Applied Epidemiology using R

Tomás Aragón, MD, DrPH, Medical Epidemiologist
Center for Infectious Disease Preparedness
UC Berkeley School of Public Health
Email: aragon@berkeley.edu
URL: http://www.idready.org

Modified February 14, 2004

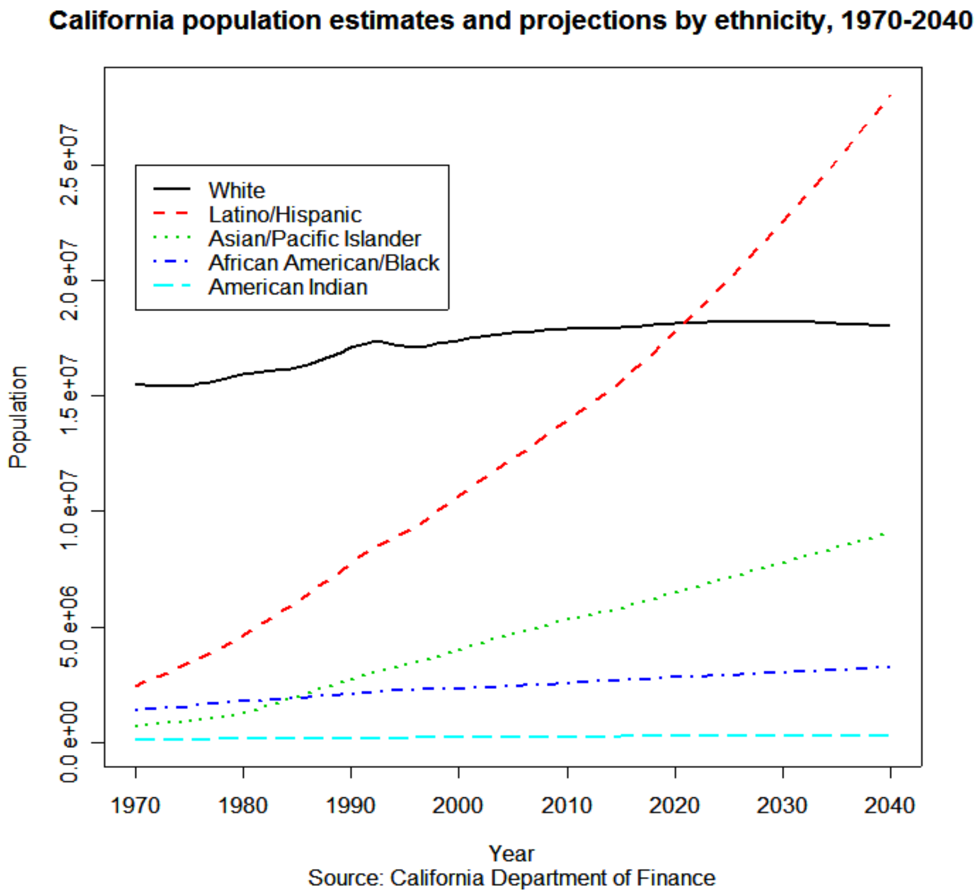


Table of Contents

Index of Tables

Preface

Acknowledgments

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 indispensible for anyone that uses and interprets data. As a physician, medical epidemiologist, and public health practitioner, I use R in the following ways:

- full-function calculator
- extensible statistical package
- high-quality graphics tool
- multi-use programming language

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

In fact, I 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, loglinear models, 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 graph data. Repetitive data analytic tasks can be automated or streamlined with the creation of simple functions (programs that execute specific tasks). The learning curve initially is challenging, but in the long run one is able to conduct analyses that would otherwise require a tremendous amounts of computer programming and time.

Some epidemiologists may find R difficult to learn. This is because R is most often used by hard-core statistician-types who feel at home with matrix algebra and using R's statistical programming capabilities. However, even for those unfamiliar with matrix algebra, there are many analyses one can accomplish in R without using any advanced mathematics and that would be very difficult in other programs. The ability to easily manipulate data in R will allow one to do good descriptive epidemiology, life table methods, graphical displays, and exploration of epidemiologic concepts. R allows one to work with data in any way it comes.

1.2 Who should learn R?

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

1.3 Why should I learn R?

In order to exercise your intuition you need a computational tool. On one end of the spectrum are *calculators* and *spreadsheets* for simple calculations, and on the end of the spectrum are specialized computer programs for such things as statistical modeling. However, many numerical problems are not easily handled by these tools. Calculators, and even spreadsheets, are too inefficient and cumbersome for numerical calculations whose scope and scale change frequently. Statistical packages are usually tailored for the statistical analysis of data sets and often lack an intuitive and extensible programming language for tackling new problems efficiently. R can do the simplest and the most complex analysis efficiently and effectively.

When you learn and use R regularly you 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. It's straightforward programming language facilitates the development of functions to extend and improve the efficiency of your analyses.

give examples -

1.4 Where can I get R?

R is available for many computer platforms, including Windows, Mac OS, Linux, and others. R comes as source or binary code. Source code needs to be compiled (which we will not expect); binary code is ready for installation. I assume most readers will be using R in the Microsoft Windows environment. Listed here are the useful links for R:

- The **R Project home page** is at http://www.r-project.org.
- The **R download page** is at the Comprehensive R Archive Network (CRAN) at http://cran.r-project.org.
- Numerous **free tutorials** are available at http://cran.r-project.org/other-docs.html.

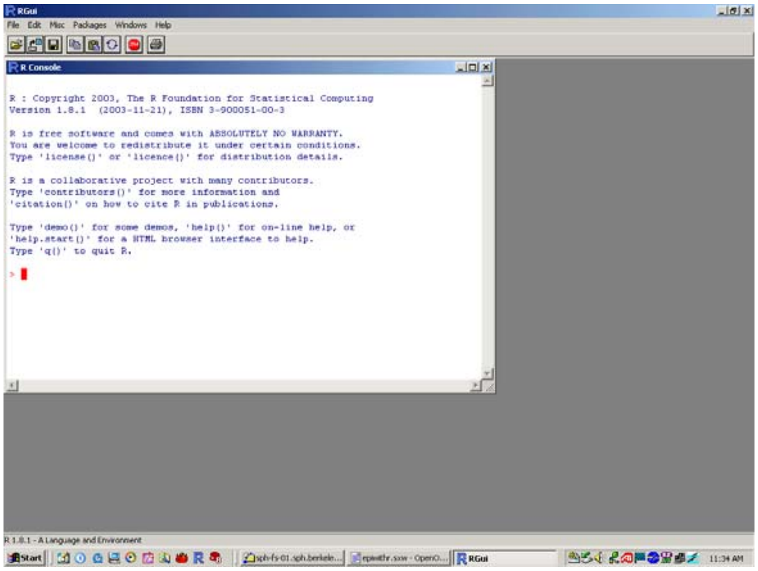
Example of installation

- On your desktop computer create a directory for download R (e.g., `C:\downloads\R for Windows`)
- Go to http://r-project.org URL.
- To download click on “CRAN” link on the left menu list.
- Click on http://cran.us.r-project.org URL.
- Click on “Windows (95 and later)” link.
- Click on “base” link.
- To download installation program click on “`rw1081.exe`” and select to save to your computer (the file is about 20 MB). For example, you can save to `C:\downloads\R for Windows`.
- Run the installation program `rw1081.exe`. That's it!

1.5 How do I use R?

Using R on your computer

You use R by typing in commands at the R command prompt (>) and pressing Enter on your keyboard. This is how to use R interactively. From the R command line, you can also run a list of R commands that you have saved in a text file.



RGui
File Edit Misc Packages Windows Help
R Console
R 1.8.1 - A Language and Environment
Start
RGui
11:34 AM

```
R : Copyright 2003, The R Foundation for Statistical Computing
Version 1.8.1  (2003-11-21), ISBN 3-900051-00-3

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for a HTML browser interface to help.
Type 'q()' to quit R.

>
```

Figure 1 R graphical user interface (GUI)

Using R on the World Wide Web

Although we highly recommend installing R on your computer, for a variety of reasons you may not be able to do so. This is not a major problem because you can run R commands using **Rweb**. Rweb is a Web-based interface to R that takes the submitted code, runs R on the code (in batch mode), and returns the output (printed and graphical). Until UC Berkeley School of Public Health implements Rweb, you can try Rweb from the University of Minnesota Statistics Department at http://rweb.stat.umn.edu/Rweb/Rweb.general.html.

1.6 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. When I display R code in these notes it appears as if I am typing the code directly at the R command prompt:

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

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

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

```
> agegrps
[1] "Age < 1"      "Age 1 to 4"   "Age 5 to 14"  "Age 15 to 24"
[5] "Age 25 to 44" "Age 45 to 64" "Age 64+"
```

An equivalent way of running the above code is to type out the lines of code in a text editor (e.g., Notepad). For example, when I display R code in a text editor it will appear like this:

```
x <- matrix(1:9,3,3)
x
agegrps <- c('Age < 1', 'Age 1 to 4', 'Age 5 to 14', 'Age 15 to 24',
      'Age 25 to 44', 'Age 45 to 64', 'Age 64+')
agegrps
```

It is a good idea to save your code with a convenient file name such as job01.r (notice the .r extension; although not necessary, it is useful for searching for R command files, and this file extension is recognized by some text editors [see next section]). The code in your text editor can be run in the following ways:

- Paste the code directly into R at the command prompt
- Paste the code directly into the Rweb command window
- Run the file in batch mode from the R command prompt using the `source` command: `source(“c:/myjobs/job01.r”)`

1.7 Just do it!

Using R as your calculator

Open R and start using it as your calculator. The most common math operators are displayed in Table 1. From now on make R your default calculator!

Practice

Study the examples in Table 1 and spend a few minutes experimenting with R as a calculator. Use parentheses as needed to group operations: `1-exp(-(.5*4+.6*4))`

Hint

Use the keyboard Up-arrow to recall what you previously entered at the command line prompt.

Table 1 Selected math operators

Operator	***Description***	***Examples in R***

+	addition	`>  5 + 4`
		`[1] 9`

-	subtraction	`>  5 - 4`
		`[1] 1`

*	multiplication	`>  5 * 4`
		`[1] 20`

/	division	`>  5 / 4`
		`[1] 1.25`

^	exponentiation	`> 5^4`
		`[1] 625`

Operator	**Description**	**Examples in R**
-	unary minus (change current sign)	> -5 [1] -5 > -(+5) [1] -5 > -(-5) [1] 5
abs	absolute value	> abs(-23) [1] 23
exp	exponentiation (e to a power)	> exp(8) [1] 2980.958
log	logarithm (default is natural log)	> log(exp(8)) [1] 8
sqrt	square root	> sqrt(64) [1] 8
%/%	integer divide	> 10 %/% 3 [1] 3
%%	modulus	> 10 %% 3 [1] 1
%*%	matrix multiplication	> xx <- matrix(1:4,2,2) > xx [,1] [,2] [1,] 1 3 [2,] 2 4 > xx %*% c(1,1) [,1] [1,] 4 [2,] 6

Useful R concepts

Types of evaluable expressions

Every *expression* that is entered at the R command prompt is evaluated by R and returns a *value* (for example, R evaluates the expression 4*4 and returns the value 16).

Table 2 Types of evaluable expressions

Expression type	**Examples in R**
character	> "hello, my name is Tomas" [1] "hello, my name is Tomas"
complex	> 8+3i [1] 8+3i
numeric	> 3.5 [1] 3.5
logical	> TRUE [1] TRUE > F [1] FALSE
function	> print(x) [1] "hello, my name is Tomas"

Expression type	***Examples in R***

math operation	`> 6*7`
	`[1] 42`

comment	`> # lines preceded with pound sign (#) are not evaluated`
	`>`

data object (eg., x)	`> x <- "hello, my name is Tomás"`
	`> x`
	`[1] "hello, my name is Tomas"`

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.

Table 3 Examples of assignment operator (<-)

Examples in R

`> xx <- "hello, what's your name"`
`> xx`
`[1] "hello, what's your name"`
`> yy <- 5.5^3`
`> yy`
`[1] 166.375`
`> zz <- F`
`> zz`
`[1] FALSE`
`> zz <- T`
`> zz`
`[1] TRUE`

Practice

Study the examples in Table 3 and spend a few minutes using the assignment operator to create and call data object.

Hint

Try to use short descriptive names if possible. It is bad practice to use a single letter as an object name.

Useful R functions

When you start R you have opened a *workspace*. Every time you create a data object it is saved in the workspace. If a data object with the same name already exists the old data object will be erased, so be careful. Data objects can be saved between sessions. You will be prompted with "Save workspace image?" (You can also use `save.image()` at the command prompt.) The workspace image is saved in a file called `.RData`. Use `getwd()` to display the file path name to the to `.RData`. Table 4 has more useful R functions.

Table 4 Useful R functions

Function	***Description***	***Examples in R***
q	Quit R	`> q()`

ls
objects
	List objects	`> ls()`
		`[1] "last.warning" "mx"           "ss"`
		`[4] "x"            "xx"           "yy"`
		`> objects() #equivalent to previous`
		`[1] "last.warning" "mx"           "ss"`
		`[4] "x"            "xx"           "yy"`

rm
remove
	Remove object(s)	`> ls()`
		`[1] "xx" "yy" "zz"`
		`> rm(yy)`
		`> ls()`
		`[1] "xx" "zz"`
		`> remove(xx) #equivalent to 'rm'`
		`> ls()`
		`[1] "zz"`

help	open help instructions; or get help on specific topic.	`> help()`
		`> help(plot)`
		`> ?plot #equivalent to previous`

help.start	start help browser	`> help.start()`

getwd	get working directory	`> getwd()`
		`[1] "C:/Program Files/R/rw1081"`

setwd	set working directory	`> setwd("C:/mywork/project1/R/")`
apropos	displays of all objects in the search list matching topic	`> apropos(plot)`

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

example	runs example of function	`> example(plot)`

data	information on available R data sets; load data set	`> data() #displays available data sets`
		`> data(Titanic) #loads Titanic data set`
		`>`

save.image	saves current workspace	`> save.image()`
		`>`

Practice

Study the examples in Table 3 and spend a few minutes experimenting with these useful R functions.

What are packages?

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

```
 > search()
[1] ".GlobalEnv"      "package:methods" "package:ctest"
     "package:mva"
[5] "package:modreg"  "package:nls"     "package:ts"      "Autoloads"
```

```
[9] "package:base"
```

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

```
> searchpaths()
[1] ".GlobalEnv"
[2] "C:/PROGRA~1/R/rw1081/library/methods"
[3] "C:/PROGRA~1/R/rw1081/library/ctest"
[4] "C:/PROGRA~1/R/rw1081/library/mva"
[5] "C:/PROGRA~1/R/rw1081/library/modreg"
[6] "C:/PROGRA~1/R/rw1081/library/nls"
[7] "C:/PROGRA~1/R/rw1081/library/ts"
[8] "Autoloads"
[9] "C:/PROGRA~1/R/rw1081/library/base"
```

To learn more about a specific package enter `library(help=`*`package_name`*`)`. Alternatively, you 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 you need to load a package enter `library(`*`package_name`*`)`. For example, when we cover survival analysis we will need to load the `survival` package.

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 interest in taking a random sample of days from the month of June which has 30 days. We want to use the `sample` function but we forgot the syntax. Let's explore:

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

Whoa! This gave more information that I need or want! What happened? Whenever you type the function name without any parentheses it usually returns the whole function code. This is useful when you start programming and you need to (1) alter an existing function, (2) borrow code for your own functions, or (3) 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 you do not need to specify the argument.

```
> dates <- 1:30
> sample(dates, 20)
 [1]  5 30  8 23  4 16 24  3 29 13  1 15 26  7 10 17 28 25 19 18
```

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

```
> sample(20, dates) #gives undesired results
[1] 14
> #No! We wanted sample of size=20
> sample(size = 20, x = dates) #gives desired result
 [1] 30 14 19 21 12  7  9 25 26  8 18  2  6 17 16 23 13 22 11  5
```

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

```
> sample(s = 20, x = dates, r = T) #sampling with replacement
 [1] 23 10 23 27 13 14  1  7 23 26 28  3 23 28  9  6 23  5 30 10
```

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

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

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

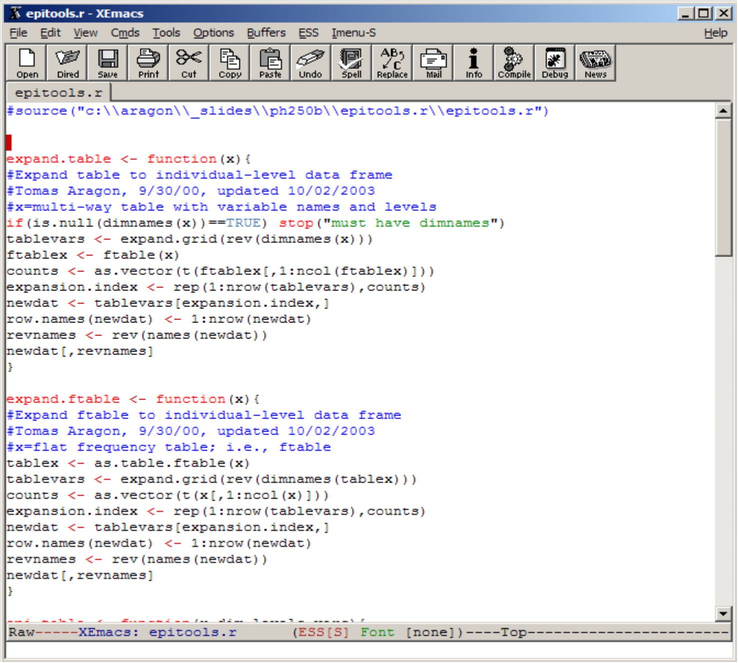
1.8 Is there anything else that I need?

Maybe. (Yes if you are serious about data analysis!) A good text editor will make your programming and data processing easier and more efficient. A text editor is a program for, you guessed it, editing text! The functionality I 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 large datasets for editing

When you are programming you want your text to wrap so you can read all your code. When you import a dataset that is wider than the screen you do not want the dataset to wrap; you want it to appear in its table format. Column editing allows you to cut and paste columns of text at will. A macro is just a way for the program 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", has any number of letters in between but not "r" and "f". Regular expression searching makes this a trivial task. All these are powerful features that once you use regularly, you will wonder how you ever got along without them.

If you do not want to install a text editing program then just use the default text editor that comes with your computer operating system (for Windows use Notepad). However, if you are game, I highly recommend installing the freely available, open source XEmacs (http://www.xemacs.org). The default Windows installations of XEmacs comes with numerous useful and powerful packages such as “Emacs Speaks Statistics” (ESS) which works with R. I maintain a web page of XEmacs short cuts at http://www.medepi.org/xemacs. To read tutorials do a Google search on “emacs tutorial” or “xemacs tutorial.”


epitools.r - XEmacs
File Edit View Cmds Tools Options Buffers ESS Imenu-S Help
Open Dired Save Print Cut Copy Paste Undo Spell Replace Mail Info Compile Debug News
epitools.r

```
#source("c:\\aragon\\_slides\\ph250b\\epitools.r\\epitools.r")


expand.table <- function(x){
#Expand table to individual-level data frame
#Tomas Aragon, 9/30/00, updated 10/02/2003
#x=multi-way table with variable names and levels
if(is.null(dimnames(x))==TRUE) stop("must have dimnames")
tablevars <- expand.grid(rev(dimnames(x)))
ftablex <- ftable(x)
counts <- as.vector(t(ftablex[,1:ncol(ftablex)]))
expansion.index <- rep(1:nrow(tablevars),counts)
newdat <- tablevars[expansion.index,]
row.names(newdat) <- 1:nrow(newdat)
revnames <- rev(names(newdat))
newdat[,revnames]
}

expand.ftable <- function(x){
#Expand ftable to individual-level data frame
#Tomas Aragon, 9/30/00, updated 10/02/2003
#x=flat frequency table; i.e., ftable
tablex <- as.table.ftable(x)
tablevars <- expand.grid(rev(dimnames(tablex)))
counts <- as.vector(t(x[,1:ncol(x)]))
expansion.index <- rep(1:nrow(tablevars),counts)
newdat <- tablevars[expansion.index,]
row.names(newdat) <- 1:nrow(newdat)
revnames <- rev(names(newdat))
newdat[,revnames]
}

Raw-----XEmacs: epitools.r       (ESS[S] Font [none])----Top-----------------------
```

Figure 2 XEmacs text editor running Emacs Speaks Statistics and editing file with R code

1.9 What's ahead?

Table 5 Concise summary and location of material covered, Modified February 14, 2004

	Atomic			***Recursive***		
	Vector	***Matrix***	***Array***	***List***	***Data Frame***	***Function***
Working with R objects						
Understanding						

	Atomic			***Recursive***		
	Vector	***Matrix***	***Array***	***List***	***Data Frame***	***Function***
Creating	Table 9 (p 28) c : seq sequence rep paste as.vector vector character complex numeric logical gl *indexing*	Table 16 (p 40) cbind rbind matrix dim array xtabs ftable as.matrix outer *indexing*	Table 22 (p 53) array table as.table dim as.array	Table 27 (p 68) list as.list vector data.frame as.data.frame read.table read.csv read.csv2 read.delim read.delim2 read.fmf	Table 33 (p 78) data.frame as.data.frame read.table read.csv read.csv2 read.delim read.delim2 read.fmf	
Naming	Table 10 (p 31) names	Table 17 (p 42) dimnames names	Table 23 (p 56) dimnames names	Table 28 (p 69) names	Table 34 (p 79) names row.names	
Indexing	Table 11 (p 32) *by name* *by position* *by logical*	Table 18 (p 43) *by name* *by position* *by logical*	Table 24 (p 58) *by name* *by position* *by logical*	Table 29 (p 70) *by name* *by position* *by logical*	(Table 35 p 79) *by name* *by position* *by logical*	
Replacing	Table 12 (p 33) *by name* *by position* *by logical*	Table 19 (p 44) *by name* *by position* *by logical*	Table 25 (p 60) *by name* *by position* *by logical*	Table 30 (p 71) *by name* *by position* *by logical*	Table 36 (p 81) *by name* *by position* *by logical*	
Operations	Table 13 (p 34) sum, cumsum diff prod, cumprod mean, median min, max, range rev order, sort rank sample quantile var, sd Table 14 (p 35) c, append cbind, rbind table, ftable outer <, > <=, >= ==, != ! &, && |, || xor	Table 20 (p 45) t apply tapply sweep margin.table prop.table	Table 26 (p 63) aperm apply sweep margin.table prop.table	Table 31 (p 72) lapply sapply mapply attach detach	Table 37 (p 82) tapply lapply sapply mapply aggregate by attach detach	
Managing objects	is.vector	is.matrix	is.array	is.list	is.data.frame	

	Atomic			**Recursive**		
	Vector	***Matrix***	***Array***	***List***	***Data Frame***	***Function***
Managing workspace						
Working with epidemiologic data						
Entering						
Sorting						
Subsetting						
Transforming						
Merging						
Exporting						
Importing						
Missing values						
Calendar dates						
Analyzing epidemiologic data						

2 Working with R data objects

2.1 Data objects in R

Atomic vs. recursive data objects

Data in R are organized as objects and have been assigned a name. The mode of an object describes the type of data it contains and is available by using the mode function (e.g., mode (*object*)). To see the list of objects available in your workspace type objects().

