the tabulation of your data:

```
gender <- sample (c('Female','Male'), 100, TRUE) # simulate data
gender[1:5]   # display first five
```

```
#> [1] "Female" "Female" "Female" "Male"   "Male"
```

```
table(gender) # tabulate vector
```

```
#> gender
#> Female   Male
#>     56     44
```

What is wrong with this tabulation? The tabulation does not have all the possible reponse levels ("Female", "Male", "Transgender", "Neither.FMT"). This is because the first 100 respondents identified only as female and male. The character vector only contains the *actual* reponses without keeping track of the *possible* responses. To remedy this we use the `factor` function to convert vectors into a categorical variable that can keep track of possible response levels. Factors is how R manages categorical variables.

```
gender.fac <- factor(gender, levels = c('Female','Male',
                     'Transgender', 'Neither.FMT'))
gender.fac[1:5]   # display first five
```

```
#> [1] Female Female Female Male   Male
#> Levels: Female Male Transgender Neither.FMT
```

```
table(gender.fac) # tabulate factor
```

```
#> gender.fac
#>      Female        Male Transgender Neither.FMT
#>          56          44           0           0
```

How does the display of factors differ from the display of character vectors? This was our first introduction to factors. Later, we cover factors in more detail.

Note: factors are actually integer (numeric) vectors with labels called *levels*. Try `unclass(gender.fac)` followed by `table(unclass(gender.fac))`.

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 2.12 [10]. 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.

TABLE 2.12: Deaths among subjects who received tolbutamide and placebo in the Unversity Group Diabetes Program (1970)

	Treatment	
Outcome status	Tolbutamide	Placebo
Deaths	30	21
Survivors	174	184

```
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
dat                            # display results
```

```
#>           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.

```
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`.

```
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 2.13). In general, we create or use matrices in the following ways:

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

TABLE 2.13: Common ways of creating a matrix

Function	Description	Try these examples
`cbind`	Column-bind	`x <- 1:3; y <- 3:1; cbind(x, y)`
`rbind`	Row-bind vectors	`rbind(x, y)`
`matrix`	Generates matrix	`matrix(1:4, nrow=2, ncol=2)`
`dim`	Assign dimensions	`mtx2 <- 1:4` `dim(mtx2) <- c(2, 2)`
`table`	Cross-tabulate	`table(infert$educ, infert$case)`
`xtabs`	Cross-tabulate	`xtabs(~education + case, data = infert)`
`ftable`	Creates 2-D flat table	`ftable(infert$educ, infert$spont, infert$case)`
`array`	Creates 2-D array	`array(1:4, dim = c(2, 2))`
`as.matrix`	Coercion	`as.matrix(1:3)`
`outer`	Outer product	`outer(1:5, 1:5, '*')`
`x[`r`,,]`	Indexing an array	`x <- array(1:8, c(2, 2, 2))`
`x[,`c`,]`		`x[1, , ]`
`x[,,`d`]`		`x[ ,1, ]`
		`x[ , ,1]`

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('~/data/ugdp1.txt', header = TRUE, sep = ',')
names(dat2) # display field names
```

```
#> [1] "Status"    "Treatment" "Agegrp"
```

```
table(dat2$Status, dat2$Treatment)
```

```
#>
#>            Placebo Tolbutamide
#>   Death         21          30
#>   Survivor     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
#>   Death         21          30
#>   Survivor     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 # xtabs produces an array
```

```
#> , , Agegrp = <55
#>
#>           Treatment
#> Status     Placebo Tolbutamide
#>   Death          5           8
#>   Survivor     115          98
#>
#> , , Agegrp = 55+
#>
#>           Treatment
#> Status     Placebo Tolbutamide
#>   Death         16          22
```

```
#>   Survivor      69          76
```

```
ftable(xtab3way) # convert to flat table
```

```
#>                      Agegrp <55 55+
#> Status   Treatment
#> Death    Placebo              5  16
#>          Tolbutamide          8  22
#> Survivor Placebo            115  69
#>          Tolbutamide         98  76
```

However, notice that `ftable`, while preserving the order of the variables, does provide the equivalent, namely, `Status` vs. `Treatment`, by `Agegrp`. We can fix this by reversing the order of the variables within the `xtabs` function:

```
ftable(xtabs(~Agegrp + Status + Treatment, data=dat2))
```

```
#>                 Treatment Placebo Tolbutamide
#> Agegrp Status
#> <55    Death                    5           8
#>        Survivor               115          98
#> 55+    Death                   16          22
#>        Survivor                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*:

$$R(0,t) = 1 - e^{-\lambda t} \tag{2.5}$$

where , λ = infection rate, and t = time.

```
lamb <- 10/100
years <- c(1, 5, 10)
risk <- 1 - exp(-lamb*years)
cbind(rate = lamb, years, cumulative.risk = risk) # display
```

```
#>      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('~/data/ugdp1.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
#> Death          21          30    51
#> Survivor      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:

$$z = xy \tag{2.6}$$

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
#> 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 matrix components

We have already seen several examples of naming components of a matrix. Table 2.14 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 field names.

TABLE 2.14: Common ways of naming matrix components

Function	Try these examples
`matrix`	`# name rows and columns only` `dat <- matrix(c(178, 79, 1411, 1486), 2, 2,` `dimnames = list(c('TypeA','TypeB'), c('Y','N')))` `# name rows, columns, and fields` `dat <- matrix(c(178, 79, 1411, 1486), 2, 2,` `dimnames = list(Behavior = c('TypeA','TypeB'),` `'Heart attack' = c('Y','N')))`
`rownames`	`dat <- matrix(c(178, 79, 1411, 1486), 2, 2)`

Function	Try these examples
	`rownames(dat) <- c('TypeA','TypeB')`
colnames	`colnames(dat) <- c('Y','N') # col names`
dimnames	`# name rows and columns only`
	`dat <- matrix(c(178, 79, 1411, 1486), 2, 2)`
	`dimnames(dat) <- list(c('TypeA','TypeB'), c('Y','N'))`
	`# name rows, columns, and fields`
	`dat <- matrix(c(178, 79, 1411, 1486), 2, 2)`
	`dimnames(dat) <- list(Behavior = c('TypeA','TypeB'),`
	`'Heart attack' = c('Y','N'))`
names	`# add field to row \& col names`
	`'dat <- matrix(c(178, 79, 1411, 1486), 2, 2,`
	`dimnames = list(c('TypeA','TypeB'), c('Y','N')))`
	`names(dimnames(dat)) <- c('Behavior', 'Heart attack')`

For example, the UGDP clinical trial 2×2 table can be created *de novo*:

```
ugdp2x2 <- matrix(c(30, 174, 21, 184), 2, 2,
              dimnames = list(Outcome = c('Deaths', 'Survivors'),
              Treatment = c('Tolbutamide', 'Placebo')))
ugdp2x2
```

```
#>            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 'Status' field, the possible values, 'Death' and 'Survivor,' are the row names.

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; add field names next
```

```
#>     Yes  No
#> Yes  30  21
#> No  174 184
```

```
names(dimnames(y)) <- c('Death', 'Tolbutamide'); y
```

```
#>       Tolbutamide
#> Death Yes  No
#>   Yes  30  21
#>   No  174 184
```

Study and test the examples in Table 2.14.

2.3.4 Indexing (subsetting) a matrix

Similar to vectors, a matrix can be indexed by position, by name, or by logical. Study and practice the examples in (Table 2.15). An important skill to master is indexing rows of a matrix using logical vectors.

TABLE 2.15: Common ways of indexing a matrix

Indexing	Try these examples
By logical	`dat[, 1] > 100` `dat[dat[, 1] > 100, ]` `dat[dat[, 1] > 100, , drop = FALSE]`
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]`

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.

```
age <- c(45, 56, 73, 44, 65)
chol <- c(145, 168, 240, 144, 210)
sbp <- c(124, 144, 150, 134, 112)
dat <- cbind(age, chol, sbp); 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, ] # index rows using logical vector '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 `&`.

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 2.16).

TABLE 2.16: Common ways of replacing matrix elements

Replacing	Try these examples
By logical	`qq <- dat[, 1] < 100 # logic vector` `dat[qq, ] <- 99; dat` `dat[dat[, 1] < 100, ] <- c(79, 1486); dat`
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`

2.3.6 Operating 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 <- 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
```

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
#> Outcome     Tolbutamide Placebo
#>   Survivors         174     184
#>   Deaths             30      21
```

```
tab[,2:1]                # reverse columns
```

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

In short, we need to be able to manipulate and execute operations on a matrix. Table 2.17 summarizes key functions (`apply` and `sweep`) for passing other functions to execute operations on a matrix. Table 2.18 summarizes additional convenience functions for operating on a matrix.

TABLE 2.17: Common ways of operating on a matrix

Function	Description	Try these examples
t	Transpose matrix	`dat=matrix(c(178,79,1411,1486),2,2)` `t(dat)`
apply	Apply a function to 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	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`

TABLE 2.18: Alternative ways of operating on a matrix (with equivalents)

Function	Description	Try these examples
`margin.table`, `rowSums`, `colSums`	Marginal sums	`dat <- matrix(5:8, 2, 2)` `margin.table(dat)` `margin.table(dat, 1)` `rowSums(dat)  # equivalent` `apply(dat, 1, sum) # equivalent` `margin.table(dat, 2)` `colSums(dat)  # equivalent` `apply(dat, 2, sum) # equivalent`
`addmargins`	Marginal totals	`addmargins(dat)`
`rowMeans`, `colMeans`	Marginal means	`rowMeans(dat)` `apply(dat, 1, mean) # equivalent` `colMeans(dat)` `apply(dat, 2, mean) # equivalent`
`prop.table`	Marginal distributions	`prop.table(dat)` `dat/sum(dat)  # equivalent` `prop.table(dat, 1)` `sweep(dat, 1, apply(y,1,sum), '/')` `prop.table(dat, 2)` `sweep(y, 2, apply(y, 2, sum), '/')`

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 2.10) 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.

```
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
```

```
addmargins(tab)
```

```
#>            Treatment
#> Outcome     Tolbutamide Placebo Sum
```

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

```
#>   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 2.10 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:

```
x <- c(1, 2, 3, 4, 5)
sumx <- sum(x) # Step 1 summation
x/sumx         # Step 2 division (the 'sweep')
```

```
#> [1] 0.06666667 0.13333333 0.20000000 0.26666667 0.33333333
```

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      0.5882353 0.4117647
#>   Survivors   0.4860335 0.5139665
```

```
ctot <- apply(tab, 2, sum)              # column totals
tab.coldist <- sweep(tab, 2, ctot, '/'); tab.coldist
```

```
#>            Treatment
#> Outcome     Tolbutamide  Placebo
#>   Deaths      0.1470588 0.102439
#>   Survivors   0.8529412 0.897561
```

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      0.5882353 0.4117647
#>   Survivors   0.4860335 0.5139665
```

```
sweep(tab, 2, apply(tab, 2, sum), '/') #column distribution
```

```
#>            Treatment
#> Outcome     Tolbutamide  Placebo
#>   Deaths      0.1470588 0.102439
#>   Survivors   0.8529412 0.897561
```

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

An *array* is the generalization of matrices from 2 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 2.19):

TABLE 2.19: Deaths among subjects who received tolbutamide (TOLB) and placebo in the Unversity Group Diabetes Program (1970) stratifying by age

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

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
```

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 2.20 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.

TABLE 2.20: Year 2000 population estimates by age, ethnicity, sex, and county

County, Sex, & Age	White	Black	AsianPI	Latino	Multirace	AmInd
Alameda						
Female, Age						
<=19	58,160	31,765	40,653	49,738	10,120	839
20–44	112,326	44,437	72,923	58,553	7,658	1,401
45–64	82,205	24,948	33,236	18,534	2,922	822
65+	49,762	12,834	16,004	7,548	1,014	246
Male, Age						
<=19	61,446	32,277	42,922	53,097	10,102	828
20–44	115,745	36,976	69,053	69,233	6,795	1,263
45–64	81,332	20,737	29,841	17,402	2,506	687
65+	33,994	8,087	11,855	5,416	711	156
San Francisco						
Female, Age						
<=$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, Age						
<=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

Displayed in Figure 2.1 is a schematic representation of the 4-dimensional array of population estimates in Table 2.20. 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 2.2 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 State', 'Florida'). For practice, using fictitious data, try the following R code and study the output:

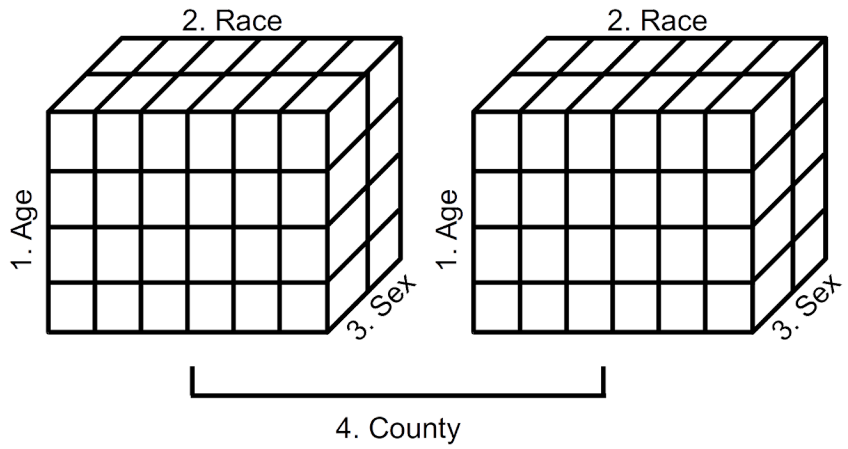


FIGURE 2.1: Schematic representation of a 4-dimensional array (Year 2000 population estimates by age, race, sex, and county)

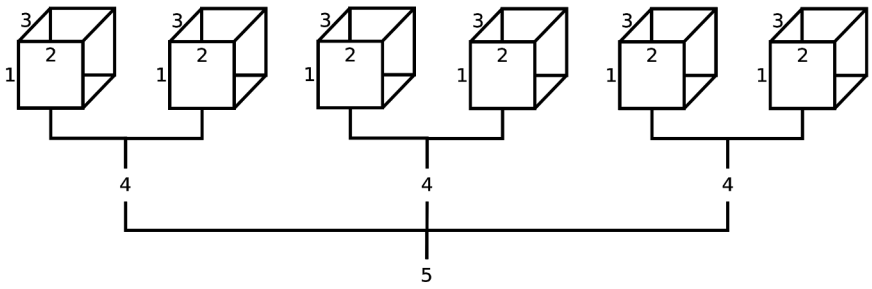


FIGURE 2.2: Schematic representation of a theoretical 5-dimensional array (possibly population estimates by age (1), race (2), sex (3), party affiliation (4), and state (5)). From this diagram, we can infer that the field 'state' has 3 levels, and the field 'party affiliation' has 2 levels; however, 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).

```
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
```

2.4.2 Creating arrays

In R, arrays are most often produced with the `array`, `table`, or `xtabs` functions (Table 2.21). 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
```

TABLE 2.21: Common ways of creating arrays

Function	Description	Try these examples
`array`	Reshapes vector	`aa<-array(1:12,dim=c(2,3,2))`
`table`	Cross-tabulate vectors	`data(infert) # load infert` `table(infert$educ, infert$spont, infert$case)`

Function	Description	Try these examples
`xtabs`	Cross-tabulate data frame	`xtabs(~education + case + parity, data = infert)`
`as.table`	Coercion	`ft <- ftable(infert$educ, infert$spont, infert$case)` `as.table(ft)`
`dim`	Assign dimensions to object	`x <- 1:12` `dim(x) <- c(2, 3, 2)`

The `table` and `xtabs` functions cross tabulate two or more categorical vectors, except that `xtabs` only works with data frames. In R, categorical data are represented by character vectors or factors.[5] First, we cross tabulate character vectors using `table` and `xtabs`.

```
 ## read data 'as is' (no factors created)
udat1 <- read.csv('~/data/ugdp1.txt', as.is = TRUE)
str(udat1)
```

```
#> 'data.frame':    409 obs. of  3 variables:
#>  $ Status   : chr  "Death" "Death" "Death" "Death" ...
#>  $ Treatment: chr  "Tolbutamide" "Tolbutamide" "Tolbutamide" "Tolbutamide" ...
#>  $ Agegrp   : chr  "<55" "<55" "<55" "<55" ...
```

```
table(udat1$Status, udat1$Treatment, udat1$Agegrp)
```

```
#> , ,  = <55
#>
#>
#>            Placebo Tolbutamide
#>   Death          5           8
#>   Survivor     115          98
#>
#> , ,  = 55+
#>
#>
#>            Placebo Tolbutamide
#>   Death         16          22
#>   Survivor      69          76
```

[5] For now, consider factors as character vectors where the possible values (called levels) have been specified. We cover factors later.

The `as~is` option assured that character vectors were not converted to factors, which is the default. Notice that the `table` function produced an array with no field names (i.e., Status, Treatment, Agegrp). This is not ideal. A better option is to use `xtabs`, which uses a formula interface (`~ V1 + V2`):

```
xtabs(~ Status + Treatment + Agegrp, data = udat1)
```

```
#> , , Agegrp = <55
#>
#>           Treatment
#> Status     Placebo Tolbutamide
#>   Death          5           8
#>   Survivor     115          98
#>
#> , , Agegrp = 55+
#>
#>           Treatment
#> Status     Placebo Tolbutamide
#>   Death         16          22
#>   Survivor      69          76
```

Notice that `xtabs` includes the field names. For practice with cross tabulating factors, repeat the above expressions with read in the data setting option `as is` to `FALSE`.

Although `table` does not generate the field names, they can be added manually:

```
table(Outcome = udat1$Status, Therapy = udat1$Treatment,
      Age = udat1$Agegrp)
```

```
#> , , Age = <55
#>
#>           Therapy
#> Outcome    Placebo Tolbutamide
#>   Death          5           8
#>   Survivor     115          98
#>
#> , , Age = 55+
#>
#>           Therapy
#> Outcome    Placebo Tolbutamide
#>   Death         16          22
#>   Survivor      69          76
```

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(Age = udat1$Agegrp, Therapy = udat1$Treatment,
               Outcome = udat1$Status); ftab
```

```
#>                 Outcome Death Survivor
#> Age Therapy
#> <55 Placebo                 5      115
#>     Tolbutamide             8       98
#> 55+ Placebo                16       69
#>     Tolbutamide            22       76
```

```
as.table(ftab)
```

```
#> , , Outcome = Death
#>
#>      Therapy
#> Age   Placebo Tolbutamide
#>   <55       5           8
#>   55+      16          22
#>
#> , , Outcome = Survivor
#>
#>      Therapy
#> Age   Placebo Tolbutamide
#>   <55     115          98
#>   55+      69          76
```

2.4.3 Naming arrays

Naming components of an array is an extension of naming components of a matrix (Table 2.14). Study and implement the examples in Table 2.22.

TABLE 2.22: Common ways of naming arrays

Function	Study and try these examples
array	`x <- c(140, 11, 280, 56, 207, 9, 275, 32)` `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`

Function	Study and try these examples
dimnames	`x <-  c(140, 11, 280, 56, 207, 9, 275, 32)` `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)` `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`

2.4.4 Indexing (subsetting) arrays

Indexing an array is an extension of indexing a matrix (Table 2.15). Study and implement the examples in (Table 2.23).

TABLE 2.23: Common ways of indexing arrays

Indexing	Try these examples
By logical	`zz <- x[ ,1,1]>50; zz` `x[zz, , ]`
By position	`x[1, , ]` `x[ ,2 , ]`
By name (if exists)	`x[ , , 'Males']` `x[ ,'Controls', 'Females']`

2.4.5 Replacing array elements

Replacing elements of an array is an extension of replacing elements of a matrix (Table 2.16). Study and implement the examples in (Table 2.24).

TABLE 2.24: Common ways of replacing array elements

Replacing	Try these examples
By logical	`x > 200` `x[x > 200] <- 999`

Replacing	Try these examples
	`x`
By position	`# use x from` Table 2.22
	`x[1, 1, 1] <- NA`
	`x`
By name (if exists)	`x[ ,'Controls', 'Females']  <- 99; x`

2.4.6 Operating on an array

With the exception of the `aperm` function, operating on an array (Table 2.25) is an extension of operating on a matrix (Table 2.17). The second group of functions, such as `margin.table`, are shortcuts provided for convenience. In other words, these operations are easily accomplished using `apply` and/or `sweep`.

TABLE 2.25: Common ways of operating on an array

Function	Description
`aperm`	Transpose an array by permuting its dimensions
`apply`	Apply a function to the margins of an array
`sweep`	Sweep out a summary statistic from an array
	These short-cuts function use ***`apply`*** *and/or* ***`sweep`***
`margin.table`	For array, sum of table entries for a given index
`rowSums`	Sum across rows of an array
`colSums`	Sum down columns of an array
`addmargins`	Calculate and display marginal totals of an array
`rowMeans`	Calculate means across rows of an array
`colMeans`	Calculate means down columns of an array
`prop.table`	Generates distribution

The `aperm` function is a bit tricky so we illustrated its use.

```
x <-  c(140, 11, 280, 56, 207, 9, 275, 32)
x <- array(x, dim = c(2, 2, 2))
rn <- c('>=1 cups per day', '0 cups per day') # dim 1 values
cn <- c('Cases', 'Controls')                  # dim 2 values
dn <- c('Females', 'Males')                   # dim 3 values
dimnames(x) <- list(Coffee = rn, Outcome = cn, Gender = dn); x
```

```
#> , , Gender = Females
#>
```

```
#>                   Outcome
#> Coffee             Cases Controls
#>   >=1 cups per day   140      280
#>   0 cups per day      11       56
#>
#> , , Gender = Males
#>
#>                   Outcome
#> Coffee             Cases Controls
#>   >=1 cups per day   207      275
#>   0 cups per day       9       32
```

```
aperm(x, c(3, 2, 1))
```

```
#> , , Coffee = >=1 cups per day
#>
#>          Outcome
#> Gender    Cases Controls
#>   Females   140      280
#>   Males     207      275
#>
#> , , Coffee = 0 cups per day
#>
#>          Outcome
#> Gender    Cases Controls
#>   Females    11       56
#>   Males       9       32
```

TABLE 2.26: Example of 3-dimensional array with marginal totals: Primary and secondary syphilis morbidity by age, race, and sex, United State, 1989 (Source: CDC Summary of Notifiable Diseases, United States, 1989, MMWR 1989;38(54))

Age (yrs)	Sex	White	Black	Other	Total
<=19	Male	90	1,443	217	1,750
	Female	267	2,422	169	2,858
	Total	357	3,865	386	4,608
20–29	Male	957	8,180	1,287	10,424
	Female	908	8,093	590	9,591
	Total	1,865	16,273	1,877	20,015
30–44	Male	1,218	8,311	1,012	10,541
	Female	559	4,133	316	5,008

Age (yrs)	Sex	White	Black	Other	Total
	Total	1,777	12,444	1,328	15,549
45+	Male	539	2,442	310	3,291
	Female	79	484	55	618
	Total	618	2,926	365	3,909
Total (by sex & ethnicity)	Male	2,804	20,376	2,826	26,006
	Female	1,813	15,132	1,130	18,075
Total (by ethnicity)		4617	35508	3956	44081

Consider the number of primary and secondary syphilis cases in the United State, 1989, stratified by sex, ethnicity, and age (Table 2.26). This table contains the marginal and joint distribution of cases. Let's read in the original data and reproduce the table results.

```
sdat5 <- read.csv('~/data/syphilis89d.csv')
str(sdat5) # structure of data frame
```

```
#> 'data.frame':    44081 obs. of  3 variables:
#>  $ Sex : Factor w/ 2 levels "Female","Male": 2 2 2 2 2 2 2 2 2 2 ...
#>  $ Race: Factor w/ 3 levels "Black","Other",..: 3 3 3 3 3 3 3 3 3 3 ...
#>  $ Age : Factor w/ 4 levels "<=19","20-29",..: 1 1 1 1 1 1 1 1 1 1 ...
```

```
lapply(sdat5, levels) # levels for each variable
```

```
#> $Sex
#> [1] "Female" "Male"
#>
#> $Race
#> [1] "Black" "Other" "White"
#>
#> $Age
#> [1] "<=19"  "20-29" "30-44" "45+"
```

In R, categorical data are converted into factors. The possible values for each factor are called levels. We used `lapply` to pass the `levels` function and display the levels for each factor. In order to match the table display we need to reorder the levels for Sex and Race, and we recode Age into four levels.

