R Cookbook

by Paul Teetor

S Kwon, Department of Applied Statistics, Konkuk University

shkwon0522@konkuk.ac.kr

Chapter 1
Getting Started and Help

This chapter sets the groundwork for the other chapters. It explains how to download, install, and run R.

1.1 Downloading and Installing R

- Windows
  1. Open http://www.r-project.org/ in your browser.
  2. Click on CRAN? You'll see a list of mirror sites, organized by country.
  3. Select a site near you.
  4. Click on indows?under ownload and Install R?
  5. Click on ase?
  6. Click on the link for downloading the latest version of R (an .exe file).
  7. When the download completes, double-click on the .exe file and answer the usual questions.

1.3 Entering Commands

- ```
  > 1+1
  [1] 2
  ```

- ```
  > max(1,3,5)
  [1] 5
  ```

- ```
  > max(1,3,5)
  [1] 5
  ```
  ```

1.7 Getting Help on a Function

- Use **help** to display the documentation for the function:

```
> help(mean)
```

- Use **args** for a quick reminder of the function arguments:

```
> args(mean)
function (x, ...)
NULL
```

- Use **example** to see examples of using the function:

```
> example(mean)
mean> x <- c(0:10, 50)
mean> xm <- mean(x)
mean> c(xm, mean(x, trim = 0.10))
[1] 8.75 5.50
```

1.8 Searching the Supplied Documentation

- If you want to search the installed documentation for a keyword,

```
> help.search("pattern")
> ??pattern
```

1.9 Getting Help on a Package

- `> help(package="tseries")`
- `> vignette()`
  `> vignette(package="car")`

1.11 Finding Relevant Functions and Packages

- Of the 2,000+ packages for R, you have no idea which ones would be useful to you.
  - Visit the list of task views at http://cran.r-project.org/web/views/.
  - Find and read the task view for your area, which will give you links to and descriptions of relevant packages.
  - Or visit http://rseek.org, search by keyword, click on the Task Views tab, and select an applicable task view.
  - Visit http://crantastic.org/ and search for packages by keyword.
  - To find relevant functions, visit http://rseek.org, search by name or keyword, and click on the Functions tab.

Chapter 2
Some Basics

The recipes in this chapter lie somewhere between problem-solving ideas and tutorials.

2.1 Printing Something

- If you simply enter the variable name or expression at the command prompt, R will print its value.
- **print** function

```
> pi
[1] 3.141593
> sqrt(2)
[1] 1.414214

> print(pi)
[1] 3.141593
> print(sqrt(2))
[1] 1.414214
```

2.1 Printing Something

- Trying to print multiple items gives this mind-numbing error message:

```
> print("The zero occurs at", 2*pi, "radians.")
Error in print.default("The zero occurs at", 2 * pi,
"radians.") : unimplemented type 'character' in
'asLogical'
```

- The only way to print multiple items is to print them one at a time.

```
> print("The zero occurs at"); print(2*pi)
[1] "The zero occurs at"
[1] 6.283185
```

2.1 Printing Something

- **cat** function

```
> fib <- c(0,1,1,2,3,5,8,13,21,34)
> cat("radians.", fib, "...\n")
radians. 0 1 1 2 3 5 8 13 21 34 ...
```

- A serious limitation, however, is that it cannot print compound data structures such as matrices and lists.

```
> cat(list("a","b","c"))
Error in cat(list(...), file, sep, fill, labels,
append) : argument 1 (type 'list') cannot be
handled by 'cat'
```

2.2 Setting Variables

- If you want to save a value in a variable, use the assignment operator $<-$ or $=$.

```
> x <- 3
> y <- 4
> z <- sqrt(x^2 + y^2)
> z
[1] 5
```

2.3 Listing Variables

- **ls** function

```
> x <- 10; y <- 50
> z <- c("three"); f <- function(a){a+1}
> ls()
[1] "f" "x" "y" "z"
```

- **ls.str**

```
> ls.str()
f : function (a)
x :  num 10
y :  num 50
z :  chr "three"
```

2.4 Deleting Variables

- **rm** function.

```
> ls()
[1] "f" "x" "y" "z"
> rm(x)
> ls()
[1] "f" "y" "z"
> rm(list=ls())
> ls()
character(0)
```

2.5 Creating a Vector

- The **c** operator can construct a vector from simple elements:

```
> c(1,1,2,3,5,8,13,21)
[1]  1  1  2  3  5  8 13 21
> c(1*pi, 2*pi, 3*pi, 4*pi)
[1]  3.141593  6.283185  9.424778 12.566371
> c("Everyone", "loves", "stats.")
[1] "Everyone" "loves"    "stats."
> c(TRUE,TRUE,FALSE,TRUE)
[1]  TRUE  TRUE FALSE  TRUE
```

2.5 Creating a Vector

- If the arguments to **c** are themselves vectors, it flattens them and combines them into one single vector:

```
> v1 <- c(1, 2, 3); v2 <- c(4, 5, 6)
> c(v1, v2)
[1] 1 2 3 4 5 6
```

- Vectors cannot contain a mix of data types, such as numbers and strings.

```
> v1 <- c(1, 2); v3 <- c("A", "B")
> c(v1, v3)
[1] "1" "2" "A" "B"
```

2.5 Creating a Vector

- Technically speaking, two data elements can coexist in a vector only if they have the same mode.

```
> mode(3.1415)
[1] "numeric"
> mode("foo")
[1] "character"
```

- Those modes are incompatible. To make a vector from them, R converts 3.1415 to character mode so it will be compatible with "foo":

```
> c(3.1415, "foo")
[1] "3.1415" "foo"
> mode(c(3.1415, "foo"))
[1] "character"
```

2.6 Computing Basic Statistics

- Calculate basic statistics: mean, median, standard deviation, variance, correlation, or covariance.

```
> x <- c(0,1,1,2,3,5,8,13,21,34)
> mean(x)
[1] 8.8
> median(x)
[1] 4
> sd(x)
[1] 11.03328
> var(x)
[1] 121.7333
```

2.6 Computing Basic Statistics

- **cor** and **cov** functions

```
> x <- c(0,1,1,2,3,5,8,13,21,34)
> y <- log(x+1)
> cor(x, y)
[1] 0.9068053
> cov(x, y)
[1] 11.49988
```

2.7 Creating Sequences

- **n:m**

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

- **seq** function

```
> seq(from=1, to=5, by=2)
[1] 1 3 5
```

- **rep** function

```
> rep(1, times=5)
[1] 1 1 1 1 1
```

2.7 Creating Sequences

- Alternatively, you can specify a length for the output sequence and then R will calculate the necessary increment:

```
> seq(from=0, to=20, length.out=5)
[1]  0  5 10 15 20
> seq(from=0, to=100, length.out=5)
[1]   0  25  50  75 100

> seq(from=1.0, to=2.0, length.out=5)
[1] 1.00 1.25 1.50 1.75 2.00
```

2.8 Comparing Vectors

- R has two logical values, TRUE and FALSE.

```
> a <- 3
> pi
[1] 3.141593
> a == pi
[1] FALSE
> a == pi      # Test for equality
[1] FALSE
> a != pi      # Test for inequality
[1] TRUE
```

2.8 Comparing Vectors

- More details: Read textbook!

```
> a < pi
[1] TRUE
> a > pi
[1] FALSE
> a <= pi
[1] TRUE
> a <= pi      # less or equal than
[1] TRUE
> a >= pi      # greater or eqaul than
[1] FALSE
```

2.9 Selecting Vector Elements

- If you want to extract one or more elements from a vector,

```
> fib <- c(0,1,1,2,3,5,8,13,21,34)
> fib
[1]  0  1  1  2  3  5  8 13 21 34
> fib[1]
[1] 0
> fib[4]
[1] 2
> fib[4:9]       # Select elements 4 through 9
[1]  2  3  5  8 13 21
> fib[c(1,2,4,8)]
[1]  0  1  2 13
> fib[-1]        # Ignore first element
[1]  1  1  2  3  5  8 13 21 34
```

2.9 Selecting Vector Elements

- By combining vector comparisons, logical operators, and vector indexing, you can perform powerful selections with very little R code:

```
> fib <- c(0,1,1,2,3,5,8,13,21,34)
> fib[fib>median(fib)]
[1]  5  8 13 21 34
> fib[(fib<quantile(fib,0.05))|(fib>quantile(fib,0.95))]
[1]  0 34
> fib[abs(fib-mean(fib))>2*sd(fib)]
[1] 34
```

2.10 Performing Vector Arithmetic

- If you want to operate on an entire vector at once,

```
> v <- c(11, 12, 13)
> w <- c(1, 2, 3)
> v+w
[1] 12 14 16
> v-w
[1] 10 10 10
> v*w
[1] 11 24 39
> v/w
[1] 11.000000  6.000000  4.333333
> w^v
[1]       1    4096 1594323
```

2.10 Performing Vector Arithmetic

- You can recenter an entire vector in one expression simply by subtracting the mean of its contents:

```
> w <- 1:5
> mean(w)
[1] 3
> w-mean(w)
[1] -2 -1  0  1  2
```

- Likewise, you can calculate the z-score of a vector in one expression: subtract the mean and divide by the standard deviation:

```
> w
[1] 1 2 3 4 5
> (w-mean(w))/sd(w)
[1] -1.2649111 -0.6324555  0.0000000  0.6324555
1.2649111
```

2.10 Performing Vector Arithmetic

- vector-level operations

```
> sqrt(w)
[1] 1.000000 1.414214 1.732051 2.000000 2.236068
> log(w)
[1] 0.0000000 0.6931472 1.0986123 1.3862944
1.6094379
> sin(w)
[1]  0.8414710  0.9092974  0.1411200 -0.7568025
-0.9589243
```

2.12 Defining a Function

- **function**

```
> cv <- function(x){sd(x)/mean(x)}
> cv(1:10)
[1] 0.5504819
> f.fun <- function(a,b){
+  if(b==0) return(a)
+  else return(a+1/b)
+ }
> f.fun(1,0)
[1] 1
> f.fun(1,1)
[1] 2
```

Chapter 3
Navigating the Software

There is nothing here about numerics, statistics, or graphics. This is all about dealing with R as software.

3.1 Getting and Setting the Working Directory

- Use **getwd** to report the working directory, and use **setwd** to change it:

```
> getwd()
[1] "C:/Users/User/Documents"
> setwd("Newfolder")
> getwd()
[1] "C:/Users/User/Documents/Newfolder"
```

3.2 Saving Your Workspace

- **save.image** function:

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

3.3 Viewing Your Command History

- **up arrow** or **Ctrl-P** or use the **history** function:

```
> history()
> history(100) # Show 100 most recent lines of history
> history(Inf) # Show entire saved history
```

3.4 Saving the Result of the Previous Command

- A special variable called **.Last.value** saves the value of the most recently evaluated expression.