The analysis of data in R involves creating, naming, manipulating, and operating on data objects using functions. You have already been introduced to several R data objects. We will now make some additional distinctions. Data objects can be 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, complex, numeric, or logical. Vectors, matrices, arrays, are atomic data objects.

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

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

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

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

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

```
> x <- 1:8
> x
[1] 1 2 3 4 5 6 7 8
> y <- array(x, dim=c(2, 2, 2))
> y
```

```
, , 1

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

, , 2

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

If one tries 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 > complex > numeric > logical. In other words, if an atomic data object contains any character element, all elements will be coerced to character.

```
> c("hello", 5+3i, 4.56, FALSE) #will coerce to character
[1] "hello" "5+3i"  "4.56"  "FALSE"
> c(5+3i, 4.56, FALSE) #will coerce to complex
[1] 5.00+3i 4.56+0i 0.00+0i
> c(4.56, FALSE) #will coerce to numeric
[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` (see Table 6, p. 21).

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

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

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

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

A data frame is a list with a 2-dimensional (table) structure. Epidemiologists are very experienced working with data frames where each row represents data collected on individual study subjects and columns represent fields for each type of data collected.

```
> 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"
> class(mydat)
[1] "data.frame"
```

Assessing structure of data objects

Table 6 summary summarizes key attributes of atomic data objects (vectors, matrices, and arrays) and recursive data objects (lists, data frames, and functions). 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 special methods designed specifically for that class of object (for example, printing and plotting display). For our purposes, you do not need to know any more about classes.

Table 6 Summary of types of data objects in R

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

Frequently, we will need to assess the structure of data objects. At a minimum, 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.

```
> 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 with 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 ...
 $ age           : num  26 42 39 34 35 36 23 32 21 28 ...
 $ parity        : num  6 1 6 4 3 4 1 2 1 2 ...
 $ induced       : num  1 1 2 2 1 2 0 0 0 0 ...
 $ case          : num  1 1 1 1 1 1 1 1 1 1 ...
 $ spontaneous   : num  2 0 0 0 1 1 0 0 1 0 ...
 $ stratum       : int  1 2 3 4 5 6 7 8 9 10 ...
```

Great! 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 (see Table 7). We will see extensive use of this when we start programming in R.

Practice

At the command prompt, enter `data()` to display the available data sets in R. Then enter `data(`*dataset*`)` to load a data set. Study the examples in Table 7 and spend a few minutes exploring the structure of the data sets you have loaded.

Hint

To display detailed information about a specific data set use `?`*dataset* at the command prompt (e.g., `?infert`).

Table 7 Useful functions to assess structure of R data objects

Function	***Description***	***Examples in R using infert data frame***
Returns summary objects		
str	displays summary of data object structure	see text

attributes	return list with data object attributes	`> attributes(infert)`
		`$names`
		`[1] "education"      "age"`
		`[3] "parity"         "induced"`
		`[5] "case"           "spontaneous"`
		`[7] "stratum"        "pooled.stratum"`
		
		`$class`
		`[1] "data.frame"`
		
		`$row.names`
		`  [1] "1"   "2"   "3"   "4"   "5"   "6"   "7"`
		`  [8] "8"   "9"   "10"  "11"  "12"  "13"  "14"`
		`...`
		`[239] "239" "240" "241" "242" "243" "244" "245"`
		`[246] "246" "247" "248"`

Function	***Description***	***Examples in R using infert data frame***

`attr`	assign user-defined attributes	`> attr(infert, "design") <- "Case-control"`
		`> attr(infert, "analyst") <- "John Snow"`
		`> attributes(infert)`
		`$names`
		`[1] "education"      "age"`
		`[3] "parity"         "induced"`
		`[5] "case"           "spontaneous"`
		`[7] "stratum"        "pooled.stratum"`
		
		`$class`
		`[1] "data.frame"`
		
		`$row.names`
		`  [1] "1"   "2"   "3"   "4"   "5"   "6"   "7"`
		`  [8] "8"   "9"   "10"  "11"  "12"  "13"  "14"`
		`...`
		`[239] "239" "240" "241" "242" "243" "244" "245"`
		`[246] "246" "247" "248"`
		
		`$design`
		`[1] "Case-control"`
		
		`$analyst`
		`[1] "John Snow"`

`Returns specific information`		

`mode`	return mode of object	`> mode(infert)`
		`[1] "list"`

`length`	returns length of object	`> length(infert)`
		`[1] 8`

`dim`	returns vector with object dimensions, if applicable	`> dim(infert)`
		`[1] 248   8`

`nrow`	returns number of rows, if applicable	`> nrow(infert)`
		`[1] 248`

`ncol`	returns number of columns, if applicable	`> ncol(infert)`
		`[1] 8`

`dimnames`	returns list containing vectors of names for each dimension, if applicable	`> dimnames(infert)`
		`[[1]]`
		`  [1] "1"   "2"   "3"   "4"   "5"   "6"   "7"`
		`  [8] "8"   "9"   "10"  "11"  "12"  "13"  "14"`
		`...`
		`[239] "239" "240" "241" "242" "243" "244" "245"`
		`[246] "246" "247" "248"`
		
		`[[2]]`
		`[1] "education"      "age"`
		`[3] "parity"         "induced"`
		`[5] "case"           "spontaneous"`
		`[7] "stratum"        "pooled.stratum"`

Function	***Description***	***Examples in R using infert data frame***

names	returns vector of names for the list, if applicable (for a data frame it returns the variable names)	`> names(infert)`
		`[1] "education"      "age"`
		`[3] "parity"         "induced"`
		`[5] "case"           "spontaneous"`
		`[7] "stratum"        "pooled.stratum"`

2.2 A vector is a collection of like elements

Understanding vectors

A vector is a collection of like elements (i.e., the elements all have the same mode). There are many ways to create vectors (see Table 9). The most common way of creating a vector is using the `c` function:

```
> #numeric
> x <- c(1/2, 2/2, 3/2, 4/2, 5/2)
> x
[1] 0.5 1.0 1.5 2.0 2.5
> #character
> y <- c("Hello", "What's your name?", "Your address?")
> y
[1] "Hello"             "What's your name?" "Your address?"
> #logical
> z <- c(T, T, F, T, F)
> z
[1]  TRUE  TRUE FALSE  TRUE FALSE
```

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

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

Logical vector operations

So what's the deal with logical vectors? Logical vectors are used for boolean operations. Boolean operations is a methodological workhorse of data analysis. For example, suppose you have a vector of female movie stars and a corresponding vector of their ages (as of January 16, 2004), and you want to select a subset of actors based on age criteria.

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

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

```
> #logical vector for stars with ages >=30
> ms.ages >= 30
[1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
> #logical vector for stars with ages <40
> ms.ages < 40
```

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

We also saw that we can compare logical vectors using *logical operators* (&, |, !). For more examples see Table 8. The expression movie.stars[thirtysomething] is an example of *indexing* using a logical vector.

Now, we can use the ! function to select the stars that are *not* “thirtysomething.” Study the following:

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

To summarize:

- logical vectors are created using boolean comparisons,
- boolean comparisons are conducted using relational and logical operators
- logical vectors are most commonly used for the following:
  - Indexing (subsetting) another data object
  - Determine the control of analytic tasks in the functions we will learn to program

Before moving on, make sure you understand the previous examples, then study the examples in Table 8. We will be using boolean operations again and again and again!

Table 8 Boolean operations using relational and logical operators

Operator	***Description***	***Examples in R***
Relational Operators		
<	less than	> position <- c("P1", "P2", "P3", "P4", "P5") > x <- c(1, 2, 3, 4, 5) > y <- c(5, 4, 3, 2, 1) > x < y [1] TRUE TRUE FALSE FALSE FALSE > position[x < y] [1] "P1" "P2"
>	greater than	> x < y [1] TRUE TRUE FALSE FALSE FALSE > position[x > y] [1] "P4" "P5"
<=	less than or equal to	> x <= y [1] TRUE TRUE TRUE FALSE FALSE > position[x <= y] [1] "P1" "P2" "P3"

Operator	***Description***	***Examples in R***

`>=`	greater than or equal to	`> x >= y`
		`[1] FALSE FALSE  TRUE  TRUE  TRUE`
		`> position[x >= y]`
		`[1] "P3" "P4" "P5"`

`==`	equal to	`> x == y`
		`[1] FALSE FALSE  TRUE FALSE FALSE`
		`> position[x == y]`
		`[1] "P3"`

`!=`	not equal to	`> x != y`
		`[1]  TRUE  TRUE FALSE  TRUE  TRUE`
		`> position[x != y]`
		`[1] "P1" "P2" "P4" "P5"`

Logical Operators		

`!`	NOT	`> position <- c("P1", "P2", "P3", "P4", "P5")`
		`> x <- c(1, 2, 3, 4, 5)`
		`> x > 2`
		`[1] FALSE FALSE  TRUE  TRUE  TRUE`
		`> !(x > 2)`
		`[1]  TRUE  TRUE FALSE FALSE FALSE`
		`> position[!(x > 2)]`
		`[1] "P1" "P2"`

`&`	element-wise AND	`> (x > 1) & (x < 5)`
		`[1] FALSE  TRUE  TRUE  TRUE FALSE`
		`> position[(x > 1) & (x < 5)]`
		`[1] "P2" "P3" "P4"`

`&&`	similar to & but only evaluates the first element of each logical vector and returns only either TRUE or FALSE	`> if(T && T) {print("Both TRUE")}`
		`[1] "Both TRUE"`
		`> if(T && F) {print("Both TRUE")}`
		`>`

`|`	element-wise OR	`> (x <= 1) | (x > 4)`
		`[1]  TRUE FALSE FALSE FALSE  TRUE`
		`> position[(x <= 1) | (x > 4)]`
		`[1] "P1" "P5"`

`||`	similar to | but only evaluates the first element of each logical vector and returns only either TRUE or FALSE	`> if(T || F) {print("Either TRUE")}`
		`[1] "Either TRUE"`
		`> if(F || F) {print("Either TRUE")}`
		`>`

`xor`	similar to | for comparing two vectors	`> xx <- x <= 1`
		`> yy <- x > 4`
		`> xor(xx, yy)`
		`[1]  TRUE FALSE FALSE FALSE  TRUE`
		`> xx | yy`
		`[1]  TRUE FALSE FALSE FALSE  TRUE`

Practice

Study the examples in Table 8 and spend a few minutes creating simple numerical 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.

Hint

“For the things we have to learn before we can do them, we learn by doing them.”

Aristotle

Creating vectors

Here is some “quick and dirty” R code to graphically display the sine and cosine function:

```
> x <- seq(1, 20, by=.001)
> sinx <- sin(x)
> cosx <- cos(x)
> sincosx <- cbind(sinx, cosx)
> matplot(x, sincosx, type="l", lwd=2, col=2:3)
```

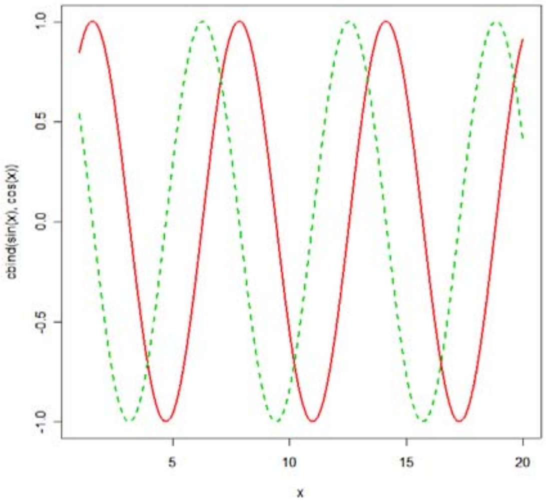


Figure 3 Graphical display of sine and cosine using R

Let's look at the same code but now highlight the vectors.

```
> x <- seq(1, 20, by=.001)
> sinx <- sin(x)
> cosx <- cos(x)
> sincosx <- cbind(sinx, cosx)
> matplot(x, sincosx, type="l", lwd=2, col=2:3)
```

In R, we will be creating, extracting, subsetting, combining, plotting, and operating on vectors frequently. Let's review these highlighted vectors in more detail:

```
> x <- seq(1, 20, by=.001)
> x
 [1] 1.000 1.001 1.002 1.003 1.004 1.005 1.006 1.007 1.008 1.009
     1.010
[12] 1.011 1.012 1.013 1.014 1.015 1.016 1.017 1.018 1.019 1.020
     1.021
...
```

```
[18991] 19.990 19.991 19.992 19.993 19.994 19.995 19.996 19.997
     19.998
[19000] 19.999 20.000
> sin(x)[1:50]
 [1] 0.8414710 0.8420109 0.8425499 0.8430881 0.8436255 0.8441620
 [7] 0.8446976 0.8452325 0.8457664 0.8462996 0.8468318 0.8473633
...
[18997]  9.113056e-01  9.117169e-01  9.121273e-01  9.125367e-01
[19001]  9.129453e-01
> cos(x)[1:50]
 [1] 0.5403023 0.5394606 0.5386183 0.5377755 0.5369321 0.5360882
 [7] 0.5352438 0.5343988 0.5335533 0.5327073 0.5318607 0.5310136
...
[18997]  4.117306e-01  4.108191e-01  4.099071e-01  4.089948e-01
[19001]  4.080821e-01
> 2:3
[1] 2 3
```

We created the vector `x` using the `seq` function (notice that the length of `x` is 19,001!). We created two new vectors by transforming `x` using the `sin` and `cos` functions (also of length = 19,001). We also used the `:` operator to create a vector to designate the colors in the `matplot` function. This is typical of the kind of efficient numerical analysis you can do in R. In fact, this example of R code could have been reduced to one line of code![1]

```
matplot(x<-seq(1,20,by=.001),cbind(sin(x),cos(x)),
     type="l",lwd=2,col=2:3)
```

The point of this example was to illustrate R's efficiency in creating and using vectors (also called *vectorized operations*). As you get more proficient in R you will appreciate the efficiency, power, and elegance of R's programming language. For now, review Table 9 which summarizes the most common methods of creating vectors.

Table 9 Common ways of creating vectors

Function	***Description***	***Examples in R***

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

`:`	generates integer sequence	`> 1:10`
		` [1]  1  2  3  4  5  6  7  8  9 10`
		`> 10:(-4)`
		` [1] 10  9  8  7  6  5  4  3  2  1  0 -1 -2 -3 -4`

`seq`	generates sequence of numbers	`> seq(1, 5, by=.5)`
		`[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0`
		`> seq(1, 5, length=3)`
		`[1] 1 3 5`
		`> zz <- c("a","b","c")`
		`> seq(along=zz)`
		`[1] 1 2 3`

[1] For the sake of clarity (and debugging), I recommend breaking down the steps.

Function	***Description***	***Examples in R***
sequence	concatenates sequences of numbers by giving upper bound of each sequence	> sequence(c(3,2)) [1] 1 2 3 1 2
rep	replicates argument	> rep("Tomas",3) [1] "Tomas" "Tomas" "Tomas" > rep(1:3,4) [1] 1 2 3 1 2 3 1 2 3 1 2 3 > rep(1:3,3:1) [1] 1 1 1 2 2 3
paste	pastes elements creating a character string	> paste(c("A","B","C"), 1:3) [1] "A 1" "B 2" "C 3" > paste(c("A","B","C"), 1:3, sep="") [1] "A1" "B2" "C3"
[*row.num*,] or [,*col.num*]	indexing a matrix returns a vector	> xx <- matrix(1:8,nrow=2,ncol=4) > xx [,1] [,2] [,3] [,4] [1,] 1 3 5 7 [2,] 2 4 6 8 > xx[2,] [1] 2 4 6 8 > xx[,3] [1] 5 6
as.vector	coerces data objects into a vector	> mx <- matrix(1:4, nrow=2, ncol=2) > mx [,1] [,2] [1,] 1 3 [2,] 2 4 > as.vector(mx) [1] 1 2 3 4
vector	creates vector of specified mode and length	> vector("character",5) [1] "" "" "" "" "" > vector("complex",5) [1] 0+0i 0+0i 0+0i 0+0i 0+0i > vector("numeric",5) [1] 0 0 0 0 0 > vector("logical",5) [1] FALSE FALSE FALSE FALSE FALSE > vector("list",2) [[1]] NULL [[2]] NULL
character	creates empty character vector	> character(5) [1] "" "" "" "" ""
complex	creates complex vector with 0+0i	> complex(5) [1] 0+0i 0+0i 0+0i 0+0i 0+0i
numeric	creates numeric vector with 0	> numeric(5) [1] 0 0 0 0 0

Function	Description	Examples in R
logical	creates character vector with FALSE	> logical(5) [1] FALSE FALSE FALSE FALSE FALSE
gl	generate factors by specifying the pattern of their levels	> ## First control, then treatment: > gl(2, 8, label = c("Male", "Female")) [1] Male Male Male Male Male Male [7] Male Male Female Female Female Female [13] Female Female Female Female Levels: Male Female > ## 20 alternating 1s and 2s > gl(2, 1, 20) [1] 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 Levels: 1 2 > ## alternating pairs of 1s and 2s > gl(2, 2, 20) [1] 1 1 2 2 1 1 2 2 1 1 2 2 1 1 2 2 1 1 2 2 Levels: 1 2

Practice

Study the examples in Table 9 and spend a few minutes creating simple vectors.

Hint

If you need help with a function remember enter ?*function_name* or help(*function_name*).

Naming vectors

The first way of naming the elements of a vector 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 assigning that vector to the numeric vector using the names function:

```
> x <- c(234, 148, 78, 54)
> x
[1] 234 148  78  54
> names(x) <- c("chol", "sbp", "dbp", "age")
> x
chol  sbp  dbp  age
 234  148   78   54
```

The names function, without a assignment, will return the character vector of element names, if they exist. This character vector can be used to name elements of other vectors.

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

Table 10 Naming vector elements

Function	***Description***	***Examples in R***

`c`	name vector elements at time that vector is created	`> z <- c(a=1, b=2, c=3, d=4)`
		`> z`
		`a b c d`
		`1 2 3 4`

`names`	name vector elements	`> x <- 1:5`
		`> x`
		`[1] 1 2 3 4 5`
		`> names(x) <- c("1st","2nd","3rd","4th","5th")`
		`> x`
		`1st 2nd 3rd 4th 5th`
		`  1   2   3   4   5`
		`> #without assignment operator returns vector of names`
		`> names(x)`
		`[1] "1st" "2nd" "3rd" "4th" "5th"`
		`> names(x) <- NULL`
		`> x`
		`[1] 1 2 3 4 5`

Practice

Study the examples in Table 10 and spend a few minutes creating and naming simple vectors.

Indexing vectors

Table 11 Common ways of indexing vectors

Description	***Examples in R***

Indexing by position	`> x`
	`chol  sbp  dbp  age`
	` 234  148   78   54`
	`> x[2] #positions to include`
	`sbp`
	`148`
	`> x[c(2, 3)]`
	`sbp dbp`
	`148  78`
	`> x[-c(1, 3, 4)] #positions to exclude`
	`sbp`
	`148`
	`> x[-c(1, 4)]`
	`sbp dbp`
	`148  78`
	
	`> #double brackets extract single element without name`
	`> x[[2]]`
	`[1] 148`
	`> x[[2:3]] #does not work`
	`Error: attempt to select more than one element`

Indexing by name	`> x["sbp"]`
	`sbp`
	`148`
	`> x[c("sbp", "dbp")]`
	`sbp dbp`
	`148  78`

Indexing using a logical vector	`> x < 100`
	` chol   sbp   dbp   age`
	`FALSE FALSE  TRUE  TRUE`
	`> x[x < 100]`
	`dbp age`
	` 78  54`
	`> (x < 150) & (x > 70)`
	` chol   sbp   dbp   age`
	`FALSE  TRUE  TRUE FALSE`
	`> bp <- (x < 150) & (x > 70)`
	`> x[bp]`
	`sbp dbp`
	`148  78`

Indexing the unique values	`> samp <- sample(1:5, 50, replace=T)`
	`> samp`
	` [1] 3 5 3 3 3 3 4 1 5 4 3 5 3 2 4 5 2 2 1 2 3 2`
	`[23] 2 3 2 1 5 1 4 3 3 4 3 3 2 4 5 5 5 1 3 2 1 3`
	`[45] 1 2 1 4 3 1`
	`> unique(samp)`
	`[1] 3 5 4 1 2`

Description	***Examples in R***

Indexing the duplicated values
(this is an example of indexing using a logical vector)
	`> duplicated(samp)`
	` [1] FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE`
	` [8] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE`
	`[15]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE`
	`[22]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE`
	`[29]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE`
	`[36]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE`
	`[43]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE`
	`[50]  TRUE`
	`> samp[duplicated(samp)]`
	` [1] 3 3 3 3 5 4 3 5 3 4 5 2 2 1 2 3 2 2 3 2 1 5`
	`[23] 1 4 3 3 4 3 3 2 4 5 5 5 1 3 2 1 3 1 2 1 4 3`
	`[45] 1`

Practice

Study the examples in Table 11 and spend a few minutes creating, naming, and indexing simple vectors.

Hint

Try using the `sample` function (with and without replacement) to create vectors of random values, for example try:

`> x <- sample(c('Heads', 'Tails'), 1000, replace=T)`).

Replacing vector elements (by indexing and assignment)

To replace vector elements we combine indexing and assignment. Any elements of a vector that can be indexed can also be replaced.