```
sdat5$Sex <- factor(sdat5$Sex, levels = c('Male', 'Female'))
sdat5$Race <- factor(sdat5$Race,levels=c('White','Black','Other'))
lapply(sdat5, levels) # verify
```

```
#> $Sex
#> [1] "Male"   "Female"
#>
#> $Race
#> [1] "White" "Black" "Other"
#>
#> $Age
#> [1] "<=19"  "20-29" "30-44" "45+"
```

We used the `factor` function and the `levels` option to reorder the levels. Now we are ready to create an array and calculate the marginal totals to replicate Table 2.26.

```
sdat <- xtabs(~Sex+Race+Age, data=sdat5) # array
sdat
```

```
#> , , Age = <=19
#>
#>         Race
#> Sex      White Black Other
#>   Male      90  1443   217
#>   Female   267  2422   169
#>
#> , , Age = 20-29
#>
#>         Race
#> Sex      White Black Other
#>   Male     957  8180  1287
#>   Female   908  8093   590
#>
#> , , Age = 30-44
#>
#>         Race
#> Sex      White Black Other
#>   Male    1218  8311  1012
#>   Female   559  4133   316
#>
#> , , Age = 45+
#>
#>         Race
```

```
#> Sex      White Black Other
#>   Male     539  2442   310
#>   Female    79   484    55
```

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(sdat, 1, 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
```

```
#>  <=19 20-29 30-44   45+
#>  4608 20015 15549  3909
```

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      <=19 20-29 30-44  45+
#>   Male   1750 10424 10541 3291
```

```
#>   Female 2858  9591  5008  618
```

```
apply(sdat, c(3, 2), sum) # by age and race
```

```
#>        Race
#> Age     White Black Other
#>   <=19    357  3865   386
#>   20-29  1865 16273  1877
#>   30-44  1777 12444  1328
#>   45+     618  2926   365
```

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    <=19 20-29 30-44  45+
#>   White   90   957  1218  539
#>   Black 1443  8180  8311 2442
#>   Other  217  1287  1012  310
#>
#> , , Sex = Female
#>
#>        Age
#> Race    <=19 20-29 30-44  45+
#>   White  267   908   559   79
#>   Black 2422  8093  4133  484
#>   Other  169   590   316   55
```

Using `aperm` we moved the 2nd dimension (race) into the first position, the 3rd dimension (age) into the second position, and the 1st dimension (sex) into the third position.

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 2.26 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)) # arrange age, sex, race
ftable(sdat.asr)                  # convert 2-D flat table
```

```
#>              Race White Black Other
#> Age   Sex
#> <=19  Male           90  1443   217
#>       Female        267  2422   169
#> 20-29 Male          957  8180  1287
#>       Female        908  8093   590
#> 30-44 Male         1218  8311  1012
#>       Female        559  4133   316
#> 45+   Male          539  2442   310
#>       Female         79   484    55
```

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

```
ftable(addmargins(sdat.asr))
```

```
#>              Race White Black Other   Sum
#> Age   Sex
#> <=19  Male           90  1443   217  1750
#>       Female        267  2422   169  2858
#>       Sum           357  3865   386  4608
#> 20-29 Male          957  8180  1287 10424
#>       Female        908  8093   590  9591
#>       Sum          1865 16273  1877 20015
#> 30-44 Male         1218  8311  1012 10541
#>       Female        559  4133   316  5008
#>       Sum          1777 12444  1328 15549
#> 45+   Male          539  2442   310  3291
#>       Female         79   484    55   618
#>       Sum           618  2926   365  3909
#> Sum   Male         2804 20376  2826 26006
#>       Female       1813 15132  1130 18075
#>       Sum          4617 35508  3956 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':    24 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/ 4 levels "<=19","20-29",..: 1 1 1 1 1 1 2 2 2 2 ...
#>  $ Freq: int  90 267 1443 2422 217 169 957 908 8180 8093 ...
```

```
sdat.df[1:8, ]
```

```
#>      Sex  Race   Age Freq
#> 1   Male White  <=19   90
#> 2 Female White  <=19  267
#> 3   Male Black  <=19 1443
#> 4 Female Black  <=19 2422
#> 5   Male Other  <=19  217
#> 6 Female Other  <=19  169
#> 7   Male White 20-29  957
#> 8 Female White 20-29  908
```

The group-level data frame is practical only if all the fields are categorical data, as is the case here. For additional practice, study and implement the examples in Table 2.25.

2.5 Selected applications

2.5.1 Probabilistic reasoning with Bayesian networks

A Bayesian network (BN) is a graphical model for representing probabilistic, but not necessarily causal, relationships between variables called *nodes* [11,12]. The nodes are connected by lines called *edges* which, for our purposes, are always *directed* with an arrow. Consider this *noncausal* BN:

$$\text{Smell smoke} \rightarrow \text{Fire nearby}$$

Smelling smoke increases the probability of a fire burning nearby, but obviously smoke alone does not cause a fire. In other words, does knowing X (smell smoke) change the *credibility* of Y (fire nearby)? In contrast, now consider this *causal* BN:[6]

$$\text{Fire} \rightarrow \text{Smoke}$$

This causal BN depicts fire causing smoke. Notice that both noncausal and causal BNs have probabilistic dependence which we will use for probabilistic reasoning. Noncausal BNs are commonly used in *influence diagrams*[7] for decision analysis which we cover later in the book.

[6] also called a *causal graph* or a *directed acyclic graph* (DAG)
[7] also called *decision networks* or *relevance diagrams*

A two-node causal BN has two types of proabilistic reasoning (Table 2.27).

$$\text{Cause} \rightarrow \text{Effect}$$

TABLE 2.27: Probabilistic reasoning for 2-node causal BN

Probabilistic reasoning	Conditional probability
Causal (predictive) reasoning	$P(\text{Effect} \mid \text{Cause})$
Evidential (diagnostic) reasoning	$P(\text{Cause} \mid \text{Effect})$

When a causal effect is not firmly established, the BN asserts this concept:

$$\text{Hypothesis} \rightarrow \text{Evidence}$$

TABLE 2.28: Bayesian network involving a hypothesis and evidence

Conditional probability	Probabilistic reasoning
$P(\text{Evidence} \mid \text{Hypothesis})$	Causal reasoning
$P(\text{Hypothesis} \mid \text{Evidence})$	Evidential reasoning

Evidential reasoning requires Bayes Theorem.

$$\begin{aligned} P(H \mid E) &= \frac{P(E \mid H)P(H)}{P(E)} \\ &= \frac{P(E \mid H)P(H)}{P(E \mid H)P(H) + P(E \mid \neg H)P(\neg H)} \end{aligned}$$

$P(H)$ is the *prior (old) belief*, $P(H \mid E)$ is the *posterior (new) belief*, and $P(E \mid H)$ is called the *likelihood*. The likelihood is critical because it is usually measurable. The challenge is that we need Bayes Theorem for two reasons:

1. to use evidence and theory to update our belief from $P(H)$ to $P(H \mid E)$, and
2. to avoid the *fallacy of the transposed conditional*; i.e., confusing $P(E \mid H)$ with $P(H \mid E)$.[8]

2.5.1.1 Example: HIV testing

For example, from Neapolitan (p. 491, [13]), suppose Sam takes a test (evidence) to determine whether he has HIV infection (hypothesis). Here is the BN:

[8] For example, because African Americans make up a high proportion in our criminal justice system, some people believe, mistakenly, that a high proportion African Americans are involved in crimes. To understand this phenomenon we would need a full causal model.

HIV infection → Test results

In diagnostic testing we use Bayes Theorem to calculate the post-test probability from the test results, pre-test probability (prevalence of infection), and test characteristics (sensitivity, specificity). Table 2.29 displays the data we need for applying Bayes Theorem.

TABLE 2.29: Probabilities for using Bayes Theorem in diagnostic testing

Name	Probabilities	Value
Pre-test (prior) probability of HIV+	$P(\text{HIV}+)$	0.00001
Sensitivity	$P(\text{Test}+ \mid \text{HIV}+)$	0.999
Specificity	$P(\text{Test}- \mid \text{HIV}-)$	0.998
Post-test (posterior) probability	$P(\text{HIV} \mid \text{Test})$	TBD

For illustrative purposes, we calculate the "positive predictive value" (PPV).

$$P(H+ \mid T+) = \frac{P(T+ \mid H+)P(H+)}{P(T+ \mid H+)P(H+) + P(T+ \mid H-)P(H-)}$$

This is easy to calculate in R.

```
prior <- 0.00001
sens <- 0.999
spec <- 0.998
(sens*prior)/(sens*prior+(1-spec)*(1-prior)) # PPV
```

```
#> [1] 0.004970223
```

Calculating the PPV (or NPV) is evidential reasoning and it requires applying Bayes Theorem. Our brains cannot calculate posterior probabilities reliably; we need computational tools. In other words, for the simplest 2-node graph, our brains are not able to "flip the arrow" from $P(\text{Evidence} \mid \text{Hypothesis})$ to $P(\text{Hypothesis} \mid \text{Evidence})$ and make valid Bayesian calculations.

2.5.2 Causal graphs—the story behind the data

2.5.2.1 Example: A retrospective observation study

A vector (one variable) is represented by a single node. A matrix (cross-tabulation of two variables) is represented by two nodes. A 3-dimensional array (cross-tabulation of two variables) is represented by three nodes. And, an n-dimensional array is represented by n nodes. Therefore, we can think of these arrays as **stratified contigency tables**.

The *story* is how the data was generated (data generating process) *and* how

the data was measured (study design). With a causal graph we encode the salient parts of the story that can only come from human knowledge. Data, by themselves, cannot tell us how they were generated.

In 1986, Charig published a retrospective observational study comparing open surgery (Treatment A) to percutaneous nephrolithotomy (Treatment B) for the treatment of kidney stones [14,15]. Treatment B is noninvasive. Success was defined as no stones at three months. The data are presented in Table 2.30.

TABLE 2.30: Comparison of Treatment A to B for kidney stones, by size

Stone size	Treatment A	Success (%)	Treatment B	Success (%)
Small stones	81/87	**93**	234/270	87
Large stones	192/263	**73**	55/80	69
Combined	273/350	78	289/350	**83**

If you believe in "data-driven" decision-making then you have a big problem! When you stratify the data and know the stone size, Treatment A is better for both size. However, if you do not stratify and do not know the stone size, Treatment B is better. Which is correct? No amount of data or data analysis will answer this question—that's why it's call a paradox—in this case, **Simpson's paradox**. We need more information (the story) and we need a method (causal graphs) to encode this story.

We have three variables: X: treatment (A, B), Y: success (yes, no), Z: stone size (small, large). How are they related? The obvious relationship is the causal effect of Treatment on Success.

FIGURE 2.3: Causal graph of treatment (X) and success (Y)

```
dat1 <- c(273, 350-273, 289, 350-289)
dat1 <- array(dat1, dim = c(2, 2), dimnames = list("Success (Y)"
                = c('Yes', 'No'), "Treatment (X)" = c('A', 'B')))
dat1
```

```
#>             Treatment (X)
#> Success (Y)   A   B
#>         Yes 273 289
#>         No   77  61
```

```
prop.table(dat1, margin = 2)
```

```
#>             Treatment (X)
#> Success (Y)    A         B
#>         Yes 0.78 0.8257143
#>         No  0.22 0.1742857
```

Now, here is the missing story: It turns out that urologists select open surgery (Treatment A) for more severe cases, and larger stones are considered more severe. In other words, severity causes a selection of treatment, and we can see the resulting association in the data.

```
dat2 <- c(87, 270, 263, 80)
dat2 <- array(dat2, dim = c(2,2), dimnames = list("Treatment (X)"
          = c('A', 'B'), "Stone.Size (Z)" = c('Small', 'Large')))
dat2
```

```
#>              Stone.Size (Z)
#> Treatment (X) Small Large
#>             A    87   263
#>             B   270    80
```

```
prop.table(dat2, margin = 2)
```

```
#>              Stone.Size (Z)
#> Treatment (X)     Small     Large
#>             A 0.2436975 0.7667638
#>             B 0.7563025 0.2332362
```

Of course, stone size (i.e., severity) has a causal effect on success. We now have enough information to construct the full causal graph (Figure 2.4).

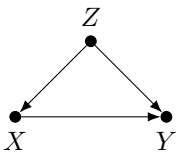


FIGURE 2.4: Causal graph of treatment (X), success (Y), and stone size (Z).

Figure 2.4 depicts the three possible patterns we can observe with three nodes:

1. **chain** (sequential cause): $Z \to X \to Y$

2. **fork** (common cause): $X \leftarrow Z \rightarrow Y$
3. **collider** (common effect): $X \rightarrow Y \leftarrow Z$

In this causal graph (Figure 2.4), the stone size (Z) is a common cause (fork) and is also called a **confounder** because it distorts the causal effect of treatment (X) on success (Y).

Later we will learn how to use the causal graph to derive an adjustment formula to **de-confound** the causal effect of treatment on success. This is not possible with the data table alone; the causal graph encodes the story, allowing us to adjust for confounding.

KEY IDEA: All causal graphs, regardless of complexity, are constructed from the three core patterns: chain (sequential cause), fork (common cause), and collider (common effect). The causal graph encodes the "story" of the data generating process. By itself the data tables never tell the full story—we need a causal graph.

2.5.2.2 Conditional probability tables in R

A directed acyclic graph (DAG) is a causal Bayesian network (BN). A BN is a probabilistic graphical model where nodes are conditional probability tables and connecting arrows encode probabilistic dependence. Depending on the probabilistic assessment, in BNs arrows can "flip." However, in DAGs the arrows cannot flip because they represent *causal effects*, yet we can still evaluate a DAG as a probabilistic graphical model.

For example, Figure 2.4 represents data that has a *joint probability distribution*. Using *product decomposition* we can rewrite it as a product of conditional probabilities that we read directly from the DAG.

$$\Pr(X, Y, Z) = \Pr(Z) \Pr(X \mid Z) \Pr(Y \mid X, Z)$$

There are no arrows into node Z so this is represented by the marginal probability $\Pr(Z)$. There is an arrow from *parent* node Z to *child* node X; therefore, we have the conditional probability $\Pr(X \mid Z)$. Finally, we have an arrow from X to Y and from Z to Y; therefore, we have the conditional probability $\Pr(Y \mid X, Z)$.

Using the data from Table 2.30, let's use R to construct these conditional probability tables. We start with $\Pr(Y \mid X, Z)$.

```
dat.yxz <- c(81, 87-81, 234, 270-234, 192, 263-192, 55, 80-55)
dat.yxz <- array(dat.yxz, dim = c(2,2,2), dimnames
   = list("Success (Y)"=c('Yes','No'), "Treatment (X)"=c('A','B'),
                          "Stone.size (Z)" = c('Small','Large')))
dat.yxz
```

```
#> , , Stone.size (Z) = Small
#>
#>            Treatment (X)
#> Success (Y)  A   B
#>         Yes 81 234
#>         No   6  36
#>
#> , , Stone.size (Z) = Large
#>
#>            Treatment (X)
#> Success (Y)   A  B
#>         Yes 192 55
#>         No   71 25
```

```
prop.table(dat.yxz, margin = c(2,3))
```

```
#> , , Stone.size (Z) = Small
#>
#>            Treatment (X)
#> Success (Y)          A         B
#>         Yes 0.93103448 0.8666667
#>         No  0.06896552 0.1333333
#>
#> , , Stone.size (Z) = Large
#>
#>            Treatment (X)
#> Success (Y)        A      B
#>         Yes 0.730038 0.6875
#>         No  0.269962 0.3125
```

Notice what the option `margin = c(2, 3)` accomplished: "Stratified by dimensions 2 (X) and 3 (Z), provide the distribution along the remaining dimension(s) (in this case, dimension 1 or Y)." This enables us to assess the probability of success given treatment, stratified by stone size (severity).

Now, let's construct $\Pr(X \mid Z)$. Recall, we already constructed this two-way table *de novo*, but now we use the `apply` function instead.

```
(dat.xz <- apply(dat.yxz, c(2,3), sum)) # assign and display
```

```
#>              Stone.size (Z)
#> Treatment (X) Small Large
#>             A    87   263
#>             B   270    80
```

```
prop.table(dat.xz, margin = 2)
```

```
#>              Stone.size (Z)
#> Treatment (X)     Small     Large
#>             A 0.2436975 0.7667638
#>             B 0.7563025 0.2332362
```

This table is the crude, combined or unstratified table.

Finally, let's construct $\Pr(Z)$, again using `apply`.

```
(dat.z <- apply(dat.xz, 2, sum)) # assign and display
```

```
#> Small Large
#>   357   343
```

```
prop.table(dat.z)
```

```
#> Small Large
#>  0.51  0.49
```

Consider DAGs (Figure 2.4) and data (Table 2.30) two sides of the same coin. They both give us different but complementary information. Causal models represent our beliefs or hypotheses of how the world works (the story). As the world unfolds it generates data which we can measure and analyze. Because one particular instance of data (e.g., a stratified contingency table) can be generated from many different data generating processes we need a causal model to guide us. For now, we are learning to use R to encode this information (data) and knowledge (DAG).

2.6 Exercises

Exercise 2.1 (Find file path to working directory)**.** From RStudio main menu, select `File` → `New Project` → `New Directory` → `Empty Project`. Name the

new directory `ph251d-hwk`. At the R console, use R to display the file path to the work directory containing `.RData`?

Exercise 2.2 (Recreate 2 x 2 Table)**.** The **Whickham** data set is "Data on age, smoking, and mortality from a one-in-six survey of the electoral roll in Whickham, a mixed urban and rural district near Newcastle upon Tyne, in the UK. The survey was conducted in 1972–1974 to study heart disease and thyroid disease. A follow-up on those in the survey was conducted twenty years later. This dataset contains a subset of the survey sample: women who were classified as current smokers or as never having smoked." (source: `mosaicData` package)

A data frame with 1314 observations on women for the following variables:

- `outcome`: survival status after 20 years: a factor with levels Alive Dead
- `smoker`: smoking status at baseline: a factor with levels No Yes
- `age`: age (in years) at the time of the first survey

Using the counts in Table 2.31 recreate the table using any combination of the `matrix`, `cbind`, `rbind`, `dimnames`, or `names` functions.

TABLE 2.31: Risk of death in a 20-year period among women in Whickham, England, according to smoking status at the beginning of the period [10]

Outcome	Smoker	
	Yes	No
Dead	139	230
Alive	443	502

Exercise 2.3 (Create table marginal totals)**.** Starting with the 2x2 matrix object we created in Table 2.31, using any combination of `apply`, `cbind`, `rbind`, `names`, and `dimnames` functions, recreate the Table 2.32.

TABLE 2.32: Risk of death in a 20-year Period among women in Whickham, England, according to smoking status at the beginning of the period

Outcome	Smoker		
	Yes	No	Total
Dead	139	230	329
Alive	443	502	945
Total	582	732	1314

Exercise 2.4 (Create marginal and joint probability distributions)**.** Using the 2×2 data from Table 2.31, use the `sweep` and `apply` functions to calculate row, column, and joint probability distributions (i.e., create three tables with proportions).

Exercise 2.5 (Create measures of association)**.** Using the data from the previous problems, recreate Table 2.33 and interpret the results.

TABLE 2.33: 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

Measure	Smoker	
	Yes	No
Risk	0.24	0.31
Risk Ratio	0.76	1.00
Odds	0.31	0.46
Odds Ratio	0.68	1.00

Exercise 2.6 (Conduct analyses)**.** Install the `mosaicData` R package using `install.packages("mosaicData")`. Load the `Whickham` data set. Using the `xtabs` function create two-way and three-way contingency tables. Calculate "measures of associations." What is your interpretation? Here is some R code to get you started.

```
library(mosaicData)
data(Whickham)
wdat <- Whickham
## discretize age into 4-level categorical variable
wdat$age4 <- cut(wdat$age, breaks=c(15, 25, 45, 65, 100),
  right = FALSE)
```

Exercise 2.7 (Practice tapply)**.** Use the `tapply` function to calculate the mean age at study entry comparing

(a) smokers to non-smokers,
(b) dead vs. alive,
(c) smoker status stratified by outcome status (2 x 2 table).

Exercise 2.8 (California home prices, July, 2019)**.** Read into R the median prices for single family homes in selected California counties for July, 2019.

The R code is provided. The data object will be a data frame, but can be manipulated like a matrix.

```
hp <-
read.csv("https://raw.githubusercontent.com/taragonmd/data/master/homes.csv",
as.is = TRUE)
```

(a) Create a character vector of the county names.
(b) Create a numeric vector of the home prices.
(c) Which county has the lowest price?
(d) Which county has the highest price?
(e) What is the median price for California counties? (i.e., the median of the median prices)
(f) What is the mean price? (i.e., the mean of the median prices)
(g) The data frame (`hp`) can be treated like a matrix. Sort the data frame by county name (hint: use the `order` function).
(h) Sort the data frame by home price (hint: use the `order` function).
(i) List the counties that have medican home prices between the 25% and 75% percentiles (hint: use the `quantile` function and index using a Boolean query).

3

Working with lists and data frames

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

3.1.1 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. 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.

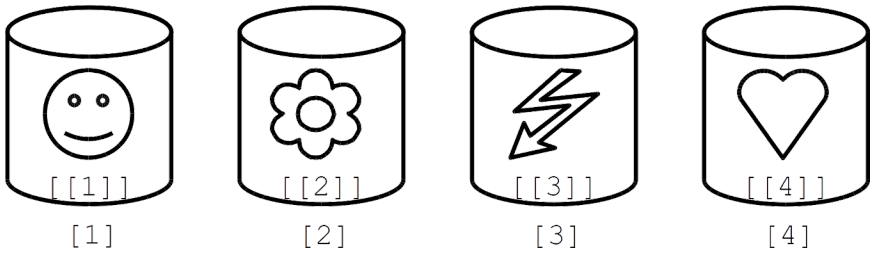


FIGURE 3.1: Schematic representation of a list of length four.

Figure 3.1 is a 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.

For example, using the UGDP clinical trial data,[1] 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.

[1] https://github.com/taragonmd/data

```
udat <- read.csv("~/data/ugdp1.txt")
tab <- xtabs(~Status+Treatment, data = udat)[,2:1]; tab
```

```
#>           Treatment
#> Status     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:
#>  0.8013768 2.8872863
#> sample estimates:
#> odds ratio
#>   1.509142
```

The default display shows partial results. All the 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   : num [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
```

```
#>   1.509142
```

```
ftab$conf.int
```

```
#> [1] 0.8013768 2.8872863
#> attr(,"conf.level")
#> [1] 0.95
```

```
ftab$p.value
```

```
#> [1] 0.1812576
```

The `str` function returns the structure of the output object. to extract additional results either for display or for further analysis. In this case, `ftab` is a list with 7 bins, each with a name.

3.1.2 Creating lists

To create a list directly, use the `list` function. Table 3.1 summarizes how to create lists.