```
> sqrt(2)
[1] 1.414214
> x <- .Last.value
> x
[1] 1.414214
```

3.5 Displaying the Search Path

- **search** function

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

3.6 Accessing the Functions in a Package

- **library** function or the **require** function

```
> library(MASS)
> Iris <- data.frame(rbind(iris3[,,1], iris3[,,2], iris3[,,3]),
+                    Sp = rep(c("s","c","v"), rep(50,3)))
> z <- lda(Sp ~ ., Iris, prior = c(1,1,1)/3)
```

- The **detach** function will unload a package that is currently loaded:

```
> detach(package:MASS)
> z <- lda(Sp ~ ., Iris, prior = c(1,1,1)/3)
Error: could not find function "lda"
```

3.7 Accessing Built-in Datasets

- You can use the built-in dataset called **pressure**:

```
> head(pressure)
temperature pressure
1           0   0.0002
2          20   0.0012
3          40   0.0060
4          60   0.0300
5          80   0.0900
6         100   0.2700
```

- If you want to know more about **pressure**, use the **help** function to learn about it and other datasets:

```
> help(pressure)
```

3.7 Accessing Built-in Datasets

- You can see a table of contents for **datasets** by calling the **data** function with no arguments:

```
> data()
```

- **MASS** includes a dataset called **Cars93**, which you can access in this way:

```
> data(Cars93, package="MASS")
```

- After this call to **data**, the **Cars93** dataset is available to you; then you can execute **summary(Cars93)**, **head(Cars93)**, and so forth.

```
> summary(Cars93)
> head(Cars93)
```

3.7 Accessing Built-in Datasets

- You can see a list of available datasets in **MASS**, or any other package, by using the **data** function with a package argument and no dataset name:

```
> data(package="MASS")
```

3.8 Viewing the List of Installed Packages

- **library** function
- **installed.packages**

```
> library()
> installed.packages()[,c("Package","Version")]
```

3.9 Installing Packages from CRAN

- Use the **install.packages** function, putting the name of the package in quotes:

```
> # Installing the package "ts"
> install.packages("ts")
```

3.12 Running a Script

- **source** function

```
> source("hello.R")
[1] "Hello, World!"
```

- Setting **echo=TRUE** will echo the script lines before they are executed, with the R prompt shown before each line:

```
> source("hello.R", echo=TRUE)
> print("Hello, World!")
[1] "Hello, World!"
```

4.1 Entering Data from the Keyboard

- Enter the data as literals using the **c** constructor for vectors:

```
> score <- c(61, 66, 90, 88, 100)
> score
[1]  61  66  90  88 100
```

- Alternatively, you can create an empty data frame and then invoke the built-in, spreadsheet-like editor to populate it:

```
> score <- data.frame() # Create empty data frame
> score <- edit(score) # Invoke editor, overwrite with edited data
```

4.1 Entering Data from the Keyboard

- The function **data.frame** works for data frames, too, by entering each variable (column) as a vector

```
> points <- data.frame(
+    label=c("Low", "Mid", "High"),
+    lbound=c( 0, 0.67, 1.64),
+    ubound=c(0.674, 1.64, 2.33)
+    )
> points
label lbound ubound
1   Low   0.00  0.674
2   Mid   0.67  1.640
3  High   1.64  2.330
```

4.2 Printing Fewer Digits (or More Digits)

- R normally formats floating-point output to have seven digits:

```
> pi
[1] 3.141593
> 100*pi
[1] 314.1593
```

- You can also alter the format of all output by using the **options** function to change the default for digits.

```
> options(digits=15)
> pi
[1] 3.14159265358979
```

4.2 Printing Fewer Digits (or More Digits)

- The **print** function lets you vary the number of printed digits using the digits parameter:

```
> print(pi, digits=4)
[1] 3.142
> print(100*pi, digits=4)
[1] 314.2
```

- Use the **format** function to format your numbers before calling **cat**:

```
> cat(pi, "\n")
3.141593
> cat(format(pi,digits=4), "\n")
3.142
```

4.2 Printing Fewer Digits (or More Digits)

- Both **print** and **format** will format entire vectors at once:

```
> pnorm(-3:3)
[1] 0.001349898 0.022750132 0.158655254 0.500000000
[5] 0.841344746 0.977249868 0.998650102
> print(pnorm(-3:3), digits=3)
[1] 0.00135 0.02275 0.15866 0.50000 0.84134 0.97725
[7] 0.99865
```

- The **print** finds the number of digits necessary to format the smallest number and then formats all numbers to have the same width:

```
> q <- seq(from=0,to=3,by=0.5)
> tbl <- data.frame(Quant=q, Lower=pnorm(-q), Upper=pnorm(q))
> print(tbl) # Unformatted print
> print(tbl,digits=2) # Formatted print: fewer digits
```

4.3 Redirecting Output to a File

- You can redirect the output of the **cat** function by using its file argument:

```
> answer = 1e+10
> cat("The answer is", answer, "\n", file="output.txt")
> getwd()
```

- Use the **sink** function to begin redirecting all console output from both **print** and **cat**:

```
> sink("output.txt") # Redirect output to file
> cat(answer, file="output.txt")
> cat(answer, file="output.txt", append=TRUE)
> cat(answer,"\n", file="output.txt", append=TRUE)
> cat(answer, file="output.txt", append=TRUE)
> cat(answer, file="output.txt")
#> source("script.R") # Run the script, capturing its output
> sink() # Resume writing output to console
```

4.3 Redirecting Output to a File

- The **file** function opens a connection to a file and writing your output to the connection. If your are done use the **close** function to disconnect it:

```
> ?file
> con <- file("long_and_complex_name.txt", "w") # Open a connection
> cat(answer, file=con)
> cat(answer*100, file=con)
> cat(answer/100, file=con)
> close(con) # Close the connection
```

4.4 Listing Files

- The **list.files** function shows the contents of your working directory.

```
> list.files()
```

- To see all the files in your subdirectories, use **recursive** argument:

```
> list.files(recursive=TRUE)
```

- A possible "gotcha" of **list.files** is that it ignores hidden files and then try **all.files** argument:

```
> list.files(all.files=TRUE)
```

4.5 Dealing with "Cannot Open File"

- You can open data files by using many functions built in R but problems arise when the name contains backslashes (\):

```
> samp <- read.csv("C:\Data\sample-data.csv")
Error: '\D' is an unrecognized escape ...
```

- Use forward slashes instead of backslashes:

```
> samp <- read.csv("C:/Data/sample-data.csv")
Error in file(file, "rt") : cannot open the connection
In addition: Warning message:
In file(file, "rt") :
cannot open file 'C:/Data/sample-data.csv': No such file or directory
```

- An alternative solution is to double the backslashes:

```
> samp <- read.csv("C:\\Data\\sample-data.csv")
Error in file(file, "rt") : cannot open the connection
...
```

4.6 Reading Fixed-Width Records

- Suppose we want to read an entire file of fixed-width records in fixed-width.txt, shown here:

```
Fisher     R.A.       1890 1962
Pearson    Karl       1857 1936
Cox        Gertrude   1900 1978
Yates      Frank      1902 1994
Smith      Kirstine   1878 1939
```

- We need to know the column widths and use the **read.fwf** function:

```
> records <- read.fwf("fixed-width.txt", widths=c(10,10,5,4))
```

- The -1 in the widths argument says there is a one-character column that should be ignored:

```
> records <- read.fwf("fixed-width.txt", widths=c(6,-4,8,-2,4,-1,4))
```

4.6 Reading Fixed-Width Records

- R supplied some funky, synthetic column names. We can override that default by using a **col.names** argument.

```
> records <- read.fwf("fixed-width.txt", widths=c(6,-4,8,-2,4,-1,4),
+ col.names=c("Last","First","Born","Died"))
> records
Last    First Born Died
1 Fisher R.A.     1890 1962
2 Pearso Karl     1857 1936
3 Cox    Gertrude 1900 1978
4 Yates  Frank    1902 1994
5 Smith  Kirstine 1878 1939
```

4.7 Reading Tabular Data Files

- Here are records in statisticians.txt, using a space character between fields.

```
Fisher R.A. 1890 1962
Pearson Karl 1857 1936
Cox Gertrude 1900 1978
Yates Frank 1902 1994
Smith Kirstine 1878 1939
```

4.7 Reading Tabular Data Files

- The **read.table** function is built to read this file. By default, it assumes the data fields are separated by white space (blanks or tabs).

```
> dfrm <- read.table("statisticians.txt")
> print(dfrm)
V1       V2   V3   V4
1  Fisher     R.A. 1890 1962
2 Pearson     Karl 1857 1936
3     Cox Gertrude 1900 1978
4   Yates    Frank 1902 1994
5   Smith Kirstine 1878 1939
```

- If our file used colon (:) as the field separator, we would read it this way.

```
> dfrm <- read.table("statisticians-colon.txt") # not separated
> dfrm <- read.table("statisticians-colon.txt", sep=":") # separated
```

4.7 Reading Tabular Data Files

- Use the **str** function to see the class of the resulting column.