Table 12 Common ways of replacing vectors elements

Description	***Examples in R***

Replacing by position	`> y`
	`chol  sbp  dbp  age`
	` 234  148   78   54`
	`> y[2] <- 180`
	`> y`
	`chol  sbp  dbp  age`
	` 234  180   78   54`

Replacing by element name, if it exists	`> y["dbp"] <- 110`
	`> y`
	`chol  sbp  dbp  age`
	` 234  180  110   54`

Description	***Examples in R***

Replacing using a logical vector	`> x`
	`chol  sbp  dbp  age`
	` 234  148   78   54`
	`> bp <- (x < 150) & (x > 70)`
	`> bp`
	` chol   sbp   dbp   age`
	`FALSE  TRUE  TRUE FALSE`
	`> x[bp] <- c(180, 110)`
	`> x`
	`chol  sbp  dbp  age`
	` 234  180  110   54`

Operations on vectors

Operations on single vectors

Table 13 Simple operations on single vectors

Function	***Description***	***Examples in R***

sum	summation	`> xx <- c(5, 13, 1, 19, 10)`
		`> sum(xx)`
		`[1] 48`

cumsum	cumulative sum	`> xx <- c(5, 13, 1, 19, 10)`
		`> cumsum(xx)`
		`[1]  5 18 19 38 48`

diff	x[i+1]-x[i]	`> xx <- c(5, 13, 1, 19, 10)`
		`> diff(xx)`
		`[1]   8 -12  18  -9`

prod	product	`> xx <- c(5, 13, 1, 19, 10)`
		`> prod(xx)`
		`[1] 12350`

cumprod	cumulative product	`> xx <- c(5, 13, 1, 19, 10)`
		`> cumprod(xx)`
		`[1]     5    65    65  1235 12350`

mean	mean	`> xx <- c(5, 13, 1, 19, 10)`
		`> mean(xx)`
		`[1] 9.6`

median	median	`> xx <- c(5, 13, 1, 19, 10)`
		`> median(xx)`
		`[1] 10`

min	minimum	`> xx <- c(5, 13, 1, 19, 10)`
		`> min(xx)`
		`[1] 1`

max	maximum	`> xx <- c(5, 13, 1, 19, 10)`
		`> max(xx)`
		`[1] 19`

range	range	`> xx <- c(5, 13, 1, 19, 10)`
		`> range(xx)`
		`[1]  1 19`

Function	***Description***	***Examples in R***
rev	reverse order	> yy <- c(1, 2, 3, 4, 5) > rev(yy) [1] 5 4 3 2 1
order	order	> xx <- c(5, 13, 1, 19, 10) > order(xx) [1] 3 1 5 2 4
sort	sort	> xx <- c(5, 13, 1, 19, 10) > sort(xx) [1] 1 5 10 13 19 > xx[order(xx)] [1] 1 5 10 13 19
rank	rank	> xx <- c(5, 13, 1, 19, 10) > rank(xx) [1] 2 4 1 5 3
sample	random sample	> vv <- 1:5 > sample(vv, 10, replace=TRUE) [1] 2 3 1 4 3 4 5 4 5 1

quantile	percentile	> ss <- sample(1:100, 1000, replace=TRUE)
		> quantile(ss)
		  0% 25% 50% 75% 100%
		   1 24 50 75 100

var	variance	> ss <- sample(1:100, 1000, replace=TRUE) > var(ss) [1] 824.5992
sd	standard deviation	> ss <- sample(1:100, 1000, replace=TRUE) > sd(ss) [1] 28.71584

Operations on multiple vectors

Table 14 Simple operations on multiple vectors

Function	***Description***	***Examples in R***
c	concatenates vectors	> x <- 1:5 > y <- 6:10 > z <- 11:15 > c(x, y, z) [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 [15] 15
append	appends a vector to another vector (default is to append at the end of the first vector)	> x <- c(10, 9, 2, 1) > x [1] 10 9 2 1 > y <- 8:3 > y [1] 8 7 6 5 4 3 > append(x, y, after=2) [1] 10 9 8 7 6 5 4 3 2 1

Function	***Description***	***Examples in R***

`cbind`	column-bind vectors or matrices	`> xyz <- cbind(x, y, z)`
		`> xyz`
		`     x  y  z`
		`[1,] 1  6 11`
		`[2,] 2  7 12`
		`[3,] 3  8 13`
		`[4,] 4  9 14`
		`[5,] 5 10 15`

`rbind`	row-bind vectors or matrices	`> xyz2 <- rbind(x, y, z)`
		`> xyz2`
		`  [,1] [,2] [,3] [,4] [,5]`
		`x    1    2    3    4    5`
		`y    6    7    8    9   10`
		`z   11   12   13   14   15`

`table`	creates contingency table from any number of vectors	`> infert$education`
		`  [1] 0-5yrs  0-5yrs  0-5yrs  0-5yrs  6-11yrs`
		`...`
		`[241] 12+ yrs 12+ yrs 12+ yrs 12+ yrs 12+ yrs`
		`[246] 12+ yrs 12+ yrs 12+ yrs`
		`Levels: 0-5yrs 6-11yrs 12+ yrs`
		`> infert$case`
		`  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1`
		`...`
		`[221] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0`
		`[241] 0 0 0 0 0 0 0 0`
		`> infert$parity`
		`  [1] 6 1 6 4 3 4 1 2 1 2 2 4 1 3 2 2 5 1 3 1`
		`...`
		`[221] 2 2 2 1 2 2 1 1 2 2 1 1 3 3 1 1 1 1 6 2`
		`[241] 1 2 1 1 1 2 1 1`
		`> table(infert$educ, infert$par, infert$case)`
		`, ,  = 0`
		
		`          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`
		
		`, ,  = 1`
		
		`          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`

<table>
<tr><th>Function</th><th>Description</th><th>Examples in R</th></tr>
<tr><td>ftable</td><td>creates contingency table from any number of vectors</td><td>> ftable(infert$educ, infert$case, infert$par)
            1 2 3 4 5 6

0-5yrs 0   2 0 0 2 0 4
        1   1 0 0 1 0 2
6-11yrs 0 28 28 14 8 2 0
        1 14 14 7 4 1 0
12+ yrs 0 36 26 10 2 2 1
        1 18 13 5 1 1 1</td></tr>
<tr><td>outer</td><td>outer product</td><td>> 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</td></tr>
<tr><td>mapply</td><td>applies a function to the first elements of each argument, the second elements, the third elements, and so on. Arguments are recycled if necessary.</td><td>> mapply("*",1:5, 1:5)
[1] 1 4 9 16 25</td></tr>
<tr><td><
>
<=
>=
==
!=</td><td>Relational operators</td><td>See Table 8, p 25</td></tr>
<tr><td>!
&
&&
|
||
xor</td><td>Logical operators</td><td>See Table 8, p 25</td></tr>
</table>

Practice

Study the examples in Table 13 and Table 14 and spend a few minutes creating and operating on simple vectors.

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

Understanding matrices

A matrix is a 2-dimensional table of like elements. Contingency tables in epidemiology are represented in R as matrices or arrays. An array is the generalization of matrices to n-dimensions (this is equivalent to stratified tables). For now we will focus on 2-dimensional tables. Consider the following 2x2 table of crude data that is presented in baby Rothman (ref). In this randomized clinical trial (RCT), diabetic subjects were randomly assigned to received either tolbutamide or placebo. Because this was a prospective study we can calculate risks, odds, risk ratio, and odds ratio. We will do this using R as a calculator. Later we will learn to program customized functions to automate repetitive or frequently used tasks.

Table 15 Deaths among subjects who received tolbutamide and placebo in the Unversity Group Diabetes Program (1970)

	Tolbutamide	Placebo
Deaths	30	21
Survivors	174	184
Risks	?	?
Risk ratio	?	Reference
Odds	?	?
Odd ratio	?	Reference

```
> dat <- matrix(c(30, 174, 21, 184), 2, 2)
> dimnames(dat) <- list(c('Deaths', 'Survivors'), c('Tolbutamide',
     'Placebo'))
> dat
          Tolbutamide Placebo
Deaths             30      21
Survivors         174     184
> trt.tot <- apply(dat, 2, sum)
> trt.tot
Tolbutamide     Placebo
        204         205
> risks <- dat['Deaths',]/trt.tot
> risks
Tolbutamide     Placebo
  0.1470588   0.1024390
> risk.ratio <- risks/risks['Placebo']
> risk.ratio
Tolbutamide     Placebo
   1.435574    1.000000
> odds <- risks/(1-risks)
> odds
Tolbutamide     Placebo
  0.1724138   0.1141304
> odds.ratio <- odds/odds['Placebo']
> odds.ratio
Tolbutamide     Placebo
   1.510673    1.000000
> results <- rbind(risks, risk.ratio, odds, odds.ratio)
> #display everything
> dat
          Tolbutamide Placebo
Deaths             30      21
Survivors         174     184
> results
           Tolbutamide   Placebo
risks        0.1470588 0.1024390
risk.ratio   1.4355742 1.0000000
odds         0.1724138 0.1141304
odds.ratio   1.5106732 1.0000000
```

Here is the same analysis without displaying intermediate results:

```
> dat <- matrix(c(30, 174, 21, 184), 2, 2)
> dimnames(dat) <- list(c('Deaths', 'Survivors'), c('Tolbutamide',
     'Placebo'))
> trt.tot <- apply(dat, 2, sum)
> risks <- dat['Deaths',]/trt.tot
> risk.ratio <- risks/risks['Placebo']
> odds <- risks/(1-risks)
> odds.ratio <- odds/odds['Placebo']
> results <- rbind(risks, risk.ratio, odds, odds.ratio)
> #display everything
> dat
          Tolbutamide Placebo
Deaths             30      21
Survivors         174     184
> results
           Tolbutamide   Placebo
risks        0.1470588 0.1024390
risk.ratio   1.4355742 1.0000000
odds         0.1724138 0.1141304
odds.ratio   1.5106732 1.0000000
```

Next is the R code as it would appear in a text editor. The most efficient approach is to build up your analysis in a text editor and then execute using R's batch mode (more on this later, for now just cut and paste your code into R or Rweb).

```
dat <- matrix(c(30, 174, 21, 184), 2, 2)
dimnames(dat) <- list(c('Deaths', 'Survivors'), c('Tolbutamide',
     'Placebo'))
trt.tot <- apply(dat, 2, sum)
risks <- dat['Deaths',]/trt.tot
risk.ratio <- risks/risks['Placebo']
odds <- risks/(1-risks)
odds.ratio <- odds/odds['Placebo']
results <- rbind(risks, risk.ratio, odds, odds.ratio)
#display everything
dat
results
```

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

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

In the above line we used the `matrix` function to take a vector and convert it into a matrix with 2 rows and 2 columns. Notice the `matrix` function reads in the vector column-wise. To read the vector in row-wise we would add the `byrow=T` option (matrix(*vector*, *nrow*, *ncol*, byrow=T)). Try creating a matrix reading in a vector column-wise (default) and row-wise.

```
dimnames(dat) <- list(c('Deaths', 'Survivors'), c('Tolbutamide',
     'Placebo'))
```

In the above line we used the `dimnames` function to assign row and column names to the matrix `dat`. The row names and the column names are both character vectors, and these vectors are contained in a `list`.

```
trt.tot <- apply(dat, 2, sum)
```

In the above line we used the `apply` function to sum the columns. `apply` is a versatile function for applying any function to matrices or arrays.

```
risks <- dat['Deaths', ]/trt.tot
```

In the above line we calculated the risks of death for each treatment group. We got the numerator by indexing the `dat` matrix using the row name `'Deaths'`. The numerator is a vector containing the deaths for each group and the denominator is the total number of subjects in each group.

```
risk.ratio <- risks/risks['Placebo']
```

In the above line we calculated the risk ratios using the placebo group as the reference.

```
odds <- risks/(1-risks)
```

In the above line we calculated the odds using the vector of risks.

```
odds.ratio <- odds/odds['Placebo']
```

In the above line we calculated the odds ratios using the vector of odds.

```
results <- rbind(risks, risk.ratio, odds, odds.ratio)
```

In the above line we used the `rbind` function to row bind the result vectors into a matrix data object we named `results`.

```
#display everything
dat
results
```

In the above lines we displayed our 2x2 table called `dat` and our results matrix called `results`. Here they are again:

```
> dat
          Tolbutamide Placebo
Deaths             30      21
Survivors         174     184
> results
           Tolbutamide   Placebo
risks        0.1470588 0.1024390
risk.ratio   1.4355742 1.0000000
odds         0.1724138 0.1141304
odds.ratio   1.5106732 1.0000000
```

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

Creating matrices

Table 16 Common ways of creating matrices

Function	*Description*	*Examples in R*

cbind	column-bind vectors or matrices	`> x <- 1:3`
		`> y <- 3:1`
		`> z <- cbind(x, y)`
		`> z`
		`     x y`
		`[1,] 1 3`
		`[2,] 2 2`
		`[3,] 3 1`

Function	***Description***	***Examples in R***

`rbind`	row-bind vectors or matrices	`> z2 <- rbind(x, y)`
		`> z2`
		`  [,1] [,2] [,3]`
		`x    1    2    3`
		`y    3    2    1`

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

`dim`	assign dimensions to a data object	`> mtx2 <- 1:4`
		`> mtx2`
		`[1] 1 2 3 4`
		`> dim(mtx2) <- c(2, 2)`
		`> mtx2`
		`     [,1] [,2]`
		`[1,]    1    3`
		`[2,]    2    4`

`array`	generates matrix when array is 2-dimensional	`> mtx <- array(1:4, dim = c(2, 2))`
		`> mtx`
		`     [,1] [,2]`
		`[1,]    1    3`
		`[2,]    2    4`

`xtabs`	create a contingency table from cross-classifying factors, usually contained in a data frame, using a formula interface	`> xtabs(~education + case, data = infert)`
		`         case`
		`education 0  1`
		`  0-5yrs   8  4`
		`  6-11yrs 80 40`
		`  12+ yrs 77 39`

`ftable`	creates matrix with class ftable	`> ftable(infert$educ, infert$spont, infert$case)`
		`            0  1`
		
		`0-5yrs  0   6  3`
		`        1   1  0`
		`        2   1  1`
		`6-11yrs 0  56 15`
		`        1  17 16`
		`        2   7  9`
		`12+ yrs 0  51 10`
		`        1  22 15`
		`        2   4 14`

`as.matrix`	coerces object into a matrix	`> 1:3`
		`[1] 1 2 3`
		`> as.matrix(1:3)`
		`     [,1]`
		`[1,]    1`
		`[2,]    2`
		`[3,]    3`

Function	Description	Examples in R
outer	outer product of two vectors	> 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
x[*row*, ,] or x[,*col* ,] or x[, ,dep]	indexing an array can return a matrix	> x <- array(1:8, c(2, 2, 2)) > x[1, ,] [,1] [,2] [1,] 1 5 [2,] 3 7 > x[,1 ,] [,1] [,2] [1,] 1 5 [2,] 2 6 > x[, ,1] [,1] [,2] [1,] 1 3 [2,] 2 4

Naming matrices

Table 17 Common ways of naming matrices

Function	Examples in R
dimnames	> x <- matrix(c(1, 5, 3, 86), 2, 2) > x [,1] [,2] [1,] 1 3 [2,] 5 86 > > #example 1 > dimnames(x) <- list(Disease=c("Case","Control"), Exposure=c("Yes","No")) > x Exposure Disease Yes No Case 1 3 Control 5 86

Function	***Examples in R***

`names`	`> #example 2`
	`> y`
	`        Yes No`
	`Case      1  3`
	`Control   5 86`
	`>`
	`> #add variable names`
	`> names(dimnames(y)) <- c("Disease","Exposure")`
	`> y`
	`         Exposure`
	`Disease   Yes No`
	`  Case      1  3`
	`  Control   5 86`

Indexing matrices

Table 18 Common ways of indexing matrices

Description	***Examples in R***

Indexing by position	`> x <- matrix(1:16, 4, 4)`
	`> x`
	`     [,1] [,2] [,3] [,4]`
	`[1,]    1    5    9   13`
	`[2,]    2    6   10   14`
	`[3,]    3    7   11   15`
	`[4,]    4    8   12   16`
	`> x[2,]`
	`[1]  2  6 10 14`
	`> x[,3]`
	`[1]  9 10 11 12`
	`> x[2,3]`
	`[1] 10`
	`> x[2, , drop = F] #preserve matrix structure`
	`     [,1] [,2] [,3] [,4]`
	`[1,]    2    6   10   14`

Description	***Examples in R***

Indexing by name	`> y <- matrix(c(34, 67, 23, 89), 2, 2)`
	`> dimnames(y) <- list(c('Exposed', 'Unexposed'),`
	`      c('Case','Control'))`
	`> y`
	`          Case Control`
	`Exposed     34      23`
	`Unexposed   67      89`
	`> y['Exposed', ]`
	`   Case Control`
	`     34      23`
	`> y['Unexposed', ]`
	`   Case Control`
	`     67      89`
	`> y[,'Case']`
	`  Exposed Unexposed`
	`       34        67`
	`> y[,c('Case', 'Control')]`
	`          Case Control`
	`Exposed     34      23`
	`Unexposed   67      89`

Indexing using a logical vector	`> x`
	`     [,1] [,2] [,3] [,4]`
	`[1,]    1    5    9   13`
	`[2,]    2    6   10   14`
	`[3,]    3    7   11   15`
	`[4,]    4    8   12   16`
	`> z <- x[,1] < 3`
	`> z`
	`[1]  TRUE  TRUE FALSE FALSE`
	`> x[z, ]`
	`     [,1] [,2] [,3] [,4]`
	`[1,]    1    5    9   13`
	`[2,]    2    6   10   14`

Replacing matrix elements

Table 19 Common ways of replacing matrix elements

Description	***Examples in R***

Replacing by position	`> x[3, ] <- c(55, 65, 75, 85)`
	`> x`
	`     [,1] [,2] [,3] [,4]`
	`[1,]    1    5    9   13`
	`[2,]    2    6   10   14`
	`[3,]   55   65   75   85`
	`[4,]    4    8   12   16`
	`> x[c(2, 3), c(2, 3)] <- matrix(99, 2, 2)`
	`> x`
	`     [,1] [,2] [,3] [,4]`
	`[1,]    1    5    9   13`
	`[2,]    2   99   99   14`
	`[3,]   55   99   99   85`
	`[4,]    4    8   12   16`

Description	***Examples in R***

Replacing by element name, if it exists	`> y[, 'Case'] <- NA #insert column of missing values`
	`> y`
	`          Case Control`
	`Exposed     NA      23`
	`Unexposed   NA      89`

Replacing using a logical vector	`> x`
	`     [,1] [,2] [,3] [,4]`
	`[1,]    1    5    9   13`
	`[2,]    2   99   99   14`
	`[3,]   55   99   99   85`
	`[4,]    4    8   12   16`
	`> x==99`
	`      [,1]  [,2]  [,3]  [,4]`
	`[1,] FALSE FALSE FALSE FALSE`
	`[2,] FALSE  TRUE  TRUE FALSE`
	`[3,] FALSE  TRUE  TRUE FALSE`
	`[4,] FALSE FALSE FALSE FALSE`
	`> x[x==99] <- 0`
	`> x`
	`     [,1] [,2] [,3] [,4]`
	`[1,]    1    5    9   13`
	`[2,]    2    0    0   14`
	`[3,]   55    0    0   85`
	`[4,]    4    8   12   16`

Operations on matrices

Table 20 Common operations on matrices

Function	***Description***	***Examples in R***

`t`	transpose matrix	`> x <- matrix(1:4,2,2)`
		`> x`
		`     [,1] [,2]`
		`[1,]    1    3`
		`[2,]    2    4`
		`> t(x)`
		`     [,1] [,2]`
		`[1,]    1    2`
		`[2,]    3    4`

Function	***Description***	***Examples in R***
apply	apply a function to the margins of a matrix	> y [,1] [,2] [1,] 1 3 [2,] 2 4 > apply(X = y, MARGIN = 2, FUN = sum) [1] 3 7 > apply(y, 1, FUN=sum) [1] 4 6 > apply(y, 1, mean) [1] 2 3 > apply(y, 2, cumprod) [,1] [,2] [1,] 1 3 [2,] 2 12
tapply	apply a function to each cell of a ragged array	> z <- rep(1:4,1:4) > z [1] 1 2 2 3 3 3 4 4 4 4 > tapply(X = z, INDEX = z, FUN = sum) 1 2 3 4 1 4 9 16 > tapply(z, z, cumsum) $"1" [1] 1 $"2" [1] 2 4 $"3" [1] 3 6 9 $"4" [1] 4 8 12 16
sweep	Return an array obtained from an input array by sweeping out a summary statistic	> y [,1] [,2] [1,] 1 3 [2,] 2 4 > z <- apply(x, 1, mean) > z [1] 2 3 > sweep(y, MARGIN=1, STATS = z, FUN="-") [,1] [,2] [1,] -1 1 [2,] -1 1

Function	***Description***	***Examples in R***
margin.table	For a contingency table in array form, compute the sum of table entries for a given index This 'margin.table' function is really just the 'apply' function using 'sum'.	> y [,1] [,2] [1,] 1 3 [2,] 2 4 > margin.table(y) [1] 10 > margin.table(y, 1) [1] 4 6 > apply(y, 1, sum) [1] 4 6 > margin.table(y, 2) [1] 3 7 > apply(y, 2, sum)
prop.table	Short cut that uses the 'sweep' and 'apply' functions to get margin and joint distributions	> y [,1] [,2] [1,] 1 3 [2,] 2 4 > prop.table(y) [,1] [,2] [1,] 0.1 0.3 [2,] 0.2 0.4 > y/sum(y) [,1] [,2] [1,] 0.1 0.3 [2,] 0.2 0.4 > prop.table(y, 1) [,1] [,2] [1,] 0.2500000 0.7500000 [2,] 0.3333333 0.6666667 > sweep(y, 1, apply(y, 1, sum), "/") [,1] [,2] [1,] 0.2500000 0.7500000 [2,] 0.3333333 0.6666667 > prop.table(y, 2) [,1] [,2] [1,] 0.3333333 0.4285714 [2,] 0.6666667 0.5714286 > sweep(y, 2, apply(y, 2, sum), "/") [,1] [,2] [1,] 0.3333333 0.4285714 [2,] 0.6666667 0.5714286

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

While a matrix is a 2-dimensional table of like elements, an array is the generalization of matrices to n-dimensions. Stratified contingency tables in epidemiology are represented as array data objects in R. Table 21 is an example of a 3-dimensional array. The counts of primary and secondary syphilis in the United States in 1989 are stratified by sex, race and age. The core data necessary to calculate the margin totals are highlighted in gray. In R, arrays are most often produced by directed with the `array` functions or applying the `table` function to variables of a data frame. For example, `table(syphilis.df$sex, syphilis.df$race,` `syphilis.df$age)` would produce a sex, race, and age stratified array from the data frame `syphilis.df`.