TABLE 3.1: Common ways of creating a list

Function	Description	Try these examples
`list`	Creates list object	`y <- matrix(4:1, 2,2)` `z <- c("Pedro", "Paulo")` `mm <- list(y, z); mm`
`data.frame`	Vectors to tabular list	`id <- 1:3` `sex <- c("M","F", "M")` `df <- data.frame(id, sex); df`
`as.data.frame`	Coercion	`x <- matrix(1:6, 2, 3); x` `y <- as.data.frame(x); y`
`read.table`	Read ASCII file	`read.table('~/data/ugdp.txt')`
`read.csv`		`read.csv('~/data/ugdp.txt')`
`vector`	Empty list length n	`vector("list", 2)`
`as.list`	Coercion into list	`list(1:2)` `as.list(1:2)`

A great use for a list is to collecte the results of a function into an object. 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
```

```
#>           Treatment
#> Status     Tolbutamide Placebo
#>   Death              30      21
#>   Survivor          174     184
```

```
orcalc(tab) # run function
```

```
#> $data
#>           Treatment
#> Status     Tolbutamide Placebo
#>   Death              30      21
#>   Survivor          174     184
#>
#> $odds.ratio
#> [1] 1.510673
#>
#> $p.value
#> [1] 0.1812576
```

For additional practice, study and implement the examples in Table 3.1.

3.1.3 Naming lists

TABLE 3.2: Common ways of naming lists

Function	Comment	Try these examples
names	After list creation	`z <- list(rnorm(20), 'Luis'); z` `names(z) <- c('bin1', 'bin2')`
	At list creation	`z <- list(bin1 = rnorm(20), bin2 = 'Luis')` `names(z) # returns names`

The components (or '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 3.2.

3.1.4 Indexing lists

TABLE 3.3: Common ways of indexing lists

Indexing	Try these examples
By position	`z <- list(bin1 = rnorm(20), bin2 = 'Luiz')`
	`z[1]   # indexes bin #1`
	`z[[1]] # indexes contents of bin #1`
By name (if exists)	`z$bin1 # indexes contents of bin #1`
By logical vector	`num <- sapply(z, is.numeric); num`
	`z[num]`

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 a single bin only.

```
mylist1 <- list(1:5, matrix(1:4,2,2), c('Juan', 'Wilfredo'))
mylist1[c(1, 3)] # index bins 1 and 3
```

```
#> [[1]]
#> [1] 1 2 3 4 5
#>
#> [[2]]
#> [1] "Juan"     "Wilfredo"
```

```
mylist1[[3]]     # index contents of 3rd bin
```

```
#> [1] "Juan"     "Wilfredo"
```

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.

```
library(survival)
data(infert)
names(infert)  # display field names
```

```
#> [1] "education"      "age"            "parity"
#> [4] "induced"        "case"           "spontaneous"
#> [7] "stratum"        "pooled.stratum"
```

```
mod1 <- clogit(case ~ spontaneous + induced + strata(stratum),
               data = infert)
mod1           # default display of model
```

```
#> Call:
#> clogit(case ~ spontaneous + induced + strata(stratum), data = infert)
#>
#>               coef exp(coef) se(coef)     z        p
#> spontaneous 1.9859    7.2854   0.3524 5.635 1.75e-08
#> induced     1.4090    4.0919   0.3607 3.906 9.38e-05
#>
#> Likelihood ratio test=53.15  on 2 df, p=2.869e-12
#> n= 248, number of events= 83
```

```
is.list(mod1)
```

```
#> [1] TRUE
```

```
names(mod1)    # names of list components
```

```
#>  [1] "coefficients"      "var"               "loglik"
#>  [4] "score"             "iter"              "linear.predictors"
#>  [7] "residuals"         "means"             "method"
#> [10] "n"                 "nevent"            "terms"
#> [13] "assign"            "wald.test"         "concordance"
#> [16] "y"                 "timefix"           "formula"
#> [19] "xlevels"           "call"              "userCall"
```

```
mod1$coeff
```

```
#> spontaneous     induced
#>    1.985876    1.409012
```

The `stratum` option is used to specify the field that identifies the cases and their matched controls. The `mod1` object has a default display (shown). However, it is a list. We can use `str` to evaluate the list structure. Instead `names` was used to display the list component names. Sometimes it is more convenience to display the names for the list rather than the complete structure.

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 summary results
```

```
#> Call:
#> coxph(formula = Surv(rep(1, 248L), case) ~ spontaneous + induced +
#>     strata(stratum), data = infert, method = "exact")
#>
#>   n= 248, number of events= 83
#>
#>                coef exp(coef) se(coef)     z Pr(>|z|)
#> spontaneous 1.9859    7.2854   0.3524 5.635 1.75e-08 ***
#> induced     1.4090    4.0919   0.3607 3.906 9.38e-05 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#>             exp(coef) exp(-coef) lower .95 upper .95
#> spontaneous     7.285     0.1373     3.651    14.536
#> induced         4.092     0.2444     2.018     8.298
#>
#> Concordance= 0.776  (se = 0.044 )
#> Likelihood ratio test= 53.15  on 2 df,   p=3e-12
#> Wald test            = 31.84  on 2 df,   p=1e-07
#> Score (logrank) test = 48.44  on 2 df,   p=3e-11
```

```
names(summod1) # names of list components
```

```
#>  [1] "call"         "fail"         "na.action"    "n"
#>  [5] "loglik"       "nevent"       "coefficients" "conf.int"
#>  [9] "logtest"      "sctest"       "rsq"          "waldtest"
#> [13] "used.robust"  "concordance"
```

```
summod1$coef
```

```
#>                 coef exp(coef)  se(coef)        z     Pr(>|z|)
#> spontaneous 1.985876  7.285423 0.3524435 5.634592 1.754734e-08
#> induced     1.409012  4.091909 0.3607124 3.906191 9.376245e-05
```

3.1.5 Replacing lists components

TABLE 3.4: Common ways of replacing list components

Replacing	Try these examples
By position	`z <- list(bin1 = rnorm(20), bin2 = 'Luis')` `z[1] <- list(c(1, 2)) # replaces bin contents` `z[[1]] <- c(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(5)); z`

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 3.4.

3.1.6 Operating 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` "simplifies" the final result, if possible.

TABLE 3.5: Common ways of operating on a list

Function	Description with examples
`lapply`	Applies function to list and returns list `x <- list(1:5, 6:10)` `lapply(x, mean)`
`sapply`	Applies function to 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 1st bin of each argument, the 2nd, the 3rd, and so on. `x <- list(1:4, 1:4); y <- list(4, rep(4, 4))` `mapply(rep, x, y, SIMPLIFY = FALSE)` `mapply(rep, x, y)`

The `do.call` function applies a function to the entire list using each each com-

ponent 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 3.5.

3.2 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 operations that can be done with a matrix can be done with a data frame.

3.2.1 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","6-11yrs",..: 1 1 1 1 2 2 2 2 2 2 ...
#>  $ 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 ...
#>  $ pooled.stratum: num  3 1 4 2 32 36 6 22 5 19 ...
```

```
infert[1:5, 1:6]
```

```
#>   education age parity induced case spontaneous
#> 1    0-5yrs  26      6       1    1           2
#> 2    0-5yrs  42      1       1    1           0
#> 3    0-5yrs  39      6       2    1           0
#> 4    0-5yrs  34      4       2    1           0
#> 5   6-11yrs  35      3       1    1           1
```

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

```
infert$age[1:10]        # first 10 of age variable
```

```
#>  [1] 26 42 39 34 35 36 23 32 21 28
```

```
mode(infert$age)
```

```
#> [1] "numeric"
```

```
class(infert$age)
```

```
#> [1] "numeric"
```

```
infert$stratum[1:10]    # first 10 of stratum variable
```

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

```
mode(infert$stratum)
```

```
#> [1] "numeric"
```

```
class(infert$stratum)
```

```
#> [1] "integer"
```

```
infert$education[1:6]   # first 10 of education variable
```

```
#> [1] 0-5yrs  0-5yrs  0-5yrs  0-5yrs  6-11yrs 6-11yrs
#> 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 an integer (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[1:10]
```

```
#>  [1] 1 1 1 1 2 2 2 2 2 2
```

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

```
cointoss <- sample(c('Head','Tail'), 100, replace = TRUE)
cointoss[1:7]
```

```
#> [1] "Tail" "Head" "Head" "Head" "Tail" "Head" "Head"
```

```
fct <- factor(cointoss); fct[1:7]
```

```
#> [1] Tail Head Head Head Tail Head Head
#> Levels: Head Tail
```

```
unclass(fct)[1:7]
```

```
#> [1] 2 1 1 1 2 1 1
```

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

```
as.character(cointoss)[1:7]
```

```
#> [1] "Tail" "Head" "Head" "Head" "Tail" "Head" "Head"
```

```
as.character(fct)[1:7]
```

```
#> [1] "Tail" "Head" "Head" "Head" "Tail" "Head" "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 = TRUE)
ofac1[1:7]
```

```
#> [1] High   Medium High   Low    Low    Low    High
#> Levels: High < Low < Medium
```

```
table(ofac1) # levels and labels not in natural order
```

```
#> ofac1
#>   High    Low Medium
#>     25     29     46
```

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

```
ofac2 <- factor(samp, levels = c('Low', 'Medium', 'High'),
                ordered = TRUE)
ofac2[1:7]
```

```
#> [1] High   Medium High   Low    Low    Low    High
#> Levels: Low < Medium < High
```

```
table(ofac2)
```

```
#> ofac2
#>    Low Medium   High
#>     29     46     25
```

Great—this is exactly what we want! For review, Table 3.6 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. Notice that in R the internal representation of categorical variables are of mode numeric. We can see this using the `unclass` function on the variable.

TABLE 3.6: Variable types in data and their representations in R

Variable type	Examples	`mode`	`typeof`	`class`
Numeric				
Continuous	1, 2, 3.45, 2/3	numeric	double	numeric
Discrete	1, 2, 3, 4, …	numeric	integer	integer
Categorical				
Nominal	male vs. female	numeric	integer	factor
Ordinal	low < medium < high	numeric	integer	ordered factor

3.2.2 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. Common ways of creating data frames are summarized in Table 3.7. Factors can be created directly from vectors.

```
wt <- c(59.5, 61.4, 45.2)
age <- c(11, 9, 6)
gender <- c('Male', 'Male', 'Female')
df <- data.frame(age, gender, wt); df
```

```
#>   age gender   wt
#> 1  11   Male 59.5
#> 2   9   Male 61.4
```

```
#> 3   6 Female 45.2
```

```
str(df)
```

```
#> 'data.frame':    3 obs. of  3 variables:
#>  $ age   : num  11 9 6
#>  $ gender: Factor w/ 2 levels "Female","Male": 2 2 1
#>  $ wt    : num  59.5 61.4 45.2
```

```
table(df$gender)
```

```
#>
#> Female   Male
#>      1      2
```

Notice that the character vector was automatically converted to a factor. In this example, possible values of `gender` were derived from the data vector. However, if possible values of `gender` included "transgender" (although none were recorded), we can specify this using using levels option.

```
df$gender <- factor(df$gender, levels = c('Male', 'Female',
                    'Transgender'))
table(df$gender)
```

```
#>
#>        Male      Female Transgender
#>           2           1           0
```

Factors allow us to keep track of possible values in categorical data even if specific values were not recorded in a given data set.

TABLE 3.7: Common ways of creating data frames

Function	Description	Try these examples
`data.frame`	Create tabular list	`id <- 1:2; sex <- c('M','F')` `x <- data.frame(id, sex)`
	Convert fully labeled array	`data(Titanic) # load data` `data.frame(Titanic)`
`as.data.frame`	Coercion	`x <- matrix(1:6, 2, 3); x` `as.data.frame(x)`
`read.table`	Reads ASCII text file	see `help(read.table)`
`expand.grid`	All combinations of vectors	see `help(expand.grid)`

Sometimes we want to convert an array into a data frame for analysis or sharing. Consider the Titanic array data that come with R. We can convert an array to a data frame (first 6 rows shown):

```
data(Titanic)                          # load data
Titanic[,,'Child',]                    # display children
```

```
#> , , Survived = No
#>
#>       Sex
#> Class  Male Female
#>   1st     0      0
#>   2nd     0      0
#>   3rd    35     17
#>   Crew    0      0
#>
#> , , Survived = Yes
#>
#>       Sex
#> Class  Male Female
#>   1st     5      1
#>   2nd    11     13
#>   3rd    13     14
#>   Crew    0      0
```

```
dft <- data.frame(Titanic); dft[1:6, ] # convert and display
```

```
#>   Class    Sex   Age Survived Freq
#> 1   1st   Male Child       No    0
#> 2   2nd   Male Child       No    0
#> 3   3rd   Male Child       No   35
#> 4  Crew   Male Child       No    0
#> 5   1st Female Child       No    0
#> 6   2nd Female Child       No    0
```

3.2.3 Naming data frames

TABLE 3.8: Common ways of naming data frames

Function	Try these examples
names	`x <- data.frame(var1 = 1:3, var2 = c('M', 'F', 'F'))` `names(x) <- c('Subjno', 'Sex')`
row.names	`row.names(x) <- c('Sub 1', 'Sub 2', 'Sub 3'); x`

Everything that applies to naming list components (Table 3.2) also applies to naming data frame components (Table 3.8). In general, we may be interested in renaming variables (fields) or row names of a data frame, or renaming the levels (possible values) for a given factor (categorical variable). For example, consider the Oswego data set.

```
odat <- read.table('~/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[1:4, 1:6]             #Display partial data frame
```

```
#>   id age Gender meal.time ill onset.date
#> 1  2  52 Female   8:00 PM   Y       4/19
#> 2  3  65   Male   6:30 PM   Y       4/19
#> 3  4  59 Female   6:30 PM   Y       4/19
#> 4  6  63 Female   7:30 PM   Y       4/18
```

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)[1:10]
```

```
#>  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
```

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

```
row.names(odat) <- sample(101:199, size = nrow(odat))
odat[1:4, 1:6]
```

```
#>     id age Gender meal.time ill onset.date
#> 145  2  52 Female   8:00 PM   Y       4/19
#> 139  3  65   Male   6:30 PM   Y       4/19
#> 105  4  59 Female   6:30 PM   Y       4/19
#> 113  6  63 Female   7:30 PM   Y       4/18
```

3.2.4 Indexing data frames

TABLE 3.9: 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 # create logical vector`
	`infert[agelt30, c("education", "induced", "parity")]`
	`# can also use 'subset' function`
	`subset(infert, agelt30, c("education", "induced", "parity"))`

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 ile virus human disease surveillance data. Suppose we are interested in summarizing the Los Angeles cases with neuroinvasive disease ("WNND").

```
wdat <- read.csv("~/data/wnv/wnv2004fin.csv")
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 14 14 14 14 14 14 14 8 12 .
#>  $ 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 2 2 ...
#>  $ syndrome   : Factor w/ 3 levels "Unknown","WNF",..: 2 2 2 2 2 2 3 3 2 3 ...
#>  $ date.onset : Factor w/ 130 levels "2004-05-14","2004-05-16",..: 3 4 4 2 1 5 7 10 7 9
#>  $ date.tested: Factor w/ 104 levels "2004-06-02","2004-06-16",..: 1 2 2 2 2 3 4 5 6 6
#>  $ death      : Factor w/ 2 levels "No","Yes": 1 1 1 1 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[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
```

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.

3.2.5 Replacing data frame components

TABLE 3.10: 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)`

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 3.10.

3.2.6 Operating on data frames

TABLE 3.11: Common ways of operating on a data frame

Function	Description with examples
`tapply`	Applies function to strata of a vector `data(infert)` `args(tapply) # display argument` `tapply(infert$age, infert$educ, mean, na.rm=TRUE)`
`lapply`	Applies function to list and returns list `lapply(infert[, 1:3], table)`
`sapply`	Applies function to list and simplifies

Function	Description with examples
	`sapply(infert[,c("age","parity")],mean,na.rm=TRUE)`
`aggregate`	Split data into subsets, and computes statistic for each `aggregate(infert[, c("age", "parity")], by = list(Education = infert$education, Induced = infert$induced), mean)`
`mapply`	Applies function to x bin of each argument, for $x = 1, 2, \cdots$ `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 = FALSE)`

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 year California population estimates for the year 2000.

```
capop <- read.csv("~/data/calpop/CalCounties2000.csv")
str(capop)
```

```
#> 'data.frame':    11918 obs. of  11 variables:
#>  $ County   : int  1 1 1 1 1 1 1 1 1 1 ...
#>  $ Year     : int  2000 2000 2000 2000 2000 2000 2000 2000 2000 2000 ...
#>  $ Sex      : Factor w/ 2 levels "F","M": 1 1 1 1 1 1 1 1 1 1 ...
#>  $ Age      : int  0 1 2 3 4 5 6 7 8 9 ...
#>  $ White    : int  2751 2673 2654 2646 2780 2826 2871 2942 3087 3070 ...
#>  $ Hispanic : int  2910 2837 2770 2798 2762 2738 2872 2837 2756 2603 ...
#>  $ Asian    : int  1921 1820 1930 1998 2029 1998 1984 1993 1957 2031 ...
#>  $ Pac.Isl  : int  63 70 64 64 85 93 78 94 85 83 ...
#>  $ Black    : int  1346 1343 1442 1455 1558 1632 1657 1732 1817 1819 ...
#>  $ Amer.Ind : int  32 37 37 39 38 45 53 51 43 44 ...
#>  $ Multirace: int  711 574 579 588 575 576 548 514 545 537 ...
```

```
capop[1:5, 1:8]
```

```
#>   County Year Sex Age White Hispanic Asian Pac.Isl
#> 1      1 2000   F   0  2751     2910  1921      63
#> 2      1 2000   F   1  2673     2837  1820      70
#> 3      1 2000   F   2  2654     2770  1930      64
#> 4      1 2000   F   3  2646     2798  1998      64
#> 5      1 2000   F   4  2780     2762  2029      85
```

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. Excluding the "Sex"

variable, the following code will return the range (i.e., minimum and maximum values) of the `capop` data frame.

```
lapply(capop[-3], range) # exclude column 3 (Sex)
```

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

```
apply(capop[,-3], 2, range) # exclude column 3 (Sex)
```

```
#>      County Year Age  White Hispanic Asian Pac.Isl Black
#> [1,]      1 2000   0      0        0     0       0     0
#> [2,]     59 2000 100 148246   127415 35240    1066 21865
#>      Amer.Ind Multirace
#> [1,]        0         0
#> [2,]     1833     11299
```

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]) # disply only columns 1 to 7
```

```
#>      County        Year      Sex           Age
#>  Min.   : 1   Min.   :2000   F:5959   Min.   :  0
#>  1st Qu.:15   1st Qu.:2000   M:5959   1st Qu.: 25
#>  Median :30   Median :2000            Median : 50
#>  Mean   :30   Mean   :2000            Mean   : 50
#>  3rd Qu.:45   3rd Qu.:2000            3rd Qu.: 75
#>  Max.   :59   Max.   :2000            Max.   :100
#>      White           Hispanic           Asian
#>  Min.   :     0   Min.   :     0.0   Min.   :    0.0
#>  1st Qu.:   100   1st Qu.:    10.0   1st Qu.:    2.0
#>  Median :   385   Median :    76.0   Median :   16.0
#>  Mean   :  2693   Mean   :  1859.9   Mean   :  628.7
#>  3rd Qu.:  1442   3rd Qu.:   589.8   3rd Qu.:  139.0
#>  Max.   :148246   Max.   :127415.0   Max.   :35240.0
```

3.2.6.1 The `aggregate` function

The `aggregate` function is almost identical to the `tapply` function. Recall that `tapply` allows us to apply a function to a *single* 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:

```
to.keep <- c("White", "Hispanic", "Asian", "Pac.Isl",
             "Black", "Amer.Ind", "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  151238   231822   41758
#> 2    [1,5)   F 2000  627554   929222  172296
#> 3   [5,15)   F 2000 1849860  2249146  482094
#> 4  [15,25)   F 2000 1737534  1893896  545692
#> 5  [25,45)   F 2000 4720500  3484732 1335912
#> 6  [45,65)   F 2000 4204180  1470124  890078
#> 7      65+   F 2000 2943684   559730  417132
#> 8    [0,1)   M 2000  159360   243170   43930
#> 9    [1,5)   M 2000  662386   968136  182746
#> 10  [5,15)   M 2000 1958466  2350768  515148
#> 11 [15,25)   M 2000 1850710  2161736  558628
#> 12 [25,45)   M 2000 4930388  3843792 1229216
```

```
#> 13 [45,65)   M 2000 4149666  1375098  768022
#> 14     65+   M 2000 2150452   404598  309932
```

3.3 Managing data objects and workspace

When we work in R we have a workspace. Think of the workspace as our office desk top that have the "objects" (data and tools) we use to conduct our work. To view all the objects in our workspace use `ls()` or `objects()` To view objects that match a pattern use the `pattern` option. To remove all data objects use `rm(list = ls())` with extreme caution. See Table 3.12 for summary.

```
obj1 <- 1; obj2 <- 2; obj3 <- 3; obj4 <- 4; obj5 <- 5
ls(pattern = "obj")
```

```
#> [1] "obj1" "obj2" "obj3" "obj4" "obj5"
```

```
rm(obj2, obj4); ls(pattern = "obj")
```

```
#> [1] "obj1" "obj3" "obj5"
```

TABLE 3.12: Common ways of managing data objects

Function	Description and examples
`ls`, `objects`	List objects
`rm`, `remove`	Remove object(s)
`save.image`	Saves workspace
`save`	Saves objects to external file.
`load`	Loads objects previously saved

Object names in the workspace 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 3.13. In general, we never go wrong using the `str`, `mode`, and `class` functions.

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 `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 <- runif(20); y <- list(a = 1, b = TRUE, c = "oops")
save(x, y, file = "xy.RData") # external file
ls() # display objects in current workking directory
```

```
#> [1] "x" "y"
```

```
rm(list = ls()) # clean current working directory
ls()
```

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

```
load("xy.RData") # load 'x' and 'y' from external file
ls()
```

```
#> [1] "x" "y"
```

```
unlink("xy.RData") # unlinks and removes external file
ls() # 'x' and 'y' remain in current working directory
```

```
#> [1] "x" "y"
```

TABLE 3.13: Assessing and coercing data objects

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

Table 3.13 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.

```
x <- as.matrix(1:3); x
```

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

```
is.matrix(x)
```

```
#> [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")); sex
```

```
#> [1] M M M M F F F
#> Levels: F M
```

```
unclass(sex)      # does not coerce into character vector
```

```
#> [1] 2 2 2 2 1 1 1
#> attr(,"levels")
#> [1] "F" "M"
```

```
as.character(sex) # yes, works
```

```
#> [1] "M" "M" "M" "M" "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
```

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] "/Users/tja/Dropbox/tja/Rproj/home"
```

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

```
setwd("~/Dropbox/tja/Rproj/panflu")
```

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

3.4 Exercises

Some of the exercises uses data that is available from https://github.com/taragonmd/data. For example, suppose we are interested in the `mi.txt` data. Locate the data set and select "Raw" tab to view the raw data. Copy the URL for reading the data into our R environment. Here is the URL for `mi.txt`: https://raw.githubusercontent.com/taragonmd/data/master/mi.txt.

Exercise 3.1 (Data-driven decision making)**.** Consider an observational study where 700 patients were given access to a new drug for an ailment. A total of 350 patients chose to take the drug and 350 patients did not. The patients were assessed for clinical recovery. Table 3.14

TABLE 3.14: Recovery outcomes of 700 patients given access to a new drug, Stratified by gender

	Men			Women		
	Drug	No drug	Total	Drug	No drug	Total
Recovered	81	234	315	192	55	247
No recovery	6	36	42	71	25	96
Total at risk	87	270	357	263	80	343

Here we read in the data set ("drugrx-pearl2.csv") and create a stratified contingency table.

```
jp2 <-
read.csv('https://raw.githubusercontent.com/taragonmd/data/master/drugrx-pearl2.csv')
str(jp2)
```

```
#> 'data.frame':    700 obs. of  4 variables:
#>  $ X         : int  1 2 3 4 5 6 7 8 9 10 ...
#>  $ Recovered: Factor w/ 2 levels "No","Yes": 2 2 2 2 2 2 2 2 2 2 ...
```

```
#>  $ Drug     : Factor w/ 2 levels "No","Yes": 2 2 2 2 2 2 2 2 2 2 ...
#>  $ Gender   : Factor w/ 2 levels "Men","Women": 1 1 1 1 1 1 1 1 1 1 ...
```

```
(tab2.rdg <- xtabs(~ Recovered + Drug + Gender, data = jp2))
```

```
#> , , Gender = Men
#>
#>          Drug
#> Recovered  No Yes
#>       No   36   6
#>       Yes 234  81
#>
#> , , Gender = Women
#>
#>          Drug
#> Recovered  No Yes
#>       No   25  71
#>       Yes  55 192
```

For the variables Recovered (R), Drug (D), and Gender (G), use R to calculate the following probabilities (proportions).

1. $P(R = yes \mid D = no)$
2. $P(R = yes \mid D = yes)$
3. $P(R = yes \mid D = no, G = men)$
4. $P(R = yes \mid D = yes, G = men)$
5. $P(R = yes \mid D = no, G = women)$
6. $P(R = yes \mid D = yes, G = women)$

Discuss your findings and make drug treatment recommendations, and your rationale. Assume the results are clinically meaningful and statistically significant, and there were no measurement errors or misclassification biases.

Exercise 3.2 (Practice cross-tabulations)**.** Use the `read.csv` function to read in syphilis counts from 1989. Do not attach the data frame (yet).