```
> str(dfrm)
'data.frame':   5 obs. of  4 variables:
$ V1: Factor w/ 5 levels "Cox","Fisher",..: 2 3 1 5 4
$ V2: Factor w/ 5 levels "Frank","Gertrude",..: 5 3 2 1 4
$ V3: int  1890 1857 1900 1902 1878
$ V4: int  1962 1936 1978 1994 1939
```

- Set the **stringsAsFactors** argument to FALSE:

```
> dfrm <- read.table("statisticians.txt", stringsAsFactor=FALSE)
> str(dfrm)
'data.frame':   5 obs. of  4 variables:
$ V1: chr  "Fisher" "Pearson" "Cox" "Yates" ...
$ V2: chr  "R.A." "Karl" "Gertrude" "Frank" ...
...
```

4.7 Reading Tabular Data Files

- Your data file might employ a different string to signal missing values, in which case use the **na.strings** argument:

```
dfrm <- read.table("statisticians.txt", na.strings="1890")
> dfrm
V1       V2   V3   V4
1  Fisher     R.A.   NA 1962
2 Pearson     Karl 1857 1936
3     Cox Gertrude 1900 1978
4   Yates    Frank 1902 1994
5   Smith Kirstine 1878 1939
```

4.7 Reading Tabular Data Files

- Any line that begins with a pound sign (#) is ignored, so you can put comments on those lines.

```
# This is a data file of famous statisticians.
# Last edited on 1994-06-18
lastname firstname born died
Fisher R.A. 1890 1962
Pearson Karl 1857 1936
Cox Gertrude 1900 1978
Yates Frank 1902 1994
Smith Kirstine 1878 1939
```

4.7 Reading Tabular Data Files

- Now we can tell read.table that our file contains a header line, and it will use the column names when it builds the data frame.

```
> dfrm <- read.table("statisticians-header.txt", header=TRUE)
> print(dfrm)
lastname firstname born died
1  Fisher      R.A. 1890 1962
2 Pearson      Karl 1857 1936
3     Cox  Gertrude 1900 1978
4   Yates     Frank 1902 1994
5   Smith  Kirstine 1878 1939
```

4.8 Reading from CSV Files

- The file table-data.csv includes a very simple comma-separated-values (CSV) where the first line is a header line that contains the column names, also separated by commas.

```
label,lbound,ubound
low,0,0.674
mid,0.674,1.64
high,1.64,2.33
```

- The **read.csv** function can read CSV files.

```
> tbl <- read.csv("table-data.csv")
> str(tbl)
> tbl <- read.csv("table-data.csv", as.is=TRUE)
> str(tbl)
> tbl <- read.csv("table-data.csv", header=FALSE)
> str(tbl)
```

4.8 Reading from CSV Files

- The label variable in the tbl data frame just shown is actually a factor, not a character variable.You see that by inspecting the structure of tbl.

```
> str(tbl)
'data.frame': 3 obs. of 3 variables:
$ label : Factor w/ 3 levels "high","low","mid": 2 3 1
$ lbound: num 0 0.674 1.64
$ ubound: num 0.674 1.64 2.33
```

- In that case, set the as.is parameter to TRUE; this indicates that R should not interpret nonnumeric data as a factor.

```
> tbl <- read.csv("table-data.csv", as.is=TRUE)
> str(tbl)
'data.frame': 3 obs. of 3 variables:
$ label : chr "low" "mid" "high"
$ lbound: num 0 0.674 1.64
$ ubound: num 0.674 1.64 2.33
```

4.9 Writing to CSV Files

- The write.csv function can write a CSV file.

```
> write.csv(x, file="filename", row.names=FALSE, col.names=TRUE )
```

4.10 Reading Tabular or CSV Data from the Web

- Use the read.csv, read.table, and scan functions, but substitute a URL for a file name.

```
> tbl <- read.csv("http://www.example.com/
+ download/data.csv")
```

- Remember that URLs work for FTP servers, not just HTTP servers. This means that R can also read data from FTP sites using URLs.

```
> tbl <- read.table("ftp://ftp.example.com/
+ download/data.txt")
```

4.11 Reading Data from HTML Tables

- Use the readHTMLTable function in the XML package.

```
> library(XML)
> url <- 'http://www.example.com/data/table.html'
> tbls <- readHTMLTable(url)
```

- To read only specific tables, use the which parameter. This example reads the third table on the page.

```
> tbl <- readHTMLTable(url, which=3)
```

4.11 Reading Data from HTML Tables

- The following example, which is taken from the help page for readHTMLTable, loads all tables from the Wikipedia page entitled World population.

```
> library(XML)
> url <- 'http://en.wikipedia.org/wiki/
+ World_population'
> tbls <- readHTMLTable(url)
```

- As it turns out, that page contains 17 tables.

```
> length(tbls)
[1] 17
```

- In this example we care only about the third table (which lists the largest populations by country), so we specify which=3.

```
> tbl <- readHTMLTable(url, which=3)
```

4.11 Reading Data from HTML Tables

- In that table, columns 2 and 3 contain the country name and population, respectively.

```
> tbl[,c(2,3)]
Country / Territory    Population
1  People's Republic of China[44] 1,338,460,000
2                           India 1,182,800,000
3                   United States   309,659,000
4                       Indonesia   231,369,500
5                          Brazil   193,152,000
6                        Pakistan   169,928,500
7                      Bangladesh   162,221,000
8                         Nigeria   154,729,000
9                          Russia   141,927,297
10                          Japan   127,530,000
```

4.14 Saving and Transporting Objects

- Write the objects to a file using the save function.

```
> save(myData, file="myData.RData")
```

- Read them back using the load function, either on your computer or on any platform that supports R.

```
> load("myData.RData")
```

- The save function writes binary data. To save in an ASCII format, use dput or dump instead.

```
> dput(myData, file="myData.txt")
> dump("myData", file="myData.txt")
# Note quotes around variable name
```

4.14 Saving and Transporting Objects

- The first time I used load, I did this:

```
> myData <- load("myFile.RData")
# Achtung! Might not do what you think
```

4.14 Saving and Transporting Objects

- suppose we save the object in a file called z.RData. The following sequence of functions will create some confusion.

```
> load("z.RData")
# Create and populate the z variable
> plot(z)
# Does not plot what we expected: zoo pkg not loaded
```

- We should have loaded the zoo package before printing or plotting any zoo objects, like this:

```
> library(zoo) # Load the zoo package into memory
> load("z.RData")
# Create and populate the z variable
> plot(z) # Ahhh. Now plotting works correctly
```

6.1 Splitting a Vector into Groups

- Suppose x (vector) includes values that correspond groups in f (factor). You can use the **split** function:

```
> x = c(40,35,25,15,50,40,25,20)
> f = c("a","c","b","b","a","c","c","a")
> groups <- split(x, f)
```

- Alternatively, you can use the **unstack** function:

```
> groups <- unstack(data.frame(x,f))
```

6.1 Splitting a Vector into Groups

- The **Cars93** dataset contains a factor called textbfOrigin that has two levels, **USA** and **non-USA**. It also contains a column called **MPG.city**. We can split the MPG data according to origin as follows:

```
> library(MASS)
> split(Cars93$MPG.city, Cars93$Origin)
$USA
[1] 22 19 16 19 16 16 25 25 19 21 18 15 17 17 20 23 20 29 23 22 17 21 18 29 20
[26] 31 23 22 22 24 15 21 18 17 18 23 19 24 23 18 19 23 31 23 19 19 19 28
$non-USA
[1] 25 18 20 19 22 46 30 24 42 24 29 22 26 20 17 18 18 29 28 26 18 17 20 19 29
[26] 18 29 24 17 21 20 33 25 23 39 32 25 22 18 25 17 21 18 21 20
```

6.2 Applying a Function to Each List Element

- Use either the **lapply** function or the **sapply** function, depending upon the desired form of the result. **lapply** always returns the results in list, whereas **sapply** returns the results in a vector if that is possible:

```
> lst <- lapply(lst, fun)
> vec <- sapply(lst, fun)
```

6.2 Applying a Function to Each List Element

- Lets say I teach an introductory statistics class four times and administer comparable final exams each time. Here are the exam scores from the four semesters:

```
> scores
$S1
[1] 89 85 85 86 88 89 86 82 96 85 93 91 98 87 94 77 87 98 85 89
[21] 95 85 93 93 97 71 97 93 75 68 98 95 79 94 98 95
$S2
[1] 60 98 94 95 99 97 100 73 93 91 98 86 66 83 77
[16] 97 91 93 71 91 95 100 72 96 91 76 100 97 99 95
[31] 97 77 94 99 88 100 94 93 86
$S3
[1] 95 86 90 90 75 83 96 85 83 84 81 98 77 94 84 89 93 99 91 77
[21] 95 90 91 87 85 76 99 99 97 97 97 77 93 96 90 87 97 88
$S4
[1] 67 93 63 83 87 97 96 92 93 96 87 90 94 90 82 91 85 93 83 90
[21] 87 99 94 88 90 72 81 93 93 94 97 89 96 95 82 97
```

6.2 Applying a Function to Each List Element

- **length** function: **lapply** will return a list of lengths, and **sapply** will return a vector of lengths:

```
> lapply(scores, length)
$S1
[1] 36
$S2
[1] 39
$S3
[1] 38
$S4
[1] 36
> sapply(scores, length)
S1 S2 S3 S4
36 39 38 36
```

6.2 Applying a Function to Each List Element

- We can see the mean and standard deviation of the scores just as easily:

```
> sapply(scores, mean)
S1       S2       S3       S4
88.77778 89.79487 89.23684 88.86111
> sapply(scores, sd)
S1        S2       S3       S4
7.720515 10.543592 7.178926 8.208542
```

- If the called function returns a vector, **sapply** will form the results into a matrix. The **range** function, for example, returns a two-element vector:

```
> sapply(scores, range)
S1  S2 S3 S4
[1,] 68  60 75 63
[2,] 98 100 99 99
```

6.3 Applying a Function to Every Row

- Use the **apply** function. Set the second argument to 1 to indicate row-by-row application of a function:

```
> results <- apply(mat, 1, fun) # mat is a matrix, fun is a function
```

- The **apply** function will call **fun** once for each row, assemble the returned values into a vector, and then return that vector.
- Suppose your matrix **long** is longitudinal data. Each row contains data for one subject, and the columns contain the repeated observations over time:

```
> long
         trial1    trial2     trial3     trial4     trial5
Moe   -1.8501520 -1.406571 -1.0104817 -3.7170704 -0.2804896
Larry  0.9496313  1.346517 -0.1580926  1.6272786  2.4483321
Curly -0.5407272 -1.708678 -0.3480616 -0.2757667 -1.2177024
```

6.3 Applying a Function to Every Row

- You could calculate the average observation for each subject by applying the **mean** function to the rows. The result is a vector:

```
> apply(long, 1, mean)
Moe     Larry      Curly
-1.6529530 1.2427334 -0.8181872
```

6.3 Applying a Function to Every Row

- In the vector case, **apply** assembles the results into a matrix. The **range** function returns a vector of two elements, the minimum and the maximum, so applying it to **long** produces a matrix:

```
> apply(long, 1, range)
           Moe      Larry      Curly
[1,] -3.7170704 -0.1580926 -1.7086779
[2,] -0.2804896  2.4483321 -0.2757667
```

6.4 Applying a Function to Every Column

- For a matrix, use the **apply** function.