Understanding arrays

Table 21 Primary and secondary syphilis morbidity by age, race, and sex, United State, 1989

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

Source: CDC Summary of Notifiable Diseases, United States, 1989, MMWR 1989;38(54)

In contrast to the `table` function, you can use the `array` function to shape a numeric vector into a numeric array. The following R code creates the 3-dimensional array displayed in Table 21.

```
sdat <- c(2, 14, 31, 165, 7, 11,88, 253, 1412, 2257, 210, 158, 407,
     475, 4059, 4503, 654, 307, 550, 433, 4121, 3590, 633, 283, 564,
     316, 4453, 2628, 520, 167, 654, 243, 3858, 1505, 492, 149, 323,
     55, 1619, 392, 202, 40, 216, 24, 823, 92, 108, 15)
sdat <- array(sdat,dim = c(2, 3, 8))
dimnames(sdat) <- list(Sex = c('Male', 'Female'), Race = c('White',
     'Black', 'Other'), Age = c('<=14', '15-19', '20-24', '25-29',
     '30-34', '35-44', '45-54', '>=55'))
sdat
```

Now let's run this code in R or Rweb:

```
> sdat <- c(2, 14, 31, 165, 7, 11,88, 253, 1412, 2257, 210, 158, 407,
     475, 4059, 4503, 654, 307, 550, 433, 4121, 3590, 633, 283, 564,
     316, 4453, 2628, 520, 167, 654, 243, 3858, 1505, 492, 149, 323,
     55, 1619, 392, 202, 40, 216, 24, 823, 92, 108, 15)
> sdat <- array(sdat,dim = c(2, 3, 8))
> dimnames(sdat) <- list(Sex = c('Male', 'Female'), Race = c('White',
     'Black', 'Other'), Age = c('<=14', '15-19', '20-24', '25-29',
     '30-34', '35-44', '45-54', '>=55'))
> sdat
, , Age = <=14

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

, , Age = 15-19

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

, , Age = 20-24

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

, , Age = 25-29

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

, , Age = 30-34

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

```
, , Age = 35-44

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

, , Age = 45-54

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

, , Age = >=55

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

Let now explore the structure of this array data object using the `str` function:

```
> str(sdat)
 num [1:2, 1:3, 1:8]   2  14  31 165   7 ...
 - attr(*, "dimnames")=List of 3
  ..$ Sex : chr [1:2] "Male" "Female"
  ..$ Race: chr [1:3] "White" "Black" "Other"
  ..$ Age : chr [1:8] "<=14" "15-19" "20-24" "25-29" ...
```

To extract the variable values use the `dimnames` function.

```
> dnames <- dimnames(sdat)
> dnames
$Sex
[1] "Male"   "Female"

$Race
[1] "White" "Black" "Other"

$Age
[1] "<=14"  "15-19" "20-24" "25-29" "30-34" "35-44" "45-54" ">=55"

> dnames$Age
[1] "<=14"  "15-19" "20-24" "25-29" "30-34" "35-44" "45-54" ">=55"
```

To extract the variable names use the `names` function applied to the dimnames object.

```
> names(dnames)
[1] "Sex"  "Race" "Age"

> names(dimnames(sdat)) #also works
[1] "Sex"  "Race" "Age"
```

To extract to numeric vector that specifies the dimensions use the `dim` function.

```
> dim(sdat)
[1] 2 3 8
```

The attributes function applies to the sdat array is equivalent to list(dim = dim(sdat), dimnames = dimnames(sdat)):

```
> attributes(sdat)
$dim
[1] 2 3 8

$dimnames
$dimnames$Sex
[1] "Male"   "Female"

$dimnames$Race
[1] "White" "Black" "Other"

$dimnames$Age
[1] "<=14"  "15-19" "20-24" "25-29" "30-34" "35-44" "45-54" ">=55"
```

Arrays are convenient for analyzing multi-dimensional contingency tables, however, for display purposes, use the ftable function to convert an array into a *flat* (2-dimensional) contingency table for displaying data in a compact, convenient form.

```
> ftable(sdat)
             Age <=14 15-19 20-24 25-29 30-34 35-44 45-54 >=55
Sex    Race
Male   White        2    88   407   550   564   654   323  216
       Black       31  1412  4059  4121  4453  3858  1619  823
       Other        7   210   654   633   520   492   202  108
Female White       14   253   475   433   316   243    55   24
       Black      165  2257  4503  3590  2628  1505   392   92
       Other       11   158   307   283   167   149    40   15
```

To change the order of displaying variables in arrays or frequency tables, use the aperm function. aperm(sdat, perm = c(3, 1, 2)) means take the array sdat and move dimension 3 into the *first* position, move dimension 1 into the *second* position, and move dimension 2 into the *third* position. Study the example that follows.

```
> sdat2 <- aperm(sdat, perm = c(3, 1, 2))
> sdat2
, , Race = White

       Sex
Age     Male Female
  <=14     2     14
  15-19   88    253
  20-24  407    475
  25-29  550    433
  30-34  564    316
  35-44  654    243
  45-54  323     55
  >=55   216     24

, , Race = Black

       Sex
Age     Male Female
```

```
  <=14    31    165
  15-19 1412   2257
  20-24 4059   4503
  25-29 4121   3590
  30-34 4453   2628
  35-44 3858   1505
  45-54 1619    392
  >=55   823     92

, , Race = Other

       Sex
Age     Male Female
  <=14     7     11
  15-19  210    158
  20-24  654    307
  25-29  633    283
  30-34  520    167
  35-44  492    149
  45-54  202     40
  >=55   108     15

> ftable(sdat2) #looks like Table 21 on page 48
             Race White Black Other
Age   Sex
<=14  Male             2    31     7
      Female          14   165    11
15-19 Male            88  1412   210
      Female         253  2257   158
20-24 Male           407  4059   654
      Female         475  4503   307
25-29 Male           550  4121   633
      Female         433  3590   283
30-34 Male           564  4453   520
      Female         316  2628   167
35-44 Male           654  3858   492
      Female         243  1505   149
45-54 Male           323  1619   202
      Female          55   392    40
>=55  Male           216   823   108
      Female          24    92    15
```

Now study the syntax used in the `array` (or `table`) function compared to the `ftable` function. In the `array` (or `table`) function, the first two arguments determines the adjacent two dimensions that are displayed “flat.” In contrast, in the `ftable` function, the last two arguments determines the adjacent two dimensions that are displayed “flat.”

```
> AGE <- sample(c("Old", "Young"), 500, replace = T)
> SEX <- sample(c("Male", "Female"), 500, replace = T)
> RACE <- sample(c("White", "Latino", "Black", "Asian", "Other"),
     500, replace = T)
> table(AGE, RACE, SEX)
, , SEX = Female
```

```
       RACE
AGE     Asian Black Latino Other White
  Old   34    25    24     25    26
  Young 25    25    17     29    20

, , SEX = Male

       RACE
AGE     Asian Black Latino Other White
  Old   16    37    23     28    20
  Young 28    28    26     25    19

> ftable(SEX, AGE, RACE)
             RACE Asian Black Latino Other White
SEX    AGE
Female Old           34    25     24    25    26
       Young         25    25     17    29    20
Male   Old           16    37     23    28    20
       Young         28    28     26    25    19
```

Creating arrays

Table 22 Common ways of creating arrays

Function	***Description***	***Examples in R***

`array`	generates matrix when array is 2-dimensional	`> aa <- array(1:4, dim = c(2, 2, 2))`
		`> aa`
		`, , 1`
		
		`     [,1] [,2]`
		`[1,]    1    3`
		`[2,]    2    4`
		
		`, , 2`
		
		`     [,1] [,2]`
		`[1,]    1    3`
		`[2,]    2    4`

Function	***Description***	***Examples in R***

`table`	creates n-dimensional contingency table from n vectors	`> data(infert)`
		`> table(infert$educ, infert$spont, infert$case)`
		`, ,  = 0`
		
		`          0  1  2`
		`  0-5yrs   6  1  1`
		`  6-11yrs 56 17  7`
		`  12+ yrs 51 22  4`
		
		`, ,  = 1`
		
		`          0  1  2`
		`  0-5yrs   3  0  1`
		`  6-11yrs 15 16  9`
		`  12+ yrs 10 15 14`

`as.table`	creates n-dimensional contingency table from n-dimensional ftable	`> ft <- ftable(infert$ed, infert$sp, infert$ca)`
		`> ft`
		`            0  1`
		
		`0-5yrs  0   6  3`
		`        1   1  0`
		`        2   1  1`
		`6-11yrs 0  56 15`
		`        1  17 16`
		`        2   7  9`
		`12+ yrs 0  51 10`
		`        1  22 15`
		`        2   4 14`
		`> as.table(ft)`
		`, ,  = 0`
		
		`          0  1  2`
		`  0-5yrs   6  1  1`
		`  6-11yrs 56 17  7`
		`  12+ yrs 51 22  4`
		
		`, ,  = 1`
		
		`          0  1  2`
		`  0-5yrs   3  0  1`
		`  6-11yrs 15 16  9`
		`  12+ yrs 10 15 14`

Function	***Description***	***Examples in R***
dim	assign dimensions to a data object	> x <- 1:8 > x [1] 1 2 3 4 5 6 7 8 > dim(x) <- c(2, 2, 2) > x , , 1 [,1] [,2] [1,] 1 3 [2,] 2 4 , , 2 [,1] [,2] [1,] 5 7 [2,] 6 8

Naming arrays

Table 23 Common ways of naming arrays

<table>
<tr><th><i>Function</i></th><th><i>Examples in R</i></th></tr>
<tr><td>dimnames</td><td>

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

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

, , 2

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

> dimnames(x) <- list(Exposed = c('Yes', 'No'), Disease = c('Yes',
      'No'), Confounder = c('Yes', 'No'))
> x
, , Confounder = Yes

       Disease
Exposed Yes No
    Yes   1  3
    No    2  4

, , Confounder = No

       Disease
Exposed Yes No
    Yes   5  7
    No    6  8

> dimnames(x)  #dimnames without an assignment
$Exposed
[1] "Yes" "No"

$Disease
[1] "Yes" "No"

$Confounder
[1] "Yes" "No"
```

</td></tr>
</table>

Function	***Examples in R***
names	> x <- array(1:8, c(2, 2, 2)) > dimnames(x) <- list(c('Yes', 'No'), c('Yes', 'No'), c('Yes', 'No')) > x , , Yes Yes No Yes 1 3 No 2 4 , , No Yes No Yes 5 7 No 6 8 > names(dimnames(x)) <- c("Exposure", "Disease", "Confounder") > x , , Confounder = Yes Disease Exposure Yes No Yes 1 3 No 2 4 , , Confounder = No Disease Exposure Yes No Yes 5 7 No 6 8 > names(dimnames(x)) #without an assignment [1] "Exposed" "Disease" "Confounder"

Indexing arrays

Table 24 Common ways of indexing arrays

<table>
<tr><th><em>Description</em></th><th><em>Examples in R</em></th></tr>
<tr><td>Indexing by position</td><td><pre>
> x
, , Confounder = Yes

        Disease
Exposure Yes No
     Yes   1 3
     No   2 4

, , Confounder = No

        Disease
Exposure Yes No
     Yes   5 7
     No   6 8

> x[1, ,]
       Confounder
Disease Yes No
    Yes   1 5
    No   3 7
> x[,1 ,]
        Confounder
Exposure Yes No
     Yes   1 5
     No   2 6
> x[, ,1]
        Disease
Exposure Yes No
     Yes   1 3
     No   2 4
> x[, ,1 ,drop = F]
, , Confounder = Yes

        Disease
Exposure Yes No
     Yes   1 3
     No   2 4
</pre></td></tr>
</table>

<table>
<tr><th>Description</th><th>Examples in R</th></tr>
<tr><td>Indexing by name</td><td>

```
> x["Yes", , ]
       Confounder
Disease Yes No
    Yes   1  5
    No    3  7
> x["Yes", , ,drop = F]
, , Confounder = Yes

        Disease
Exposure Yes No
     Yes   1  3

, , Confounder = No

        Disease
Exposure Yes No
     Yes   5  7
```

</td></tr>
<tr><td>Indexing using a logical vector</td><td>

```
> x
, , Confounder = Yes

        Disease
Exposure Yes No
     Yes   1  3
     No    2  4

, , Confounder = No

        Disease
Exposure Yes No
     Yes   5  7
     No    6  8

> x[, 1, 1]
Yes  No
  1   2
> x[, 1, 1] < 2
  Yes    No
 TRUE FALSE
> x[x[, 1, 1] < 2, , ]
       Confounder
Disease Yes No
    Yes   1  5
    No    3  7
```

</td></tr>
</table>

Replacing array elements

Table 25 Common ways of replacing array elements

Description	***Examples in R***

Replacing by position	> x
	, , Confounder = Yes
	
	        Disease
	Exposure Yes No
	     Yes 1 3
	     No 2 4
	
	, , Confounder = No
	
	        Disease
	Exposure Yes No
	     Yes 5 7
	     No 6 8
	
	> x[1, ,] <- NA
	> x
	, , Confounder = Yes
	
	        Disease
	Exposure Yes No
	     Yes NA NA
	     No 2 4
	
	, , Confounder = No
	
	        Disease
	Exposure Yes No
	     Yes NA NA
	     No 6 8

<table>
<tr><th><strong><em>Description</em></strong></th><th><strong><em>Examples in R</em></strong></th></tr>
<tr><td>Replacing by element name</td><td>

```
> x
, , Confounder = Yes

        Disease
Exposure Yes No
     Yes   1  3
     No    2  4

, , Confounder = No

        Disease
Exposure Yes No
     Yes   5  7
     No    6  8

> x['Yes',,]
       Confounder
Disease Yes No
    Yes   1  5
    No    3  7
> x['Yes',,] <- 3*x['Yes',,]
> x
, , Confounder = Yes

        Disease
Exposure Yes No
     Yes   3  9
     No    2  4

, , Confounder = No

        Disease
Exposure Yes No
     Yes  15 21
     No    6  8
```

</td></tr>
</table>

<table>
<tr><th><strong><em>Description</em></strong></th><th><strong><em>Examples in R</em></strong></th></tr>
<tr><td>Replacing using a logical vector</td><td>

```
> x
, , Confounder = Yes

        Disease
Exposure Yes No
     Yes   1  3
     No    2  4

, , Confounder = No

        Disease
Exposure Yes No
     Yes   5  7
     No    6  8

> x >= 7
, , Confounder = Yes

        Disease
Exposure   Yes    No
     Yes FALSE FALSE
     No  FALSE FALSE

, , Confounder = No

        Disease
Exposure   Yes   No
     Yes FALSE TRUE
     No  FALSE TRUE

> x[x >= 7] <- 99
> x
, , Confounder = Yes

        Disease
Exposure Yes No
     Yes   1  3
     No    2  4

, , Confounder = No

        Disease
Exposure Yes No
     Yes   5 99
     No    6 99
```

</td></tr>
</table>

Operations on arrays

Table 26 Common operations on arrays

Function	***Description***	***Examples in R***

aperm	Transpose an array by permuting its dimensions and optionally resizing it.	> x
		, , Confounder = Yes
		
		        Disease
		Exposure Yes No
		     Yes 1 3
		     No 2 4
		
		, , Confounder = No
		
		        Disease
		Exposure Yes No
		     Yes 5 7
		     No 6 8
		
		> aperm(x, c(3, 2, 1))
		, , Exposure = Yes
		
		          Disease
		Confounder Yes No
		       Yes 1 3
		       No 5 7
		
		, , Exposure = No
		
		          Disease
		Confounder Yes No
		       Yes 2 4
		       No 6 8

<table>
<tr><th><b><i>Function</i></b></th><th><b><i>Description</i></b></th><th><b><i>Examples in R</i></b></th></tr>
<tr><td>apply</td><td>apply a function to the margins of an array</td><td><pre>
> x
, , Confounder = Yes

        Disease
Exposure Yes No
     Yes   1 3
     No   2 4

, , Confounder = No

        Disease
Exposure Yes No
     Yes   5 7
     No   6 8

> apply(x, 1, sum)
Yes No
 16 20
> apply(x, 2, sum)
Yes No
 14 22
> apply(x, c(1, 2), sum)
        Disease
Exposure Yes No
     Yes   6 10
     No   8 12
> apply(x, c(2, 3), sum)
       Confounder
Disease Yes No
    Yes   3 11
    No   7 15
</pre></td></tr>
</table>

<table>
<tr><th>Function</th><th>Description</th><th>Examples in R</th></tr>
<tr><td>sweep</td><td>Return an array obtained from an input array by sweeping out a summary statistic</td><td><pre>
> x
, , Confounder = Yes

        Disease
Exposure Yes No
     Yes   1 3
     No   2 4

, , Confounder = No

        Disease
Exposure Yes No
     Yes   5 7
     No   6 8

> sweep(x, c(2, 3), apply(x, c(2, 3), sum), "/")
, , Confounder = Yes

        Disease
Exposure   Yes   No
     Yes 0.3333333 0.4285714
     No 0.6666667 0.5714286

, , Confounder = No

        Disease
Exposure   Yes   No
     Yes 0.4545455 0.4666667
     No 0.5454545 0.5333333
</pre></td></tr>
</table>

<table>
<tr><th>Function</th><th>Description</th><th>Examples in R</th></tr>
<tr><td>margin.table</td><td>For a contingency table in array form, compute the sum of table entries for a given index</td><td>

```
> x
, , Confounder = Yes

        Disease
Exposure Yes No
     Yes   1  3
     No    2  4

, , Confounder = No

        Disease
Exposure Yes No
     Yes   5  7
     No    6  8

> margin.table(x, c(1, 2))
        Disease
Exposure Yes No
     Yes   6 10
     No    8 12

> apply(x, c(1, 2), sum)
        Disease
Exposure Yes No
     Yes   6 10
     No    8 12
```

</td></tr>
</table>

Function	Description	Examples in R

prop.table		> x
		, , Confounder = Yes
		
		        Disease
		Exposure Yes No
		     Yes 1 3
		     No 2 4
		
		, , Confounder = No
		
		        Disease
		Exposure Yes No
		     Yes 5 7
		     No 6 8
		
		> prop.table(x, c(2, 3))
		, , Confounder = Yes
		
		        Disease
		Exposure Yes No
		     Yes 0.3333333 0.4285714
		     No 0.6666667 0.5714286
		
		, , Confounder = No
		
		        Disease
		Exposure Yes No
		     Yes 0.4545455 0.4666667
		     No 0.5454545 0.5333333
		
		> sweep(x, c(2, 3), apply(x, c(2, 3), sum), "/")
		, , Confounder = Yes
		
		        Disease
		Exposure Yes No
		     Yes 0.3333333 0.4285714
		     No 0.6666667 0.5714286
		
		, , Confounder = No
		
		        Disease
		Exposure Yes No
		     Yes 0.4545455 0.4666667
		     No 0.5454545 0.5333333

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

Understanding lists

Think of list objects 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.

Creating lists

Table 27 Common ways of creating lists

Function	***Description***	***Examples in R***
list	creates list object	> x <- 1:3 > y <- matrix(c("a", "c", "b", "d"), 2, 2) > z <- c("Pedro", "Paulo", "Maria") > mm <- list(x, y, z) > mm [[1]] [1] 1 2 3 [[2]] [,1] [,2] [1,] "a" "b" [2,] "c" "d" [[3]] [1] "Pedro" "Paulo" "Maria"
as.list	coercion into list object	> list(1:2) #compare to as.list [[1]] [1] 1 2 > as.list(1:2) [[1]] [1] 1 [[2]] [1] 2
vector	creates empty list of length *n*	> vector("list", 2) [[1]] NULL [[2]] NULL
data.frame	data frames are of mode list	> x <- data.frame(id = 1:3, sex = c("M", "F", "T")) > x id sex 1 1 M 2 2 F 3 3 T > mode(x) [1] "list"
as.data.frame	coerces data object into a data frame	> x <- matrix(1:6, 2, 3) > y <- as.data.frame(x) > y V1 V2 V3 1 1 3 5 2 2 4 6

Function	Description	Examples in R

read.table
read.csv
read.csv2
read.delim
read.delim2
read.fmf
	reads in ASCII text data file into data frame object	> wcgs <- read.csv(".../data/wcgs.csv", header=T)
		> str(wcgs)
		`data.frame': 3154 obs. of 14 variables:
		 $ id : int 2001 2002 2003 2004 2005 ...
		 $ age0 : int 49 42 42 41 59 44 44 40 ...
		 $ height0: int 73 70 69 68 70 72 72 71 72 ...
		 $ weight0: int 150 160 160 152 150 204 164 ...
		 $ sbp0 : int 110 154 110 124 144 150 130 ...
		...

Naming lists

Table 28 Common ways of naming lists

Function	Examples in R
names	> z <- list(1, "c", 1:3) > z [[1]] [1] 1 [[2]] [1] "c" [[3]] [1] 1 2 3 > names(z) <- c('bin1', 'bin2', 'bin3') > z $bin1 [1] 1 $bin2 [1] "c" $bin3 [1] 1 2 3 > z <- list(Bin1 = 1, Bin2 = "c", Bin3 = 1:3) #name at creation of list > z $Bin1 [1] 1 $Bin2 [1] "c" $Bin3 [1] 1 2 3 > names(z) #names without an assignment [1] "Bin1" "Bin2" "Bin3"

Indexing lists

Table 29 Common ways of indexing lists

Description	Examples in R

Indexing by position	`> z`
	`$Bin1`
	`[1] 1`
	
	`$Bin2`
	`[1] "c"`
	
	`$Bin3`
	`[1] 1 2 3`
	
	`> z[3] #index bin`
	`$Bin3`
	`[1] 1 2 3`
	
	`> z[[3]] #index bin contents`
	`[1] 1 2 3`

Indexing by name	`> z$Bin3  #indexing by name retrieves bin contents`
	`[1] 1 2 3`

Indexing using a logical vector	`> zz <- c(T, T, F)`
	`> zz`
	`[1]  TRUE  TRUE FALSE`
	`> z[zz]`
	`$Bin1`
	`[1] 1`
	
	`$Bin2`
	`[1] "c"`

Replacing lists components

Table 30 Common ways of replacing list components

Description	***Examples in R***

Replacing by position	`> z <- list(bin1 = 1:3, bin2 = "c")`
	`> z`
	`$bin1`
	`[1] 1 2 3`
	
	`$bin2`
	`[1] "c"`
	
	`> z[1] <- list(replacement1=c(2, 3, 4)) #replace w/ vector`
	`> z`
	`$bin1`
	`[1] 2 3 4`
	
	`$bin2`
	`[1] "c"`
	
	`> z[[1]] <- list(replacement1=c(2, 3, 4)) #replace w/ list`
	`> z`
	`$bin1`
	`$bin1$replacement1`
	`[1] 2 3 4`
	
	`$bin2`
	`[1] "c"`

Replacing by name	`> z <- list(bin1 = 1:3, bin2 = "c")`
	`> z`
	`$bin1`
	`[1] 1 2 3`
	
	`$bin2`
	`[1] "c"`
	
	`> z$bin1 <- list(replacement1=c(2, 3, 4)) #replace w/ list`
	`> z`
	`$bin1`
	`$bin1$replacement1`
	`[1] 2 3 4`
	
	`$bin2`
	`[1] "c"`

Description	***Examples in R***
Replacing using a logical vector	> z <- list(bin1 = 1:3, bin2 = "c") > z $bin1 [1] 1 2 3 $bin2 [1] "c" > chars <- lapply(z, is.character) > chars $bin1 [1] FALSE $bin2 [1] TRUE > z[chars] Error: invalid subscript type > unlist(chars) #unlist to create logical vector bin1 bin2 FALSE TRUE > z[unlist(chars)] $bin2 [1] "c"

Operations on lists

Table 31 Common operations on lists

Function	***Description***	***Examples in R***
lapply	applies a function to a list	> x <- list(1:5, 5:10) > x [[1]] [1] 1 2 3 4 5 [[2]] [1] 5 6 7 8 9 10 > lapply(x, mean) #applies function, returns list [[1]] [1] 3 [[2]] [1] 7.5
sapply	applies a function to a list and simplifies	> sapply(x, mean) #applies function, returns vector [1] 3.0 7.5

Function	Description	Examples in R
mapply	Apply a function to the first elements of each argument, the second elements, the third elements, and so on. Arguments are recycled if necessary.	> y <- list(1:3, 1:4) > mapply(outer, y, y) [[1]] [,1] [,2] [,3] [1,] 1 2 3 [2,] 2 4 6 [3,] 3 6 9 [[2]] [,1] [,2] [,3] [,4] [1,] 1 2 3 4 [2,] 2 4 6 8 [3,] 3 6 9 12 [4,] 4 8 12 16
attach detach	Attach or detach list or data frame to search path	> z $bin1 [1] 1 2 3 $bin2 [1] "c" > search() [1] ".GlobalEnv" "package:methods" ... [9] "package:base" > attach(z) > search() [1] ".GlobalEnv" "z" ... [9] "Autoloads" "package:base" > bin1 [1] 1 2 3 > detach(z) > search() [1] ".GlobalEnv" "package:methods" ... [9] "package:base"

2.6 A data frame is a list in the form of 2-dimensional data table

Understanding data frames and factors

Epidemiologists are familiar with data sets that come in the form of tables 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 usually either a factor or numeric vector, although it can have complex, character, or logical vectors. Data frame 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:

```
> data(infert)
> str(infert)
```

```
`data.frame':   248 obs. of  8 variables:
 $ education     : Factor w/ 3 levels "0-5yrs","6-11yrs",..: 1 1 ...
 $ age           : num  NA 45 NA 23 35 36 23 32 21 28 ...
 $ parity        : num  6 1 6 4 3 4 1 2 1 2 ...
 $ induced       : num  1 1 2 2 1 2 0 0 0 0 ...
 $ case          : num  1 1 1 1 1 1 1 1 1 1 ...
 $ spontaneous   : num  2 0 0 0 1 1 0 0 1 0 ...
 $ stratum       : int  1 2 3 4 5 6 7 8 9 10 ...
 $ pooled.stratum: num  3 1 4 2 32 36 6 22 5 19 ...
> infert[1:10, 1:6]
   education age parity induced case spontaneous
1     0-5yrs  NA      6       1    1           2
2     0-5yrs  45      1       1    1           0
3     0-5yrs  NA      6       2    1           0
4     0-5yrs  23      4       2    1           0
5    6-11yrs  35      3       1    1           1
6    6-11yrs  36      4       2    1           1
7    6-11yrs  23      1       0    1           0
8    6-11yrs  32      2       0    1           0
9    6-11yrs  21      1       0    1           1
10   6-11yrs  28      2       0    1           0
```

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

```
> #age variable
> infert$age
  [1] NA 45 NA 23 35 36 23 32 21 28 29 37 31 29 31
 [16] 27 30 26 25 44 40 35 28 36 27 40 38 34 28 30
 ...
[226] 35 25 34 31 26 32 21 28 37 25 32 25 31 38 26
[241] 31 31 25 31 34 35 29 23
> mode(infert$age)
[1] "numeric"
> class(infert$age)
[1] "numeric"

> #stratum variable
> infert$stratum
  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
 [16] 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
 ...
[226] 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75
[241] 76 77 78 79 80 81 82 83
> mode(infert$stratum)
[1] "numeric"
> class(infert$stratum)
[1] "integer"

> #education variable
> infert$education
  [1] 0-5yrs  0-5yrs  0-5yrs  0-5yrs  6-11yrs
  [6] 6-11yrs 6-11yrs 6-11yrs 6-11yrs 6-11yrs
 ...
[241] 12+ yrs 12+ yrs 12+ yrs 12+ yrs 12+ yrs
[246] 12+ yrs 12+ yrs 12+ yrs
Levels: 0-5yrs 6-11yrs 12+ yrs
> mode(infert$education)
[1] "numeric"
> class(infert$education)
[1] "factor"
```

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

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

```
> z <- unclass(infert$education)
> z
  [1] 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 [23] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 [45] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
 [67] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 1 1 1 1 2
 [89] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
[111] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3
[133] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
[155] 3 3 3 3 3 3 3 3 3 3 3 1 1 1 1 2 2 2 2 2 2 2
[177] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
[199] 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3
[221] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
[243] 3 3 3 3 3 3
attr(,"levels")
[1] "0-5yrs"  "6-11yrs" "12+ yrs"
> mode(z)
[1] "numeric"
> class(z)
[1] "integer"
```

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

```
> cointoss <- sample(c("Head","Tail"), 100, replace = T)
> cointoss
  [1] "Head" "Head" "Head" "Tail" "Tail" "Head"
  [7] "Tail" "Head" "Tail" "Tail" "Tail" "Tail"
  ...
 [91] "Head" "Head" "Tail" "Tail" "Tail" "Head"
 [97] "Tail" "Head" "Tail" "Head"
> fct <- factor(cointoss)
> fct
  [1] Head Head Head Tail Tail Head Tail Head Tail
 [10] Tail Tail Tail Tail Head Tail Tail Head Tail
 ...
 [82] Tail Tail Tail Tail Head Head Tail Tail Tail
 [91] Head Head Tail Tail Tail Head Tail Head Tail
[100] Head
Levels: Head Tail
> unclass(fct)
  [1] 1 1 1 2 2 1 2 1 2 2 2 2 2 1 2 2 1 2 2 2 1 2
 [23] 1 1 2 2 1 2 2 1 2 2 2 2 1 1 2 2 2 1 1 1 1 2
 [45] 1 1 1 1 1 1 2 2 2 1 2 2 1 2 1 2 1 1 1 1 1 2
 [67] 2 1 1 1 2 2 1 1 1 1 2 1 1 1 2 2 2 2 2 1 1 2
 [89] 2 2 1 1 2 2 2 1 2 1 2 1
attr(,"levels")
[1] "Head" "Tail"
```

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

```
> as.character(cointoss)
  [1] "Head" "Head" "Head" "Tail" "Tail" "Head"
  [7] "Tail" "Head" "Tail" "Tail" "Tail" "Tail"
  ...
 [91] "Head" "Head" "Tail" "Tail" "Tail" "Head"
 [97] "Tail" "Head" "Tail" "Head"
> as.character(fct)
  [1] "Head" "Head" "Head" "Tail" "Tail" "Head"
  [7] "Tail" "Head" "Tail" "Tail" "Tail" "Tail"
  ...
 [91] "Head" "Head" "Tail" "Tail" "Tail" "Head"
 [97] "Tail" "Head" "Tail" "Head"
```

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

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

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

```
> ofac2 <- factor(samp, levels=c("Low","Medium","High"), ordered=T)
> ofac2 #yes, much better!
  [1] High   High   Medium Medium High   Low
  [7] Medium High   High   High   Low    High
  ...
 [91] Medium Medium High   Medium High   High
 [97] High   Medium Medium Low
Levels: Low < Medium < High
```

```
> table(ofac2)
ofac2
   Low Medium   High
    25     32     43
```

Great this is exactly what we want! For review, Table 32 summarizes the variable types in epidemiology and how they are represented in R. Factors (unordered and ordered) are use to represent nominal and ordinal categorical variables. The `infert` data set contains vectors of class factor, numeric, and integer

```
> 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 ...
```

Table 32 Variable types in epidemiologic data and their representations in R data frames

Representations in data		***Representations in R***		
Variable type	***Examples***	***Mode***	***Class***	***Examples from infert data***
Numeric				

Continuous	3.45, 2/3	numeric	numeric	> infert$age
				  [1] 26 42 39 34 35 36 23 32
				  [9] 21 28 29 37 31 29 31 27
				  ...
				[233] 28 37 25 32 25 31 38 26
				[241] 31 31 25 31 34 35 29 23

Discrete	1, 2, 3, 4, ...	numeric	integer	> infert$stratum
				  [1] 1 2 3 4 5 6 7 8
				  [9] 9 10 11 12 13 14 15 16
				  ...
				[233] 68 69 70 71 72 73 74 75
				[241] 76 77 78 79 80 81 82 83

Categorical				
Nominal	male vs. female	numeric	factor	> infert$education [1] 0-5yrs 0-5yrs 0-5yrs [4] 0-5yrs 6-11yrs 6-11yrs ... [244] 12+ yrs 12+ yrs 12+ yrs [247] 12+ yrs 12+ yrs 3 Levels: 0-5yrs ... 12+ yrs
Ordinal	low < medium < high	numeric	factor ordered	

Creating data frames

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

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

Table 33 Common ways of creating data frames

Function	*Description*	*Examples in R*

`data.frame`	data frames are of mode list	`> x <- data.frame(id=1:3,sex=c("M","F","T"))`
		`> x`
		`  id sex`
		`1  1   M`
		`2  2   F`
		`3  3   T`
		`> mode(x)`
		`[1] "list"`

`as.data.frame`	coerces data object into a data frame	`> x <- matrix(1:6,2,3)`
		`> y <- as.data.frame(x)`
		`> y`
		`  V1 V2 V3`
		`1  1  3  5`
		`2  2  4  6`

	can combine with as.table to convert an array into a data frame	`> x <- array(1:8, c(2, 2, 2))`
		`> dimnames(x) <- list(Exp = c("Y", "N"),`
		`      Dis = c("Y", "N"), Conf = c("Y", "N"))`
		`> x`
		`, , Conf = Y`
		
		`   Dis`
		`Exp Y N`
		`  Y 1 3`
		`  N 2 4`
		
		`, , Conf = N`
		
		`   Dis`
		`Exp Y N`
		`  Y 5 7`
		`  N 6 8`
		
		`> as.data.frame(as.table(x))`
		`  Exp Dis Conf Freq`
		`1   Y   Y    Y    1`
		`2   N   Y    Y    2`
		`3   Y   N    Y    3`
		`4   N   N    Y    4`
		`5   Y   Y    N    5`
		`6   N   Y    N    6`
		`7   Y   N    N    7`
		`8   N   N    N    8`

Function	***Description***	***Examples in R***

read.table
read.csv
read.csv2
read.delim
read.delim2
read.fmf
	reads in ASCII text data file into data frame object	> wcgs <- read.csv(".../data/wcgs.csv", header=T)
		> str(wcgs)
		`data.frame': 3154 obs. of 14 variables:
		 $ id : int 2001 2002 2003 2004 2005 ...
		 $ age0 : int 49 42 42 41 59 44 44 40 ...
		 $ height0: int 73 70 69 68 70 72 72 71 72 ...
		 $ weight0: int 150 160 160 152 150 204 164 ...
		 $ sbp0 : int 110 154 110 124 144 150 130 ...
		...

Naming data frames

Table 34 Common ways of naming data frames

Function	***Examples in R***
names	> x <- data.frame(cbind(1:3, c("M","F","F"))) > x X1 X2 1 1 M 2 2 F 3 3 F > names(x) <- c("Subjno","Sex") > x Subjno Sex 1 1 M 2 2 F 3 3 F
row.names	> row.names(x) <- c("Subj 1","Subj 2","Subj 3") > x Subjno Sex Subj 1 1 M Subj 2 2 F Subj 3 3 F

Indexing data frames

Table 35 Common ways of indexing data frames

Description	***Examples in R***
Replacing by position	> data(infert) > infert[1:5, 1:3] education age parity 1 0-5yrs 26 6 2 0-5yrs 42 1 3 0-5yrs 39 6 4 0-5yrs 34 4 5 6-11yrs 35 3

Description	***Examples in R***

Replacing by name	`> infert[1:5, c("education", "age", "parity")]`
	`  education age parity`
	`1    0-5yrs  26      6`
	`2    0-5yrs  42      1`
	`3    0-5yrs  39      6`
	`4    0-5yrs  34      4`
	`5   6-11yrs  35      3`

Replacing using a logical vector	`> infert$age<30`
	`  [1]  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE`
	`  [8] FALSE  TRUE  TRUE  TRUE FALSE FALSE  TRUE`
	`...`
	`[239] FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE`
	`[246] FALSE  TRUE  TRUE`
	`> infert[infert$age<30, c("education", "induced", "parity")]`
	`    education induced parity`
	`1      0-5yrs       1      6`
	`7     6-11yrs       0      1`
	`9     6-11yrs       0      1`
	`...`
	`243   12+ yrs       0      1`
	`247   12+ yrs       0      1`
	`248   12+ yrs       0      1`
	
	`> #can also use subset function`
	`> subset(infert, age<30, c("education", "induced",`
	`      "parity"))`
	`    education induced parity`
	`1      0-5yrs       1      6`
	`7     6-11yrs       0      1`
	`9     6-11yrs       0      1`
	`...`
	`243   12+ yrs       0      1`
	`247   12+ yrs       0      1`
	`248   12+ yrs       0      1`

Replacing data frame components

Table 36 Common ways of replacing data frame components

<table>
<tr><th>Description</th><th>Examples in R</th></tr>
<tr><td>Replacing by position</td><td>

```
> data(infert)
> infert[1:4, 1:2]
  education age
1    0-5yrs  26
2    0-5yrs  42
3    0-5yrs  39
4    0-5yrs  34
> infert[1:4, 2] <- c(NA, 45, NA, 23)
> infert[1:4, 1:2]
  education age
1    0-5yrs  NA
2    0-5yrs  45
3    0-5yrs  NA
4    0-5yrs  23
```

</td></tr>
<tr><td>Replacing by name</td><td>

```
> data(infert)
> names(infert)
[1] "education"      "age"
[3] "parity"         "induced"
[5] "case"           "spontaneous"
[7] "stratum"        "pooled.stratum"
> infert[1:4, c("education","age")]
  education age
1    0-5yrs  26
2    0-5yrs  42
3    0-5yrs  39
4    0-5yrs  34
> infert[1:4, c("age")] <- c(NA, 45, NA, 23)
> infert[1:4, c("education","age")]
  education age
1    0-5yrs  NA
2    0-5yrs  45
3    0-5yrs  NA
4    0-5yrs  23
```

</td></tr>
<tr><td>Replacing using a logical vector</td><td>

```
> data(infert)
> table(infert$parity)

 1  2  3  4  5  6
99 81 36 18  6  8
> #change values of 5 or 6 to missing (NA)
> infert$parity[infert$parity==5 | infert$parity==6] <- NA
> table(infert$parity)

 1  2  3  4
99 81 36 18
> table(infert$parity, exclude=NULL)

   1    2    3    4 <NA>
  99   81   36   18   14
```

</td></tr>
</table>

Operations on data frames

Table 37 Common operations on data frames

Function	***Description***	***Examples in R***
tapply	Apply a function to each cell of a ragged array, that is to each (non-empty) group of values given by a unique combination of the levels of certain factors.	> args(tapply) function (X, INDEX, FUN = NULL, ..., simplify = TRUE) NULL > tapply(infert$age, infert$education, mean) 0-5yrs 6-11yrs 12+ yrs 35.25000 32.85000 29.72414
lapply	Apply a function to each component of the list	> data(infert) > lapply(infert[,1:3], table) $education 0-5yrs 6-11yrs 12+ yrs 12 120 116 $age 21 23 24 25 26 27 28 29 30 31 32 34 35 36 37 38 6 6 3 15 15 15 30 12 12 21 15 18 18 15 12 8 39 40 41 42 44 9 6 3 6 3 $parity 1 2 3 4 5 6 99 81 36 18 6 8
sapply	Apply a function to each component of the list, and simplify if possible	> x <- list(1:3, 5:8) > x [[1]] [1] 1 2 3 [[2]] [1] 5 6 7 8 > sapply(x, mean) [1] 2.0 6.5 > lapply(x, mean) [[1]] [1] 2 [[2]] [1] 6.5
mapply	Apply a function to the first elements of each argument, the second elements, the third elements, and so on. Arguments are recycled if necessary.	> df <- data.frame(var1 = 1:4, var2 = 4:1) > mapply("*", df$var1, df$var2) [1] 4 6 6 4 > mapply(c, df$var1, df$var2) [,1] [,2] [,3] [,4] [1,] 1 2 3 4 [2,] 4 3 2 1

<table>
<tr><th>Function</th><th>Description</th><th>Examples in R</th></tr>
<tr><td>aggregate</td><td>Splits the data into subsets, computes summary statistics for each, and returns the result in a convenient form.</td><td><pre>
> data(infert)
> aggregate(infert[,c("age", "parity")],
      by = list(Education = infert$educ,
      Induced = infert$induced), mean)
  Education Induced   age   parity
1   0-5yrs   0 38.00000 2.500000
2   6-11yrs   0 33.23077 1.833333
3   12+ yrs   0 30.04918 1.524590
4   0-5yrs   1 34.00000 3.500000
5   6-11yrs   1 32.51852 2.333333
6   12+ yrs   1 29.46154 1.974359
7   0-5yrs   2 33.83333 5.666667
8   6-11yrs   2 31.46667 3.066667
9   12+ yrs   2 29.12500 2.875000
</pre></td></tr>
<tr><td>attach
detach</td><td>The database is attached to the R search path. This means that the database is searched by R when evaluating a variable, so objects in the database can be accessed by simply giving their names.</td><td><pre>
> attach(infert)
> table(induced, education)
       education
induced 0-5yrs 6-11yrs 12+ yrs
      0 4   78   61
      1 2   27   39
      2 6   15   16

> detach(infert)
> table(induced, education)
Error in table(induced, education) : Object
      "induced" not found
</pre></td></tr>
</table>

2.7 Managing data objects

Table 38 Common ways of managing data objects

<table>
<tr><th>Function</th><th>Description</th><th>Examples in R</th></tr>
<tr><td>ls
objects</td><td>List objects</td><td><pre>
> ls()
[1] "last.warning" "mx"   "ss"
[4] "x"   "xx"   "yy"
> objects() #equivalent to previous
[1] "last.warning" "mx"   "ss"
[4] "x"   "xx"   "yy"
</pre></td></tr>
<tr><td>rm
remove</td><td>Remove object(s)</td><td><pre>
> ls()
[1] "xx" "yy" "zz"
> rm(yy)
> ls()
[1] "xx" "zz"
> remove(xx) #equivalent to 'rm'
> ls()
[1] "zz"

## remove (almost) everything working environment.
## Don't do this unless you are really sure
> rm(list = ls())
</pre></td></tr>
</table>

Function	***Description***	***Examples in R***

`apropos`	displays of all objects in the search list matching topic	`> apropos(plot)`
		` [1] ".__C__recordedplot"  "biplot"`
		` [3] "screeplot"           "lag.plot"`
		` [5] "monthplot"           "plot.spec"`
		` ...`
		`[43] "preplot"             "print.recordedplot"`
		`[45] "qqplot"              "sunflowerplot"`
		`[47] "termplot"`

`save.image`	saves current workspace	`> save.image()`
		`>`

`save`
`load`
	writes a external representation of R objects to the specified file. The objects can be read back from the file at a later date by using the function 'load'	`x <- runif(20)`
		`y <- list(a = 1, b = TRUE, c = "oops")`
		`save(x, y, file = "c:/temp/xy.Rdata")`
		`> rm(x,y)`
		`> x`
		`Error: Object "x" not found`
		`> y`
		`Error: Object "y" not found`
		`> load(file = "c:/temp/xy.Rdata")`
		`> x`
		` [1] 0.2887683 0.5891149 0.7900659 0.2806621`
		` [5] 0.2585261 0.2429649 0.6663309 0.8029014`
		` [9] 0.1938921 0.6188805 0.3239679 0.8038926`
		`[13] 0.9732817 0.1010119 0.5107601 0.8798169`
		`[17] 0.8679555 0.6202131 0.6596147 0.5545634`
		`> y`
		`$a`
		`[1] 1`
		
		`$b`
		`[1] TRUE`
		
		`$c`
		`[1] "oops"`

2.8 Managing your workspace

Getting and setting your working directory

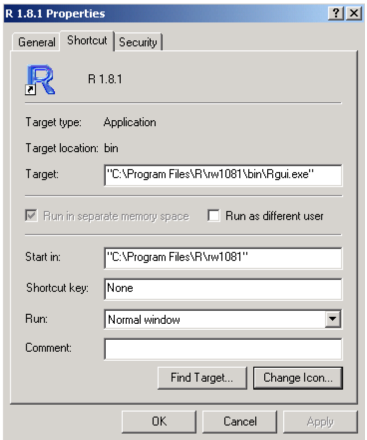
When you start R, the program sets up a default file path to the working directory that contains or will contain the `.Rdata` file. To see the default path to the working directory use the `getwd` function.

```
> getwd()
[1] "C:/Program Files/R/rw1081"
```

If you are running multiple projects, you can set the file path to a specific project working directory.

```
> setwd(“C:/myprojects/R/project01”)
> getwd()
[1] "C:/myprojects/R/project01"
```

Some epidemiologists like to set up a separate R icon for each major project. You can set a specific R icon start in a project working directory. First, right click on the R icon to open a context menu. Select Properties from the menu to open a dialog box (see Figure 4). From the Shortcut tab, edit the Start in box to contain the file path to your working directory. If your working directory does not have a `.Rdata` file, R will create a new file.



R 1.8.1

Target type: Application

Target location: bin

Target: "C:\Program Files\R\rw1081\bin\Rgui.exe"

☑ Run in separate memory space ☐ Run as different user

Start in: "C:\Program Files\R\rw1081"

Shortcut key: None

Run: Normal window

Comment:

Figure 4 R icon Properties dialog box to set the path to the working directory

Changing R options

To display current options use the `options` function without arguments (`options()`). The default settings of some of these options are the following:

```
'prompt'              '"> "'     'continue'             '"+ "'
'width'               '80'       'digits'               '7'
'expressions'         '500'      'keep.source'          'TRUE'
'show.signif.stars'   'TRUE'     'show.coef.Pvalues'    'TRUE'
'na.action'           'na.omit'  'ts.eps'               '1e-5'
'error'               'NULL'     'show.error.messags'   'TRUE'
'warn'                '0'        'warning.length'       '1000'
'echo'                'TRUE'     'verbose'              'FALSE'
'scipen'              '0'        'locatorBell'          'TRUE'
```

Yon can change the prompt symbol:

```
> options(prompt="cidp> ")
cidp>
```

You can change the screen display width:

```
cidp> options(width=50)
cidp> 1:30
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
[16] 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
cidp> options(width=25)
cidp> 1:30
 [1]  1  2  3  4  5  6  7
 [8]  8  9 10 11 12 13 14
[15] 15 16 17 18 19 20 21
[22] 22 23 24 25 26 27 28
[29] 29 30
```

Table 39 Common ways of managing your workspace

Function	***Description***	***Examples in R***
getwd	get working directory	> getwd() [1] "C:/Program Files/R/rw1081"
setwd	set working directory	> setwd("C:/mywork/project1/R/")
options	display or set options	> options(width=50) > 1:30 [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 [16] 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 > options(width=25) > 1:30 [1] 1 2 3 4 5 6 7 [8] 8 9 10 11 12 13 14 [15] 15 16 17 18 19 20 21 [22] 22 23 24 25 26 27 28 [29] 29 30
.First		

2.9 Exercises

3 Managing epidemiologic data in R

3.1 Entering data

There are many ways of getting your data into R for analysis. In the section that follow we will see how to enter the UGDB data in the 2x2 table (Table 40) as well as the original data from a comma-delimited text file.

Table 40 Deaths among subjects who received tolbutamide and placebo in the Unversity Group Diabetes Program (1970)

	Tolbutamide	Placebo
Deaths	30	21
Survivors	174	184

Entering data at the command prompt

Method 1 (recommended)

For review, the quickest and safest way to enter data at the command prompt is to do it in the following ways:

```
> #enter data for a vector
> vec <- c(30, 174, 21, 184)
> vec
[1]  30 174  21 184

> #enter data for a matrix
> vec <- c(30, 174, 21, 184)
> udat <- matrix(vec, 2, 2)
> udat
     [,1] [,2]
[1,]   30   21
[2,]  174  184

> #enter data for an array
> udat3 <- array(vec2, c(2, 2, 2))
> udat3
, , 1

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

, , 2

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

```
> #enter a list
> x <- list(crude.data = udat, stratified.data = udat3)
> x
$crude.data
     [,1] [,2]
[1,]   30   21
[2,]  174  184

$stratified.data
, , 1

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

, , 2

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

> #enter simple data frame
> subjno <- 1:3
> subjname <- c("Pedro", "Paulo", "Maria")
> age <- c(34, 56, 56)
> dat <- data.frame(subjno, subjname, age)
> dat
  subjno subjname age
1      1    Pedro  34
2      2    Paulo  56
3      3    Maria  56

> #enter a simple function
> odds.ratio <- function(a, b, c, d){ a*d / (b*c)}
> odds.ratio(30, 174,  21, 184)
[1] 1.510673
```

The above method is the safest for entering data at the command prompt. Alternatively, you can paste in the code from a text editor and get the same result. However, some students are accustomed to a different method of entering data at the command prompt. I will call this Method 2.