```
std89c <-
read.csv("https://raw.githubusercontent.com/taragonmd/data/master/syphilis89c.csv")
```

(a) Evaluate the structure of data frame.
(b) The `Age` variable levels are not ordered correctly. Why is this? Fix it by creating a new variable with the correctly ordered levels.

(c) Create 3-dimensional arrays using both the `table` and `xtabs` function, with and without attaching the data frame (use `atach` and `detach` functions). Describe how the `table` function output differs with and without attachment of the data frame.

Exercise 3.3 (Practice calculating marginal totals)**.** Use the `apply` function to get marginal totals for the syphilis 3-dimensional array (Sex by Race by Age).

Exercise 3.4 (Practice calculating marginal and joint distributions)**.** Use the `sweep` and `apply` functions to get marginal and joint distributions for a 3-D array.

Exercise 3.5 (Convert individual-level data frame into group-level data frame)**.** When the individual-level data frame has a few categorical variables, it's easy to convert it into a group-level data frame with frequency counts. Try these in R.

```
table(std89c)
data.frame(table(std89c))
```

Exercise 3.6 (Working with group-level data frames)**.** Review and read in the group-level data set (syphilis89b.csv) of primary and secondary syphilis cases in the United States in 1989. This is the same data frame that was created in Exercise 3.5

```
std89b <-
read.csv("https://raw.githubusercontent.com/taragonmd/data/master/syphilis89b.csv")
```

Use the `rep` function and the concept of indexing rows of a data frame to recreate the individual-level data frame with over 40,000 observations. Instead of indexing to subset rows of a data frame, we are indexing to replicate rows of a group-level data frame using the frequency counts.

Exercise 3.7 (Working with population estimates)**.** Working with population estimates can be challenging because of the amount of data manipulation. Study the 2000 population estimates for California Counties (CalCounties2000.csv). 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. To read in the data you will need to get and use the GitHub raw data URL as before.

```
#### 1 Read county names
url1 <-
"https://raw.githubusercontent.com/taragonmd/data/master/calcounty.txt"
cty <- scan(url1, what="")

#### 2 Read county population estimates
url2 <-
"https://raw.githubusercontent.com/taragonmd/data/master/CalCounty2000.csv"
calpop <- read.csv(url2)

#### 2 Replace county number with county name
calpop$CtyName <- calpop$County ## create duplicate
for(i in 1:length(cty)){
    calpop$CtyName[calpop$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$Pac.Isl

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

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

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

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

4

Managing epidemiologic data in R

4.1 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 (4.1) 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

TABLE 4.1: Deaths among subjects who received tolbutamide and placebo in the Unversity Group Diabetes Program (1970), stratifying by age

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

4.1.1 Entering data

We review four methods. For Methods 1 and 2, data are entered directly at the R console prompt . Method 3 uses the same R expressions and data as Methods 1 and 2, but they are entered into a text editor (R script editor), saved as a text file with a `.R` extension (e.g., `job02.R`), and then executed from the R console prompt using the `source` function. Alternatively, the R expressions and data can be copied and pasted into R. And, for Method 4 we use RStudio's spreadsheet editor (least preferred).

4.1.1.1 Method 1: Enter data at the console prompt

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
```

```
#### 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
#> [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
```

4.1.1.2 Method 2: Enter data at the console prompt using `scan` function

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
```

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. Study this example:

```
#### list includes 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
```

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 = TRUE)) )
#> 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
```

4.1.1.3 Method 3: Execute script (.R) files using the source function

Method 3 uses the same R expressions and data as Methods 1 and 2, but they are entered into the R script (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. For example, the following expressions are in a R script 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('~/Rproj/home/job01.R')
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

In general, we highly recommend using RStudio's script editor for all your work. The script file (e.g., `job01.R`) created with the editor facilitates documenting, reviewing, and debugging our code; replicating our analytic steps; and auditing by external reviewers.

4.1.1.4 Method 4: Use a spreadsheet editor (optional read)

Method 4 uses R's spreadsheet editor. 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 saves 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 a new object name (even if it is the original name).

To enter a vector we need to initialize a vector and then use the `data.entry` function (4.1).

```
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] 34 56 34 56 45 34 23 67 87 99
```

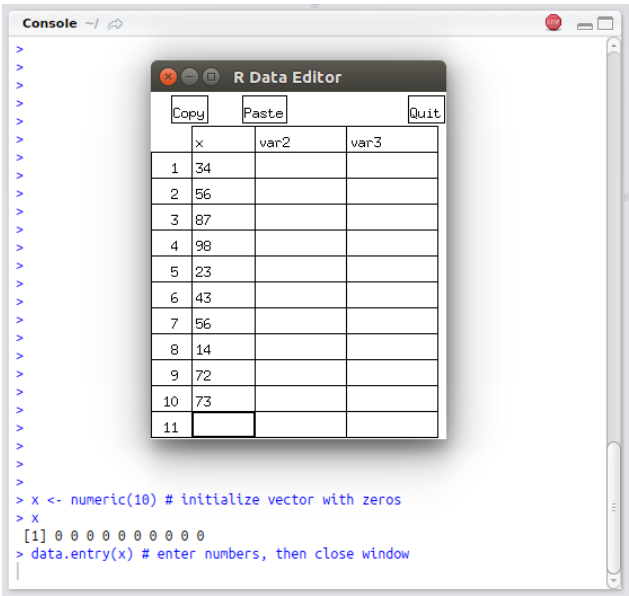


FIGURE 4.1: Example of using the 'data.entry' function in RStudio console

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   17
#> 2  Luisito   16
#> 3 Angelita   13
```

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
```

4.1.2 Importing data from a file

4.1.2.1 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).[1] 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
...
408,Survived,Placebo,55+
409,Survived,Placebo,55+
```

Notice that the header row has 3 items (field names), 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.[2] Here we read in and display the first four lines of this data set:

```
udat1 <- read.table('~/data/ugdp1.txt', header = TRUE, sep = ",")
udat1[1:4,] # display first 4 lines
```

```
#>   Status   Treatment Agegrp
#> 1  Death Tolbutamide    <55
#> 2  Death Tolbutamide    <55
#> 3  Death Tolbutamide    <55
#> 4  Death Tolbutamide    <55
```

Next (below) is the same data (`ugdp2.txt`) but without a header and without row names:

```
Dead,Tolbutamide,<55
Dead,Tolbutamide,<55
...
Survived,Placebo,55+
Survived,Placebo,55+
```

And here is how we read this comma-delimited data set (`ugdp2.txt`):

[1] Available at ~/data/ugdp1.txt
[2] If row names are supplied, they must be unique.

```
cnames <- c('Status', 'Treatment', 'Agegrp')
udat2 <- read.table('~/data/ugdp2.txt', header = FALSE, sep = ",",
                    col.names = cnames)
udat2[1:4,] # display first 4 lines
```

```
#>   Status   Treatment Agegrp
#> 1   Dead Tolbutamide    <55
#> 2   Dead Tolbutamide    <55
#> 3   Dead Tolbutamide    <55
#> 4   Dead Tolbutamide    <55
```

Here is the same data (`ugdp3.txt`) 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.

```
Dead    Tolbutamide<55
...
Dead    Tolbutamide55+
...
Dead    Placebo    <55
...
SurvivedTolbutamide<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('~/data/ugdp3.txt', width = c(8, 11, 3),
                  col.names = cnames, strip.white = TRUE)
udat3[1:4,]
```

```
#>   Status   Treatment Agegrp
#> 1   Dead Tolbutamide    <55
#> 2   Dead Tolbutamide    <55
#> 3   Dead Tolbutamide    <55
#> 4   Dead Tolbutamide    <55
```

Finally, here is the same data file as a *fixed width formatted* file but with integer 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
...
122
122
...
211
211
...
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('~/data/ugdp4.txt', width = c(1, 1, 1),
                  col.names = cnames)
udat4[1:4,]
```

```
#>   Status Treatment Agegrp
#> 1      1         2      1
#> 2      1         2      1
#> 3      1         2      1
#> 4      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.

4.1.2.2 Reading data from a binary format (e.g., Stata)

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.

```
library(foreign)
idat <- read.dta('~/data/infert.dta')
idat[1:4, 1:7]
```

```
#>   id education age parity induced case spontaneous
```

```
#> 1  1        0  26      6       1    1          2
#> 2  2        0  42      1       1    1          0
#> 3  3        0  39      6       2    1          0
#> 4  4        0  34      4       2    1          0
```

4.1.3 Importing data using a URL

As we have already seen, text data files can be read directly off a web server into R using `read.table` or equivalent function.

4.2 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.

4.2.1 Text editor

Figure 4.2 displays West Nile virus (WNV) infection surveillance data in the GNU Emacs text editor.[3] In RStudio, this is equivalent to the script editor (left upper window), and the console (left lower window).

4.2.2 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 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 "Unknown" and change it to NA for missing. We will also change "." to NA.

```
wd <- read.table('~/data/wnv/wnv2004raw.csv', header = TRUE,
                 sep = ",",  as.is = TRUE)
```

[3] Raw data set available at [~/data/wnv/wnv2004raw.csv], and clean data set available at [~/data/wnv/wnv2004fin.csv]. This file is a comma-delimited data file with a header.

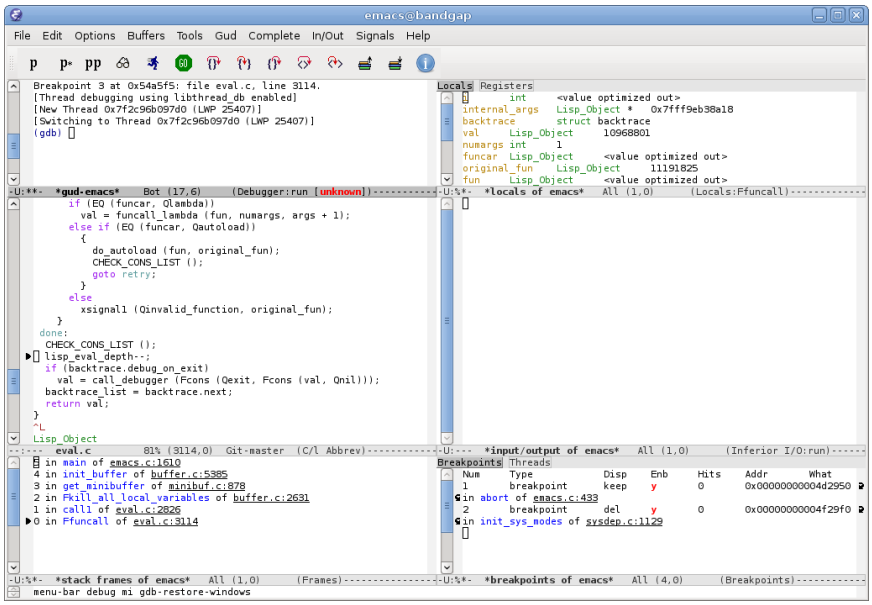


FIGURE 4.2: Left frame: Editing West Nile virus human surveillance data in GNU Emacs text editor. Right frame: Running R using Emacs Speaks Statistics (ESS). Data source: California Department of Health Services, 2004

```
wd[wd$syndrome=='Unknown',][1:3, -7]  # before edits

#>      id      county age sex syndrome date.onset death
#> 128 128 Los Angeles  81   M  Unknown 07/28/2004     .
#> 129 129   Riverside  44   F  Unknown 07/25/2004     .
#> 133 133 Los Angeles  36   M  Unknown 08/04/2004    No

fix(wd) # opens R data editor and edits made -- not shown
wd[c(128, 129, 133), -7]  # after edits (3 records)

#>      id      county age sex syndrome date.onset death
#> 128 128 Los Angeles  81   M       NA 07/28/2004    NA
#> 129 129   Riverside  44   F       NA 07/25/2004    NA
#> 133 133 Los Angeles  36   M       NA 08/04/2004    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, we manually changed the missing values "Unknown" to NA (R's representation of missing values). However, the manual

approach is 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 can 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('~/data/wnv/wnv2004raw.csv', 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
#> 128 128 Los Angeles  81   M     <NA> 07/28/2004  08/11/2004
#> 129 129   Riverside  44   F     <NA> 07/25/2004  08/11/2004
#> 133 133 Los Angeles  36   M     <NA> 08/04/2004  08/11/2004
#>     death
#> 128  <NA>
#> 129  <NA>
#> 133    No
```

4.2.3 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. We use the following code to read in (again) and evaluate the raw data.

```
wd <- read.table('~/data/wnv/wnv2004raw.csv',
                 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" "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" "05/22/2004" "05/16/2004" ...
#>  $ date.tested: chr  "06/02/2004" "06/16/2004" "06/16/2004" "06/16/2004" ...
#>  $ death      : chr  "No" "No" "No" "No" ...
```

```
lapply(wd, table)[c(2:5, 8)] # apply 'table' function to list
```

```
#> $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 27
#>  6  1  1  1  3  2  3  3  1  4  6  5  1  4  2  3  6  8  3  9  4
#> 28 29 30 31 32 33 34 35 36 37 38 39  4 40 41 42 43 44 45 46 47
#>  8  1  8  6  3  6 15 15  8  8  5  7  1  9 17 15 19 11 14 23 25
#> 48 49  5 50 51 52 53 54 55 56 57 58 59  6 60 61 62 63 64 65 66
#> 19 22  2 14 16 24 17 14 17 13 16  8 17  2 29 13 15  7 12 10  7
#> 67 68 69  7 70 71 72 73 74 75 76 77 78 79  8 80 81 82 83 84 85
#>  8  8 14  2 12 10  9 11 12  7 10  7  7  7  1  6 11 10  5  6  4
#> 86 87 88 89  9 91 93 94
#>  2  2  1  6  1  4  1  1
#>
#> $sex
#>
#>   .   F   M
#>   2 294 483
#>
#> $syndrome
#>
#> Unknown     WNF    WNND
#>     105     391     283
#>
#> $death
#>
#>   .  No Yes
#>  66 686  27
```

What did we learn? First, there are 779 observations and 781 id's; therefore, three 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:

```
wd$age[wd$age=='.'] <- NA
wd$sex[wd$sex=='.'] <- NA
wd$syndrome[wd$syndrome == 'Unknown'] <- NA
wd$death[wd$death=='.'] <- NA
table(wd$death)  # verify
```

```
#>
#>  No Yes
#> 686  27
```

```
table(wd$death, exclude = NULL)  # show missing NAs
```

```
#>
#>   No  Yes <NA>
#>  686   27   66
```

Note that the NA replacement could have been done globally like this:

```
wd[wd=='.' | wd=='Unknown'] <- NA
```

We also notice that the entry for one of the counties, San Luis Obispo, was misspelled (`Sn Luis Obispo`). We can use replacement to correct this:

```
wd$County[wd$county=='Sn Luis Obispo'] <- 'San Luis Obispo'
```

4.2.4 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."[4] We'll start by using the `substr` function. This function extracts characters from a character vector based on position.

[4]ISO 8601 is an international standard for date and time representations issued by the International Organization for Standardization (ISO). See http://www.iso.org

```
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"
```

TABLE 4.2: R functions for processing text in character vectors

Function	Description with 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')` `strsplit(some.dates, '/')`

4.3 Sorting data

The `sort` function sorts a vector as expected:

```
x <- sample(1:10, 10); x
```

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

```
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
```

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 <- c(15, 7, 20, 16,  8)
sort(x)     # sorts as expected
```

```
#> [1]  7  8 15 16 20
```

```
y <- c(6, 18, 14,  9, 3)
order(y)    # integer vector specifying position change
```

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

```
x[order(y)] # sort elements of x based on order(y)
```

```
#> [1]  8 15 16 20  7
```

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', 'Black', 'Latino', 'Asian'), 2)
age <- c(57, 93, 7, 65, 38, 27, 66, 72)
dat <- data.frame(age, sex, ethnicity)
dat <- dat[sample(1:8, 8), ] # randomly order rows
dat
```

```
#>   age    sex ethnicity
#> 3   7   Male    Latino
#> 4  65   Male     Asian
#> 7  66 Female    Latino
#> 1  57   Male     White
#> 8  72 Female     Asian
#> 2  93   Male     Black
#> 6  27 Female     Black
#> 5  38 Female     White
```

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
#> 3   7   Male    Latino
#> 6  27 Female     Black
#> 5  38 Female     White
#> 1  57   Male     White
#> 4  65   Male     Asian
#> 7  66 Female    Latino
#> 8  72 Female     Asian
#> 2  93   Male     Black
```

```
dat[order(dat$sex, dat$age),] # sort based on 2 variables
```

```
#>   age    sex ethnicity
#> 6  27 Female     Black
#> 5  38 Female     White
#> 7  66 Female    Latino
#> 8  72 Female     Asian
#> 3   7   Male    Latino
#> 1  57   Male     White
#> 4  65   Male     Asian
#> 2  93   Male     Black
```

4.4 Indexing (subsetting) data

In R, we use indexing to subset (or extract) parts of a data object. For this section, please load the well known Oswego foodborne illness data frame using:

```
odat <- read.table('~/data/oswego/oswego.txt', header = TRUE,
                   as.is = TRUE, sep = "")
names(odat) # display field names
```

```
#>  [1] "id"             "age"            "sex"
#>  [4] "meal.time"      "ill"            "onset.date"
#>  [7] "onset.time"     "baked.ham"      "spinach"
#> [10] "mashed.potato"  "cabbage.salad"  "jello"
#> [13] "rolls"          "brown.bread"    "milk"
#> [16] "coffee"         "water"          "cakes"
#> [19] "van.ice.cream"  "choc.ice.cream" "fruit.salad"
```

4.4.1 Indexing

Now, we 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'``. Foris equivalent to" we use the `==` relational operator.

```
cases <- odat$ill=='Y'  # logical vector to index cases
odat.ca <- odat[cases, ]
odat.ca[1:4, 1:7] # display part of data set
```

```
#>   id age sex meal.time ill onset.date onset.time
#> 1  2  52   F   8:00 PM   Y       4/19   12:30 AM
#> 2  3  65   M   6:30 PM   Y       4/19   12:30 AM
#> 3  4  59   F   6:30 PM   Y       4/19   12:30 AM
#> 4  6  63   F   7:30 PM   Y       4/18   10:30 PM
```

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$van.ice.cream == 'Y' & odat$age < median(odat$age)
odat.fcv <- odat[fem.cases.vic, ]
odat.fcv[ , c(1:6, 19)]
```

```
#>    id age sex meal.time ill onset.date van.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.

4.4.2 Using the `subset` function

Indexing 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' &
             van.ice.cream == 'Y' & age < median(odat$age)},
                   select = c(id:onset.date, van.ice.cream))
odat.fcv
```

```
#>    id age sex meal.time ill onset.date van.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 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.

4.5 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")
- Recoding factor levels (categorical variables)

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 R workspace). For the examples that follow load the well known Oswego foodborne illness dataset:

```
odat <- read.table('~/data/oswego/oswego.txt',
                   header = TRUE, as.is = TRUE, sep = "")
```

4.5.1 Numerical transformation

In this example we "center" the `age` field by substracting the mean age from each age.

```
summary(odat$age)
```

```
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>    3.00   16.50   36.00   36.81   57.50   77.00
```

```
#### create new field in same data frame
odat$age.centered <- odat$age - mean(odat$age)
summary(odat$age.centered)
```

```
#>     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
#> -33.8133 -20.3133  -0.8133   0.0000  20.6867  40.1867
```

```
#### create new vector in workspace; data frame unchanged
age.centered <- odat$age - mean(odat$age)
summary(age.centered)
```

```
#>     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
#> -33.8133 -20.3133  -0.8133   0.0000  20.6867  40.1867
```

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))
```

TODO

4.5.2 Recoding vector values

4.5.2.1 Recoding using replacement

Here is a straightforward example of recoding a vector by replacement:

```
x <- c("m", "f", "u", "f", "f", "m", "m")
y <- x  # create copy
y[x=="m"] <- "Male"
y[x=="f"] <- "Female"
y[x=="u"] <- NA
table(y)
```

```
#> y
#> Female   Male
#>      3      3
```

4.5.2.2 Recoding using a lookup table

Alternatively, we can use character matching to create a lookup table which is just indexing by name (or the data character vector).

```
x <- c("m", "f", "u", "f", "f", "m", "m")
lookup <- c(m = "Male", f = "Female", u = NA)
lookup[x]
```

```
#>        m        f        u        f        f        m        m
#>   "Male" "Female"       NA "Female" "Female"   "Male"   "Male"
```

Note that if you don't want names in the result, use unname() to remove them.

```
y <- unname(lookup[x]); y
```

```
#> [1] "Male"   "Female" NA       "Female" "Female" "Male"
#> [7] "Male"
```

4.5.3 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 six age categories (in years): `<5`, `5 to 14`, `15 to 24`, `25 to 44`, `45 to 64`, and `65+`. We will demonstrate this using several methods.

4.5.3.1 Using `cut` function (preferred method)

Here we use the `cut` function.

```
agecat <- cut(odat$age, breaks = c(0, 5, 15, 25, 45, 65, 100))
table(agecat)
```

```
#> agecat
#>    (0,5]   (5,15]  (15,25]  (25,45]  (45,65] (65,100]
#>        1       17       12       17       22        6
```

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, in the United States, for age categories, the age boundaries are 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, 5, 15, 25, 45, 65, 100), right = FALSE)
table(agecat)
```

```
#> agecat
#>    [0,5)   [5,15)  [15,25)  [25,45)  [45,65) [65,100)
#>        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('<5', '5-14', '15-24', '25-44', '45-64', '65+')
agecat <- cut(odat$age, breaks = c(0, 5, 15, 25, 45,
    65, 100), right = FALSE, labels = agelabs)
table(Illness = odat$ill, 'Age category' = agecat) # display
```

```
#>        Age category
#> Illness <5 5-14 15-24 25-44 45-64 65+
#>       N  0    8     5     8     5   3
#>       Y  1    6     8    10    15   6
```

4.5.3.2 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!