```
> results <- apply(mat, 2, fun)
```

- For a data frame, use the **lapply** or **sapply** functions.

```
> lst <- lapply(dfrm, fun)
> vec <- sapply(dfrm, fun)
```

6.4 Applying a Function to Every Column

- You can use **lapply** and **sapply** to process the columns, as described in Recipe 6.2:

```
> lst <- lapply(dfrm, fun) # Returns a list
> vec <- sapply(dfrm, fun) # Returns a vector
```

- The function **fun** should expect one argument: a column from the data frame.

6.4 Applying a Function to Every Column

- I often use this recipe to check the types of columns in data frames. The **batch** column of this data frame seems to contain numbers:

```
> head(batches)
batch clinic dosage   shrinkage
1     1      IL      3 -0.11810714
2     3      IL      4 -0.29932107
3     2      IL      4 -0.27651716
4     1      IL      5 -0.18925825
5     2      IL      2 -0.06804804
6     3      NJ      5 -0.38279193
```

- But printing the classes of the columns reveals it to be a factor instead:

```
> sapply(batches, class)
batch    clinic    dosage  shrinkage
"factor" "factor" "integer" "numeric"
```

6.5 Applying a Function to Groups of Data

- use the **tapply** function, which will apply a function to each group of data:

```
> tapply(x, f, fun)
```

6.5 Applying a Function to Groups of Data

- Suppose I have a vector with the populations of the 16 largest cities in the greater Chicago metropolitan area, taken from the data frame called **suburbs**:

```
> attach(suburbs)
> pop
[1] 2853114 90352 171782 94487 102746 106221 147779 76031 70834
[10] 72616 74239 83048 67232 75386 63348 91452
```

- We can easily compute sums and averages for all the cities:

```
> sum(pop)
[1] 4240667
> mean(pop)
[1] 265041.7
```

6.5 Applying a Function to Groups of Data

- What if we want the sum and average broken out by county? We will need a factor, say **county**, the same length as **pop**

```
county
[1] Cook Kenosha Kane Kane     Lake(IN) Kendall DuPage Cook
[9] Will Cook    Cook Lake(IN) Cook     Cook    Cook   Lake(IL)
Levels: Cook DuPage Kane Kendall Kenosha Lake(IL) Lake(IN) Will
```

- This example shows summing the populations by county:

```
> tapply(pop,county,sum)
Cook DuPage   kane Kendall Kenosha Lake(IL) Lake(IN)  Will
3281966 147779 266269  106221   90352    91452   185794 70834
```

6.5 Applying a Function to Groups of Data

- The next example computes average populations by county:

```
> tapply(pop,county,mean)
Cook   DuPage     Kane  Kendall Kenosha Lake(IL) Lake(IN)   Will
468852.3 147779.0 133134.5 106221.0 90352.0 91452.0  92897.0 70834.0
```

- The function given to **tapply** should expect a single argument: a vector containing all the members of one group. A good example is the **length** function, which takes a vector parameter and returns the vectors length. Use it to count the number of data in each group; in this case, the number of cities in each county:

```
> tapply(pop,county,length)
Cook DuPage Kane Kendall Kenosha Lake(IL) Lake(IN) Will
7      1    2       1       1       1        2     1
```

6.6 Applying a Function to Groups of Rows

- The **by** function collects the returned values into a list and returns the list:

```
> by(dfrm, fact, fun)
```

- Here, **dfrm** is the data frame, **fact** is the grouping factor, and **fun** is a function. The function should expect one argument, a data frame.

6.6 Applying a Function to Groups of Rows

- Suppose you have a data frame from clinical trials, called **trials**, where the dosage was randomized to study its effect:

```
> trials
sex      pre dose1 dose2     post
1   F 5.931640     2     1 3.162600
2   F 4.496187     1     2 3.293989
3   M 6.161944     1     1 4.446643
4   F 4.322465     2     1 3.334748
5   M 4.153510     1     1 4.429382
. . (etc.) .
```

6.6 Applying a Function to Groups of Rows

- The data includes a factor for the subjects sex, so **by** can split the data according to sex and call **summary** for the two groups. The result is two summaries, one for men and one for women:

6.6 Applying a Function to Groups of Rows

- ```
  > by(trials, trials$sex, summary)
  trials$sex: F
  sex       pre              dose1            dose2            post
  F:7 Min.   :4.156  Min.   :1.000  Min.   :1.000  Min.   :2.886
  M:0 1st Qu.:4.409  1st Qu.:1.000  1st Qu.:1.000  1st Qu.:3.075
  Median :4.895  Median :1.000  Median :2.000  Median :3.163
  Mean   :5.020  Mean   :1.429  Mean   :1.571  Mean   :3.174
  3rd Qu.:5.668  3rd Qu.:2.000  3rd Qu.:2.000  3rd Qu.:3.314
  Max.   :5.932  Max.   :2.000  Max.   :2.000  Max.   :3.389
  ------------------------------------------------------------
  trials$sex: M
  sex       pre              dose1            dose2            post
  F:0 Min.   :3.998  Min.   :1.000  Min.   :1.000  Min.   :3.738
  M:9 1st Qu.:4.773  1st Qu.:1.000  1st Qu.:1.000  1st Qu.:3.800
  Median :5.110  Median :2.000  Median :1.000  Median :4.194
  Mean   :5.189  Mean   :1.556  Mean   :1.444  Mean   :4.148
  3rd Qu.:5.828  3rd Qu.:2.000  3rd Qu.:2.000  3rd Qu.:4.429
  Max.   :6.658  Max.   :2.000  Max.   :2.000  Max.   :4.517
  ```
  ```

7.1 Getting the length of a string

- Use the nchar function, not the length function.
- The nchar function takes a string and returns the number of characters in the string:

```
> nchar("Moe")
[1] 3
> nchar("Curly")
[1] 5
```

- If you apply nchar to a vector of strings, it returns the length of each string:

```
> s <- c("Moe", "Larry", "Curly")
> nchar(s)
[1] 3 5 5
```

7.1 Getting the length of a string

- When you apply the length function to a single string, R returns the value 1 because it views that string as a singleton vectora vector with one element:

```
> length("Moe")
[1] 1
> length(c("Moe","Larry","Curly"))
[1] 3
```

7.2 Concatenating Strings

- The paste function creates a new string by joining the given strings end to end :

```
> paste("Everybody", "loves", "stats.")
[1] "Everybody loves stats."
```

7.2 Concatenating Strings

- By default, paste inserts a single space between pairs of strings

```
> paste("Everybody", "loves", "stats.", sep="-")
[1] "Everybody-loves-stats."
> paste("Everybody", "loves", "stats.", sep="")
[1] "Everybodylovesstats."
```

7.2 Concatenating Strings

- The function is very forgiving about nonstring arguments. It tries to convert them to strings using the as.character function:

```
> paste("The square root of twice pi is approximately", sqrt(2*pi))
[1] "The square root of twice pi is approximately 2.506628274631"
```

- If one or more arguments are vectors of strings, paste will generate all combinations of the arguments:

```
> stooges <- c("Moe", "Larry", "Curly")
> paste(stooges, "loves", "stats.")
[1] "Moe loves stats." "Larry loves stats." "Curly loves stats."
```

7.2 Concatenating Strings

- The function is very forgiving about nonstring arguments. It tries to convert them to strings using the as.character function:

```
> paste("The square root of twice pi is approximately", sqrt(2*pi))
[1] "The square root of twice pi is approximately 2.506628274631"
```

- If one or more arguments are vectors of strings, paste will generate all combinations of the arguments:

```
> stooges <- c("Moe", "Larry", "Curly")
> paste(stooges, "loves", "stats.")
[1] "Moe loves stats." "Larry loves stats." "Curly loves stats."
```

7.2 Concatenating Strings

- Sometimes you want to join even those combinations into one, big string. The collapse parameter lets you define a toplevel separator and instructs paste to concatenate the generated strings using that separator:

```
> paste(stooges, "loves", "stats", collapse=", and ")
[1] "Moe loves stats, and Larry loves stats, and Curly loves stats"
```

7.3 Extracting Substrings

- The substr function takes a string, a starting point, and an ending point. It returns the substring between the starting to ending points:

```
> substr("Statistics", 1, 4) # Extract first 4 characters
[1] "Stat"
> substr("Statistics", 7, 10) # Extract last 4 characters
[1] "tics"
```

7.3 Extracting Substrings

- Just like many R functions, substr lets the first argument be a vector of strings. In that case, it applies itself to every string and returns a vector of substrings :

```
> ss <- c("Moe", "Larry", "Curly")
> substr(ss, 1, 3) # Extract first 3 characters of each string
[1] "Moe" "Lar" "Cur"
```

```
> cities <- c("New York, NY", "Los Angeles, CA","Peoria, IL")
> substr(cities, nchar(cities)-1, nchar(cities))
[1] "NY" "CA" "IL"
```

7.4 Splitting a String According to a Delimiter

- You want to split a string into substrings. The substrings are separated by a delimiter. $\rightarrow$ Use strsplit

```
> strsplit(string, delimiter)
```