Method 2 (not recommended)

I do not recommend the following method of interactively entering data primarily for these reasons:

- If you make an error you must start again from the beginning
- You cannot easily paste your R code from a text editor and get reliable results

In spite of this, some students will still like Method 2. Here it is:

To read in a vector at the command prompt use the `scan` function. It does not matter if you enter the numbers on different lines, it will still be a vector.

```
> udat <- scan()
1: 30 174
3: 21 184
5:
Read 4 items
> udat
[1]  30 174  21 184
```

To read in a matrix at the command prompt combine the `matrix` and `scan` function. Again, it does not matter on what lines you enter the data, as long as they are in the correct order because the `matrix` function reads data in data column-wise.

```
> udat <- matrix(scan(), 2, 2)
1: 30 174 21 184
5:
Read 4 items
> udat
     [,1] [,2]
[1,]   30   21
[2,]  174  184

> udat <- matrix(scan(), 2, 2, byrow=T) #read data row-wise
1: 30 21 174 184
5:
Read 4 items
> udat
     [,1] [,2]
[1,]   30   21
[2,]  174  184
```

To read in an array at the command prompt combine the `array` and `scan` function. Again, it does not matter on what lines you enter the data, as long as they are in the correct order because the `array` function reads data in data column-wise. In this example I include the `dimnames` argument.

```
> udat3 <- 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
> udat3
, , 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 at the command prompt combine the `list` and `scan` function. Notice that you will need to specify the type of data that will go into each bin. In the example that follow this is done by specifying the `what` argument.

```
> dat <- scan("", what = list("", "", 0, 0))
1: John Paul 84 250
2: Jane Doe 34 154
3:
Read 2 records
> dat
[[1]]
[1] "John" "Jane"

[[2]]
[1] "Paul" "Doe"

[[3]]
[1] 84 34

[[4]]
[1] 250 15

> #same example but naming each bin
> dat <- scan("",list(firstname="",lastname="",age=0,chol=0))
1: John Paul 84 250
2: Jane Doe 34 154
3:
Read 2 records
> dat
$firstname
[1] "John" "Jane"

$lastname
[1] "Paul" "Doe"

$age
[1] 84 34

$chol
[1] 250 154
```

To read in a `data frame` at the command prompt combine the `data.frame`, `scan`, and `list` functions. You will need to specify the type of data that will go into each field using the `what` argument of the `scan` function.

```
> df <- data.frame(scan("", what = list(name="", age=0)))
1: Pedro 34
2: Paulo 56
3: Maria 56
4:
Read 3 records
> df
   name age
```

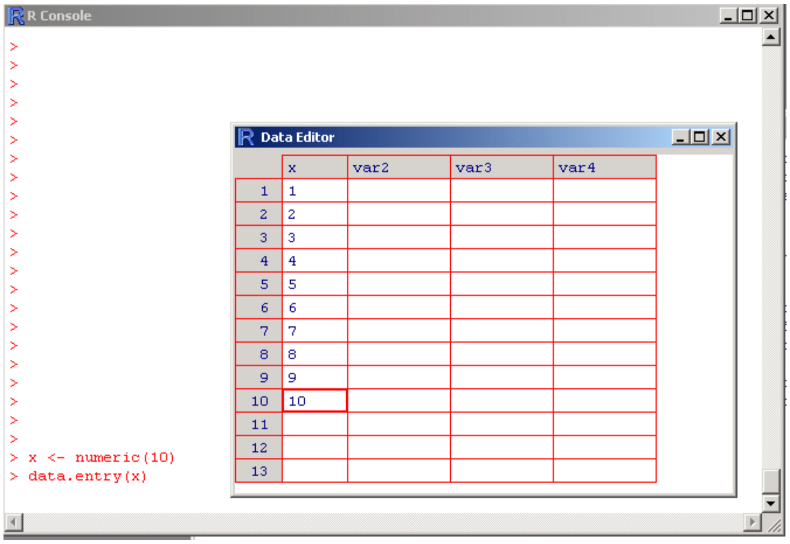
```
1 Pedro  34
2 Paulo  56
3 Maria  56
```

Method 3

Method 3 uses R's spreadsheet editor. Again, this is not a preferred method for me because I like to have my original data in a text editor. But for those that like Method 2, I think you might like Method 3 even more. We will be using the `edit` and `data.entry` functions.

To enter a vector you need to initialize a vector and then use the `data.entry` function. Also try the `edit` function like this `xnew <- edit(numeric(10))` and see what happens.

```
> x <- numeric(10)
> x
 [1] 0 0 0 0 0 0 0 0 0 0
> data.entry(x) #See Figure 5
> x
 [1]  1  2  3  4  5  6  7  8  9 10
```



```
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
>
> x <- numeric(10)
> data.entry(x)
```

	x	var2	var3	var4
1	1			
2	2			
3	3			
4	4			
5	5			
6	6			
7	7			
8	8			
9	9			
10	10			
11				
12				
13				

Figure 5 Spreadsheet editor in R from the data.entry function. Here we display the numbers that have already been entered.

To enter a matrix you need to initialize the matrix and then use the `edit` function. Notice that the editor added default column names. However, to add your own column names just click on the column heading with your mouse pointer (unfortunately you cannot give row names).

```
> #open spreadsheet in two steps
> xnew <- matrix(numeric(4),2,2)
> xnew <- edit(xnew)                   #Spreadsheet screen not shown

> #open spreadsheet in one step
> xnew <- edit(matrix(numeric(4),2,2)) #Spreadsheet screen not shown
> xnew
     col1 col2
[1,]   11   33
[2,]   22   44
```

Arrays and complex lists cannot be entered using a spreadsheet editor. Hence, we begin to see the limitations of spreadsheet-type approach to data analysis. One type of list, the data frame, can be entered using the `edit` function.

To enter a data frame use the `edit` function. However, you do not need to initialize a data frame (unlike with a matrx). Again, click on the column headings to enter column names.

```
> df <- edit(data.frame()) #Spreadsheet screen not shown
> df
    mykids age
1 Tomasito   7
2  Luisito   6
3 Angelita   3
```

When using the `edit` function to create a new data frame you 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 your edits directly back to `data_object` without the need to make an assignment. The equivalent using the `edit` functions looks like this: *data_object* `<- edit(data_object)`.

Entering from a file

Reading an ASCII text data file

For our purposes we will review how to read the following types of text data files:

- Comma-delimited data file (with or without headers and/or row names)
- Fixed width formatted data file (with or without headers and/or 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 (ugdp1.csv). The header is the first line that contains the field variable names. The row names is the first column that starts on the second line and uniquely identifies each row. A data file can come with either row names or header, neither, or both. My 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.

```
Vital.Status,Treatment,Age.Group
1,Dead,Tolbutamide,<55
2,Dead,Tolbutamide,<55
3,Dead,Tolbutamide,<55
4,Dead,Tolbutamide,<55
...
406,Survived,Placebo,55+
407,Survived,Placebo,55+
408,Survived,Placebo,55+
409,Survived,Placebo,55+
```

Let's enter this data using the `read.table` function.

```
> udat1 <- read.table("c:/.../ugdp1.csv", header=T, sep=",")
> udat1[1:10,]
  Vital.Status   Treatment Age.Group
1         Dead Tolbutamide       <55
2         Dead Tolbutamide       <55
3         Dead Tolbutamide       <55
4         Dead Tolbutamide       <55
```

```
5          Dead Tolbutamide       <55
6          Dead Tolbutamide       <55
7          Dead Tolbutamide       <55
8          Dead Tolbutamide       <55
9          Dead Tolbutamide       55+
10         Dead Tolbutamide       55+
```

Here is the same data file as it would appear without row names and without a header (ugdp2.csv).

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

Let's enter this data using the `read.table` function.

```
> cnames <- c("Status", "Treatment", "Age")
> udat2 <- read.table("c:/.../ugdp2.csv", header=F, sep=",",
     col.names = cnames)
> udat2[1:10,]
   Status   Treatment Age
1    Dead Tolbutamide <55
2    Dead Tolbutamide <55
3    Dead Tolbutamide <55
4    Dead Tolbutamide <55
5    Dead Tolbutamide <55
6    Dead Tolbutamide <55
7    Dead Tolbutamide <55
8    Dead Tolbutamide <55
9    Dead Tolbutamide 55+
10   Dead Tolbutamide 55+
```

Here is the same data file as it might appear as a fix formatted file. In this file, columns 1 to 8 are for field #1, columns 9 to 19 are for field #2, and columns 20 to 22 are for field #3. This type of data file is more compact. One needs a data dictionary to know which columns contain which fields.

```
1234567890123456789012
Dead    Tolbutamide<55
Dead    Tolbutamide<55
Dead    Tolbutamide<55
Dead    Tolbutamide<55
Dead    Tolbutamide<55
...
SurvivedPlacebo    55+
SurvivedPlacebo    55+
SurvivedPlacebo    55+
SurvivedPlacebo    55+
SurvivedPlacebo    55+
```

Finally, here is the same data file as it might appear as a fixed width formatted file but with numeric codes (ugdp3.fwf). 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.

```
123
121
121
121
121
121
121
...
212
212
212
212
212
```

Let's enter this data using the `read.fwf` function.

```
> cnames <- c("Status", "Treatment", "Age")
> udat3 <- read.fwf("c:/.../ugdp3.fwf", widths = c(1,1,1),
     col.names = cnames)
> udat3[1:10,]
   Status Treatment Age
1       1         2   1
2       1         2   1
3       1         2   1
4       1         2   1
5       1         2   1
6       1         2   1
7       1         2   1
8       1         2   1
9       1         2   2
10      1         2   2
```

R has other functions for reading text data files (`read.csv`, `read.csv2`, `read.delim`, `read.delim2`)

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

To read data that comes in a proprietary format load the `foreign` library.

```
> library(foreign)
> help(package=foreign) #R 1.8.1
Title: Read data stored by Minitab, S, SAS, SPSS, Stata, ...
...
Index:
lookup.xport            Lookup information on a SAS XPORT format
                        library
S3 read functions       Read an S3 Binary File
read.dta                Read Stata binary files
read.epiinfo            Read Epi Info data files
read.mtp                Read a Minitab Portable Worksheet
read.spss               Read an SPSS data file
read.ssd                obtain a data frame from a SAS permanent
                        dataset, via read.xport
read.xport              Read a SAS XPORT format library
write.dta               Write files in Stata binary format
```

For example, here I read in the `infert` data which is also available as a Stata data file.

```
> idat <- read.dta("c:/.../infert.dta")
> names(idat)
[1] "id"             "education"      "age"            "parity"
[5] "induced"        "case"           "spontaneous"    "stratum"
[9] "pooledstratum"
> str(idat)
`data.frame':   248 obs. of  9 variables:
 $ id           : int  1 2 3 4 5 6 7 8 9 10 ...
 $ education    : int  0 0 0 0 1 1 1 1 1 1 ...
 $ age          : int  26 42 39 34 35 36 23 32 21 28 ...
 $ parity       : int  6 1 6 4 3 4 1 2 1 2 ...
 $ induced      : int  1 1 2 2 1 2 0 0 0 0 ...
 $ case         : int  1 1 1 1 1 1 1 1 1 1 ...
 $ spontaneous  : int  2 0 0 0 1 1 0 0 1 0 ...
 $ stratum      : int  1 2 3 4 5 6 7 8 9 10 ...
 $ pooledstratum: int  3 1 4 2 32 36 6 22 5 19 ...
 ...
```

Entering using a URL

Text data files can be read directly off a web server into R using the `read.table` function. Here I load the Western Collaborative Group Study data directly off a web server.

```
> wdat <- read.table("http://www.../wcgs.csv", header = T, sep = ",")
> str(wdat)
`data.frame':   3154 obs. of  14 variables:
 $ id     : int  2001 2002 2003 2004 2005 2006 2007 2008 2009 ...
 $ age0   : int  49 42 42 41 59 44 44 40 43 42 ...
 $ height0: int  73 70 69 68 70 72 72 71 72 70 ...
 $ weight0: int  150 160 160 152 150 204 164 150 190 175 ...
 $ sbp0   : int  110 154 110 124 144 150 130 138 146 132 ...
 $ dbp0   : int  76 84 78 78 86 90 84 60 76 90 ...
 $ chol0  : int  225 177 181 132 255 182 155 140 149 325 ...
 $ behpat0: int  2 2 3 4 3 4 4 2 3 2 ...
 $ ncigs0 : int  25 20 0 20 20 0 0 0 25 0 ...
 $ dibpat0: int  1 1 0 0 0 0 0 1 0 1 ...
 $ chd69  : int  0 0 0 0 1 0 0 0 0 0 ...
 $ typechd: int  0 0 0 0 1 0 0 0 0 0 ...
 $ time169: int  1664 3071 3071 3064 1885 3102 3074 3071 3064 ...
 $ arcus0 : int  0 1 0 0 1 0 0 0 0 1 ...
```

3.2 Editing data

Text editor

For small data sets, it may be very convenience to edit the data in your favorite text editor. Key-recording macros, and search and replace tools can be very useful and efficient.

data.entry, edit, or fix function

For vector and matrices you 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 you assign it to a new or other object name. In contrast, changes made with the `fix` function are saved back to the original data object name. Therefore, be careful when you use the `fix` function because you made unintentionally loose data. To play it safe use the `edit` function.

Vectorized approaches

You can combine relational and logical operators to replace vector components in data frames.

`EXAMPLES PENDING`

R also has versatile text processing functions. Study the following related examples.

Table 41 R function for processing text in character vectors

Function	***Description***	***Examples in R***

`nchar`	returns the number of characters in each element of a character vector	`> x <- c("a", "ab", "abc", "abcd")`
		`> nchar(x)`
		`[1] 1 2 3 4`

`strsplit`	Split the elements of a character vector into substrings according to the presence of substring 'split' within them.	`> some.dates <- c("10/02/70", "02/04/67")`
		`> some.dates`
		`[1] "10/02/70" "02/04/67"`
		`> strsplit(some.dates, "/")`
		`[[1]]`
		`[1] "10" "02" "70"`
		
		`[[2]]`
		`[1] "02" "04" "67"`

`substr`	Extract or replace substrings in a character vector	`> months <- substr(some.dates, 1, 2)`
		`> months`
		`[1] "10" "02"`
		`> days <- substr(some.dates, 4, 5)`
		`> days`
		`[1] "02" "04"`
		`> years <- substr(some.dates, 7, 8)`
		`> years`
		`[1] "70" "67"`

`paste`	Concatenate vectors after converting to character.	`> recover.dates <- paste(months, "/", days, "/",`
		`      years, sep = "")`
		`> recover.dates`
		`[1] "10/02/70" "02/04/67"`

Regular expressions

grep, sub

3.3 Sorting data

order

sort

3.4 Subsetting data

Indexing

Subsetting

3.5 Transforming data

Table 42 R functions for transforming variables in data frames

Function	***Description***	***Examples in R***
cut		

<-	transforming a vector and assigning it to a new data frame variable name	`> df <- data.frame(v1=1:3, v2=3:1)`
		`> df`
		`  v1 v2`
		`1  1  3`
		`2  2  2`
		`3  3  1`
		`> df$v3 <- log(df$v1) #creates new variable in df`
		`> df$v4 <- exp(df$v2)`
		`> df`
		`  v1 v2        v3        v4`
		`1  1  3 0.0000000 20.085537`
		`2  2  2 0.6931472  7.389056`
		`3  3  1 1.0986123  2.718282`

transform	transform one or more variables from a data frame and add it to the data frame	`> df <- data.frame(v1=1:3, v2=3:1)`
		`> df`
		`  v1 v2`
		`1  1  3`
		`2  2  2`
		`3  3  1`
		`> df2 <- transform(df, v3=log(v1), v4=exp(v2))`
		`> df2`
		`  v1 v2        v3        v4`
		`1  1  3 0.0000000 20.085537`
		`2  2  2 0.6931472  7.389056`
		`3  3  1 1.0986123  2.718282`

cut		
ifelse		
relevel		

3.6 Merging data

merge

3.7 Exporting data

write.table

write

3.8 Working with missing values

NA, is.na

NaN, is.nan

3.9 Working with dates

Table 43 R functions for handling calendar dates (from the survival package)

Function	Description	Examples in R
as.date	Converts character vector of dates into a vector of Julian dates (number of days since January 1, 1960)	> library(survival) > dd <- c("8/31/56", "8-31-1956", "083156", "31Aug56", "August 31 1956") > as.date(dd) [1] 31Aug56 31Aug56 31Aug56 31Aug56 31Aug56 > as.numeric(as.date(dd)) [1] -1218 -1218 -1218 -1218 -1218
mdy.date	converts a vector of months, a vector of days, and a vector of years into a vector of Julian dates (number of days since January 1, 1960)	> mons <- sample(1:12, 5, replace=T) > days <- sample(1:31, 5, replace=T) > years <- sample(1940:2003, 5, replace=T) > jd <- mdy.date(mons, days, years) > jd [1] 30Sep61 11Oct75 2Jun78 7Mar88 5Jan84 > as.numeric(jd) [1] 638 5762 6727 10293 8770
date.mdy	Convert a vector of Julian dates to a list of vectors with the corresponding values of month, day and year, and optionally weekday	> date.mdy(jd) $month [1] 9 10 6 3 1 $day [1] 30 11 2 7 5 $year [1] 1961 1975 1978 1988 1984 > date.mdy(jd, weekday=T) $month [1] 9 10 6 3 1 $day [1] 30 11 2 7 5 $year [1] 1961 1975 1978 1988 1984 $weekday [1] 7 7 6 2 5
date.mmddyy date.ddmmmyy date.mmddyyyy	Given a vector of Julian dates, returns a character vector of the form “mm/dd/yy” or “ddmmmyy”, or “mm/dd/yyyy”	> date.mmddyy(jd) [1] "9/30/61" "10/11/75" "6/2/78" "3/7/88" "1/5/84" > date.ddmmmyy(jd) [1] "30Sep61" "11Oct75" "2Jun78" "7Mar88" "5Jan84" > date.mmddyyyy(jd) [1] "9/30/1961" "10/11/1975" "6/2/1978" "3/7/1988" [5] "1/5/1984"

3.10 Exercises

4 Analyzing simple epidemiologic data

4.1 Overview

1. Assign input values
2. Do calculations
3. Collect results into one object

4.2 Evaluating a single measure of occurrence

Risk and prevalence data

Approximation methods

Risk and prevalence estimates based on binomial data can be represented by $R=x/n$, where *x* is the number of events or the number of persons with a condition, and *n* is the number of persons at risk at a given time. The following formula provides a normal distribution approximation for binomial data (ref Dalgaard).

$$z=\frac{x-p_0}{\sqrt{(p_0(1-p_0)/n)}}$$

This formula gives improved results with the Yate's continuity correction which shrinks the observed value by half a unit toward the expected value when calculating *z*.

$$z=\frac{|x-p_0|-\frac{1}{2}}{\sqrt{(p_0(1-p_0)/n)}}$$

```
#assign input values
x <- 39
n <- 215
p0 <- .15
# do the calculations
z <- (abs(x-n*p0)-.5)/sqrt(n*p0*(1-p0))
p.value <- 2*(1-pnorm(z)) #two-sided test
#collect results into one object
c(x=x, n=n, p1=x/n, p0=p0, z=z, p.value=p.value)
```

Let's run this code in R.

```
> #assign input values
> x <- 39
> n <- 215
> p0 <- .15
> # do the calculations
> z <- (abs(x-n*p0)-.5)/sqrt(n*p0*(1-p0))
> p.value <- 2*(1-pnorm(z)) #two-sided test
> #collect results into one object
> c(x=x, n=n, p1=x/n, p0=p0, z=z, p.value=p.value)
        x         n          p1          p0           z     p.value
 39.00000 215.00000   0.1813953   0.1500000   1.1937289   0.2325840
```

We can improve the output by collecting the results using the `cbind` function rather than `c` function.

```
> cbind(x=x, n=n, p1=x/n, p0=p0, z=z, p.value=p.value)
      x   n        p1   p0        z   p.value
[1,] 39 215 0.1813953 0.15 1.193729 0.2325840
```

An additional advantage of using the `cbind` function is that you can test one or more proportions against one or more reference proportions, and then collect and display the results as a matrix. Suppose we want to test the proportions 39/215, 30/225, 50/200 against the value 0.15. Here's the new code run in R

```
> #assign input values
> x <- c(39, 30, 50)
> n <- c(215, 225, 200)
> p0 <- .15
> # do the calculations
> z <- (abs(x-n*p0)-.5)/sqrt(n*p0*(1-p0))
> p.value <- 2*(1-pnorm(z)) #two-sided test
> #collect results into one object
> cbind(x=x, n=n, p1=x/n, p0=p0, z=z, p.value=p.value)
      x   n        p1   p0        z      p.value
[1,] 39 215 0.1813953 0.15 1.193729 0.2325840460
[2,] 30 225 0.1333333 0.15 0.606788 0.5439915886
[3,] 50 200 0.2500000 0.15 3.861575 0.0001126582
```

Now let's review the following line:

```
> p.value <- 2*(1-pnorm(z)) #two-sided test
```

The `pnorm` function takes the argument `z` and returns the appropriate probability from the cumulative distribution function; that is, it returns Pr(Z≤z) from the standard normal distribution. Notice we had to multiply the probability by 2 in order to conduct a two-sided test. Because z^2 has an approximate χ^2 distribution with 1 degree of freedom, we can also use the `pchisq` function and get the same answer:

```
> p.value <- 1-pchisq(z^2,df=1) #two-sided test
> cbind(x=x, n=n, p1=x/n, p0=p0, z=z, p.value=p.value)
      x   n        p1   p0        z      p.value
[1,] 39 215 0.1813953 0.15 1.193729 0.2325840460
[2,] 30 225 0.1333333 0.15 0.606788 0.5439915886
[3,] 50 200 0.2500000 0.15 3.861575 0.0001126582
```

Notice that with the χ^2 distribution you do not need to multiply the probability by 2 to get a two-sided test. Now, if you plan on testing one sample proportions often, then it makes sense to create your own function to automate the calculation and increase your productivity.