```
agecat <- odat$age
agecat[odat$age < 5] <- 1
agecat[odat$age >= 5 & odat$age < 15] <- 2
agecat[odat$age >= 15 & odat$age < 25] <- 3
agecat[odat$age >= 25 & odat$age < 45] <- 4
agecat[odat$age >= 45 & odat$age < 65] <- 5
agecat[odat$age >= 65] <- 6
#### create factor
agelabs <- c('<5', '5-14', '15-24', '25-44', '45-64', '65+')
agecat <- factor(agecat, levels = 1:6, labels = agelabs)
table(Illness = odat$ill, 'Age category' = agecat) # display
```

```
#>         Age category
#> Illness <5 5-14 15-24 25-44 45-64 65+
#>       N  0    8     5     8     5   3
#>       Y  1    6     8    10    15   6
```

In these previous examples, notice that `agecat` is a factor object in the workspace and not a field in the `odat` data frame.

4.5.4 Recoding factor levels (categorical variables)

In the previous example the categorical variable was a numeric vector (1, 2, 3, 4, 5, 6) that was converted to a factor and provided labels ('<5', '5-14', …). 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 = TRUE)
ethnicity <- factor(ethnicity, levels = ethlabs)
table(ethnicity) # display frequency
```

```
#> ethnicity
```

```
#>  White  Black Latino  Asian
#>     28     26     24     22
```

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.[5]

```
ethcode <- unclass(ethnicity)
table(ethcode) # display frequency
```

```
#> ethcode
#>  1  2  3  4
#> 28 26 24 22
```

```
levels(ethcode) # display levels
```

```
#> [1] "White"  "Black"  "Latino" "Asian"
```

To recover the original factor,

```
eth.orig <- factor(ethcode, labels = levels(ethcode))
table(eth.orig) # display frequency
```

```
#> eth.orig
#>  White  Black Latino  Asian
#>     28     26     24     22
```

Although we can extract the integer code, why would we 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.[6] 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 the order of the factor levels.

```
levels(ethnicity) # display levels
```

```
#> [1] "White"  "Black"  "Latino" "Asian"
```

[5] The `as.integer` function also works but does not preserve the levels attribute.
[6] For example, [~/data/oswego/oswego.txt]

```
eth2 <- ethnicity # create duplicate
levels(eth2) <- c('Caucasion', 'Afr. American', 'Hispanic', 'Asian') # assign new levels
table(eth2) # display frequency with new levels
```

```
#> eth2
#>     Caucasion Afr. American      Hispanic         Asian
#>            28            26            24            22
```

In R, we can re-order and re-label at the same time using the `levels` function and assigning to it a list. The `list` function is necessary to assure the re-ordering.

```
eth3 <- ethnicity # create duplicate
levels(eth3) <- list(Hispanic = 'Latino', Asian = 'Asian',
        Caucasion = 'White', 'Afr. American' = 'Black')
table(eth3) # display frequency with new levels & labels
```

```
#> eth3
#>      Hispanic         Asian     Caucasion Afr. American
#>            24            22            28            26
```

To re-order *without* re-labeling use the `list` function

```
table(ethnicity)
```

```
#> ethnicity
#>  White  Black Latino  Asian
#>     28     26     24     22
```

```
eth4 <- ethnicity
levels(eth4) <- list(Latino = 'Latino', Asian = 'Asian',
    White = 'White', Black = 'Black')
table(eth4)
```

```
#> eth4
#> Latino  Asian  White  Black
#>     24     22     28     26
```

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
#>     28     26     24     22
```

```
eth5a <- factor(ethnicity, sort(levels(ethnicity)))
table(eth5a)
```

```
#> eth5a
#>  Asian  Black Latino  White
#>     22     26     24     28
```

```
eth5b <- factor(as.character(ethnicity))
table(eth5b)
```

```
#> eth5b
#>  Asian  Black Latino  White
#>     22     26     24     28
```

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.

4.5.4.1 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"
```

```
eth6 <- relevel(ethnicity, ref = 'Asian')
levels(eth6)
```

```
#> [1] "Asian"  "White"  "Black"  "Latino"
```

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.

4.5.5 Use factors instead of dummy variables

TABLE 4.3: Categorical variable represented as a factor or a set of dummy variables

Factor	Dummy	variable	codes
Ethnicity	Asian	Black	Latino
White	0	0	0
Asian	1	0	0
Black	0	1	0
Latino	0	0	1

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 4.3). 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.

4.5.6 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 = TRUE); x
```

```
#>  [1] "M" "M" "F" "F" "F" "M" "F" "M" "F" "F"
```

```
y <- ifelse(x=='M', 'Male', 'Female'); y
```

```
#>  [1] "Male"   "Male"   "Female" "Female" "Female" "Male"
#>  [7] "Female" "Male"   "Female" "Female"
```

TABLE 4.4: 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))`

Function	Description	Try these examples
		`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 <- transform(dat, logx = log(x)) dat`
cut	Creates a factor by dividing the range of a numeric vector into intervals	`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	Gives access to the levels attribute of a factor	`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)`
relevel	Set the reference level for a factor	`sex2 <- relevel(sex, ref = 'Women')`

Function	Description	Try these examples
ifelse	Conditionally operate on elements of a vector based on a test	`table(sex2)` `age <- sample(1:100, 1000, replace = TRUE)` `agecat <- ifelse(age<=50, '<=50', '>50')` `table(agecat)`

4.6 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 telephone 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`.[7] Data is more efficiently stored in `tab2`, and adding new types of telephone

[7] This approach is the basis for designing and implementing relational databases. A relational database consists of multiple tables linked by an indexing field.

numbers only requires assigning a new value (e.g., Pager) to the `teletype` field.

```
tab2 # pager added to Tomas Aragon
```

```
#>             name telephone 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
#> 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 cvsiador@berkeley.edu     Work
#> 6 Chris    cvsiador@yahoo.com Personal
```

To merge these two data frames use the `merge` function. The `by.x` and `by.y` options identify the indexing fields.

```
dat <- merge(phone, email, by.x='fname', by.y='name')
dat
```

```
#>    fname phonenum phonetype                 mail mailtype
#> 1  Chris 643-3932      work cvsiador@berkeley.edu     Work
#> 2  Chris 643-3932      work    cvsiador@yahoo.com Personal
#> 3  Chris 643-4926       fax cvsiador@berkeley.edu     Work
#> 4  Chris 643-4926       fax    cvsiador@yahoo.com Personal
#> 5  Tomas 643-4926       fax   aragon@berkeley.edu     Work
#> 6  Tomas 643-4926       fax     aragon@medepi.net Personal
#> 7  Tomas 643-4935      work   aragon@berkeley.edu     Work
#> 8  Tomas 643-4935      work     aragon@medepi.net Personal
#> 9  Tomas 847-9139    mobile   aragon@berkeley.edu     Work
#> 10 Tomas 847-9139    mobile     aragon@medepi.net Personal
#> 11 Wayne 643-4926       fax enanoria@berkeley.edu     Work
#> 12 Wayne 643-4926       fax  enanoria@idready.org     Work
#> 13 Wayne 643-4934      work enanoria@berkeley.edu     Work
#> 14 Wayne 643-4934      work  enanoria@idready.org     Work
```

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 cvsiador@berkeley.edu     Work
#> 2  Chris 643-3932      work    cvsiador@yahoo.com Personal
#> 3  Chris 643-4926       fax cvsiador@berkeley.edu     Work
#> 4  Chris 643-4926       fax    cvsiador@yahoo.com Personal
#> 5  Diana 643-3931      work                  <NA>     <NA>
#> 6    Ray 643-4926       fax                  <NA>     <NA>
#> 7    Ray 643-4933      work                  <NA>     <NA>
#> 8  Tomas 643-4926       fax   aragon@berkeley.edu     Work
#> 9  Tomas 643-4926       fax     aragon@medepi.net Personal
#> 10 Tomas 643-4935      work   aragon@berkeley.edu     Work
#> 11 Tomas 643-4935      work     aragon@medepi.net Personal
#> 12 Tomas 847-9139    mobile   aragon@berkeley.edu     Work
#> 13 Tomas 847-9139    mobile     aragon@medepi.net Personal
#> 14 Wayne 643-4926       fax enanoria@berkeley.edu     Work
#> 15 Wayne 643-4926       fax  enanoria@idready.org     Work
```

```
#> 16 Wayne 643-4934      work enanoria@berkeley.edu     Work
#> 17 Wayne 643-4934      work  enanoria@idready.org     Work
```

%%TODO: MAKE TABLE OF MERGE, STACK, reshape

To ‘`reshape'' tabular data look up and study the`reshape`and`stack‘ functions.

4.7 Executing commands from, and directing output to, a file

4.7.1 The `source` function

The `source` function is used to evaluate R expressions that have beenn collected in a R script file. For example, consider the contents this script file (`outer-multiplication-table.R`):

```
i <- 1:10
x <- outer(i, i, '*')
show(x)
```

Here the `source` function is used at the R console to “source” (evaluate and execute) the `outer-multiplication-table.R` file:

```
source('~/git/phds/example/outer-multiplication-table.R')
```

```
#>       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
#>  [1,]    1    2    3    4    5    6    7    8    9    10
#>  [2,]    2    4    6    8   10   12   14   16   18    20
#>  [3,]    3    6    9   12   15   18   21   24   27    30
#>  [4,]    4    8   12   16   20   24   28   32   36    40
#>  [5,]    5   10   15   20   25   30   35   40   45    50
#>  [6,]    6   12   18   24   30   36   42   48   54    60
#>  [7,]    7   14   21   28   35   42   49   56   63    70
#>  [8,]    8   16   24   32   40   48   56   64   72    80
#>  [9,]    9   18   27   36   45   54   63   72   81    90
#> [10,]   10   20   30   40   50   60   70   80   90   100
```

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 the `source` function is used with the option `echo = TRUE`:

```
source('~/git/phds/example/outer-multiplication-table2.R', echo = TRUE)
```

```
#>
#> > i <- 1:10
#>
#> > x <- outer(i, i, '*')
#>
#> > x
#>       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
#>  [1,]    1    2    3    4    5    6    7    8    9    10
#>  [2,]    2    4    6    8   10   12   14   16   18    20
#>  [3,]    3    6    9   12   15   18   21   24   27    30
#>  [4,]    4    8   12   16   20   24   28   32   36    40
#>  [5,]    5   10   15   20   25   30   35   40   45    50
#>  [6,]    6   12   18   24   30   36   42   48   54    60
#>  [7,]    7   14   21   28   35   42   49   56   63    70
#>  [8,]    8   16   24   32   40   48   56   64   72    80
#>  [9,]    9   18   27   36   45   54   63   72   81    90
#> [10,]   10   20   30   40   50   60   70   80   90   100
```

4.7.2 The `sink` and `capture.output` functions

To send console output to a separate file use the `sink` or `capture.output` functions. To work correctly, the `sink` function is used twice: first, to open a connection to an output file and, second, to close the connection. By default, `sink` creates a new output file. To append to an existing file set the option `append = TRUE`. Consider the following script file (`sink-outer.R`):

```
i <- 1:5
x <- outer(i, i, '*')
sink('~/git/phds/example/sink-outer.log') # open connection
cat('Here are results of outer function', fill = TRUE)
```

```
show(x)
sink() # close connection
```

Now the script file (`sink-outer.R`) is sourced:

```
source('~/git/phds/example/sink-outer.R')
```

Notice that nothing was printed to the console because `sink` sent it to the output file. Here are the contents of `sink-outer.log`.

```
Here are results of 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 (`sink-outer.log`). The `cat` and `show` functions printed results to output file. The second `sink` closed 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. At the R console, the `source` option is set to `echo = TRUE`:

```
source('~/git/phds/example/sink-outer.R', echo = TRUE)
```

```
#>
#> > i <- 1:5
#>
#> > x <- outer(i, i, '*')
#>
#> > sink('~/git/phds/example/sink-outer.log') # open connection
```

Notice that nothing appears after `sink`. That is because the output was redirected to the new output file `sink-outer.log`, shown here:

```
> cat('Here are results of outer function', fill = TRUE)
Here are results of outer function

> show(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() # close connection
```

The `sink` and `capture.output` functions accomplish the same task: sending results to an output file. The `sink` function works in pairs: opening 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 script file (`capture-output.R`) using `capture.output` instead of `sink`:

```
i <- 1:5
x <- outer(i, i, '*')
capture.output({
x # display x
}, file = '~/git/phds/example/capture-output.log')
```

Now the script file is sourced at the R console:

```
source('~/git/phds/example/capture-output.R')
```

And, here are the contents of `capture-output.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 expressions (between the 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 evaluates and executes R expression collected in a script file. Setting the `echo` option to TRUE prints expressions and results to the console as if the commands were directly entered at the command prompt. The `sink` and `capture.output` functions direct results to an output file.

4.8 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
```

```
#> [1]  2  4 NA  5
```

```
y <- c("M", NA, "M", "F"); y
```

```
#> [1] "M" NA  "M" "F"
```

```
z <- c(F, NA, F, T); z
```

```
#> [1] FALSE    NA FALSE  TRUE
```

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:

```
as.numeric(c("4", "six")) # Inappropriate coercion
```

```
#> Warning: NAs introduced by coercion
```

```
#> [1]  4 NA
```

```
c(1:5)[7] # Indexing out of range
```

```
#> [1] NA
```

```
df <- data.frame(v1 = 1:3, v2 = 4:6)
df[4, ] # There is no 4th row
```

```
#>    v1 v2
#> NA NA NA
```

```
df["4th",] # Indexing with non-existing name
```

```
#>    v1 v2
#> NA NA NA
```

```
NA + 8 # Operations with NAs
```

```
#> [1] NA
```

```
var(55) # Variance of a single number
```

```
#> [1] NA
```

In general, these "not available" NAs indicate a missing information issue that must or should be addressed.

4.8.1 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; x  # replacement
```

```
#> [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 # recode vector
```

```
#> [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 (this is because a matrix is just a reshaped vector).

```
m <- m2 <- matrix (c(1, -99, 3, 4, -88, 5), 2, 3)
m[m[,1]==-99, 1] <- NA # Replacement one column at a time
m[m[,3]==-88, 3] <- NA; m
```

```
#>      [,1] [,2] [,3]
#> [1,]    1    3   NA
#> [2,]   NA    4    5
```

```
m2[m2==-99 | m2==-88] <- NA; m2 # Global replacement
```

```
#>      [,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
```

```
z1$fname[z1$fname=="Unknown"] <- NA # Replacement
z1$age[z1$age==-999] <- NA; z1
```

```
#>   fname age
#> 1   Tom  56
#> 2  <NA>  34
#> 3 Jerry  NA
```

```
z2[z2=="Unknown" | z2==-999] <- NA; z2 # Global replacement
```

```
#>   fname age
#> 1   Tom  56
#> 2  <NA>  34
#> 3 Jerry  NA
```

4.8.2 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 ␣ (space) 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 fields to factors.

4.8.3 Working with NA values in data frames and factors

There are several functions 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 gender age
#> 1    Jose      M  34
#> 2     Ana      F  NA
#> 3 Roberto      M  22
#> 4  Isabel   <NA>  18
#> 5     Jen      F  34
```

```
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 --- returns error
```

```
#> 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 row observations for any field that contain NAs. The `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
```

4.8.3.1 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.

```
#> [1] M    F    M    <NA> F
#> Levels: F M
```

```
#> gender.na
#>    F    M <NA>
#>    2    2    1
```

```
df$gender
```

```
#> [1] M    F    M    <NA> F
#> Levels: F M
```

```
xtabs(~gender, data = df)
```

```
#> Error in na.fail.default(list(gender = c(2L, 1L, 2L, NA, 1L))) :
#>   missing values in object
```

```
df$gender.na <- factor(df$gender, exclude = NULL)
xtabs(~gender.na, data = df)
```

```
#> gender.na
#>    F    M <NA>
#>    2    2    1
```

%%% HERE

4.8.3.2 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 gender.na
#> NA    <NA>   <NA>  NA      <NA>
#> 3  Roberto      M  22         M
#> 4   Isabel   <NA>  18      <NA>
```

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 gender.na
#> 3 Roberto      M  22         M
#> 4  Isabel   <NA>  18      <NA>
```

This differs from the `na.omit`, `na.exclude`, and `complete.cases` functions that remove all missing values from the data frame first.

4.8.4 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 <NA>
#>    2    2    1
```

```
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
```

```
#> Error in na.fail.default(list(gender = c(2L, 1L, 2L, NA, 1L))) :
#>   missing values in object
```

```
xtabs(~gender.chr, data=df, exclude=NULL) # still does not work
```

```
#> Error in na.fail.default(list(gender.chr = c("M", "F", "M", NA, "F"))) :
#>   missing values in object
```

```
df$gender.na <- factor(df$gender, exclude = NULL) #works
xtabs(~gender.na, data = df)
```

```
#> gender.na
#>    F    M <NA>
#>    2    2    1
```

4.8.5 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.fail"
```

```
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 theoptionsfunction. Globally (inside theoptionsfunction) 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.[8] With `na.pass`, the data object is returned unchanged.

4.8.6 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
```

[8] If `na.omit` removes cases, the row numbers of the cases form the "na.action" attribute of the result, of class "omit". The `na.exclude` function differs from `na.omit` only in the class of the "na.action" attribute of the result, which is "exclude". See help for more details.

```
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
```

4.9 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 4.3 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.

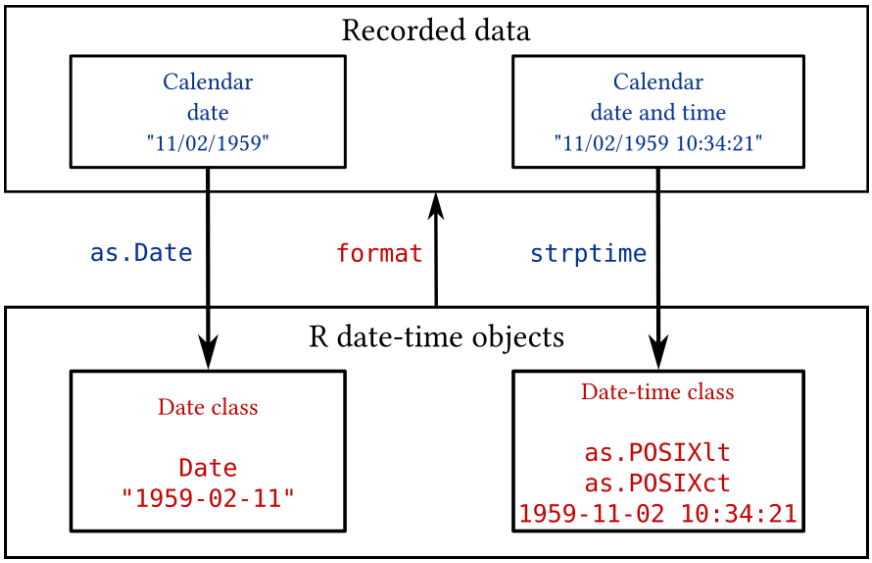


FIGURE 4.3: Displayed are functions to convert calendar date and time data into R date-time classes ('as.Date', 'strptime', 'as.POSIXlt', 'as.POSIXct'), and the 'format' function converts date-time objects into character dates, days, weeks, months, times, etc.

4.9.1 Date functions in the `base` package

4.9.1.1 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 # character vector
```

```
#> [1] "11/2/1959" "1/1/1970"
```

```
#### convert to Julian dates
bdays.julian <- as.Date(bdays, format = "%m/%d/%Y")
bdays.julian # numeric vector of class data
```

```
#> [1] "1959-11-02" "1970-01-01"
```

```
#### display Julian dates
as.numeric(bdays.julian)
```

```
#> [1] -3713     0
```

```
#### calculate age as of today's date
date.today <- Sys.Date()
date.today
```

```
#> [1] "2019-12-22"
```

```
age <- (date.today - bdays.julian)/365.25
age
```

```
#> Time differences in days
#> [1] 60.13689 49.97125
```

```
#### the display of 'days' is not correct
#### truncate number to get "age"
age2 <- trunc(as.numeric(age))
age2
```

```
#> [1] 60 49
```

```
#### 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  60
#> 2  1/1/1970 1970-01-01      0  49
```

Notice that although `bdays.julian` appears like a character vector, it is actually a numeric vector of class `date`. Use the `class` and `mode` functions to verify this.

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)`.

Abbrev.	Description
`%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). Century you get is system-specific. So don't!.
`%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"
```

```
as.Date("11/2/59", format = "%m/%d/%y") # caution using 2-digit year
```

```
#> [1] "2059-11-02"
```

```
as.Date("02Nov1959", format = "%d%b%Y")
```

```
#> [1] "1959-11-02"
```

```
as.Date("02Nov59", format = "%d%b%y") # caution using 2-digit year
```

```
#> [1] "2059-11-02"
```

```
as.Date("1959-11-02") # standard format does not require format option
```

```
#> [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"
```

4.9.1.2 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"
```

4.9.1.3 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("~/data/oswego/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" "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" "12:30 AM" "12:30 AM" "10:30 PM" ...
#>  $ baked.ham     : chr  "Y" "Y" "Y" "Y" ...
#>  $ spinach       : chr  "Y" "Y" "Y" "Y" ...
#>  $ mashed.potato : chr  "Y" "Y" "N" "N" ...
#>  $ cabbage.salad : chr  "N" "Y" "N" "Y" ...
#>  $ jello         : chr  "N" "N" "N" "Y" ...
#>  $ rolls         : chr  "Y" "N" "N" "N" ...
#>  $ brown.bread   : chr  "N" "N" "N" "N" ...
#>  $ milk          : chr  "N" "N" "N" "N" ...
#>  $ coffee        : chr  "Y" "Y" "Y" "N" ...
#>  $ water         : chr  "N" "N" "N" "Y" ...
```

```
#>  $ cakes         : chr  "N" "N" "Y" "N" ...
#>  $ van.ice.cream : chr  "Y" "Y" "Y" "Y" ...
#>  $ choc.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" "4/18/1940 6:30 PM"
#> [4] "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 PST" "1940-04-18 18:30:00 PST"
#> [3] "1940-04-18 18:30:00 PST" "1940-04-18 19:30:00 PST"
```

```
#### 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 PST" "1940-04-19 00:30:00 PST"
#> [3] "1940-04-19 00:30:00 PST" "1940-04-18 22:30:00 PST"
```

```
#### calculate incubation period
incub.period <- onset.dt - meal.dt
incub.period
```

```
#> Time differences in hours
#>  [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  NA
#> [16]  NA  NA 3.0  NA  NA 3.0 3.0  NA  NA 3.0  NA  NA  NA  NA  NA
#> [31] 6.0  NA 4.0  NA  NA  NA 3.0 7.0 4.0 3.0  NA  NA 5.5 4.5  NA
#> [46]  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA
#> [61]  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA
```

```
mean(incub.period, na.rm = T)
```

```
#> Time difference of 4.295455 hours
```

```
median(incub.period, na.rm = T)
```

```
#> Time difference of 4 hours
```

```
#### 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)
```

```
#>  'difftime' num [1:75] 4.5 6 6 3 ...
#>  - 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)[20] # show 20 values
```

```
#> [1] NA
```

Now, what kind of objects were created by the `strptime` function?

```
str(meal.dt)
```

```
#>  POSIXlt[1:75], format: "1940-04-18 20:00:00" "1940-04-18 18:30:00" ...
```

```
str(onset.dt)
```

```
#>  POSIXlt[1:75], format: "1940-04-19 00:30:00" "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."[9]

4.9.1.4 The `POSIXlt` and `POSIXct` functions

The POSIXlt list contains the date-time data in human readable forms. The named list contains the following vectors:

TABLE 4.6: POSIXlt list contains the date-time data in human readable forms

Names	Values: description
'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.
'isdst'	Daylight savings time flag. Positive if in force, zero if not, negative if unknown.

[9] For more information visit the Portable Application Standards Committee site at [http://www.pasc.org/]

Let's examine the `onset.dt` object we created from the Oswego data.

```
is.list(onset.dt)
```

```
#> [1] TRUE
```

```
names(onset.dt)
```

```
#> NULL
```

```
onset.dt$min[10]
```

```
#> [1] 30
```

```
onset.dt$hour[10]
```

```
#> [1] 10
```

```
onset.dt$mday[10]
```

```
#> [1] 19
```

```
onset.dt$mon[10]
```

```
#> [1] 3
```

```
onset.dt$year[10]
```

```
#> [1] 40
```

```
onset.dt$wday[10]
```

```
#> [1] 5
```

```
onset.dt$yday[10]
```

```
#> [1] 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)[10]
onset.dt.ct[1:5]
```

```
#> [1] "1940-04-19 10:30:00 PST" NA
#> [3] NA                        NA
#> [5] NA
```

```
as.numeric(onset.dt.ct[1:5])
```

```
#> [1] -937287000         NA         NA         NA         NA
```

4.9.1.5 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 TODO-REF, 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"
#>  [5] "2003-12-19" "2003-12-20" "2003-12-21" "2003-12-22"
#>  [9] "2003-12-23" "2003-12-24" "2003-12-25" "2003-12-26"
#> [13] "2003-12-27" "2003-12-28" "2003-12-29" "2003-12-30"
#> [17] "2003-12-31" "2004-01-01" "2004-01-02" "2004-01-03"
#> [21] "2004-01-04" "2004-01-05" "2004-01-06" "2004-01-07"
#> [25] "2004-01-08" "2004-01-09" "2004-01-10" "2004-01-11"
#> [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" "51"
#> [13] "51" "52" "52" "52" "52" "00" "00" "00" "01" "01" "01" "01"
#> [25] "01" "01" "01" "02" "02" "02" "02" "02"
```

4.9.2 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.[10]

4.10 Exporting data objects

TABLE 4.7: R functions for exporting data objects

Function and description	Try these examples
Export to Generic ASCII text file	
write.table	
Write tabular data as a data frame to an ASCII text file. Read file back in using read.table function	`write.table(infert,"infert.txt")`
write.csv	
Write tabular data as a data frame to an ASCII text .csv file. Read file back in using read.csv function	`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;	`dump("Titanic", "titanic.R")`

[10] [http://cran.r-project.org/doc/Rnews]

Function and description	Try these examples
Read back in using `source` function	
dput	
Writes an R object as R code 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 non-R text files (SPSS, Stata, SAS). Use `foreign` package to read in	`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")`

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 - Non-R ASCII text files - Non-R binary file

4.10.1 Exporting to a generic ASCII text file

4.10.1.1 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"), fileEncoding = "")
#> NULL
```

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,"~/git/phds/example/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,"~/git/phds/example/infert.csv", 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.