7.4 Splitting a String According to a Delimiter

- It is common for a string to contain multiple substrings separated by the same delimiter. One example is a file path, whose components are separated by slashes (/):

```
> path <- "/home/mike/data/trials.csv"
```

- We can split that path into its components by using strsplit with a delimiter of /:

```
> strsplit(path, "/")
[[1]]
[1] "" "home" "mike" "data" "trials.csv"
```

7.4 Splitting a String According to a Delimiter

- Also notice that strsplit returns a list and that each element of the list is a vector of substrings.

```
> paths <- c("/home/mike/data/trials.csv",
+ "/home/mike/data/errors.csv",
+ "/home/mike/corr/reject.doc")
> strsplit(paths, "/")
[[1]]
[1] "" "home" "mike" "data" "trials.csv"
[[2]]
[1] "" "home" "mike" "data" "errors.csv"
[[3]]
[1] "" "home" "mike" "corr" "reject.doc"
```

7.5 Replacing Substrings

- Within a string, you want to replace one substring with another.
- → Use sub to replace the first instance of a substring:
- → Use gsub to replace all instances of a substring:

```
> sub(old, new, string)
> gsub(old, new, string)
```

- The sub function finds the first instance of the old substring within string and replaces it with the new substring:

```
with the new substring:
> s <- "Curly is the smart one. Curly is funny, too."
> sub("Curly", "Moe", s)
[1] "Moe is the smart one. Curly is funny, too."
```

7.5 Replacing Substrings

- gsub does the same thing, but it replaces all instances of the substring (a global replace), not just the first:

```
> gsub("Curly", "Moe", s)
[1] "Moe is the smart one. Moe is funny, too."
```

- To remove a substring altogether, simply set the new substring to be empty:

```
> sub(" and SAS", "", "For really tough problems, you need R and SAS.")
[1] "For really tough problems, you need R."
```

7.6 Seeing the Special Characters in a String

- In this example, the string seems to contain 13 characters but cat shows only a 6-character output:

```
> nchar(s)
[1] 13
> cat(s)
second
```

- The reason is that the string contains special characters, which are characters that do not display when printed. When we use print, it uses escapes (backslashes) to show the special characters:

```
> print(s)
[1] "first\rsecond\n"
```

7.7 Generating All Pairwise Combinations of Strings

- You have two sets of strings, and you want to generate all combinations from those two sets (their Cartesian product).

```
> m <- outer(strings1, strings2, paste, sep="")
```

7.7 Generating All Pairwise Combinations of Strings

- We can apply outer and paste to generate all combinations of test sites and treatments:

```
> outer(locations, treatments, paste, sep="-")
[,1] [,2] [,3]
[1,] "NY-T1" "NY-T2" "NY-T3"
[2,] "LA-T1" "LA-T2" "LA-T3"
[3,] "CHI-T1" "CHI-T2" "CHI-T3"
[4,] "HOU-T1" "HOU-T2" "HOU-T3"
```

7.7 Generating All Pairwise Combinations of Strings

- In the special case when you are combining a set with itself and order does not matter, the result will be duplicate combinations:

```
> outer(treatments, treatments, paste, sep="-")
[,1] [,2] [,3]
[1,] "T1-T1" "T1-T2" "T1-T3"
[2,] "T2-T1" "T2-T2" "T2-T3"
[3,] "T3-T1" "T3-T2" "T3-T3"
```

7.7 Generating All Pairwise Combinations of Strings

- The lower.tri function identifies that triangle, so inverting it identifies all elements outside the lower triangle:

```
> m <- outer(treatments, treatments, paste, sep="-")
> m[!lower.tri(m)]
[1] "T1-T1" "T1-T2" "T2-T2" "T1-T3" "T2-T3" "T3-T3"
```

7.8 Getting the Current Date

- You need to know todays date. → The Sys.Date function returns the current date:

```
> Sys.Date()
[1] "2010-02-11"
```

```
> class(Sys.Date())
[1] "Date"
```

7.9 Converting a String into a Date

- This example shows the default format assumed by as.Date, which is the ISO 8601 standard format of yyyy-mm-dd:

```
> as.Date("2010-12-31")
[1] "2010-12-31"
```

- I often mistakenly try to convert the usual American date format (mm/dd/yyyy) into a Date object, with these unhappy results:

```
> as.Date("12/31/2010")
Error in charToDate(x) :
character string is not in a standard unambiguous format
```

- Here is the correct way to convert an American-style date:

```
> as.Date("12/31/2010", format="%m/%d/%Y")
[1] "2010-12-31"
```

7.10 Converting a Date into a String

- You want to convert a Date object into a character string, usually because you want to print the date. → Use either format or as.character:

```
> format(Sys.Date())
[1] "2010-04-01"
> as.character(Sys.Date())
[1] "2010-04-01"
```

- Both functions allow a format argument that controls the formatting. Use format="%m/%d/%Y" to get American-style dates, for example:

```
> format(Sys.Date(), format="%m/%d/%Y")
[1] "04/01/2010"
```

7.10 Converting a Date into a String

- Each two-letter combination of a percent sign (%) followed by another character has special meaning. Some common ones are:

```
%b ; Abbreviated month name (Jan)
%B ; Full month name (January)
%d ; Day as a two-digit number
%m ; Month as a two-digit number
%y ; Year without century (0099)
%Y ; Year with century
```

7.11 Converting Year, Month, and Day into a Date

- You have a date represented by its year, month, and day. You want to merge these elements into a single Date object representation. $\rightarrow$ Use the ISOdate function:
- The result is a POSIXct object that you can convert into a Date object:

```
> ISOdate(year, month, day)
> as.Date(ISOdate(year, month, day))
```

7.11 Converting Year, Month, and Day into a Date

- It is common for input data to contain dates encoded as three numbers: year, month, and day. The ISOdate function can combine them into a POSIXct object:

```
> ISOdate(2012,2,29)
[1] "2012-02-29 12:00:00 GMT"
```

- You can keep your date in the POSIXct format. However, when working with pure dates (not dates and times), I often convert to a Date object and truncate the unused time information:

```
> as.Date(ISOdate(2012,2,29))
[1] "2012-02-29"
```

- Trying to convert an invalid date results in NA:

```
> ISOdate(2013,2,29) # Oops! 2013 is not a leap year
[1] NA
```

7.11 Converting Year, Month, and Day into a Date

- ISOdate can process entire vectors of years, months, and days, which is quite handy for mass conversion of input data.

```
> years
[1] 2010 2011 2012 2013 2014
> months
[1] 1 1 1 1 1
> days
[1] 15 21 20 18 17
> ISOdate(years, months, days)
[1] "2010-01-15 12:00:00 GMT" "2011-01-21 12:00:00 GMT"
[3] "2012-01-20 12:00:00 GMT" "2013-01-18 12:00:00 GMT"
[5] "2014-01-17 12:00:00 GMT"
> as.Date(ISOdate(years, months, days))
[1] "2010-01-15" "2011-01-21" "2012-01-20" "2013-01-18" "2014-01-17"
```

7.11 Converting Year, Month, and Day into a Date

- Purists will note that the vector of months is redundant and that the last expression can therefore be further simplified by invoking the Recycling Rule:

```
> as.Date(ISOdate(years, 1, days))
[1] "2010-01-15" "2011-01-21" "2012-01-20" "2013-01-18" "2014-01-17"
```

- This recipe can also be extended to handle year, month, day, hour, minute, and second data by using the ISOdatetime function (see the help page for details):

```
> ISOdatetime(year, month, day, hour, minute, second)
```

7.12 Getting the Julian Date

- Given a Date object, you want to extract the Julian datewhich is, in R, the number of days since January 1, 1970. $\rightarrow$ Either convert the Date object to an integer or use the *julian* function:

```
> d <- as.Date("2010-03-15")
> as.integer(d)
[1] 14683
> julian(d)
[1] 14683
attr(,"origin")
[1] "1970-01-01"
```

7.12 Getting the Julian Date

- A Julian date is simply the number of days since a more-or-less arbitrary starting point. In the case of R, that starting point is January 1, 1970, the same starting point as Unix systems. So the Julian date for January 1, 1970 is zero, as shown here:

```
> as.integer(as.Date("1970-01-01"))
[1] 0
> as.integer(as.Date("1970-01-02"))
[1] 1
> as.integer(as.Date("1970-01-03"))
[1] 2
.
. (etc.)
.
```

7.13 Extracting the Parts of a Date

- Convert the Date object to a POSIXlt object, which is a list of date parts. Then extract the desired part from that list:

```
> d <- as.Date("2010-03-15")
> p <- as.POSIXlt(d)
> p$mday # Day of the month
[1] 15
> p$mon # Month (0 = January)
[1] 2
> p$year + 1900 # Year
[1] 2010
```

7.13 Extracting the Parts of a Date

- The POSIXlt object represents a date as a list of date parts. Convert your Date object to POSIXlt by using the as.POSIXlt function, which will give you a list with these members:

```
sec ; Seconds (061)
min ; Minutes (059)
hour ; Hours (023)
mday ; Day of the month (131)
mon ; Month (011)
year ; Years since 1900
wday ; Day of the week (06, 0 = Sunday)
yday ; Day of the year (0365)
isdst ; Daylight savings time flag
```

7.13 Extracting the Parts of a Date

- Using these date parts, we can learn that April 1, 2010, is a Thursday (wday = 4) and the 91st day of the year (because yday = 0 on January 1):

```
> d <- as.Date("2010-04-01")
> as.POSIXlt(d)$wday
[1] 4
> as.POSIXlt(d)$yday
[1] 90
```

- A common mistake is failing to add 1900 to the year, giving the impression you are living a long, long time ago:

```
> as.POSIXlt(d)$year # Oops!
[1] 110
> as.POSIXlt(d)$year + 1900
[1] 2010
```