```
prop.approx <- function(x, n, p0=0.5, correction=TRUE){
  # x = vector of numerators (successes)
  # n = vector of denominators (independent trials)
  # p0 = vector reference proportions
  # do the calculations
  if(correction) {
    yates <- .5
  } else {
      yates <- 0
    }
  z <- (abs(x-n*p0)-yates)/sqrt(n*p0*(1-p0))
  p.value <- 1-pchisq(z^2, df=1) #two-sided test
  #collect results into one object
  cbind(x=x, n=n, p1=x/n, p0=p0, chisq=z^2, p.value=p.value)
}
```

Let's test this function.

```
> prop.approx(xx, nn, p0=.15)
      x   n        p1   p0      chisq      p.value
[1,] 39 215 0.1813953 0.15  1.4249886 0.2325840460
[2,] 30 225 0.1333333 0.15  0.3681917 0.5439915886
[3,] 50 200 0.2500000 0.15 14.9117647 0.0001126582
```

First, notice that in this function we provided a default value for `p0`; if a value for `p0` is not provided then the function will use the default value 0.5. Second, we also provided for the user the option to choose not to use the Yate's continuity correction. For this we combined the `if` and `else` functions. It works like this: `if(TRUE) {'run this code'} else {'run alternative code'}`. Study the following function, test it, and experiment with it.

```
f <- function(option=TRUE){
  if(option){
    response <- 'The option is TRUE'
  } else{
    response <- 'The option is FALSE'
    }
  print(response)
}
```

Let's test this function:

```
> f(T)
[1] "The option is TRUE"
> f(F)
[1] "The option is FALSE"
```

Finally, does R have a function for testing a one sample proportion? Yes, but it's much more informative to learn how to build your own function. Why? Because R or another software package may not have the function you need, and by learning how to create you own functions you will be able to exploit many of R capabilities to solve many types of problems effectively and efficiently. Here is the same analysis using R's `prop.test` function (which additionally provides a confidence interval).

```
> prop.test(x=39, n=215, p=.15)

        1-sample proportions test with continuity correction

data:  39 out of 215, null probability 0.15
X-squared = 1.425, df = 1, p-value = 0.2326
alternative hypothesis: true p is not equal to 0.15
95 percent confidence interval:
 0.1335937 0.2408799
sample estimates:
        p
0.1813953
```

More specifically, unlike our `prop.approx` function that test each proportion, `prop.test` does an overall test of whether several proportions are equal to each other or equal to specified null proportion (see section on testing two or more proportions).

Exact method

To assess whether an observed proportion (*R*) differs from an alternative value (say p_0) you can calculate a *p* value based on the binomial distribution (`binom.test`) or a normal distribution approximation to the binomial distribution (covered in previous section).

The approximation using the normal distribution is satisfactory when the expected number of "successes" (x) and the "failures" (n-x) are both larger than 5. Here is the same analysis using `binom.test`:

```
> binom.test(x=39, n=215, p=.15)

        Exact binomial test

data:  39 and 215
number of successes = 39, number of trials = 215, p-value = 0.2135
alternative hypothesis: true probability of success is not equal to
     0.15
95 percent confidence interval:
 0.1322842 0.2395223
sample estimates:
probability of success
             0.1813953
```

4.3 Evaluating two or more measures of occurrence

```
> xx <- c(39, 30, 50)
> nn <- c(215, 225, 200)
> p0 <- rep(.15, 3)
> prop.test(xx, nn, p0) #overall test that 3 proportions = .15

        3-sample test for given proportions without continuity
     correction

data:  xx out of nn, null probabilities p0
X-squared = 17.8386, df = 3, p-value = 0.0004749
alternative hypothesis: two.sided
null values:
prop 1 prop 2 prop 3
  0.15   0.15   0.15
sample estimates:
   prop 1    prop 2    prop 3
0.1813953 0.1333333 0.2500000

> prop.test(xx, nn) #overall test that 3 proportions are equal

        3-sample test for equality of proportions without continuity
        correction

data:  xx out of nn
```

```
X-squared = 9.5653, df = 2, p-value = 0.008374
alternative hypothesis: two.sided
sample estimates:
   prop 1    prop 2    prop 3
0.1813953 0.1333333 0.2500000
```

4.4 Confidence intervals for measures of occurrence

Risk and prevalence data

$$R = \frac{x}{n}$$

Normal approximation to the binomial

$$R_L = R - Z \cdot SE(R)$$
$$R_U = R + Z \cdot SE(R)$$

$$SE(R) = \sqrt{\frac{x(n-x)}{n^3}}$$

Here is the R code in a text editor:

```
# assign input values
x <- 20
n <- 100
Z <- 1.645 #z value for 90% CI
# do calculations
SE.R <- sqrt(x*(n-x)/(n^3))
R.lower <- x/n-Z*SE.R
R.upper <- x/n+Z*SE.R
# collect results into one object
c(R = x/n, LCL = R.lower, UCL = R.upper)
```

Here is the code pasted into R or Rweb:

```
> # assign input values
> x <- 20
> n <- 100
> Z <- 1.645 #z value for 90% CI
> # do calculations
> SE.R <- sqrt(x*(n-x)/(n^3))
> R.lower <- x/n-Z*SE.R
> R.upper <- x/n+Z*SE.R
> # collect results into one object
> c(R = x/n, LCL = R.lower, UCL = R.upper)
     R    LCL    UCL
0.2000 0.1342 0.2658
```

Let's refine our calculation. Recall that for a 95% confidence interval (CI) the z value is approximately 1.96, and for a 90% CI the z value is approximately 1.645. For this we can use the `qnorm` function. The `qnorm` function returns the quantile value z for a specified "area under the normal distribution curve" (Pr{Z≤z}). More specifically, for a (1- α)% CI, the Pr{Z≤z} = 1-α/2. Then, for a given value of (1 – α/2), `qnorm` returns the value z. For example, for a 95% CI, α =

0.05 and Pr{Z≤z} = 0.975. Therefore, z = `qnorm(.975)` = 1.959964.

Let's edit the previous R code in our text editor to the following:

```
# assign input values
x <- 20
n <- 100
conf.level <- .90
# do calculations
Z <- qnorm(0.5*( 1 + conf.level))
SE.R <- sqrt(x * (n - x) / (n^3))
R.lci <- x/n - Z*SE.R
R.uci <- x/n + Z*SE.R
# collect results into one object
c(risk=x/n, lower.ci=R.lci, upper.ci=R.uci)
```

Here is the code pasted into R or Rweb:

```
> # assign input values
> x <- 20
> n <- 100
> conf.level <- .90
> # do calculations
> Z <- qnorm(0.5*(1 + conf.level))
> SE.R <- sqrt(x * (n - x) / (n^3))
> R.lci <- x/n - Z*SE.R
> R.uci <- x/n + Z*SE.R
> # collect results into one object
> c(risk=x/n, lower.ci=R.lci, upper.ci=R.uci)
     risk  lower.ci  upper.ci
0.2000000 0.1342059 0.2657941
```

Now we will create and load a function to automate these three steps (assign input values, do calculations, and collect results). Notice that I have included a default value for the `conf.level` argument and I have provided comments to define the arguments `x` and `n`. I can load the function by pasting the code that follows into R.

```
binom.approx <- function(x, n, conf.level = .95) {
  # x = number of successes
  # n = number of trials
  # do calculations
  Z <- qnorm(0.5*( 1 + conf.level))
  SE.R <- sqrt(x * (n - x) / (n^3))
  R.lci <- x/n - Z*SE.R
  R.uci <- x/n + Z*SE.R
  # collect results into one object
  c(x=x, n=n, risk=x/n, conf.level=conf.level, lower.ci=R.lci,
     upper.ci=R.uci)
}
```

Once the function is loaded we can use it:

```
> tt <- binom.approx(20, 100, .90)
> round(tt, 4)
        x          n       risk conf.level   lower.ci   upper.ci
  20.0000   100.0000     0.2000     0.9000     0.1342     0.2658
```

Great! However, what simple changes can we make to our `binom.approx` function to make it more versatile? For example, we may want it to handle vector arguments of length>1. By now we know enough to make these improvements.

```
binom.approx <- function(x, n, conf.level = .95) {
  # x = number of successes
  # n = number of trials
  # do calculations
  Z <- qnorm(0.5*(1+conf.level))
  SE.R <- sqrt(x * (n - x) / (n^3))
  R.lci <- x/n - Z*SE.R
  R.uci <- x/n + Z*SE.R
  # collect results into one object
  cbind(x=x, n=n, risk=x/n, conf.level=conf.level, lower=R.lci,
     upper=R.uci)
}
```

Once this improved function is loaded we can use it:

```
> success <- 1:9
> trials <- 10
> rr <- binom.approx(success, trials)
> rr
      x  n risk conf.level       lower     upper
 [1,] 1 10  0.1       0.95 -0.08593851 0.2859385
 [2,] 2 10  0.2       0.95 -0.04791801 0.4479180
 [3,] 3 10  0.3       0.95  0.01597423 0.5840258
 [4,] 4 10  0.4       0.95  0.09636369 0.7036363
 [5,] 5 10  0.5       0.95  0.19010248 0.8098975
 [6,] 6 10  0.6       0.95  0.29636369 0.9036363
 [7,] 7 10  0.7       0.95  0.41597423 0.9840258
 [8,] 8 10  0.8       0.95  0.55208199 1.0479180
 [9,] 9 10  0.9       0.95  0.71406149 1.0859385
```

From this example we see the limitation of calculating confidence limits using the standard normal distribution to approximate the binomial distribution. One solution is to use a formula that more accurately approximates the binomial distribution for small counts or small proportions (see Wilson's formula).

Exact approximation (using Wilson's formula)

For small counts or small proportions, Wilson's confidence limits for binomial data can be used (ref baby Rothman, p 132).

$$R_L , R_U = \frac{n}{n+Z^2} [\frac{x}{n} + \frac{Z^2}{2n} \pm Z \sqrt{\frac{x(n-x)}{n^3}} + \frac{Z^2}{4n^2}]$$

Assuming we plan to use this formula in the future, it makes sense to create a function. The easiest approach is to edit `binom.approx`.

```
binom.wilson <- function(x, n, conf.level = .95) {
  # x = number of successes
  # n = number of trials
  # do calculations
  Z <- qnorm(0.5*(1+conf.level))
  Zinsert <- Z*sqrt(((x*(n-x))/n^3) + Z^2/(4*n^2))
  R.lower <- (n/(n+Z^2))*(x/n + Z^2/(2*n) - Zinsert)
  R.upper <- (n/(n+Z^2))*(x/n + Z^2/(2*n) + Zinsert)
  # collect results into one object
  cbind(x=x, n=n, risk=x/n, conf.level=conf.level, lower.ci=R.lower,
     upper.ci=R.upper)
}
```

Okay, let's test Wilson's formula.

```
> xx <- 1
> nn <- c(10, 100, 1000, 10000, 100000, 1000000)
> binom.wilson(xx, nn)
     x     n  risk conf.level     lower.ci     upper.ci
[1,] 1 1e+01 1e-01       0.95 1.787621e-02 4.041500e-01
[2,] 1 1e+02 1e-02       0.95 1.767432e-03 5.448620e-02
[3,] 1 1e+03 1e-03       0.95 1.765464e-04 5.642559e-03
[4,] 1 1e+04 1e-04       0.95 1.765267e-05 5.662689e-04
[5,] 1 1e+05 1e-05       0.95 1.765248e-06 5.664710e-05
[6,] 1 1e+06 1e-06       0.95 1.765246e-07 5.664912e-06
```

Great! Looks good for very small proportions (i.e., no negative values). Of course, if you want an exact binomial confidence limits you can use the binom.test function in R.

Exact method (binom.test function)

To calculate exact binomial confidence limits, use the binom.test function. In addition to calculating the confidence limits, binom.test also tests the hypothesis that the observed proportion (x/n) is different from an alternative hypothesis (default p = 0.5) and reports a p value.

```
> args(binom.test)
function (x, n, p = 0.5, alternative = c("two.sided", "less",
    "greater"), conf.level = 0.95)
NULL
> binom.test(1, 100)

        Exact binomial test

data:  1 and 100
number of successes = 1, number of trials = 100, p-value = < 2.2e-16
alternative hypothesis: true probability of success is not equal to
     0.5
95 percent confidence interval:
 0.0002531460 0.0544593854
sample estimates:
probability of success
                  0.01
```

Unfortunately, binom.test cannot handle vector arguments of length > 1, and the results are more than we need. How can we change this? First, I need to know how to extract the confidence limits calculated from binom.test. To solve this I explore the data object produced by binom.test.

```
> rr <- binom.test(1, 10)
> attributes(rr)
$names
[1] "statistic"   "parameter"   "p.value"     "conf.int"
[5] "estimate"    "null.value"  "alternative" "method"
```

```
[9] "data.name"

$class
[1] "htest"
```

We see that `rr` is a list, so extracting the confidence limits is easy:

```
> rr$conf.int
[1] 0.002528579 0.445016117
attr(,"conf.level")
[1] 0.95
```

Now that I know how to extract the confidence limits from `binom.test`, I can create a new function that will use the exact binomial confidence limits from `binom.test`, but be able to handle vector arguments:

```
binom.exact <- function(x, n, conf.level=.95) {
  # x = number of successes
  # n = number of trials
  # do calculations
  xnc <- cbind(x,n,conf.level)
  lower <- numeric(nrow(xnc))
  upper <- numeric(nrow(xnc))
  for(i in 1:nrow(xnc)){
    ci <- binom.test(x=xnc[i,1], n=xnc[i,2], conf.level=xnc[i,3])
     $conf.int
    lower[i] <- ci[1]
    upper[i] <- ci[2]
  }
  # collect results into one object
  cbind(x=x, n=n, risk=x/n, conf.level=conf.level, lower.ci=lower,
     upper.ci=upper)
}
```

Now let's test `binom.exact`:

```
> binom.exact(1,100)
     x   n risk conf.level     lower.ci   upper.ci
[1,] 1 100 0.01       0.95 0.0002531460 0.05445939

> binom.exact(1:5,100)
     x   n risk conf.level     lower.ci   upper.ci
[1,] 1 100 0.01       0.95 0.0002531460 0.05445939
[2,] 2 100 0.02       0.95 0.0024313368 0.07038393
[3,] 3 100 0.03       0.95 0.0062299715 0.08517605
[4,] 4 100 0.04       0.95 0.0110044940 0.09925716
[5,] 5 100 0.05       0.95 0.0164318792 0.11283491
```

The `binom.exact` function demonstrates the use of a `for` loop. In the programming section we will cover `for` loops in more detail. For now, study the following example to get a feel how `for` loops work.

```
cumulative.sum <- function(x){
  lx <- length(x)
  for(i in 1:(lx-1)){
    x[i+1] <- x[i] + x[i+1]
  }
  x
}
```

Let's test the function.

```
> cumulative.sum(1:10)
 [1]  1  3  6 10 15 21 28 36 45 55
```

Incidence rate data

The calculation of confidence intervals for incidence rate data is based on the Poisson distribution.

$$R = \frac{x}{PT}$$

Normal approximation

$$R_L, R_U = R \pm Z \cdot SE(R)$$

$$SE(R) = \sqrt{\frac{x}{PT^2}}$$

```
pois.approx <- function(x, pt=1, conf.level = .95) {
  # x = Poisson count
  # pt = person time
  # do calculations
  Z <- qnorm(0.5*(1+conf.level))
  SE.R <- sqrt(x/pt^2)
  lower <- x/pt - Z*SE.R
  upper <- x/pt + Z*SE.R
  # collect results into one object
  cbind(x=x, pt=pt, rate=x/pt, conf.level=conf.level,
     lower.ci=lower, upper.ci=upper)
}
```

Let's test `pois.approx`.

```
> pois.approx(1:5, 2500)
     x   pt   rate conf.level      lower.ci    upper.ci
[1,] 1 2500 0.0004       0.95 -3.839856e-04 0.001183986
[2,] 2 2500 0.0008       0.95 -3.087231e-04 0.001908723
[3,] 3 2500 0.0012       0.95 -1.579029e-04 0.002557903
[4,] 4 2500 0.0016       0.95  3.202881e-05 0.003167971
[5,] 5 2500 0.0020       0.95  2.469549e-04 0.003753045
```

Exact approximation (Byar's formula)

$$R_L, R_U = \frac{x'\left(1 - \frac{1}{9x'} \pm \frac{Z}{3}\sqrt{\frac{1}{x'}}\right)^3}{PT}$$

```
pois.byar <- function(x, pt=1, conf.level = .95) {
  # x = Poisson count
  # pt = person time
  # do calculations
  Z <- qnorm(0.5*(1+conf.level))
  xprime <- x + 0.05
  Zinsert <- (Z/3)*sqrt(1/xprime)
  lower <- (xprime*(1-1/(9*xprime)-Zinsert)^3)/pt
  upper <- (xprime*(1-1/(9*xprime)+Zinsert)^3)/pt
  # collect results into one object
  cbind(x=x, pt=pt, rate=x/pt, conf.level=conf.level,
     lower.ci=lower, upper.ci=upper)
}
```

Let's test `pois.byar`.

```
> pois.byar(1:5, 2500)
     x   pt   rate conf.level     lower.ci    upper.ci
[1,] 1 2500 0.0004       0.95 7.096388e-06 0.001509448
[2,] 2 2500 0.0008       0.95 9.617719e-05 0.002260215
[3,] 3 2500 0.0012       0.95 2.498997e-04 0.002920101
[4,] 4 2500 0.0016       0.95 4.406498e-04 0.003536218
[5,] 5 2500 0.0020       0.95 6.557532e-04 0.004125255
```

Exact method

R does not have a function for calculating exact confidence limits for Poisson counts.

```
pois.exact <- function(x, pt=1, conf.level=.95){
  # x = Poisson count
  # pt = person time
  xc <- cbind(x,conf.level)
  results <- matrix(NA,nrow(xc),6)
  f1 <- function(x,ans,alpha=alp) {ppois(x,ans)-alpha/2}
  f2 <- function(x,ans,alpha=alp) 1-ppois(x,ans)+dpois(x,ans)-
     alpha/2
  for(i in 1:nrow(xc)){
    alp <- 1-xc[i,2]
    interval <- c(0,xc[i,1]*5+4)
    uci <- uniroot(f1,interval=interval,x=xc[i,1])$root/pt
    if(xc[i,1]==0){
      lci <- 0
    } else lci <- uniroot(f2,interval=interval,x=xc[i,1])$root/pt
    results[i,] <- c(xc[i,1],pt,xc[i,1]/pt,xc[i,2],lci,uci)
  }
  colnames <- c("x","pt","rate","conf.level","lower.ci","upper.ci")
  dimnames(results) <- list(NULL,colnames)
  results
}
```

Let's test `pois.exact`.

```
> pois.exact(1:5, 2500)
     x   pt   rate conf.level     lower.ci    upper.ci
[1,] 1 2500 0.0004       0.95 1.011583e-05 0.002228657
[2,] 2 2500 0.0008       0.95 9.688146e-05 0.002889877
[3,] 3 2500 0.0012       0.95 2.474685e-04 0.003506911
[4,] 4 2500 0.0016       0.95 4.359557e-04 0.004096636
```