4.10.1.2 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.

4.10.2 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.

4.10.2.1 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"),
     file = "~/git/phds/example/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.

4.10.2.2 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(66L, 33L, 54L, 27L, 24L, 12L, 12L, 6L, 4L, 2L, 5L,
#> 3L), .Dim = c(2L, 6L), .Dimnames = list(case = c("0", "1"), parity = c("1",
#> "2", "3", "4", "5", "6")), class = c("xtabs", "table"), call = 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, "~/git/phds/example/infert_tab1.R")
dget("~/git/phds/example/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
```

4.10.3 Exporting to R binary file

4.10.3.1 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 argu-

ment(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 <- x^3
save(x, y, file="~/git/phds/example/xy.RData")
rm(x, y)
load(file="~/git/phds/example/xy.RData")
x; y
```

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

```
#> [1]   1   8  27  64 125
```

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 <- x^3
save(list=c("x", "y"), file="~/git/phds/example/xy.RData")
rm(x, y)
load(file="~/git/phds/example/xy.RData")
x; y
```

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

```
#> [1]   1   8  27  64 125
```

In fact, the `save.image` function we use to save the entire workspace is just the following:

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

4.10.4 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.

4.11 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 [16]. 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 area 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.

4.11.1 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",
          "z rox z", "y xo y", "xy aba", "cba")
grep("x", vec1)
```

```
#> [1] 1 2 4 5 6 7 8 9
```

```
vec1[grep("x", vec1)] # index by position
```

```
#> [1] "x"       "xa bc"   "ax bc"   "ab xc"   "ab cx"   "z rox z"
#> [7] "y xo y"  "xy aba"
```

```
grep("x", vec1, value = TRUE) # equivalent
```

```
#> [1] "x"       "xa bc"   "ax bc"   "ab xc"   "ab cx"   "z rox z"
#> [7] "y xo y"  "xy aba"
```

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.

We distinguish between matching a pattern at the beginning or end of a line versus matching a pattern at the beginning or end of a word. The caret `^` matches the empty string at the beginning of a line. Therefore, to match this pattern at the beginning of a line we add the `^` character to the regular expression:

```
grep("^x", vec1, value = TRUE)
```

```
#> [1] "x"      "xa bc"  "xy aba"
```

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:

```
grep("x$", vec1, value = TRUE)
```

```
#> [1] "x"     "ab cx"
```

The `^` and `$` characters are examples of metacharacters which have special meanings in regular expression functions. We learn more about metacharacter later.

The symbol `\b` matches the *empty string at the edge of a word*, and `\B` matches the *empty string provided it is not at the edge of a word*. The backslash `\` is a special character; therefore, to use `\b` in a matching pattern we need to add an escape backslash (`\`) to `\b`; in other words use `\\b`. Study these examples:

```
grep("\\bx", vec1, value = TRUE) # find 'x' at beginning of a word
```

```
#> [1] "x"      "xa bc"  "ab xc"  "y xo y" "xy aba"
```

```
grep("x\\b", vec1, value = TRUE) # find 'x' at end of a word
```

```
#> [1] "x"       "ax bc"   "ab cx"   "z rox z"
```

```
grep("\\bx\\b", vec1, value = TRUE) # find 'x' at beginning and end of a word
```

```
#> [1] "x"
```

```
grep("\\Bx", vec1, value = TRUE) # find 'x' not at left edge of word
```

```
#> [1] "ax bc"   "ab cx"   "z rox z"
```

```
grep("x\\B", vec1, value = TRUE) # find 'x' not at right edge of word
```

```
#> [1] "xa bc"  "ab xc"  "y xo y" "xy aba"
```

```
grep("\\Bx\\B", vec1, value = TRUE) # find 'x' not at right and left edge of word
```

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

The symbol \< matches the pattern at the *beginning* of the word, and \> matches the pattern at the *end* of the word. For practical purposes, \b and either \< or \> will yield the same results.

The period (.) matches any single character, including a space.

```
grep(".bc", vec1, value = TRUE)
```

```
#> [1] "xa bc" "abc"   "ax bc"
```

In this example the . was followed by two characters bc. Appending characters is called concatenation and is covered later in detail.

TABLE 4.8: Anchors are regular expressions for matching any single character, and characters at edges of words or lines, or not

Regular expression	Description	Example
.	Any single character	"a.c"
^	At beginning of line	"^F"
$	At end of line	"F$"
\b	At edge of a word	"\\bF", "F\\b"
\B	Not at edge of a word	"\\BF", "F\\B"
\<	Beginning of a word	"\\<F"
\>	End of a word	"F\\>"

4.11.2 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, value = TRUE)
```

```
#> [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")
grep("^[^fhr]", vec2, value = TRUE)
```

```
#> [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. Table 4.9 displays additional selected character class regular expessions.

TABLE 4.9: Selected character class regular expessions

Regular expression	Description	Alternative
`\d`	Digits	`[0-9]`
`\D`	Non-digits	`[^0-9]`
`\w`	Word character	`[A-Za-z0-9_]`
`\W`	Not work character	`[^A-Za-z0-9_]`
`\s`	Space	
`\S`	Non-space	

```
wordvec <- c('hello', '?', '1234', ' ', '_', '\n', '&')
grep('\\w', wordvec, value = TRUE)
```

```
#> [1] "hello" "1234"  "_"
```

```
grep('\\W', wordvec, value = TRUE)
```

```
#> [1] "?"  " "  "\n" "&"
```

For convenience, certain character classes are predefined and their interpretation depend on locale.[11] 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 4.10 lists predefined character classes. This is very convenient for matching punctuation characters and multiple types of spaces (e.g., tab, newline, carriage return). The `[:punct:]` character class matches any of the following punctuation characters:

```
! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [ \ ] ^ _ ` { | } ~
```

TABLE 4.10: Predefined character classes for regular expressions

Predefined	Description	Alternative
[[:lower:]]	Lower-case letters	`[a-z]`
[[:upper:]]	Upper-case letters	`[A-Z]`
[[:alpha:]]	Alphabetic characters	`[A-Za-z]` or `[[:lower:][:upper:]]`
[[:digit:]]	Digits	`[0-9]`
[[:alnum:]]	Alphanumeric characters	`[A-Za-z0-9]` or `[[:alpha:][:digit:]]`
[[:punct:]]	Punctuation characters:	
[[:blank:]]	Blank characters[12]	
[[:cntrl:]]	Control characters[13]	
[[:space:]]	Space characters[14]	
[[:graph:]]	Graphical characters	`[[:alnum:][:punct:]]`
[[:print:]]	Printable characters	`[[:alnum:][:punct:][:space:]]`
[[:xdigit:]]	Hexadecimal digits:	`[0-9A-Fa-f]`

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:

```
grep("^.[^a].+", vec2, value = TRUE)
```

```
#> [1] "elf"
```

[11] 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).

[12] space, tab, and non-breaking space.

[13] In ASCII, octal codes 000 through 037, and 177.

[14] tab, newline, vertical tab, form feed, carriage return, and space

Combining single character matches is call concatenation and is covered next.

4.11.3 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")
grep("^[fhr]at$", vec3, value = TRUE)
```

```
#> [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")
grep("[ct]a[br]", vec4, value = TRUE)
```

```
#> [1] "cab"   "carat" "tar"   "tab"   "care"
```

To match single, 3-letter words use `"\\b[ct]a[br]\\b"`.

```
grep("\\b[ct]a[br]\\b", vec4, value = TRUE)
```

```
#> [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")
grep("f.t", vec5, , value = TRUE)
```

```
#> [1] "fate"   "fit"    "futbol"
```

4.11.4 Repetition

TABLE 4.11: Regular expressions may be followed by a repetition quantifier

Repetition quantifier	Description
`?`	Preceding pattern is optional and will be matched at most once

Repetition quantifier	Description
*	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

Regular expressions (so far: single characters, character classes, and concatenations) can be qualified by whether a pattern can repeat (Table 4.11). For example, the pattern `"^[fF].+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")
grep("^[fF].+t$", vec6, value = TRUE)
```

```
#> [1] "fat"        "feat"       "Fahrenheit" "foot"
```

Repetition quantifiers gives us great flexibility to specify how often preceding patterns can repeat.

4.11.5 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, value = TRUE) # match 1st subexpression
```

```
#>  [1] "B16"  "B160" "B161" "B162" "B163" "B164" "B165" "B167"
#>  [9] "B168" "B169"
```

```
grep("^B17[0,2,8]?$", hepb, value = TRUE) # match 2nd subexpression
```

```
#> [1] "B17"  "B170" "B172" "B178"
```

```
grep("^B18[0,1,8,9]?$", hepb, value = TRUE) # match 3rd subexpression
```

```
#> [1] "B18"  "B180" "B181" "B188" "B189"
```

```
#### match any subexpression
grep("^B16[0-9]?$|^B17[0,2,8]?$|^B18[0,1,8,9]?$", hepb, value = TRUE)
```

```
#>  [1] "B16"  "B160" "B161" "B162" "B163" "B164" "B165" "B167"
#>  [9] "B168" "B169" "B17"  "B170" "B172" "B178" "B18"  "B180"
#> [17] "B181" "B188" "B189"
```

A natural use for these pattern matches is for indexing and replacement. We illustrate this using the 2nd subexpression.

```
#### indexing by position
hepb[grep("^B17[0,2,8]?$", hepb)]
```

```
#> [1] "B17"  "B170" "B172" "B178"
```

```
#### replacement by position
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)
```

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"
```

4.11.6 Repetition > Concatenation > Alternation

Repetition takes precedence over concatenation, which in turn takes precedence over alternation. A whole subexpression may be enclosed in parenthe-

ses 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")
grep("[Tt]om{2,}", vec7, value=TRUE) # repetition takes precedence
```

```
#> [1] "Tommy"
```

```
grep("([Tt]om){2,}", vec7, value=TRUE) # concatenation takes precedence
```

```
#> [1] "Tomtom"
```

Recall that `{2,}` means repeat the previous 2 or more times.

4.11.7 Metacharacters

TABLE 4.12: Metacharacters used by regular expressions

Char.	Description	Example	Literal search
`^`	Matches empty string at beginning of line	`"^my car"`	See p. TODO
	When 1st character in character class, matches any single character not in the list	`"[^abc]"`	See p. TODO
`$`	Matches empty string at end of line	`"my car$"`	`"[$]"`
`[`	Character class		`"[[]"`
`.`	Matches any single character	`"p.t"`	`"[.]"`
`?`	Repetition quantifier (Table 4.11)	`".?"`	`"[?]"`
`*`	Repetition quantifier (Table 4.11)	`".*"`	`"[*]"`
`+`	Repetition quantifier (Table 4.11)	`".+"`	`"[+]"`
`( )`	Grouping subexpressions	`"([Tt]om){2,}"`	`"[(]"` or `"[)]"`
`|`	Join subexpresions, any of which can be matched	`"Tomas|Luis"`	`"[|]"`

RStudio provides a very useful **Cheat Sheet** for basic regular expressions here: https://rstudio.com/wp-content/uploads/2016/09/RegExCheatsheet.pdf.

4.12 Exercises

See Appendix C for description of data sets and data dictionary.

Exercise 4.1 (Practice on data frames)**.** Read in the Evans data set. Type in the R expressions below. Do not copy and paste. After each expression briefly describe what the step accomplished.

```
#### (1)
edat <- read.table("~/data/evans.txt", header = TRUE, sep = "")
str(edat)
xtabs(~chd, data = edat)
edat$chd2 <- factor(edat$chd, levels = 0:1,
                    labels = c("No", "Yes"))
xtabs(~chd2, data = edat) # test
xtabs(~cat, data = edat)
edat$cat2 <- factor(edat$cat, levels = 0:1,
                    labels = c("Normal", "High"))
xtabs(~cat2, data = edat) # test
xtabs(~smk, data = edat)
edat$smk2 <- factor(edat$smk, levels = 0:1,
                    labels = c("Never", "Ever"))
xtabs(~smk2, data = edat) # test
xtabs(~ecg, data = edat)
edat$ecg2 <- factor(edat$ecg, levels = 0:1,
                    labels = c("Normal", "Abnormal"))
xtabs(~ecg2, data = edat) # test
xtabs(~hpt, data = edat)
edat$hpt2 <- factor(edat$hpt, levels = 0:1,
                    labels = c("No", "Yes"))
xtabs(~hpt2, data = edat) # test
#### (2)
quantile(edat$age)
edat$age4 <- cut(edat$age, quantile(edat$age), right = FALSE,
                 include.lowest = TRUE)
xtabs(~age4, data = edat) # test
#### (3)
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"))
xtabs(~hpt4, data = edat) # test
#### (4)
xtabs(~hpt2 + hpt4, data = edat)
```

Exercise 4.2 (Working with disease surveillance data)**.** Review the California 2004 surveillance data on human West Nile virus cases available at `~/data/wnv/wnv2004raw.csv`. Read in the data setting `as.is = TRUE`, taking into account missing values (use `na.strings` option). Convert the calendar dates into the international standard format. Using the `write.table` function export the data as an ASCII text file.

Hint provided:

```
wdat <- read.table("~/data/wnv/wnv2004raw.csv", 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, "~/git/phds/example/wnvdat.txt",
sep = ",", row.names = FALSE)
wdat.test <- read.csv("~/git/phds/example/wnvdat.txt")
head(wdat.test)
str(wdat.test)
```

Exercise 4.3 (Outbreak investigation)**.** 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 C 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. See the raw data set at ~/data/oswego/oswego.txt.

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.

Hints provided:

Hint 1: 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 ~/data/oswego/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("~/data/oswego/oswego.txt", sep = "",
                   header = TRUE, na.strings = ".")
str(odat)
head(odat)
#### create vector w/ meal date and time
mdt <- paste("4/18/1940", odat$meal.time)
#### convert to standard date and time
meal.dt <- strptime(mdt, "%m/%d/%Y %I:%M %p")
#### create vector with onset date and time
odt <- paste(paste(odat$onset.date,"/1940",sep = ""), odat$onset.time)
#### convert into std date & time
onset.dt <- strptime(odt, "%m/%d/%Y %I:%M %p")
hist(onset.dt, breaks = 30, freq = TRUE)
```

Hint 2: 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 = TRUE))
min.obs.pos
max.obs.pos <- which(onset.dt == max(onset.dt, na.rm = TRUE))
max.obs.pos
#### index data frame to display outliers
odat[min.obs.pos, ]
odat[max.obs.pos, ]
```

Hint 3: 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
```

Hint 4: Calculate incubation periods and plot histogram using `truehist`.

```
#### Calculate incubation periods
incub.dt <- onset.dt - meal.dt
library(MASS)  #load MASS package
truehist(as.numeric(incub.dt), nbins = 7, prob = FALSE,
    ylab = 'Frequency', 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)
```

Exercise 4.4 (Adventures of Huckleberry Finn)**.** Mark Twain's book, "Adventures of Huckleberry Finn," is available online for reading into R.

View the entire text file here: https://www.inferentialthinking.com/data/huck_finn.txt. Get an idea of the structure of this text file. For example, there is a table of contents from `CHAPTER I` to `CHAPTER XLII`, followed by `CHAPTER THE LAST`.

We will scan in the book using the `scan` function, and the options `what` and `sep`. Read this in and explore the data object. What kind of data object do we have? Be specific.

```
hf.url <- "https://www.inferentialthinking.com/data/huck_finn.txt"
hf <- scan(hf.url, what="character", sep="\n")
str(hf)
```

Exercise 4.5 (Adventures of Huckleberry Finn (Part 2))**.** Run the following R expressions and then explain in plain words what happened and why.

```
grep("^CHAPTER [IVXL]+[.]\\b",hf)
grep("^CHAPTER [IVXL]+[.]\\b",hf, value = TRUE)
```

Exercise 4.6 (Adventures of Huckleberry Finn (Part 3))**.** The `grepl` function is similar to the `grep` function but generates a logical vector instead of a integer vector. Run the following R expressions.

```
table(grepl("Huckleberry", hf))
#### Use `grep`, not `grepl` to see
grep("Huckleberry", hf, value = TRUE)
table(grepl("Huck", hf))
```

How many times does the word “Huckleberry” appear in this text file? How many times does the word “Huck” appear in this text file?

5

Programming with R—An introduction

"Good programmers write good code, great programmers steal great 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.

5.1 Basic programming

Basic data science programming in R is just a list of R expressions, that are collected and executed as a batch job. The collection of R expressions are saved in an ASCII text file with a `.R` extension. In RStudio, from the main menu, select `File`, `New`, and then `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_task.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, GNU Emacs with ESS[1]).

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,

[1] See *Emacs Speaks Statistics* at https://ess.r-project.org/.

they may be involved in earlier phases of data preparation. Regardless of stage of involvement, data analysts should organize, conduct, and document their analytic 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 the order it was conducted, including corrections. We avoid editing earlier jobs. If we edit previous jobs, then we must *rerun all subsequent jobs in chronological order* to double check our work.

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.

Fourth, while preliminary and core analyses should be conducted with script (`.R`) files, the final report and analyses should be compiled with Rmarkdown (`.Rmd`) files. For now, we focus on script files.

More recently, data scientists are using R Markdown or notebook style of conducting and documenting analyses. This is fine but be sure to follow the principles above.

5.2 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,

5.2.1 Control statements

Control flow involves one or more decision points. A simple 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.

5.2.1.1 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.

5.2.1.2 The `else` function

Up to now the `if` condition had only one possible response. For two, mutually exclusive responses we add the `else` function

```
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
```

```
x
```

```
#> [1]   1   2 999   4   5
```

```
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
```

```
y
```

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

Therefore, use the `if` and `else` combination if one needs to evaluate 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.

5.2.1.3 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. In constrast, for element-wise comparisons of two or more vectors,

use the & and | operators but not the && and || operators.[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.matrix(x) && is.numeric(x) && is.vector(x)) {
  x^2
} else cat("Either not numeric, not a vector, or is a matrix\n")
```

```
#> [1]  1  4  9 16 25
```

```
if(!is.matrix(y) && is.numeric(y) && is.vector(y)) {
  y^2
} else cat("Either not numeric, not a vector, or is a matrix\n")
```

```
#> Either not numeric, not a vector, 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 complicated 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) or is.vector(y).

5.2.2 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)
(colsum <- tab[,1] + tab[,2] + tab[,3] + tab[,4])
```

```
#> [1] 22 26 30
```

However, this can be accomplished more efficiently using the apply function:

```
(colsum2 <- apply(tab, 1, sum))
```

[2] TODO: needs cross-reference

```
#> [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 perform vectorized calculations.

5.2.2.1 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")) # binary recode
```

```
#> [1] "Male"   NA       "Female" "Female" NA       "Male"
#> [7] "Female" "Male"
```

If an element of `sex` contains "M" (TRUE), it is recoded to "Male" (in `sex2`), otherwise (FALSE) it is recoded to "Female". This assumes that there are only "M"s and "F"s in the data vector.

5.2.3 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 <- c(2,4,6,8,10)
index <- 1:length(x) # create vector of sequential integers
xsum <- 0
for (i in index){
  xsum <- xsum + x[i]
}
xsum
```

```
#> [1] 30
```

A much better approach is to use the `sum` function:

```
sum(x)
```

```
#> [1] 30
```

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).

5.2.3.1 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` object contain the American English alphabet. Here we use an integer vector for indexing `letters`:

```
letters # display letters
```

```
#>  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o"
#> [16] "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

```
for(i in 1:3){
  cat(letters[i],"\n")
}
```

```
#> a
#> b
#> c
```

Somevector can be any vector:

```
kids <- c("Tomasito", "Luisito", "Angelita")
for (i in kids) {print(i)}
```

```
#> [1] "Tomasito"
#> [1] "Luisito"
#> [1] "Angelita"
```

5.2.3.2 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.

%TODO

5.2.3.3 repeat

5.2.3.4 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.

5.2.3.5 The double `for` loop

In the next example we nest two `for` loop to generate a multiplication table for two integer vectors.

```
x <- 11:15
y <- 16:25
mtab <- matrix(NA, nrow = length(x), ncol = length(y))
rownames(mtab) <- x
colnames(mtab) <- y
for (i in 1:length(x)) {
    for (j in 1:length(y)) {
        mtab[i, j] <- x[i] * y[j]
    }
}
mtab
```

```
#>     16  17  18  19  20  21  22  23  24  25
#> 11 176 187 198 209 220 231 242 253 264 275
#> 12 192 204 216 228 240 252 264 276 288 300
#> 13 208 221 234 247 260 273 286 299 312 325
#> 14 224 238 252 266 280 294 308 322 336 350
#> 15 240 255 270 285 300 315 330 345 360 375
```

Notice that the `outer` function is a more efficient way of conducting a calculation using all combinations for two numeric vectors.

```
x <- 11:15; y <- 16:25
mtab2 <- outer(x, y, '*')
rownames(mtab2) <- x; colnames(mtab2) <- y
mtab2
```

```
#>     16  17  18  19  20  21  22  23  24  25
#> 11 176 187 198 209 220 231 242 253 264 275
#> 12 192 204 216 228 240 252 264 276 288 300
#> 13 208 221 234 247 260 273 286 299 312 325
#> 14 224 238 252 266 280 294 308 322 336 350
#> 15 240 255 270 285 300 315 330 345 360 375
```

5.3 Writing R functions

Writing R functions is a great way to create "commands" to customize analyses or repeated operations. Here is the general analysis approach:

1. Prepare *inputs*
2. Do *calculations*
3. Collect *results*

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 outbreak data set.