7.14 Creating a Sequence of Dates

- A typical use of *seq* specifies a starting date (*from*), ending date (*to*), and increment (by). An increment of 1 indicates daily dates:

```
> s <- as.Date("2012-01-01")
> e <- as.Date("2012-02-01")
> seq(from=s, to=e, by=1) # One month of dates
[1] "2012-01-01" "2012-01-02" "2012-01-03" "2012-01-04" "2012-01-05" "2012-01-06"
[7] "2012-01-07" "2012-01-08" "2012-01-09" "2012-01-10" "2012-01-11" "2012-01-12"
[13] "2012-01-13" "2012-01-14" "2012-01-15" "2012-01-16" "2012-01-17" "2012-01-18"
[19] "2012-01-19" "2012-01-20" "2012-01-21" "2012-01-22" "2012-01-23" "2012-01-24"
[25] "2012-01-25" "2012-01-26" "2012-01-27" "2012-01-28" "2012-01-29" "2012-01-30"
[31] "2012-01-31" "2012-02-01"
```

7.14 Creating a Sequence of Dates

- Another typical use specifies a starting date (*from*), increment (*by*), and number of dates (*length.out*):

```
> seq(from=s, by=1, length.out=7) # Dates, one week apart
[1] "2012-01-01" "2012-01-02" "2012-01-03" "2012-01-04" "2012-01-05" "2012-01-06"
[7] "2012-01-07"
```

- The increment (*by*) is flexible and can be specified in days, weeks, months, or years:

```
> seq(from=s, by="month", length.out=12) # First of the month for one year
[1] "2012-01-01" "2012-02-01" "2012-03-01" "2012-04-01" "2012-05-01" "2012-06-01"
[7] "2012-07-01" "2012-08-01" "2012-09-01" "2012-10-01" "2012-11-01" "2012-12-01"
> seq(from=s, by="3 months", length.out=4) # Quarterly dates for one year
[1] "2012-01-01" "2012-04-01" "2012-07-01" "2012-10-01"
> seq(from=s, by="year", length.out=10) # Year-start dates for one decade
[1] "2012-01-01" "2013-01-01" "2014-01-01" "2015-01-01" "2016-01-01" "2017-01-01"
[7] "2018-01-01" "2019-01-01" "2020-01-01" "2021-01-01"
```

7.14 Creating a Sequence of Dates

- Be careful with by="month" near month-end. In this example, the end of February overflows into March, which is probably not what you wanted:

```
> seq(as.Date("2010-01-29"), by="month", len=3)
[1] "2010-01-29" "2010-03-01" "2010-03-29"
```

9.1 Summarizing Your Data

- **summary** function

```
> class(cars)
> cars$type = sample(c("Large","Small"),length(cars[,1]),replace=TRUE)
> cars$city = as.factor(sample(c("Seoul","Busan","Daejon"),length(cars[,1]
> summary(cars)
speed           dist            type               city
Min.   : 4.0   Min.   :  2.00   Length:50          Busan :18
1st Qu.:12.0   1st Qu.: 26.00   Class :character   Daejon:13
Median :15.0   Median : 36.00   Mode  :character   Seoul :19
Mean   :15.4   Mean   : 42.98
3rd Qu.:19.0   3rd Qu.: 56.00
Max.   :25.0   Max.   :120.00
```

9.1 Summarizing Your Data

- ```
  > cars.list = as.list(cars)
  > summary(cars.list)
  > lapply(cars.list, summary)
  $speed
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  4.0    12.0    15.0    15.4    19.0    25.0
  $dist
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  2.00   26.00   36.00   42.98   56.00  120.00
  $type
  Length     Class      Mode
  50 character character
  $city
  Busan Daejon  Seoul
  18     13     19
  ```
  ```

9.2 Calculating Relative Frequencies

- Identify the interesting observations by using a logical expression; then use the **mean** function to calculate the fraction of observations it identifies.

```
> x = runif(100,-1,2)
> mean(x > 0)
```

9.3 Tabulating Factors and Creating Contingency Tables

- *iris* samples

```
?iris
data <- iris
a <- rep("y",100)
b <- rep("x",100)
ab <- rbind(a,b)
xy <- sample(ab,dim(data)[1])
data <- cbind(data,xy) # iris data with 2 factor variables
head(data)
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species	xy
1	5.1	3.5	1.4	0.2	setosa	y
2	4.9	3.0	1.4	0.2	setosa	x
3	4.7	3.2	1.3	0.2	setosa	y
4	4.6	3.1	1.5	0.2	setosa	y
5	5.0	3.6	1.4	0.2	setosa	x
6	5.4	3.9	1.7	0.4	setosa	y

9.3 Tabulating Factors and Creating Contingency Tables

- ```
  > table(data$xy)
  x  y
  78 72
  ```

- ```
  > table(data$Species, data$xy)
              x  y
  setosa     20 30
  versicolor 29 21
  virginica  29 21
  ```

9.4 Testing Categorical Variables for Independence

- Use the **table** function to produce a contingency table from the two factors. Then use the **summary** function to perform a chi-squared test of the contingency table:

```
> summary(table(data$Species, data$xy))
Number of cases in table: 150
Number of factors: 2
Test for independence of all factors:
Chisq = 0.16, df = 2, p-value = 0.9231
```

9.5 Calculating Quantiles (and Quartiles) of a Dataset

- The **quantile** function can tell you which observation delimits the lower 5% of the data:

```
sam <- seq(0,1,length.out = 10^4)
vec <- sample(sam,1000)
> quantile(vec,0.05)
5%
0.04711971
> quantile(vec,c(0.05,0.95))
5%        95%
0.04711971 0.95460046
> quantile(vec)
0%        25%        50%        75%       100%
0.00110011 0.25817582 0.51705171 0.74597460 0.99849985
```

9.6 Inverting a Quantile

- Given an observation x from your data, you want to know its corresponding quantile. That is, you want to know what fraction of the data is less than x.

- Assuming your data is in a vector vec, compare the data against the observation and then use mean to compute the relative frequency of values less than x:

```
> a = mean(vec < 0.8)
> quantile(vec,a)
```

9.7 Converting Data to Z-Scores

- You want to calculate the corresponding *z*-scores for all data elements. (This is sometimes called normalizing the data.)

```
> cars = cars[c(1,2)]
> a = scale(cars$speed)
> b = (cars$speed-mean(cars$speed))/sd(cars$speed)
> plot(a,b)
> s.cars = scale(cars)
> a.cars = apply(cars,2,scale)
```

9.8 Testing the Mean of a Sample (t Test)

- Apply the t.test function for one-sample t-test.

```
> x <- rnorm(50, mean=100, sd=15)
> t.test(x, mu=95)
One Sample t-test
data: x
t = 3.2832, df = 49, p-value = 0.001897
alternative hypothesis: true mean is not equal to 95
95 percent confidence interval:
97.16167 103.98297
sample estimates:
mean of x
100.5723
> t.test(x, mu=100)
```

9.10 Forming a Confidence Interval for a Median

- Wilcoxon rank sum and signed rank tests

```
> wilcox.test(x, conf.int=TRUE,mu=100)
Wilcoxon signed rank test with continuity correction
data:  x
V = 559, p-value = 0.4515
alternative hypothesis: true location is not equal to 100
95 percent confidence interval:
94.56114 102.52374
sample estimates:
(pseudo)median
98.42834
```

9.11 Testing a Sample Proportion

- **prop.test**

```
> prop.test(11, 20, 0.5, alternative="greater")
1-sample proportions test with continuity correction
data: 11 out of 20, null probability 0.5
X-squared = 0.05, df = 1, p-value = 0.4115
alternative hypothesis: true p is greater than 0.5
95 percent confidence interval:
0.3496150 1.0000000
sample estimates:
p
0.55
```

9.12 Forming a Confidence Interval for a Proportion

- Here, the number of observations is $n = 9$ and the number of successes is $x = 6$.

```
> prop.test(6, 9)
1-sample proportions test with continuity correction
data: 6 out of 9, null probability 0.5
X-squared = 0.4444, df = 1, p-value = 0.505
alternative hypothesis: true p is not equal to 0.5
95 percent confidence interval:
0.3091761 0.9095817
sample estimates:
p
0.6666667
```

9.13 Testing for Normality

- **shapiro.test** function:

```
> shapiro.test(x)
```

- Examples

```
x <- rnorm(50,0,1)
shapiro.test(x)
> shapiro.test(x)
Shapiro-Wilk normality test
data:  x
W = 0.968, p-value = 0.192
```

- This example reports a $p - value$ of 0.192 for x, which is large(normal).

9.13 Testing for Normality

- Examples

```
y <- runif(50,0,1)
shapiro.test(y)
> shapiro.test(y)
Shapiro-Wilk normality test
data:  y
W = 0.9498, p-value = 0.03361
```

- This example reports a $p - value$ of 0.03361 for y, which is small(not normal).