```
[5,] 5 2500 0.0020       0.95 6.493943e-04 0.004667329
```

4.5 Confidence intervals for measures of association

Cohort studies with risk or prevalence data

Table 44 2 x 2 table for risk or prevalence data

	Disease	No disease	Totals
Exposed	a	b	a + b
Not exposed	c	d	c + d
Totals	a + c	b + d	T

Comparing two or more proportions

Table 45 R functions for handling calendar dates (from the survival package)

Function	*Description*	*Examples in R*
prop.test		
prop.trend.test		
chisq.test		
fishers.exact		

Risk difference

$$RD = \frac{a}{a+c} - \frac{b}{b+d}$$

Risk ratio

$$RR = \frac{a/(a+c)}{b/(b+d)}$$

Cohort studies with incidence rate data

Normal approximation

Exact approximation

Exact method

Case control studies

Normal approximation

Exact approximation

Exact method

5 Creating simple R functions

5.1 Why create functions?

5.2

5.3 Exercises

6 Controlling for confounding using stratification methods

6.1 Cohort studies with risk or prevalence data

Risk difference

$$RD_{MH}=\frac{\sum_i \frac{a_i N_{0i}-b_i N_{1i}}{T_i}}{\sum_i \frac{N_{1i} N_{0i}}{T_i}}$$

$$var(RD_{MH})=\frac{\sum_i \left(\frac{N_{1i} N_{0i}}{T_i}\right)^2 \left[\frac{a_i c_i}{N_{1i}^2 (N_{1i}-1)}+\frac{b_i d_i}{N_{0i}^2 (N_{0i}-1)}\right]}{\sum_i \left(\frac{N_{1i} N_{0i}}{T_i}\right)^2}$$

Risk ratio

$$RR_{MH}=\frac{\sum_i \frac{a_i N_{0i}}{T_i}}{\sum_i \frac{b_i N_{1i}}{T_i}}$$

$$var(\log(RR_{MH}))=\frac{\sum_i \left(\frac{M_{1i} N_{1i} N_{0i}}{T_i^2}-\frac{a_i b_i}{T_i}\right)}{\left(\sum_i \frac{a_i N_{0i}}{T_i}\right)\left(\sum_i b_i \frac{N_{1i}}{T_i}\right)}$$

6.2 Cohort studies with incidence rate data

Incidence rate difference

$$ID_{MH}=\frac{\sum_i \frac{a_i PT_{0i}+b_i PT_{1i}}{T_i}}{\sum_i \frac{PT_{1i} PT_{0i}}{T_i}}$$

$$Var(ID_{MH})=\frac{\sum_i (\frac{PT_{1i}PT_{0i}}{T_i})^2 (\frac{a_i}{PT_{1i}^2}+\frac{b_i}{PT_{0i}^2})}{(\sum_i \frac{PT_{1i}PT_{0i}}{T_i})^2}$$

Incidence rate ratio

$$IR_{MH}=\frac{\sum_i \frac{a_i PT_{0i}}{T_i}}{\sum_i \frac{b_i PT_{1i}}{T_i}}$$

$$Var[\log(\mathrm{OR}_{MH})]=\frac{\sum_i (\frac{M_{1i}PT_{1i}PT_{0i}}{T_i})^2}{(\sum_i \frac{a_i PT_{0i}}{T_i})(\sum_i \frac{b_i PT_{1i}}{T_i})}$$

6.3 Case control studies

Odds ratio

$$\mathrm{OR}_{MH}=\frac{\sum_i \frac{a_i d_i}{T_i}}{\sum_i \frac{b_i c_i}{T_i}}$$

$$Var[\log(IR_{MH})]=\frac{\sum_i G_i P_i}{2(\sum_i G_i)^2}+\frac{\sum_i G_i Q_i+H_i P_i}{2(\sum_i G_i \sum_i H_i)}+\frac{\sum_i H_i Q_i}{2(\sum_i H_i)^2}$$

where

$$G_i=\frac{a_i d_i}{T_i}, H_i=\frac{b_i c_i}{T_i}, P_i=\frac{a_i+d_i}{T_i}, Q_i=\frac{b_i+c_i}{T_i}$$

7 Using regression methods

8 Graphing basic epidemiologic data

8.1 Graphs

Arithmetic-scale line graphs

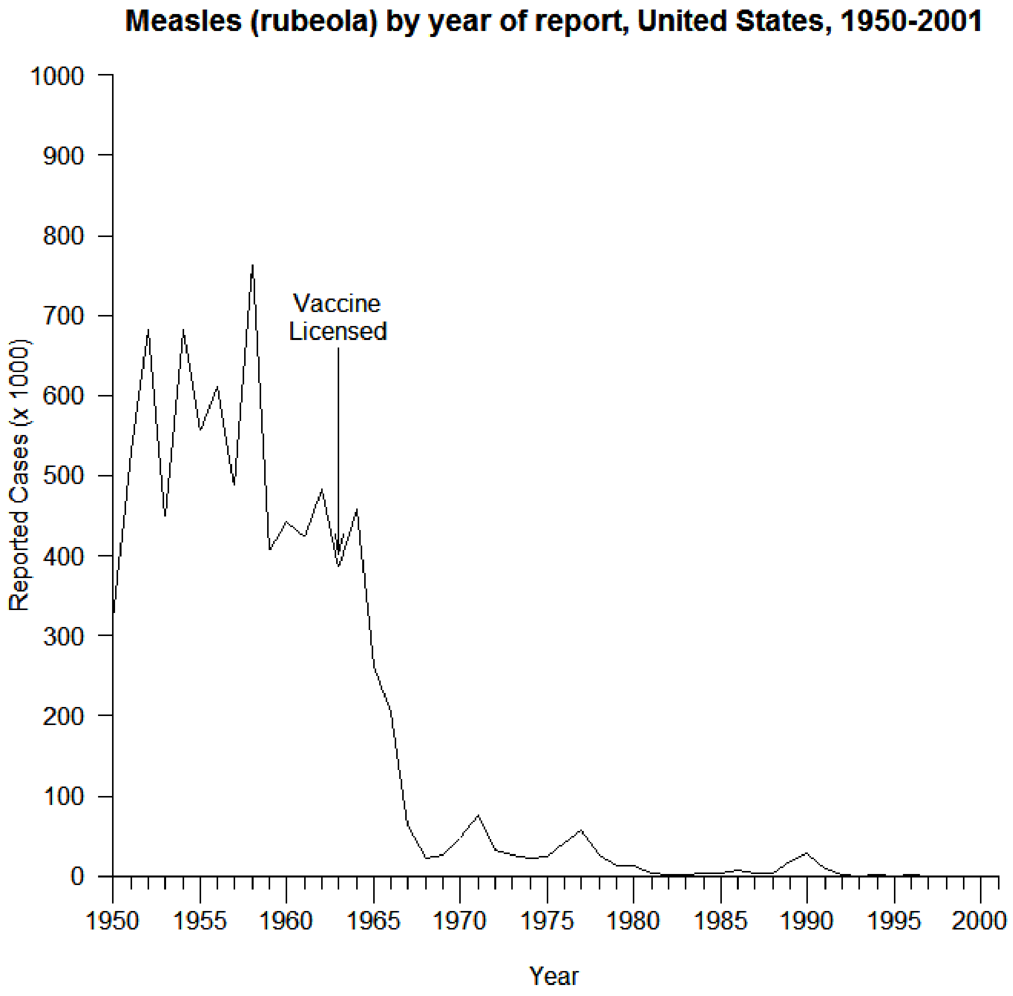


Figure 6 Example of arithmetic-scale line graph

```
year <- 1950:2001
# rate per 1000
cases <- c(319000, 530000, 683000, 449000, 683000, 555000, 612000,
     487000, 763000, 406000, 442000, 424000, 482000, 385000, 458000,
     262000, 204000, 63000, 22000, 26000, 47351, 75290, 32275,
     26690, 22094, 24374, 41126, 57345, 26871, 13597, 13506, 3124,
     1714, 1497, 2587, 2822, 6282, 3655, 3396, 18193, 27786, 9643,
     2237, 312, 963, 309, 508, 138, 100, 100, 86, 116)
plot(year, cases/1000, type='l', xlim=c(1950, 2001), ylim=c(0,1000),
     main='Measles (rubeola) by year of report, United States, 1950-
     2001', xlab='Year', ylab='Reported Cases (x 1000)', xaxs='i',
     yaxs='i', axes=F)
box(bty='l')
axis(1, at=year, labels=F, tick=T)
axis(1, at=seq(1950, 2000, 5), labels=seq(1950, 2000, 5), tick=T,
     tcl=-1)
axis(2, at=seq(0, 1000, 100), labels=seq(0, 1000, 100), las=2,
     tick=T)
arrows(1963, 660, 1963, 400, length=.15, angle=10)
text(1963, 700, 'Vaccine\nLicensed')
```

Semi-logarithmic-scale line graphs

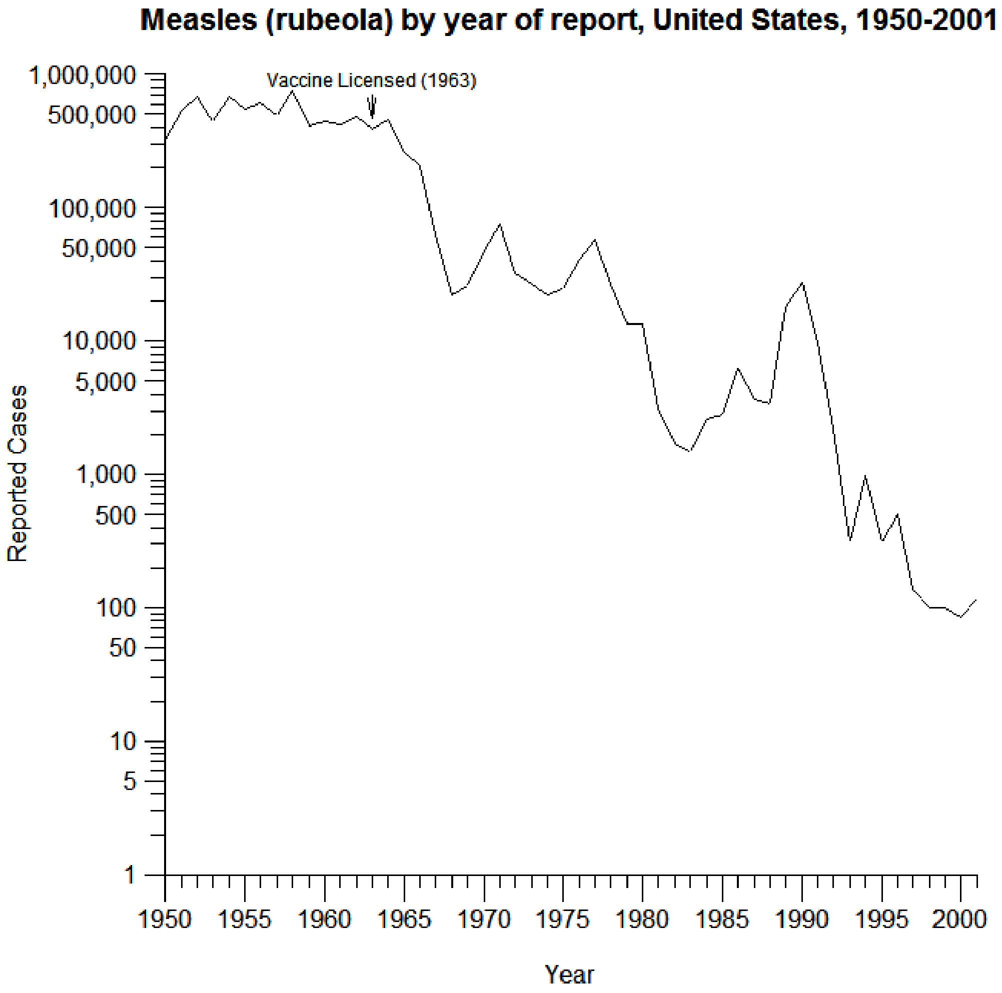


Figure 7 Example of semi-logarithmic-scale line graph

```
year <- 1950:2001
cases <- c(319000, 530000, 683000, 449000, 683000, 555000, 612000,
     487000, 763000, 406000, 442000, 424000, 482000, 385000, 458000,
     262000, 204000, 63000, 22000, 26000, 47351, 75290, 32275,
     26690, 22094, 24374, 41126, 57345, 26871, 13597, 13506, 3124,
     1714, 1497, 2587, 2822, 6282, 3655, 3396, 18193, 27786, 9643,
     2237, 312, 963, 309, 508, 138, 100, 100, 86, 116)

par(omi=c(0,.5,0,0))

plot(year, cases, type='l', log='y', xlim=c(1950, 2001), ylim=c(1,
     1000000), main='Measles (rubeola) by year of report, United
     States, 1950-2001', xlab='Year', ylab='', xaxs='i', yaxs='i',
     axes = FALSE)
box(bty='l')
axis(1, at=year, labels=FALSE, tick=T)
axis(1, at=seq(1950, 2000, 5), labels=seq(1950, 2000, 5), tick=T,
     tcl=-1)

axis(2, at=c(seq(1, 10, 1), seq(10, 100, 10), seq(100, 1000, 100),
     seq(1000, 10000, 1000), seq(10000, 100000, 10000), seq(100000,
     1000000, 100000)), labels=FALSE, tick=TRUE)

axis(2, at=c(1, 5, 10, 50, 100, 500, 1000, 5000, 10000, 50000,
     100000, 500000, 1000000), labels=c('1', '5', '10', '50', '100',
     '500', '1,000', '5,000', '10,000', '50,000', '100,000',
     '500,000', '1,000,000'), las=2, tick=T, tcl=-.75)

arrows(1963, 700000, 1963, 450000, length=.15, angle=10)
text(1963, 900000, 'Vaccine Licensed (1963)', cex=.75)
mtext(text='Reported Cases', side=2, line=5)

par(op)
```

Histograms

The hist function

```
> args(hist.default)
function (x, breaks = "Sturges", freq = NULL, probability = !freq,
    include.lowest = TRUE, right = TRUE, density = NULL, angle = 45,
    col = NULL, border = NULL, main = paste("Histogram of", xname),
    xlim = range(breaks), ylim = NULL, xlab = xname, ylab, axes =
     TRUE,
    plot = TRUE, labels = FALSE, nclass = NULL, ...)
NULL
```

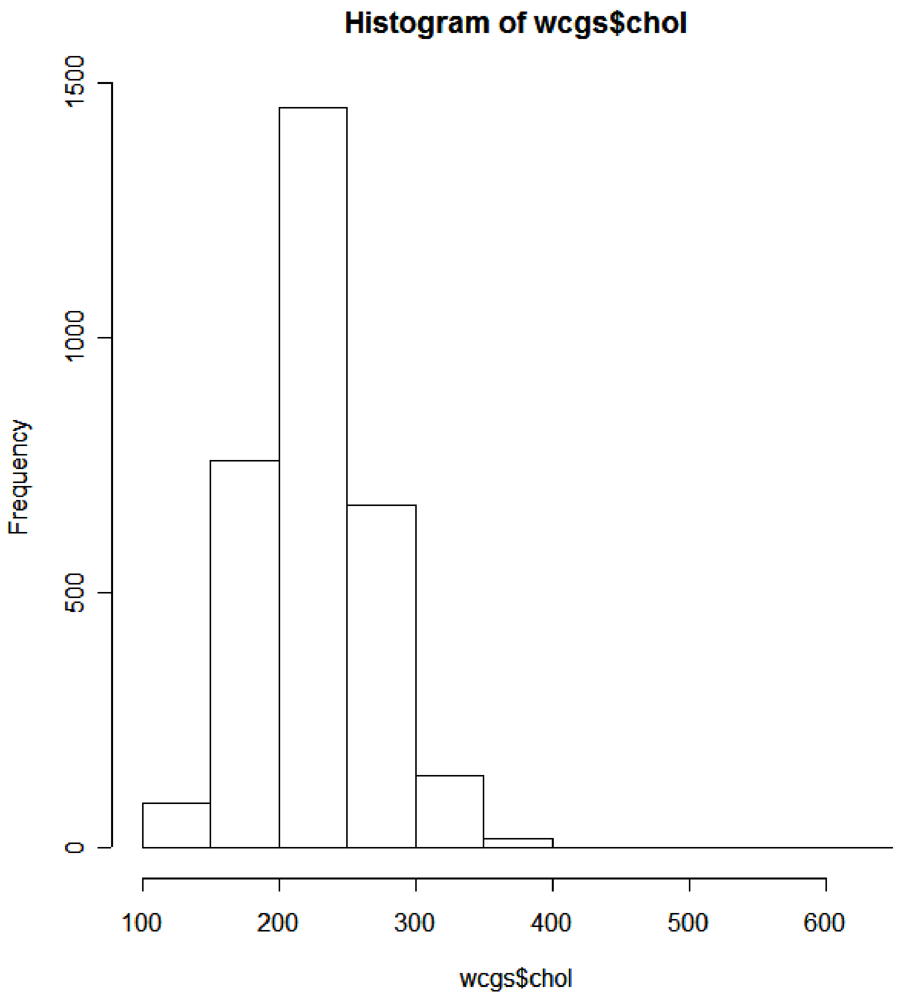


Figure 8 Example of semi-logarithmic-scale line graph

```
wcgs <- read.table('http://www.ucbcidp.org/data/wcgsdata.csv',
     header=TRUE, sep=',')
hist(wcgs$chol)
```

Frequency polygons

Cumulative frequency and survival curves

Scatter diagrams

8.2 Charts

Bar charts

Groups bar charts

Deviation bar charts

Proportional component bar charts

Pie charts

Dot plots and box plots

Maps

8.3 Miscellaneous

locator
identify

Appendix A Data sets

Data sets in R

Western Collaborative Group Study

http://www.medepi.net/data/wcgsdata.csv

http://www.crcpress.com/e_products/downloads/

Appendix B Regular Expressions as used in R

Description

This help page documents the regular expression patterns supported by grep and related functions regexpr, sub and gsub, as well as by strsplit.

This is preliminary documentation.

Details

A ‘regular expression’ is a pattern that describes a set of strings. Three types of regular expressions are used in R, extended regular expressions, used by grep(extended = TRUE) (its default), basic regular expressions, as used by grep(extended = FALSE), and Perl-like regular expressions used by grep(perl = TRUE).

Other functions which use regular expressions (often via the use of grep) include apropos, browseEnv, help.search, list.files, ls and strsplit. These will all use extended regular expressions, unless strsplit is called with argument extended = FALSE.

Patterns are described here as they would be printed by cat: do remember that backslashes need to be doubled in entering R character strings from the keyboard.

Extended Regular Expressions

This section covers the regular expressions allowed if extended = TRUE in grep, regexpr, sub, gsub and strsplit. They use the GNU implementation of the POSIX 1003.2 standard.

Regular expressions are constructed analogously to arithmetic expressions, by using various operators to combine smaller expressions.

The fundamental building blocks are the regular expressions that match a single character. Most characters, including all letters and digits, are regular expressions that match themselves. Any metacharacter with special meaning may be quoted by preceding it with a backslash. The metacharacters are . \ | ( ) [{ ^ $ * + ?.

A character class is a list of characters enclosed by [and] matches any single character in that list; if the first character of the list is the caret ^, then it matches any character not in the list. For example, the regular expression [0123456789] matches any single digit, and [^abc] matches anything except the characters a, b or c. A range of characters may be specified by giving the first and last characters, separated by a hyphen. (Character ranges are interpreted in the collation order of the current locale.)

Certain named classes of characters are predefined. Their interpretation depends on the locale (see locales); the interpretation below is that of the POSIX locale.

[:alnum:] Alphanumeric characters: [:alpha:] and [:digit:].

[:alpha:] Alphabetic characters: [:lower:] and [:upper:].

[:blank:] Blank characters: space and tab.

[:cntrl:] Control characters. In ASCII, these characters have octal codes 000 through 037, and 177 (DEL). In another character set, these are the equivalent characters, if any.

[:digit:] Digits: 0 1 2 3 4 5 6 7 8 9.

[:graph:] Graphical characters: [:alnum:] and [:punct:].

[:lower:] Lower-case letters in the current locale.

[:print:] Printable characters: [:alnum:], [:punct:] and space.

[:punct:] Punctuation characters: ! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [\ ] ^ _ ` { | } ~.

[:space:] Space characters: tab, newline, vertical tab, form feed, carriage return, and space.

[:upper:] Upper-case letters in the current locale.

[:xdigit:] Hexadecimal digits: 0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f.

For example, [[:alnum:]] means [0-9A-Za-z], except the latter depends upon the locale and the character encoding, whereas the former is independent of locale and character set. (Note that the brackets in these class names are part of the symbolic names, and must be included in addition to the brackets delimiting the bracket list.) Most metacharacters lose their special meaning inside lists. To include a literal], place it first in the list. Similarly, to include a literal ^, place it anywhere but first. Finally, to include a literal -, place it first or last. (Only these and \ remain special inside character classes.)

The period . matches any single character. The symbol \w is documented to be synonym for [[:alnum:]] and \W is its negation. However, \w also matches underscore in the GNU grep code used in R.

The caret ^ and the dollar sign $ are metacharacters that respectively match the empty string at the beginning and end of a line. The symbols \< and \> respectively match the empty string at the beginning and end of a word. 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.

A regular expression may be followed by one of several repetition quantifiers:

? The preceding item is optional and will be matched at most once.

* The preceding item will be matched zero or more times.

+ The preceding item will be matched one or more times.

{n} The preceding item is matched exactly n times.

{n,} The preceding item is matched n or more times.

{n,m} The preceding item is matched at least n times, but not more than m times.

Repetition is greedy, so the maximal possible number of repeats is used.

Two regular expressions may be concatenated; the resulting regular expression matches any string formed by concatenating two substrings that respectively match the concatenated subexpressions.

Two regular expressions may be joined by the infix operator |; the resulting regular expression matches any string matching either subexpression. For example, abba|cde matches either the string abba or the string cde. Note that alternation does not work inside character classes, where | has its literal meaning.

Repetition takes precedence over concatenation, which in turn takes precedence over alternation. A whole subexpression may be enclosed in parentheses to override these precedence rules.

The backreference \N, where N is a single digit, matches the substring previously matched by the Nth parenthesized subexpression of the regular expression.

The current code attempts to support traditional usage by assuming that { is not special if it would be the start of an invalid interval specification. (POSIX allows this behaviour as an extension but we advise users not to rely on it.)

Basic Regular Expressions

This section covers the regular expressions allowed if extended = FALSE in grep, regexpr, sub, gsub and strsplit.

In basic regular expressions the metacharacters ?, +, {, |, (, and) lose their special meaning; instead use the backslashed versions \?, \+, \ {, \|, \(, and \). Thus the metacharacters are . \ [^ $ *.

Perl Regular Expressions

The perl = TRUE argument to grep, regexpr, sub and gsub switches to the PCRE library that implements regular expression pattern matching using the same syntax and semantics as Perl 5, with just a few differences. Character tables created in the C locale at compile time are used in this version, but locale-specific tables will be used in later versions of R.

For complete details please consult the man pages for PCRE (especially man pcrepattern or if that does not exist, man pcre) on your system or from the sources at ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/. If PCRE support was compiled from the sources within R, the PCRE version is 3.9 as described here: PCRE >= 4.0 supports more of the Perl regular expressions.

All the regular expressions described for extended regular expressions are accepted except \< and \>: in Perl all backslashed metacharacters are alphanumeric and backslashed symbols always are interpreted as a literal character. { is not special if it would be the start of an invalid interval specification. There can be more than 9 backreferences.

The construct (?...) is used for Perl extensions in a variety of ways depending on what immediately follows the ?.

Perl-like matching can work in several modes, set by the options (?i) (caseless, equivalent to Perl's /i), (?m) (multiline, equivalent to Perl's /m), (?s) (single line, so a dot matches all characters, even new lines: equivalent to Perl's /s) and (?x) (extended, whitespace data characters are ignored unless escaped and comments are allowed: equivalent to Perl's /x). These can be concatenated, so for example, (?im) sets caseless multiline matching. It is also possible to unset these options by preceding the letter with a hyphen, and to combine setting and unsetting such as (?im-sx). These settings can be applied within patterns, and then apply to the remainder of the pattern. Additional options not in Perl include (?U) to set ‘ungreedy’ mode (so matching is minimal unless ? is used, when it is greedy). Initially none of these options are set.

The escape sequences \d, \s and \w represent any decimal digit, space character and and ‘word’ character (letter, digit or underscore in the current locale) respectively, and their upper-case versions represent their negation. In PCRE 3.9 the vertical tab is not regarded as a whitespace character, but it is in PCRE >= 4.0. (Perl itself changed around version 5.004.)

Escape sequence \a is BEL, \e is ESC, \f is FF, \n is LF, \r is CR and \t is TAB. In addition \cx is cntrl-x for any x, \ddd is the octal character ddd (for up to three digits unless interpretable as a backreference), and \xhh specifies a character in hex.

Outside a character class, \b matches a word boundary, \B is its negation, \A matches at start of subject (even in multiline mode, unlike ^), \Z matches at end of a subject or before newline at end, \z matches at end of a subject. and \G matches at first matching position in a subject. \C matches a single byte. including a newline.

The same repetition quantifiers as extended POSIX are supported. However, if a quantifier is followed by ?, the match is 'ungreedy', that is as short as possible rather than as long as possible (unless the meanings are reversed by the (?U) option.)

The sequence (?# marks the start of a comment which continues up to the next closing parenthesis. Nested parentheses are not permitted. The characters that make up a comment play no part at all in the pattern matching.

If the extended option is set, an unescaped # character outside a character class introduces a comment that continues up to the next newline character in the pattern.

The pattern (?:...) groups characters just as parentheses do but does not make a backreference.

Patterns (?=...) and (?!...) are zero-width positive and negative lookahead assertions: they match if an attempt to match the ... forward from the current position would succeed (or not), but use up no characters in the string being processed. Patterns (?<=...) and (?<!...) are the lookbehind equivalents: they do not allow repetition quantifiers nor \C in ....

Named subpatterns, atomic grouping, possessive qualifiers and conditional and recursive patterns are not covered here.

Author(s)

This help page is based on the documentation of GNU grep 2.4.2, from which the C code used by R has been taken, the pcre man page from PCRE 3.9 and the pcrepattern man page from PCRE 4.4.

See Also

grep, apropos, browseEnv, help.search, list.files, ls and strsplit.