```
#### Prepare inputs
odatcsv <- read.csv('~/data/oswego/oswego2.csv')
head(odatcsv)
```

```
#>   ID Age Sex   MealDate MealTime Ill OnsetDate TimeOnset
#> 1  2  52   F 04/18/1940 20:00:00   Y 4/19/1940  00:30:00
#> 2  3  65   M 04/18/1940 18:30:00   Y 4/19/1940  00:30:00
#> 3  4  59   F 04/18/1940 18:30:00   Y 4/19/1940  00:30:00
#> 4  6  63   F 04/18/1940 19:30:00   Y 4/18/1940  22:30:00
#> 5  7  70   M 04/18/1940 19:30:00   Y 4/18/1940  22:30:00
#> 6  8  40   F 04/18/1940 19:30:00   Y 4/19/1940  02:00:00
#>   Baked.Ham Spinach Mashed.Potatoes Cabbage.Salad Jello Rolls
#> 1         Y       Y               Y             N     N     Y
#> 2         Y       Y               Y             Y     N     N
#> 3         Y       Y               N             N     N     N
#> 4         Y       Y               N             Y     Y     N
#> 5         Y       Y               Y             N     Y     Y
#> 6         N       N               N             N     N     N
#>   Brown.Bread Milk Coffee Water Cakes Van.Ice.Cream
#> 1           N    N      Y     N     N             Y
#> 2           N    N      Y     N     N             Y
#> 3           N    N      Y     N     Y             Y
#> 4           N    N      N     Y     N             Y
#> 5           Y    N      Y     Y     N             Y
#> 6           N    N      N     N     N             Y
#>   Choc.Ice.Cream Fruit.Salad
#> 1              N           N
#> 2              Y           N
#> 3              Y           N
#> 4              N           N
```

```
#> 5              N           N
#> 6              Y           N
```

```
tab1 = xtabs(~ Ill + Spinach, data = odatcsv)
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 = odatcsv)
myOR(tab.test)
```

```
#> $data
#>    Spinach
#> Ill  N  Y
#>   N 12 17
#>   Y 20 26
#>
#> $odds.ratio
#> [1] 0.9176471
```

5.3.1 Arguments with 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
  if(missing(conf.level)) stop("Must specify confidence level")
  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)
}
```

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 = odatcsv)
myOR2(tab.test)

## Error in myOR2(tab.test) (from #4) : 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.level = conf.level, conf.int = CI)
}
```

We test our new function:

```
tab.test <- xtabs(~ Ill + Spinach, data = odatcsv)
myOR3(tab.test)
```

```
#> $data
#>    Spinach
#> Ill  N  Y
#>   N 12 17
#>   Y 20 26
#>
#> $odds.ratio
#> [1] 0.9176471
#>
#> $conf.level
#> [1] 0.95
#>
#> $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.level
#> [1] 0.9
#>
#> $conf.int
#> [1] 0.4165094 2.0217459
```

5.3.2 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, the optional arguments must be valid options for `plot` function.

5.4 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 value of free variable is resolved by first looking in the environment in which the function was defined. This is called *lexical scope*. If the free variable is not defined there, R looks in the parent 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.

To summarize, from Roger Peng (CITE):

1. "If the value of a symbol is not found in the environment in which a function was defined, then the search is continued in the parent environment.
2. The search continues down the sequence of parent environments until we hit the top-level environment; this usually the global environment (workspace) or the namspace of a package.
3. After the top-level environment, the search continues down the search list until we hit the empty environment."

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 parent 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.[3] 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 lexical scope (e.g., in 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.

```
cube(2)
```

```
#> [1] 8
```

Here is a variation that demonstrates lexical scoping but with an unexpected numerical answer.

[3] http://cran.r-project.org/doc/manuals/r-release/R-intro.pdf

```
n <- 3
sq <- function() {n * n}
cube <- function(n) {
  n * sq()
}
cube(2)
```

```
#> [1] 18
```

Why does `cube(2) = 18`? The `sq` function has a free variable `n`. When `sq` is evaluated inside of `cube`, `sq` must find a value for its free variable `n`. Under lexical scoping, `sq` looks in the parent environment when `sq` was defined. The `sq` function was defined in the global environment, so it looks, finds, and uses `n = 3`.

5.4.1 Functions with a mutatable state

Lexical scoping can be used to create a function with a *mutable state*. Consider the following function that simulates a banking savings account balance with deposits and withdrawals.[4]

```
open.account <- function(total) {
    list(deposit = function(amount) {
        if (amount <= 0) stop("Deposits must be positive!\n")
        total <<- total + amount
        cat(amount, "deposited.  Your balance is", total, "\n\n")
    }, withdraw = function(amount) {
        if (amount > total) stop("You don't have that much money!\n")
        total <<- total - amount
        cat(amount, "withdrawn.  Your balance is", total, "\n\n")
    }, balance = function() {
        cat("Your balance is", total, "\n\n")
    })
}
```

The first `total` is a formal parameter of the function `open.account`. There is a list with three functions: `deposit`, `withdraw`, and `balance`. Inside these functions, the first `total` is a free variable; they find their values in the enclosing environment's formal parameter (`total`) which is one level up. The special assignment operator, "`<<-`", makes an assignment to the enclosing environment, one level up, updating the value of the formal parameter `total`.

Let's open some savings accounts and make deposits and withdrawals.

[4] https://cran.r-project.org/doc/manuals/r-release/R-intro.html#Scope

```
luis <- open.account(500)
angela <- open.account(100)
luis$withdraw(75)
```

```
#> 75 withdrawn.  Your balance is 425
```

```
angela$deposit(250)
```

```
#> 250 deposited.  Your balance is 350
```

```
luis$balance()
```

```
#> Your balance is 425
```

```
angela$balance()
```

```
#> Your balance is 350
```

5.5 Recursive functions

A function that calls itself is a recursive function. Recursive functions break a problem down into small steps that require the same methodologic technique. For example, consider the factorial of 5:

$$5! = 5 \times 4! = 5 \times 4 \times 3! = 5 \times 4 \times 3 \times 2! == 5 \times 4 \times 3 \times 2 \times 1!$$

We can write a recursive function for factorial calculations:

```
recursive.factorial <- function(x) {
    if (x == 0) {
        return(1)
    } else {
        return(x * recursive.factorial(x - 1))
    }
}
recursive.factorial(5)
```

```
#> [1] 120
```

```
factorial(5)  # compare to native R function
```

```
#> [1] 120
```

5.6 Debugging and profiling R code

TODO: pending

5.7 Example: Bootstrap of risk ratio data

Consider the following data (Table 5.1) with the number of coronary heart disease events comparing men with Type A behavior to men with Type B behavior [17]. There is no formula for calculating an exact confidence interval for a risk ratio. When the distribution for a statistical estimate is not known we can use the bootstrap method. Essentially, this involves the following steps:

1. Simulate risk ratio estimates using the data
2. Estimate the mean of the risk ratios
3. Choose appropriate quantile values as confidence limits

TABLE 5.1: Counts of coronary heart disease (CHD) events for Type A and Type B subjects

	CHD		
Exposure	Yes	No	Total
Type A	178	1411	1589
Type B	79	1486	1565

```
rr.boot <- function(x1, x0, conf.level = 0.95, replicates = 5000){
  #### x1 = c(number cases among exposed, number of exposed)
  #### x0 = c(number cases among nonexposed, number of nonexposed)
  #### prepare input
  n1 <- x1[2]; n0 <- x0[2]
  p1 <- x1[1]/n1 # risk among exposed
  p0 <- x0[1]/n0 # risk among unexposed
```

```
  #### do calculations
  RR <- p1/p0 # maximum likelihood estimate of risk ratio
  r1 <- rbinom(replicates, n1, p1)/n1
  r0 <- rbinom(replicates, n0, p0)/n0
  rrboot <- r1/r0
  rrbar <- mean(rrboot)
  alpha <- 1 - conf.level
  ci <- quantile(rrboot, c(alpha/2, 1-alpha/2))

  #### collect results
  list(x = x,
       risks = c(p1 = p1, p0 = p0),
       risk.ratio = RR,
       rrboot.mean = rrbar,
       conf.int = unname(ci),
       conf.level = conf.level,
       replicates = replicates)
}

#### test function
xx1 <- c(178, 1589) # CHD among Type A
xx0 <- c(79, 1565) # CHD among Type B
rr.boot(xx1, xx0)
```

```
#> $x
#> [1] 11 12 13 14 15
#>
#> $risks
#>         p1         p0
#> 0.11202014 0.05047923
#>
#> $risk.ratio
#> [1] 2.219133
#>
#> $rrboot.mean
#> [1] 2.246046
#>
#> $conf.int
#> [1] 1.737386 2.878927
#>
#> $conf.level
#> [1] 0.95
#>
```

```
#> $replicates
#> [1] 5000
```

5.8 Exercises

Probability distributions are either continuous (e.g., normal distribution) or discrete (e.g. binomial distribution). In R, we deal with four aspects of probability distributions (Table 5.2).

TABLE 5.2: Examples of probability distribution functions in R

Aspects of a probability distribution	Normal	Binomial
probability density function	`pnorm`	`pbinom`
cumulative distribution function	`dnorm`	`dbinom`
quantile ("percentile") function	`qnorm`	`qbinom`
random sample from distribution	`rnorm`	`rbinom`

For example, to get the quantile value from a standard normal distribution (Z value) that corresponds to a 95% confidence interval we use the `qnorm` function.

```
conf.level <- 0.95
qnorm((1 + conf.level)/2)
```

```
#> [1] 1.959964
```

For a second example, suppose we wish to simulate a binomial distribution. Out of 350 patients that adhered to a new drug treatment, 273 recovered. The response proportion was 273/350 or 0.78 (78%). We can use the `rbinom` function to simulate this binomial process. Let's simulate this 10 times (i.e., 10 replicates).

```
replicates <- 10
num_of_events <- 273
num_at_risk <- 350
risk <- num_of_events/num_at_risk
sim_events <- rbinom(n = replicates, size = num_at_risk, prob = risk)
sim_events
```

```
#>  [1] 263 277 267 267 277 266 260 260 265 271
```

```
sim_risks <- sim_events/num_at_risk
round(sim_risks,3)
```

```
#>  [1] 0.751 0.791 0.763 0.763 0.791 0.760 0.743 0.743 0.757 0.774
```

Bootstrapping is a resampling method to estimate a confidence interval. For example, we could simulate 5000 replicates and use the `quantile` function to estimate a 0.95 confidence interval.

```
replicates <- 5000
num_of_events <- 273 #recovered
num_at_risk <- 350
risk <- num_of_events/num_at_risk
sim_events <- rbinom(n = replicates, size = num_at_risk, prob = risk)
conf.level <- 0.95
alpha <- 1 - conf.level
## 0.95 CI for binomial counts
quantile(sim_events, c(alpha/2, 1-alpha/2))
```

```
#>  2.5% 97.5%
#>   257   287
```

```
## 0.95 CI for binomial proportions
quantile(sim_events, c(alpha/2, 1-alpha/2))/num_at_risk
```

```
#>      2.5%     97.5%
#> 0.7342857 0.8200000
```

Exercise 5.1 (Bootstrap a confidence interval)**.** In the same study above, 350 patients did not take the new drug treatment, and 289 recovered. Calculate the bootstrap confidence interval for this proportion (289/350).

Exercise 5.2 (Function for bootstrapping a confidence interval)**.** Adapt your code from the previous problem to write and test a function for calculating the bootstrap confidence interval for a binomial proportion.

For example, if your previous code was

```
z <- 4
z^3
```

then, your function and test might be

```
mycube <- function(x) {
    x^3
}
mycube(4)  # test
```

```
#> [1] 64
```

Exercise 5.3 (Calculating risks and risk ratio)**.** Consider Table 5.3 for cohort (risk) data.

TABLE 5.3: 2×2 table for cohort (risk) data

	Exposed		
Outcome	Yes	No	Total
Yes	a	b	M_1
No	c	d	M_0
Total	N_1	N_0	T

The risk ratio is calculated by

$$RR = \frac{a/N_1}{b/N_0} = \frac{R_1}{R_0},$$

and the standard error is calculated by

$$SE(ln(RR)) = \sqrt{\frac{1}{a} - \frac{1}{N_1} + \frac{1}{b} - \frac{1}{N_0}}.$$

Consider an observational study where 700 patients were given access to a new drug for an ailment. A total of 350 patients chose to take the drug and 350 patients did not. Table 5.4 displays the results of this study.

TABLE 5.4: Study where 700 patients were given access to a new drug treatment

	Treatment		
Recovered	Yes	No	Total
Yes	273	289	562
No	77	61	138
Total	350	350	700

Using the results from Table 5.4 create and test a function that:

- Calculates the risks R_1 and R_0.
- Calculates the risk ratio (RR).
- Calculates 95% confidence intervals.
- Uses the `fisher.test` function (Fisher's Exact Test) to calculate the two-sided p-value of the null hypothesis of no association. (Hint: see `epitable` function in Exercise 5.5)
- Collects and returns all the results as a 'list'.

Exercise 5.4 (Evaluating drug treatment)**.** Consider an observational study where 700 patients were given access to a new drug for an ailment. A total of 350 patients chose to take the drug and 350 patients did not. Table 5.4 displays the results of this study.

Using the `read.csv` function import the data frame for Table 5.4. Use the code below.

```
drugrx <- read.csv('~/data/exported/jp-drugrx-p2.txt')
```

Analyze the `drugrx` data frame using the `xtabs` function, your function from the previous problem, and any R functions you wish. Is the drug treatment effective? Is there a difference in drug treatment effectiveness comparing men and women?

Exercise 5.5 (Oswego outbreak investigation)**.** Study this clean version of the Oswego data set and import it as a data frame: `~/data/oswego/oswego2.txt`. Remember this data set is available from https://github.com/taragonmd/data. Locate the data set and select the "Raw" tab. Copy the URL and use with the `read.csv` function.

Import this clean version.

```
oswego2 <- read.csv("~/data/oswego/oswego2.txt", strip.white = TRUE)
food.item <- c("Baked.ham", "Spinach", "Mashed.potato", "Cabbage.salad",
    "Jello", "Rolls", "Brown.bread", "Milk", "Coffee", "Water", "Cakes",
    "Vanilla.ice.cream", "Chocolate.ice.cream", "Fruit.salad")
names(oswego2)[9:22] <- food.item
str(oswego2)
```

```
#> 'data.frame':    75 obs. of  22 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               : Factor w/ 2 levels "F","M": 1 2 1 1 2 1 1 1 2 1 ...
```

```
#>  $ MealDate           : Factor w/ 1 level "04/18/1940": 1 1 1 1 1 1 1 1 1 1 ...
#>  $ MealTime           : Factor w/ 6 levels "11:00:00","18:30:00",..: 5 2 2 4 4 4 6 3 4
#>  $ Ill                : Factor w/ 2 levels "N","Y": 2 2 2 2 2 2 2 2 2 2 ...
#>  $ OnsetDate          : Factor w/ 2 levels "4/18/1940","4/19/1940": 2 2 2 1 1 2 2 1 2 2
#>  $ OnsetTime          : Factor w/ 17 levels "00:00:00","00:30:00",..: 2 2 2 15 15 4 3 1
#>  $ Baked.ham          : Factor w/ 2 levels "N","Y": 2 2 2 2 2 1 1 2 1 2 ...
#>  $ Spinach            : Factor w/ 2 levels "N","Y": 2 2 2 2 2 1 1 2 1 2 ...
#>  $ Mashed.potato      : Factor w/ 2 levels "N","Y": 2 2 1 1 2 1 1 2 1 1 ...
#>  $ Cabbage.salad      : Factor w/ 2 levels "N","Y": 1 2 1 2 1 1 1 1 1 1 ...
#>  $ Jello              : Factor w/ 2 levels "N","Y": 1 1 1 2 2 1 1 1 1 1 ...
#>  $ Rolls              : Factor w/ 2 levels "N","Y": 2 1 1 1 2 1 1 2 1 2 ...
#>  $ Brown.bread        : Factor w/ 2 levels "N","Y": 1 1 1 1 2 1 1 2 1 1 ...
#>  $ Milk               : Factor w/ 2 levels "N","Y": 1 1 1 1 1 1 1 1 1 1 ...
#>  $ Coffee             : Factor w/ 2 levels "N","Y": 2 2 2 1 2 1 1 1 1 2 ...
#>  $ Water              : Factor w/ 2 levels "N","Y": 1 1 1 2 2 1 1 2 1 1 ...
#>  $ Cakes              : Factor w/ 2 levels "N","Y": 1 1 2 1 1 1 2 1 1 2 ...
#>  $ Vanilla.ice.cream  : Factor w/ 2 levels "N","Y": 2 2 2 2 2 2 1 2 2 2 ...
#>  $ Chocolate.ice.cream: Factor w/ 2 levels "N","Y": 1 2 2 1 1 2 2 2 2 2 ...
#>  $ Fruit.salad        : Factor w/ 2 levels "N","Y": 1 1 1 1 1 1 1 1 1 1 ...
```

Study the following `epitable` function, especially the use of the `...` argument to past options to the `fisher.test` function.

```
epitable <- function(exposure, outcome, digits = 3, ...){
    # Updated 2016-10-16
    # 2x2 table from 'xtabs' function
    # row names are exposure status
    # col names are illness outcome
    # digits = significant digits (default = 3)
    # ... = options to fisher.test function
    x <- xtabs(~ exposure + outcome, na.action=na.pass)
    rowtot <- apply(x, 1, sum)
    rowdist <- sweep(x, 1, rowtot, '/')
    risk0 <- rowdist['N','Y']
    risk1 <- rowdist['Y','Y']
    rr <- risk1/risk0
    or <- (risk1/(1-risk1)) / (risk0/(1-risk0))
    ft <- fisher.test(x, ...)
    pv <- ft$p.value
    results <- signif(c(risk0, risk1, rr, or, pv), digits = digits)
    names(results) <- c('R_0', 'R_1', 'RR', 'OR', 'p.value')
    list(data = x, results = results)
}
```

Here is an example of testing one food item:

```
epitable(oswego2$Baked.ham, oswego2$Ill)
```

```
#> $data
#>          outcome
#> exposure  N  Y
#>        N 12 17
#>        Y 17 29
#>
#> $results
#>     R_0     R_1      RR      OR p.value
#>   0.586   0.630   1.080   1.200   0.809
```

Use the `epitable` function to test each food item to see if it is associated with developing illness. Construct a table with the following column headings:

- Food item
- R_0
- R_1
- RR (risk ratio)
- OR (odds ratio)
- Fisher p value

Discuss your findings.

Exercise 5.6 (Generate outbreak investigation results table)**.** Write a `for` loop to automate the creation of the outbreak table from previous exercise.

Hint provided:

```
outbreak <- matrix(NA, length(food.item), 5)
rownames(outbreak) <- food.item
colnames(outbreak) <- c('R_0','R_1','RR','OR','P value')
for(i in 1:length(food.item)) {
  outbreak[i,] <- epitable(oswego2[,food.item[i]], oswego2$Ill)$results
}
outbreak
```

Exercise 5.7 (Comparing lapply and sapply)**.** Study the following use of `lapply` and `sapply`. Explain how both functions work, and what are their differences.

```
myepitab <- function(x) {
    epitable(x, oswego2$Ill)$results
}
lapply(oswego2[, food.item], myepitab)
```

```
#> $Baked.ham
#>     R_0     R_1      RR      OR p.value
#>   0.586   0.630   1.080   1.200   0.809
#>
#> $Spinach
#>     R_0     R_1      RR      OR p.value
#>   0.625   0.605   0.967   0.918   1.000
#>
#> $Mashed.potato
#>     R_0     R_1      RR      OR p.value
#>   0.622   0.622   1.000   1.000   1.000
#>
#> $Cabbage.salad
#>     R_0     R_1      RR      OR p.value
#>   0.596   0.643   1.080   1.220   0.808
#>
#> $Jello
#>     R_0     R_1      RR      OR p.value
#>   0.577   0.696   1.210   1.680   0.442
#>
#> $Rolls
#>     R_0     R_1      RR      OR p.value
#>   0.658   0.568   0.863   0.682   0.482
#>
#> $Brown.bread
#>     R_0     R_1      RR      OR p.value
#>   0.583   0.667   1.140   1.430   0.622
#>
#> $Milk
#>     R_0     R_1      RR      OR p.value
#>   0.620   0.500   0.807   0.614   0.638
#>
#> $Coffee
#>     R_0     R_1      RR      OR p.value
#>   0.614   0.613   0.999   0.997   1.000
#>
#> $Water
#>     R_0     R_1      RR      OR p.value
#>   0.647   0.542   0.837   0.645   0.450
```

```
#>
#> $Cakes
#>     R_0     R_1      RR      OR p.value
#>   0.543   0.675   1.240   1.750   0.342
#>
#> $Vanilla.ice.cream
#>      R_0      R_1       RR       OR  p.value
#> 1.43e-01 7.96e-01 5.57e+00 2.35e+01 2.60e-07
#>
#> $Chocolate.ice.cream
#>     R_0     R_1      RR      OR p.value
#>  0.7410  0.5320  0.7180  0.3980  0.0891
#>
#> $Fruit.salad
#>     R_0     R_1      RR      OR p.value
#>   0.609   0.667   1.100   1.290   1.000
```

```
t(sapply(oswego2[, food.item], myepitab))  # transpose output
```

```
#>                       R_0   R_1    RR     OR  p.value
#> Baked.ham           0.586 0.630 1.080  1.200 8.09e-01
#> Spinach             0.625 0.605 0.967  0.918 1.00e+00
#> Mashed.potato       0.622 0.622 1.000  1.000 1.00e+00
#> Cabbage.salad       0.596 0.643 1.080  1.220 8.08e-01
#> Jello               0.577 0.696 1.210  1.680 4.42e-01
#> Rolls               0.658 0.568 0.863  0.682 4.82e-01
#> Brown.bread         0.583 0.667 1.140  1.430 6.22e-01
#> Milk                0.620 0.500 0.807  0.614 6.38e-01
#> Coffee              0.614 0.613 0.999  0.997 1.00e+00
#> Water               0.647 0.542 0.837  0.645 4.50e-01
#> Cakes               0.543 0.675 1.240  1.750 3.42e-01
#> Vanilla.ice.cream   0.143 0.796 5.570 23.500 2.60e-07
#> Chocolate.ice.cream 0.741 0.532 0.718  0.398 8.91e-02
#> Fruit.salad         0.609 0.667 1.100  1.290 1.00e+00
```

Exercise 5.8 (Lexical scoping)**.** Study the following R code and explain the final answer for `g(2)`:

```
a <- 1
b <- 2
f <- function(x) {
    a * x + b
}
```

```
g <- function(x) {
    a <- 2
    b <- 1
    f(x)
}
g(2)
```

```
#> [1] 4
```

Exercise 5.9 (More lexical scoping)**.** Study the following R code and explain the final answer for `g(2)`:

```
g <- function(x) {
    a <- 2
    b <- 1
    f <- function(x) {
        a * x + b
    }
    f(x)
}
g(2)
```

```
#> [1] 5
```

6

Displaying data in R—An introduction

Part II

Population health data science

7

Population health approach

7.1 Introduction

In public health we see community health through a socioecological lens focused on complex adaptive systems and life course social networks. We are called upon to lead teams and organizations in making high stakes decisions in the setting of complex environments, limited information, multiple objectives, competing trade-offs, uncertainty, and time constraints. The emerging field of population health data science provides an integrative, dynamic approach to "data-driven decision making" that applies to community *and* health care systems. Our approach to population health has been heavily influenced by public health epidemiology.

First we start with some key definitions:

- Health is a state of complete physical, mental and social well-being and not merely the absence of disease or infirmity [18].
- Public health is what we, as a society, do collectively to assure the conditions in which people can be healthy [19].
- Population health is a systems framework for studying and improving the health of populations through collective action and learning [20].
- Data science is the art and science of transforming data into actionable knowledge [20].
- Population health data science is the art and science of transforming data into actionable knowledge to improve health [20].

Our definition of population health differs from the mainstream definition of population health as "the health outcomes of a group of individuals, including the distribution of such outcomes within the group" [21]. Our definition of population health emphasizes the following:

- complex adaptive socioecological systems
- life course health development
- inter-generational transmission of risks (social, epigenetic)

Our definition supports modern approaches to community health and health

systems transformation [22]. In general, in public health we have four population health goals.

1. Protecting and promoting *health* and *equity*.
2. Transforming *people* and *place*
3. Ensuring a *healthy planet*
4. Achieving *health equity*

To make progress on these goals we focus on changing

- Policies: economic, social, environmental, and health
- Systems: economic, social, environmental, and health
- Enviroments: social, physical, and environmental

The basis for change is decision making; hence, the need for transforming data into actionable knowledge. We start by review data from an epidemiologic perspective.

7.2 Epidemiologic approach

Epidemiology is a basic science of public health, and is commonly defined as

The study of the distribution and determinants of health-related states or events in specified populations, and the application of this study to the control of health problems [23].

Operationally, the epidemiologic approach, from problem definition to public health action is summarized here:

- Epidemiologic measures
- Public health surveillance
- Field investigations (e.g., outbreaks)
- Descriptive and analytic epidemiology
- Program and project evaluation
- Causal and statistical inference
- Evidence-based public health action

7.3 Epidemiologic analyses for 2-by-2 tables

7.3.1 Cohort studies with risk data or prevalence data

	Exposed	Unexposed	Total
Cases	a	b	M_1
Noncases	c	d	M_0
Total at risk	N_1	N_0	T_i

7.3.1.1 Risk difference

7.3.1.2 Risk ratio

7.3.1.3 Odds ratio

7.4 Epidemiologic analyses for stratified 2-by-2 tables

7.4.1 Cohort studies with binomial (risk or prevalence) data

TABLE 7.2: Notation

	Exposed	Unexposed	Total
Cases	a_i	b_i	M_{1i}
Noncases	c_i	d_i	M_{0i}
Total at risk	N_{1i}	N_{0i}	T_i

7.4.1.1 Pooled risk difference with confidence limits

$$RD_{MH} = \frac{\sum_i \frac{a_i N_{0i} - b_i N_{1i}}{T_i}}{\sum_i \frac{N_{1i} N_{0i}}{T_i}}$$

$$\mathrm{Var}(RD_{MH}) = \frac{\sum_i \left(\frac{N_{1i} N_{0i}}{T_i}\right)^2 \left[\frac{a_i c_i}{N_{1i}^2 (N_{1i}-1)} + \frac{b_i d_i}{N_{0i}^2 (N_{0i}-1)}\right]}{\left(\sum_i \frac{N_{1i} N_{0i}}{T_i}\right)^2}$$