9.14 Testing for Runs

- The **tseries** package contains the **runs.test** function, which checks a sequence for randomness.

```
> library(tseries)
> runs.test(as.factor(s))
```

-
```
> library(tseries)
> s <- sample(c(0,1), 100, replace=T)
> runs.test(as.factor(s))
Runs Test
data: as.factor(s)
Standard Normal = 0.2175, p-value = 0.8279
alternative hypothesis: two.sided
```

9.15 Comparing the Means of Two Samples

- **t.test** function

  ```
  > t.test(x,y)
  ```

- ```
  > t.test(x,y,paired=TRUE)
  ```
  ```

9.15 Comparing the Means of Two Samples

Paired vs Not paired

- Paired : Randomly select one group of people. Give them the SAT test twice, once with morning coffee and once without morning coffee. For each person, we will have two SAT scores. These are *paired observations*.

- Not paired : Randomly select two groups of people. One group has a cup of morning coffee and takes the SAT test. The other group just takes the test. We have a score for each person, but the scores are not paired in any way.

9.15 Comparing the Means of Two Samples

- Examples

```
x <- rnorm(50,0,1)
y <- runif(50,0,1)
> t.test(x,y)

Welch Two Sample t-test

data:  x and y
t = -2.6841, df = 56.259, p-value = 0.009536
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
-0.7210333 -0.1047653
sample estimates:
mean of x mean of y
0.0555821 0.4684814
```

9.15 Comparing the Means of Two Samples

- Examples

```
> t.test(x,y,paired=TRUE)

Paired t-test

data:  x and y
t = -2.7776, df = 49, p-value = 0.007739
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
-0.7116274 -0.1141713
sample estimates:
mean of the differences
-0.4128993
```

9.16 Comparing the Locations of Two Samples Nonparametrically

- **wilcox.test** function

```
> wilcox.test(x, y, paired=TRUE)
```

- For unpaired observations,let paired default to *FALSE*:

```
> wilcox.test(x, y)
```

- Example for x and y

```
> wilcox.test(x,y)

Wilcoxon rank sum test with continuity correction

data:  x and y
W = 962, p-value = 0.04748
alternative hypothesis: true location shift is not equal to 0
```

9.17 Testing a Correlation for Significance

- **coe.test** function

```
> cor.test(x, y)
```

- For nonnormal populations, use the Spearman method instead:

```
> cor.test(x, y, method="Spearman")
```

9.18 Testing Groups for Equal Proportions

- Use the prop.test function with two vector arguments:

```
> ns <- c(ns1, ns2, ..., nsN)
> nt <- c(nt1, nt2, ..., ntN)
> prop.test(ns, nt)
```

9.18 Testing Groups for Equal Proportions

- Examples

```
> successes <- c(14,10)

> trials <- c(38,40)

> prop.test(successes, trials)

2-sample test for equality of proportions with continuity correction

data: successes out of trials
X-squared = 0.7872, df = 1, p-value = 0.3749
alternative hypothesis: two.sided
95 percent confidence interval:
-0.1110245 0.3478666
sample estimates:
prop 1 prop 2
0.3684211 0.2500000
```

9.19 Performing Pairwise Comparisons Between Group Means

- Use **pairwise.t.test** to perform the pairwise comparison of means:

```
> pairwise.t.test(x,f) # x contains the data, f is the grouping factor
```

- Examples ; Using iris data

```
data <- iris
pairwise.t.test(data$Sepal.Length,data$Species)
```

```
> pairwise.t.test(data$Sepal.Length,data$Species)
Pairwise comparisons using t tests with pooled SD

data:  data$Sepal.Length and data$Species

           setosa  versicolor
versicolor 1.8e-15 -
virginica  < 2e-16 2.8e-09

P value adjustment method: holm
```

9.20 Testing Two Samples for the Same Distribution

- **ks.test** function

```
> ks.test(x, y)
```

-

```
x <- rnorm(50,0,1)
y <- runif(50,0,1)
> ks.test(x,y)

Two-sample Kolmogorov-Smirnov test

data:  x and y
D = 0.5, p-value = 4.808e-06
alternative hypothesis: two-sided
```

10. Graphics

This chapter introduces the basics of "graphics".

- high-level graphics function
  : plot, boxplot, hist, qqnorm, curve
- low-level graphics function
  : points, lines, abline, segment, polygon, text

10.1 Plot

- If your data are held in two parallel vectors, x and y, then use them as arguments of plot.

```
> plot(x,y)
> plot(dfrm) # dfrm : dataframe
```

10.2 Adding a Title and Labels

- Use the *main* argument for a title.
- Use the *xlab* argument for an x-axis label.
- Use the *ylab* argument for a y-axis label.

```
> plot(x, main="The Title",
xlab="X-axis Label", ylab="Y-axis Label")
```

10.3 Adding a Grid

- Call plot with type="n" to initialize the graphics frame without displaying the data.
- Call the grid function to draw the grid.
- Call low-level graphics functions, such as points and lines, to draw the graphics overlaid on the grid.

```
> plot(x, y, type="n")
> grid()
> points(x, y)
```

10.1/2/3 Example

- 10.1/2/3 Example

```
par(mfrow=c(1,3))
---------------------------------------
plot(pressure)
---------------------------------------
plot(pressure,
main="relationship between temp and pres",
xlab="temperature(temp)",
ylab="pressure(pres)")
---------------------------------------
plot(pressure,
main="relationship between temp and pres",
xlab="temperature(temp)",
ylab="pressure(pres)",
type="n")
---------------------------------------
grid(); points(pressure)
```

10.4 Creating a Scatter Plot of Multiple Groups

- Use the pch argument of plot

```
> plot(x, y, pch=as.integer(f))
```

- Example

```
rm(list=ls()); dev.off()
par(mfrow=c(1,2))
iris = read.csv("C:\\Users\\pc\\Desktop\\Iris.csv",
header=TRUE, stringsAsFactor=FALSE)
with(iris, plot(Petal.Length, Petal.Width))
with(iris, plot(Petal.Length, Petal.Width, pch=1:3))
```

10.5 Adding a Legend

- Legend for points

```
> legend(x, y, labels,
pch=c(pointtype1, pointtype2, ...))
```

- Legend for lines according to line type

```
> legend(x, y, labels,
lty=c(linetype1, linetype2, ...))
```

- Legend for lines according to line width

```
> legend(x, y, labels,
lwd=c(width1, width2, ...))
```

10.5 Adding a Legend

- Example

```
rm(list=ls()); dev.off()
f = factor(iris$Species)
with(iris, plot(Petal.Length, Petal.Width,
pch=as.integer(f)))
legend(1.5, 2.4, as.character(levels(f)),
pch=1:length(levels(f)))
```

10.6 Plotting the Regression Line of a Scatter Plot

- **abline** function

```
> m <- lm(y ~ x)
> plot(y ~ x)
> abline(m)
```

- Example

```
##10.6 Example
rm(list=ls()); dev.off()
install.packages("faraway")
library(faraway)
data(strongx)
m = lm(crossx ~ energy, data=strongx)
plot(crossx ~ energy, data=strongx)
abline(m)
```

10.7 Plotting All Variables Against All Other Variables

- Place your data in a data frame and then plot the data frame. R will create one scatter plot for every pair of columns.

```
> plot(dfrm)
```

- Example

```
rm(list=ls()); dev.off()
head(iris)
plot(iris[,1:4])
```

10.8 Creating One Scatter Plot for Each Factor Level

- Conditioning plot: **coplot** function

```
> coplot(y ~ x | f)
```

- Example

```
rm(list=ls()); dev.off()
install.packages("MASS")
data(Cars93, package="MASS")
coplot(Horsepower ~ MPG.city | Origin, data=Cars93)
```

10.9 Creating a Bar Chart

- **barplot** function

```
> barplot(c(height1, height2, ..., heightn))
```

- Example

```
rm(list=ls()); dev.off()
heights = tapply(airquality$Temp, airquality$Month, mean)
par(mfrow=c(1,2))
barplot(heights)
barplot(heights,
main="Mean Temp. by Month",
names.arg=c("May", "Jun", "Jul", "Aug", "Sep"),
ylab="Temp (deg. F)")
```

10.10 Adding Confidence Intervals to a Bar Chart

- **barplot2** function

```
> library(gplots)
> barplot2(x, plot.ci=TRUE, ci.l=lower, ci.u=upper)
```

- Example

```
rm(list=ls()); dev.off()
attach(airquality)
heights = tapply(Temp, Month, mean)
lower = tapply(Temp, Month,
function(v) t.test(v)$conf.int[1])
upper = tapply(Temp, Month,
function(v) t.test(v)$conf.int[2])
par(mfrow=c(1,2))
barplot(heights, plot.ci=TRUE, ci.l=lower, ci.u=upper)
barplot(heights, plot.ci=TRUE, ci.l=lower, ci.u=upper,
ylim=c(50,90), xpd=FALSE,
main="Mean Temp. By Month",
names.arg=c("May","Jun","Jul","Aug","Sep"),
```

10.11 Coloring a Bar Chart

- Use the col argument of barplot

```
> barplot(heights, col=colors)
```

- Example

```
rm(list=ls()); dev.off()
attach(airquality)
heights = tapply(Temp, Month, mean)
rel.hts = rank(heights) / length(heights)
grays = gray(1 - rel.hts)
par(mfrow=c(1,2))
barplot(heights, col=grays)
rel.hts = (heights - min(heights))/
(max(heights) - min(heights))
barplot(heights,col=grays,
ylim=c(50,90), xpd=FALSE,
main="Mean Temp. By Month",
names.arg=c("May", "Jun", "Jul", "Aug", "Sep"),
ylab="Temp (deg. F)")
```

10.12 Plotting a Line from x and y Points

- Use the **plot** function with a plot type of "l".

```
> plot(x, y, type="l")
> plot(dfrm, type="l")
```

- Example

```
rm(list=ls()); dev.off()
par(mfrow=c(1,2))
plot(pressure)
plot(pressure, type="l")
```

10.13 Changing the Type, Width, or Color of a Line

- lty="solid" or lty=1 (default)
- lty="dashed" or lty=2
- lty="dotted" or lty=3
- lty="dotdash" or lty=4
- lty="longdash" or lty=5
- lty="twodash" or lty=6
- lty="blank" or lty=0 (inhibits drawing)

```
> plot(x, y, type="l", lty="")
```

10.13 Changing the Type, Width, or Color of a Line

- Example

```
rm(list=ls()); dev.off()
x = pressure$temperature
y = pressure$pressure
plot(x,y,type="l",lty=3,col="red")
```

10.14 Plotting Multiple Datasets

- Show multiple datasets in one plot

```
> xlim <- range(c(x1,x2))
> ylim <- range(c(y1,y2))
> plot(x1, y1, type="l", xlim=xlim, ylim=ylim)
> points(point.x, point.y, type="p")
```

- Example

```
rm(list=ls()); dev.off()
x = pressure$temperature
y = pressure$pressure
xlim = range(c(min(x),max(x)))
ylim = range(c(min(y),max(y)))
plot(x, y, type="l", xlim=xlim, ylim=ylim)
points(median(x), median(y),col="red",type="p")
```

10.15 Adding Vertical or Horizontal Lines

- ```
  > abline(v=x) # Draw a vertical line at x
  > abline(h=y) # Draw a horizontal line at y
  ```

- Example

```
rm(list=ls()); dev.off()
samp = rnorm(100)
plot(samp)
m = mean(samp)
abline(h=m)
stdevs = m + c(-2,-1,+1,+2)*sd(samp)
abline(h=stdevs, lty="dotted")
```
```