```
rd.mh <- function(x, conf.level = 0.95) {
    ai <- x[1, 1, ]  # exposed cases
    bi <- x[1, 2, ]  # unexposed cases
    ci <- x[2, 1, ]  # exposed noncases
    di <- x[2, 2, ]  # unexposed noncases
    N0i <- bi + di  # total exposed (at risk)
    N1i <- ai + ci  # total unexposed
    M0i <- ci + di  # total cases
    M1i <- ai + bi  # total noncases
    Ti <- ai + bi + ci + di  # stratum total
```

```
    RDmh <- (sum((ai * N0i - bi * N1i)/(Ti)))/(sum((N1i * N0i)/(Ti)))
    numer <- ((N1i * N0i)/(Ti))^2 * (((ai * ci)/(N1i^2 * (N1i - 1))) +
        ((bi * di)/(N0i^2 * (N0i - 1))))
    denom <- (sum((N1i * N0i)/(Ti)))^2
    Z <- qnorm((1 + conf.level)/2)
    SE.RDmh <- sqrt(numer/denom)
    LL <- RDmh - Z * SE.RDmh
    UL <- RDmh + Z * SE.RDmh
    list(data = x, risk.diff = RDmh, conf.int = c(LL, UL))
}
```

7.4.1.2 Pooled risk ratio with confidence limits

$$RR_{MH} = \frac{\sum_i \frac{a_i N_{0i}}{T_i}}{\sum_i \frac{b_i N_{1i}}{T_i}}$$

$$\mathrm{Var}[\log(RR_{MH})] = \frac{\sum_i \left(\frac{M_{1i} N_{1i} N_{0i}}{T_i^2} - \frac{a_i b_i}{T_i} \right)}{\left(\sum_i \frac{a_i N_{0i}}{T_i} \right) \left(\sum_i \frac{b_i N_{1i}}{T_i} \right)}$$

```
rr.mh <- function(x, conf.level = 0.95) {
    ai <- x[1, 1, ]  # exposed cases
    bi <- x[1, 2, ]  # unexposed cases
    ci <- x[2, 1, ]  # exposed noncases
    di <- x[2, 2, ]  # unexposed noncases
    N0i <- bi + di  # total exposed (at risk)
    N1i <- ai + ci  # total unexposed
    M0i <- ci + di  # total cases
    M1i <- ai + bi  # total noncases
    Ti <- ai + bi + ci + di  # stratum total
    RRmh <- sum(ai * N0i/Ti)/sum(bi * N1i/Ti)
    numer <- sum(M1i * N1i * N0i/Ti^2 - ai * bi/Ti)
    denom <- sum(ai * N0i/Ti) * sum(bi * N1i/Ti)
    Z <- qnorm((1 + conf.level)/2)
    SE.logRRmh <- sqrt(numer/denom)
    LL <- exp(log(RRmh) - Z * SE.logRRmh)
    UL <- exp(log(RRmh) + Z * SE.logRRmh)
    list(data = x, risk.ratio = RRmh, conf.int = c(LL, UL))
}
```

7.4.1.3 Odds ratio

8

Understanding epidemiologic measures

9

Displaying of epidemiologic data in R

10

Conducting descriptive analysis

11

Conducting predictive analysis

11.1 topic 1

11.2 topic 2

11.3 topic 3

12

Conducting causal analysis

13

Gaining insights with simulations

14

Optimizing decision quality (DQ)

A

Probability review

B

Mathematics review

C

Available data sets

All the data sets described here can be downloaded from https://github.com/taragonmd/data.

Latina Mothers and their Newborn

TABLE C.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)

From 1980 to 1990 data was collected on 427 Latino mothers that gave birth at the University of California, San Francisco cite[Selvin2001_9780195144895, Abrams1995_7617345]. Data was collected on the characteristics of the mothers and their newborn infants (C.1). Mothers were weighed at each prenatal visit. Rate of weight gain during each trimester was based on a linear regression interpolation.

Oswego County (outbreak)

TABLE C.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` (BH)	Consumed item: Y = Yes; N = No
`spinach` (Sp)	Consumed item: Y = Yes; N = No
`mashed.potato` (MP)	Consumed item: Y = Yes; N = No
`cabbage.salad` (CS)	Consumed item: Y = Yes; N = No
`jello` (Je)	Consumed item: Y = Yes; N = No
`rolls` (Ro)	Consumed item: Y = Yes; N = No
`brown.bread` (BB)	Consumed item: Y = Yes; N = No
`milk` (Mi)	Consumed item: Y = Yes; N = No
`coffee` (Co)	Consumed item: Y = Yes; N = No
`water` (Wa)	Consumed item: Y = Yes; N = No
`cakes` (Ca)	Consumed item: Y = Yes; N = No
`vanilla.ice.cream` (VI)	Consumed item: Y = Yes; N = No
`chocolate.ice.cream` (CI)	Consumed item: Y = Yes; N = No
`fruit.salad` (FS)	Consumed item: Y = Yes; N = No

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, vom-

iting, 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 dictionary is provided in C.2.

Western Collaborative Group Study (cohort)

TABLE C.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	68–150 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	8–3430 days

Variable	Variable name	Variable type	Possible values
arcus0	Corneal arcus	Categorical	0 = None; 1 = Yes

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 dictionary is provided in C.3.

Evans County (cohort)

TABLE C.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 $\times$ hpt	Categorical	product term

Variable	Variable name	Variable type	Possible values
cc	cat × chl	Continuous	product term

The Evans County data set is used to demonstrate a standard logistic regression (unconditional) cite[kleinbaum2002]. 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 dictionary is provided in C.4.

Myocardial infarction case-control study

TABLE C.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

The myocardial infarction (MI) data set cite[kleinbaum2002] 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 dictionary is provided in C.5.

AIDS surveillance cases

Download `aids.txt`.

Hepatitis B surveillance cases

Download `hepb.txt`.

Measles surveillance cases

Download `measles.txt`.

West Nile virus surveillance cases, California 2004

Download `./wnv/wnv2004fin.txt`.

Download `./wnv/wnv2004raw.txt`.

University Group Diabetes Program

Download `ugdp.txt`

Novel influenza A (H1N1) pandemic

United States reported cases and deaths as of July 23, 2009

Download `h1n1panflu23jul09usa.txt`.

Exercise solutions

Chapter 2

Exercise 2.2

```
tab <- matrix(c(139, 443, 230, 502), nrow = 2, ncol = 2,
        dimnames = list("Outcome" = c("Dead", "Alive"),
                        Smoker = c("Yes", "No")))
#### equivalent
tab <- matrix(c(139, 443, 230, 502), 2, 2)
dimnames(tab) <- list("Outcome" = c("Dead", "Alive"),
                  Smoker = c("Yes", "No"))
#### equivalent
tab <- matrix(c(139, 443, 230, 502), 2, 2)
rownames(tab) <- c("Dead", "Alive")
colnames(tab) <- c("Yes", "No")
names(dimnames(tab)) <- c("Outcome", "Smoker")
```

Exercise 2.3

Using the `tab` object from previous solution, study and practice the following R code to recreate Table 2.32.

```
rowt <- apply(tab, 1, sum)
tab2 <- cbind(tab, Total = rowt)
colt <- apply(tab2, 2, sum)
tab2 <- rbind(tab2, Total = colt)
names(dimnames(tab2)) <- c("Outcome", "Smoker")
```

Exercise 2.4

Study and execute the following R code:

```
rowt <- apply(tab, 1, sum)          # row distrib
rowd <- sweep(tab, 1, rowt, "/")
```

```
colt <- apply(tab, 2, sum)          # col distrib
cold <- sweep(tab, 2, colt, "/")
jtd <- tab/sum(tab)                 # joint distrib
ptabs <- list(row.distribution = rowd, col.distribution = cold,
              joint.distribution = jtd)
```

Exercise 2.5

Using the tab2 object from previous solution, study and practice the following R code to recreate Table 2.33. Note that the column distributions could also have been used.

```
risk = tab2[1, 1:2]/tab2[3, 1:2]
risk.ratio <- risk/risk[2]
odds <- risk/(1 - risk)
odds.ratio <- odds/odds[2]
results <- rbind(risk, risk.ratio, odds, odds.ratio)
results.rounded <- round(results, 2)
```

Interpretation: The risk of death among non-smokers is higher than the risk of death among smokers, suggesting that there may be some confounding.

Exercise 2.6

```
wdat.oas <- xtabs(~outcome + age4 + smoker, data = wdat)
wdat.tot.oas <- apply(wdat.oas, c(2, 3), sum)
wdat.risk.oas <- sweep(wdat.oas, c(2, 3), wdat.tot.oas, "/")
wdat.risk.oas2 <- round(wdat.risk.oas, 2)
```

Interpretation: The risk of death is not larger in non-smokers, in fact it is larger among smokers in older age groups.

Chapter 3

Exercise 3.1

Approach 1: do the arithmetic using R as a calculator with number counts from Table 3.14.

```
pR1.D0 <- (234+55)/(270+80)
pR1.D1 <- (81+192)/(87+263)
```

```
pR1.D0Gm <- 234/270
pR1.D1Gm <- 81/87
pR1.D0Gw <- 55/80
pR1.D1Gw <- 192/263
```

Approach 2: do the arithmetic using R objects.

```
jp2 <-
read.csv('https://raw.githubusercontent.com/taragonmd/data/master/drugrx-pearl2.csv')
str(jp2)
tab2.rdg <- xtabs(~ Recovered + Drug + Gender, data = jp2)
tab2.rd <- apply(tab2.rdg, c(1,2), sum)
pR1.D0 <- tab2.rd['Yes','No']/sum(tab2.rd[,'No'])
pR1.D1 <- tab2.rd['Yes','Yes']/sum(tab2.rd[,'Yes'])
pR1.D0Gm <- tab2.rdg['Yes','No','Men']/sum(tab2.rdg[,'No','Men'])
pR1.D1Gm <- tab2.rdg['Yes','Yes','Men']/sum(tab2.rdg[,'Yes','Men'])
pR1.D0Gw <- tab2.rdg['Yes','No','Women']/sum(tab2.rdg[,'No','Women'])
pR1.D1Gw <- tab2.rdg['Yes','Yes','Women']/sum(tab2.rdg[,'Yes','Women'])
```

Approach 3: do the arithmetic using `apply` and `sweep` functions. Study and try this solution.

```
sweep(tab2.rd, 2, apply(tab2.rd,2,sum), "/")
sweep(tab2.rdg, 2:3, apply(tab2.rdg, 2:3,sum), "/")
```

Approach 4: do the arithmetic using `prop.table` functions. Study and try this solution.

```
prop.table(tab2.rd, 2)
prop.table(tab2.rdg, 2:3)
```

Exercise 3.2

(a) Evaluate the structure of data frame.

```
std89c <-
read.csv("https://raw.githubusercontent.com/taragonmd/data/master/syphilis89c.csv")
str(std89c)
head(std89c)
lapply(std89c, table)
```

(b) The `Age` variable levels are not ordered correctly. This is because R, by default, converts character vectors into factors with the levels in "alphabetical" order. One option is to set `as.is=TRUE` when using `read.csv`, and set the factor levels after reading in the data.

```
std89c<-
read.csv("https://raw.githubusercontent.com/taragonmd/data/master/syphilis89c.csv",
as.is = c(FALSE, FALSE, TRUE))
str(std89c)
table(std89c$Age)
agelab<-c("<=14","15-19","20-24","25-29","30-34","35-44","45-54",">55")
std89c$Age <- factor(std89c$Age, levels = agelab)
table(std89c$Age)
```

(c) Create 3-dimensional arrays using both the `table` and `xtabs` function, with and without attaching the data frame.

```
## Without attach
table(std89c$Race, std89c$Age, std89c$Sex)
```

```
#> , ,  = Female
#>
#>
#>         <=14  >55 15-19 20-24 25-29 30-34 35-44 45-54
#>   Black  165   92  2257  4503  3590  2628  1505   392
#>   Other   11   15   158   307   283   167   149    40
#>   White   14   24   253   475   433   316   243    55
#>
#> , ,  = Male
#>
#>
#>         <=14  >55 15-19 20-24 25-29 30-34 35-44 45-54
#>   Black   31  823  1412  4059  4121  4453  3858  1619
#>   Other    7  108   210   654   633   520   492   202
#>   White    2  216    88   407   550   564   654   323
```

```
xtabs(~ Race + Age + Sex, data = std89c)
```

```
#> , , Sex = Female
#>
#>        Age
#> Race    <=14  >55 15-19 20-24 25-29 30-34 35-44 45-54
```

```
#>   Black  165   92  2257  4503  3590  2628  1505   392
#>   Other   11   15   158   307   283   167   149    40
#>   White   14   24   253   475   433   316   243    55
#>
#> , , Sex = Male
#>
#>        Age
#> Race    <=14  >55 15-19 20-24 25-29 30-34 35-44 45-54
#>   Black   31  823  1412  4059  4121  4453  3858  1619
#>   Other    7  108   210   654   633   520   492   202
#>   White    2  216    88   407   550   564   654   323
```

```
## With attach and detach
attach(std89c)
table(Race, Age, Sex)
```

```
#> , , Sex = Female
#>
#>        Age
#> Race    <=14  >55 15-19 20-24 25-29 30-34 35-44 45-54
#>   Black  165   92  2257  4503  3590  2628  1505   392
#>   Other   11   15   158   307   283   167   149    40
#>   White   14   24   253   475   433   316   243    55
#>
#> , , Sex = Male
#>
#>        Age
#> Race    <=14  >55 15-19 20-24 25-29 30-34 35-44 45-54
#>   Black   31  823  1412  4059  4121  4453  3858  1619
#>   Other    7  108   210   654   633   520   492   202
#>   White    2  216    88   407   550   564   654   323
```

```
xtabs(~ Race + Age + Sex)
```

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

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

```
detach(std89c)
```

Exercise 3.3

Here we use the `apply` function to get marginal totals for the syphilis 3-dimensional array.

```
tab.ars <- xtabs(~ Age + Race + Sex, data = std89c)
#### 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
```

Exercise 3.4

For this syphilis data example, we'll choose one 3-D array.

```
tab.ars <- xtabs(~ Age + Race + Sex, data = std89c)
rowt <- apply(tab.ars, c(1, 3), sum) # row distrib
rowd <- sweep(tab.ars, c(1, 3), rowt, "/"); rowd
apply(rowd, c(1, 3), sum) # confirm
colt <- apply(tab.ars, c(2, 3), sum) #  col distrib
cold <- sweep(tab.ars, c(2, 3), colt, "/"); cold
apply(cold, c(2, 3), sum) # confirm
jtt <- apply(tab.ars, 3, sum) # joint distrib
jtd <- sweep(tab.ars, 3, jtt, "/"); jtd
apply(jtd, 3, sum) # confirm
distr <- list(rowd, cold, jtd); distr
```

Exercise 3.5

```
table(std89c)
data.frame(table(std89c))
```

Exercise 3.6

Use the `rep` function on the data frame rows to recreate the individual-level data frame with over 40,000 observations.

It's a good idea to understand how the `rep` function works with two vectors:

```
rep(c(1,2,3), c(4,5,6))
```

```
#>  [1] 1 1 1 1 2 2 2 2 2 3 3 3 3 3 3
```

We can see that the second vector determines the frequency of the first vector elements. Now use this understanding with the syphilis data.

```
std89b <-
read.csv("https://raw.githubusercontent.com/taragonmd/data/master/syphilis89b.csv",
as.is = c(FALSE, FALSE, TRUE, FALSE))
str(std89b)
std89b
expand.rows <- rep(1:nrow(std89b),std89b$Freq)
std89b.df <- std89b[expand.rows,]
agelab<-c("<=14","15-19","20-24","25-29","30-34","35-44","45-54",">55")
std89b.df$Age <- factor(std89b.df$Age, levels = agelab)
str(std89b.df)
table(std89b.df$Age)
```

Exercise 3.7

Chapter 5

Exercise 5.1

```
replicates <- 5000
num_of_events <- 289 # recovered
num_at_risk <- 350
risk <- num_of_events/num_at_risk
sim_events <- rbinom(n = replicates, size = num_at_risk, prob = risk)
conf.level <- 0.95
alpha <- 1 - conf.level
## 0.95 CI for binomial counts
quantile(sim_events, c(alpha/2, 1-alpha/2))
```

```
## 0.95 CI for binomial proportions
quantile(sim_events, c(alpha/2, 1-alpha/2))/num_at_risk
```

Exercise 5.2

```
risk.boot <- function(x, n, conf.level = 0.95, replicates = 5000){
    ## x = number of events (numerator)
    ## n = number at risk (denominator)
    ## conf.level = confidence level (default 0.95)
    ## replicates = number of simulations
    ## prepare inputs
    num_of_events <- x
    num_at_risk <- n
    ## do calculations
    risk <- num_of_events/num_at_risk
    sim_events <- rbinom(n = replicates, size = num_at_risk, prob = risk)
    conf.level <- conf.level
    alpha <- 1 - conf.level
    ci <- quantile(sim_events, c(alpha/2, 1-alpha/2))
    ## collect results
    list(x = num_of_events,
         n = num_at_risk,
         conf.int = ci,
         conf.level = conf.level)
}

## test
risk.boot(10, 20)
```

Exercise 5.3

```
myRR <- function(x, conf.level = 0.95){
    ## x is 2x2 table with this layout
    ## |Outcome| Exposed   |
    ## |       | Yes | No  | Sum |
    ## |-------+-----+-----+-----|
    ## |   Yes |  a  |  b  | M_1 |
    ## |    No |  c  |  d  | M_0 |
    ## |-------+-----+-----|-----|
    ## |   Sum | N_1 | N_0 |  T  |
    ##
    ## Prepare input
    tab <- addmargins(x)
```

```
    aa <- tab[1, 1]
    bb <- tab[1, 2]
    N1 <- tab['Sum',1]
    N0 <- tab['Sum',2]

    ## Do calculations
    R1 <- aa/N1
    R0 <- bb/N0
    RR <- R1/R0
    Z <- qnorm((1 + conf.level)/2)
    logRR <- log(RR)
    SE_logRR <- sqrt(1/aa - 1/N1 + 1/bb - 1/N0)
    CI <- exp(logRR + c(-1, 1) * Z * SE_logRR)
    pv <- fisher.test(tab[1:2,1:2])$p.value

    ## Collect results
    list(data = tab,
         risks = prop.table(x, 2),
         risk.ratio = RR,
         conf.level = conf.level,
         conf.int = CI,
         p.value = pv)
}

## test function
tab <- matrix(c(273, 289, 77, 61), 2, 2, byrow = TRUE,
              dimnames = list (Recovered = c('Yes', 'No'),
                               Treatment = c('Yes', 'No')))
myRR(tab)
```

Exercise 5.4

```
url <- "https://raw.githubusercontent.com/taragonmd/data/master/exported/jp-drugrx-p2.txt"
drugrx <- read.csv(url)
str(drugrx)
########## ALL ##########
tab_all <- xtabs(~ Recovered + Drug, data = drugrx)[2:1,2:1]
myRR(tab_all)
########## MEN ##########
tab_men <- xtabs(~ Recovered + Drug + Gender, data = drugrx)[2:1,2:1,'Men']
myRR(tab_men)
########## WOMEN ##########
```

```
tab_women <- xtabs(~ Recovered + Drug + Gender, data = drugrx)[2:1,2:1,'Women']
myRR(tab_women)
```

Exercise 5.5

Exercise 5.6

Exercise 5.7

Exercise 5.8

Exercise 5.9

Bibliography

1. Spetzler C, Winter H, Meyer J. Decision quality: Value creation from better business decisions. 1st ed. Wiley; 2016.

2. Kahneman D. Thinking, fast and slow. New York: Farrar, Straus; Giroux; 2011.

3. Nisbett R. Mindware: Tools for smart thinking. New York: Farrar, Straus; Giroux; 2016.

4. Pearl J. The Book of Why: The new science of cause and effect. Basic Books; 2018.

5. Barrett L. How emotions are made: The secret life of the brain. Mariner Books; 2018.

6. Adhikari A, DeNero J. Computational and inferential thinking: The foundations of data science. Available from: https://www.inferentialthinking.com/; 2017.

7. 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;54(RR-2):1–20.

8. Olsen SJ, Chang H-L, Cheung TY-Y, Tang AF-Y, Fisk TL, Ooi SP-L, et al. Transmission of the severe acute respiratory syndrome on aircraft. N Engl J Med. 2003 Dec;349(25):2416–22.

9. The GenIUSS Group. Best practices for asking questions to identify transgender and other gender minority respondents on population-based surveys [Internet]. The Williams Institute; 2014. Available from: https://williamsinstitute.law.ucla.edu/wp-content/uploads/geniuss-report-sep-2014.pdf

10. Rothman K. Epidemiology: An introduction. 2nd ed. New York, NY: Oxford University Press; 2012.

11. Scutari M, Denis J-B. Bayesian networks: With examples in r. Boca Raton: Chapman; Hall; 2014.

12. Fenton N, Neil M. Risk assessment and decision analysis with bayesian networks. Chapman; Hall/CRC; 2 edition; 2019.

13. Neapolitan R, Jiang X, Ladner DP, Kaplan B. A primer on Bayesian decision analysis with an application to a kidney transplant decision. Transplantation. 2016 Mar;100(3):489–96.

14. Charig CR, Webb DR, Payne SR, Wickham JE. Comparison of treatment of renal calculi by open surgery, percutaneous nephrolithotomy, and extracorporeal shockwave lithotripsy. Br Med J (Clin Res Ed). 1986 Mar;292(6524):879–82.

15. Julious SA, Mullee MA. Confounding and simpson's paradox. BMJ. 1994 Dec;309(6967):1480–1.

16. Aragón TJ, Lichtensztajn DY, Katcher BS, Katz RRMH. Calculating expected years of life lost for assessing local ethnic disparities in causes of premature death. BMC Public Health [Internet]. 2008 Apr;8:116. Available from: https://bmcpublichealth.biomedcentral.com/articles/10.1186/1471-2458-8-116

17. Selvin S. Modern applied biostatistical methods using s-plus. New York: Oxford University Press; 1998.

18. World Health Organization. WHO definition of health. World Health Organization; World Health Organization; 1948.

19. Institute of Medicine. The future of public health [Internet]. Washington, DC: The National Academies Press; 1988. Available from: https://www.nap.edu/catalog/1091/the-future-of-public-health

20. Aragón TJ, Colfax G. We will be the best at getting better! A playbook for population health improvement. UC Berkeley eScholarship [Internet]. 2019; Available from: https://escholarship.org/uc/item/9xg5t30s

21. Kindig D, Stoddart G. What is population health? Am J Public Health. 2003 Mar;93(3):380–3.

22. Halfon N, Larson K, Lu M, Tullis E, Russ S. Lifecourse health development: Past, present and future. Matern Child Health J. 2014 Feb;18(2):344–65.

23. Last J. A dictionary of epidemiology. 4th ed. New York: Oxford University Press; 2000.