10.16 Creating a Box Plot

- **boxplot(x)**

```
> boxplot(x)
```

- Example

```
rm(list=ls()); dev.off()
samp = rnorm(100)
boxplot(samp)
```

10.17 Creating One Box Plot for Each Factor Level

- Use the **boxplot** function with a formula
  Here, x is the numeric variable and f is the factor.

```
> boxplot(x ~ f)
```

- `> plot(f, x)`

- Example

```
rm(list=ls()); dev.off()
install.packages("MASS")
par(mfrow=c(1,2))
data(UScereal, package="MASS")
boxplot(sugars ~ shelf, data=UScereal)
data(UScereal, package="MASS")
boxplot(sugars ~ shelf, data=UScereal,
main="Sugar Content by Shelf",
xlab="Shelf", ylab="Sugar (grams per portion)")
```

10.18 Creating a Histogram

- Use **hist(x)**, where x is a vector of numeric values.

```
> hist(x)
```

- Example

```
rm(list=ls()); dev.off()
data(Cars93, package="MASS")
par(mfrow=c(1,3))
hist(Cars93$MPG.city)
hist(Cars93$MPG.city, 20)
hist(Cars93$MPG.city, 20,
main="City MPG (1993)", xlab="MPG")
```

10.19 Adding a Density Estimate to a Histogram

- Use the density function to approximate the sample density, then use lines to draw the approximation.

```
> hist(x, prob=T) # with probability scale
> lines(density(x)) # Graph the approximate density
```

- Example

```
rm(list=ls()); dev.off()
samp = rgamma(500, 2, 2)
hist(samp, 20, prob=T)
lines(density(samp))
```

10.20 Creating a Discrete Histogram

- Use the table function to count occurrences.
  Then use the plot function with type="h" to graph the occurrances as a histogram.
  Here, x is a vector of discrete values.

```
> plot(table(x), type="h")
```

- Example

```
rm(list=ls()); dev.off()
par(mfrow=c(1,2))
x = sample(1:10,100,replace=T)
plot(table(x), type="h", lwd=5, ylab="Freq")
plot(table(x)/length(x), type="h", lwd=5, ylab="Freq")
```

10.21 Creating a Normal Quantile-Quantile (Q-Q) Plot

- **qqnorm** function, **qqline**

```
> qqnorm(x)
> qqline(x)
```

- Example

```
rm(list=ls()); dev.off()
par(mfrow=c(1,2))
data(Cars93, package="MASS")
qqnorm(Cars93$Price, main="Q-Q Plot: Price")
qqline(Cars93$Price)
qqnorm(log(Cars93$Price), main="Q-Q Plot: log(Price)")
qqline(log(Cars93$Price))
```

10.22 Creating Other Quantile-Quantile Plots

- Use abline to plot the diagonal line

```
> plot(qt(ppoints(y), 5), sort(y))
> abline(a=0, b=1)
```

- Example

```
rm(list=ls()); dev.off()
RATE = 1/10
N = 100
y = rexp(N, rate=RATE)
plot(qexp(ppoints(y), rate=RATE), sort(y))
abline(a=0, b=1)
plot(qexp(ppoints(y), rate=RATE), sort(y),
main="Q-Q Plot", xlab="Theoretical Quantiles",
ylab="Sample Quantiles")
abline(a=0, b=1)
```

10.23 Plotting a Variable in Multiple Colors

- Use the col argument of the plot function

```
> plot(x, col=colors)
```

- Example

```
rm(list=ls()); dev.off()
x = runif(100,-5,5)
colors = ifelse(x >= 0, "black", "gray")
plot(x, type='h', lwd=3, col=colors)
```

10.24 Graphing a Function

- **curve** function

```
> curve(sin, -3, +3)
#Graph the sine function from -3 to +3
```

- Example

```
rm(list=ls()); dev.off()
par(mfrow=c(1,2))
curve(dnorm, -3.5, +3.5,
main="Std. Normal Density")
f = function(x) exp(-abs(x)) * sin(2*pi*x)
curve(f, -5, +5, main="Dampened Sine Wave")
```

10.25 Pausing Between Plots

- ```
  > par(ask=TRUE)
  > par(ask=FALSE)
  ```

- ```
  > par(mfrow=(c(N,M))
  # Divide the graphics window into N x M matrix.
  ```

- Example

```
rm(list=ls()); dev.off()
par(mfrow=c(2,2))
Quantile <- seq(from=0, to=1, length.out=30)
plot(Quantile, dbeta(Quantile, 2, 4),
type="l", main="First")
plot(Quantile, dbeta(Quantile, 4, 2),
type="l", main="Second")
plot(Quantile, dbeta(Quantile, 1, 1),
type="l", main="Third")
plot(Quantile, dbeta(Quantile, 0.5, 0.5),
type="l", main="Fourth")
```

10.27 Opening Additional Graphics Windows

- **win.graph** function

```
> win.graph()
```

10.28 Writing Your Plot to a File

- **savePlot** function

```
> savePlot(filename="filename.ext", type="type")
#type: "png","jpeg",and etc
```

10.29 Changing Graphical Parameters

- Use the par function, which lets you set values of global graphics parameters.

Parameter and type	Purpose
bg="color"	Background color
cex=number	Height of text and plotted points
col="color"	Default plotting color
lty="linetype"	Type of line: solid, dotted, dashed, etc
lwd=number	Line width: 1 = normal, 2 = thicker, etc.
mfcol(mfrow)=c(nr,nc)	plot matrix with nr(rows) and nc(columns)
pch=pointtype	Default point type

11. Linear Regression and ANOVA

- Simple linear regression
  $y_i = \beta_0 + \beta_(1)x_i + \varepsilon_i$
- Multiple linear regression
  $y_i = \beta_0 + \beta_1 u_i + \beta_2 v_i + \beta_3 w_i + \varepsilon_i$

Linear Regression

Most of the information you need in regression is as follows.

- Is the model statistically significant?
  : Check the F statistic at the bottom of the summary.
- Are the coefficients significant?
  : Check the coefficient t statistics and p-values in the summary, or check their confidence intervals.
- Is the model useful?
  : Check the R^2 near the bottom of the summary.
- Does the model fit the data well?
  : Plot the residuals and check the regression diagnostics.
- Does the data satisfy the assumptions behind linear regression?
  : Check whether the diagnostics confirm that a linear model is reasonable for your data.

11.1 Performing Simple Linear Regression

- The lm function performs a linear regression and reports the coefficients.

```
> x.vec = c(1.9,0.8,1.1,0.1,-0.1,4.4,4.6,1.6,5.5,3.4)
> y.vec = c(0.7,-1.0,-0.2,-1.2,-0.1,3.4,0.0,0.8,3.7,2.0)
> fit = lm(y.vec~x.vec)
> fit

Call:
lm(formula = y.vec ~ x.vec)

Coefficients:
(Intercept)    x.vec
-0.7861       0.6850
```

11.2 Performing Multiple Linear Regression

- Use the lm function. Specify the multiple predictors on the righthand side of the formula,separated by plus signs (+).

```
> fit2 = lm(mpg~ hp + wt , data = mtcars)
> summary(fit2)

Call:
lm(formula = mpg ~ hp + wt, data = mtcars)

Residuals:
Min      1Q Median      3Q     Max
-3.941 -1.600 -0.182  1.050  5.854

Coefficients:
            Estimate   Std. Error t value Pr(>|t|)
(Intercept) 37.22727    1.59879  23.285  < 2e-16 ***
hp          -0.03177    0.00903  -3.519  0.00145 **
wt          -3.87783    0.63273  -6.129 1.12e-06 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 2.593 on 29 degrees of freedom
Multiple R-squared:  0.8268,Adjusted R-squared:  0.8148
F-statistic: 69.21 on 2 and 29 DF,  p-value: 9.109e-12
```

11.3 Getting Regression Statistics

- Save the regression model in a variable, say m.
  Then use functions to extract regression statistics and information below from the model.
  - anova(m) : ANOVA table
  - coefficients(m) : Model coefficients
  - confint(m) : Confidence intervals for the regression coefficients
  - deviance(m) : Residual sum of squares
  - effects(m) : Vector of orthogonal effects
  - fitted(m) : Vector of fitted y values
  - residuals(m) : Model residuals
  - summary(m) : Key statistics, such as R^2, the F statistic, and the residual standard error (σ)

11.4 Understanding the Regression Summary

- Example

```
> summary(fit)

Call:
lm(formula = y.vec ~ x.vec)

Residuals:
Min      1Q  Median      3Q     Max
-2.3651 -0.4036  0.3208  0.6613  1.1720

Coefficients:
Estimate Std. Error t value Pr(>|t|)
(Intercept)  -0.7861     0.5418  -1.451  0.18485
x.vec         0.6850     0.1802   3.801  0.00523 **
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 1.083 on 8 degrees of freedom
Multiple R-squared:  0.6436,Adjusted R-squared:  0.599
F-statistic: 14.45 on 1 and 8 DF,  p-value: 0.005231
```

11.5 Performing Linear Regression Without an Intercept

You want to perform a linear regression, but you want to force the intercept to be zero.

- `> fit = lm(y.vec~x.vec+0)`

11.6 Performing Linear Regression with Interaction Terms

- The R syntax for regression formulas lets you specify interaction terms. The interaction of two variables, u and v, is indicated by separating their names with an asterisk (*). This corresponds to the model;

$$y_i = \beta_0 + \beta_1 u_i + \beta_2 v_i + \beta_3 u_i v_i + \varepsilon_i$$

, which includes the first-order interaction term $\beta_3 u_i v_i$.

```
> lm(y ~ u*v)
```

11.6 Performing Linear Regression with Interaction Terms

- Example

```
> fit2 = lm(mpg~ hp + wt + hp*wt , data = mtcars)
> summary(fit2)

Call:
lm(formula = mpg ~ hp + wt + hp * wt, data = mtcars)

Residuals:
Min      1Q  Median      3Q     Max
-3.0632 -1.6491 -0.7362  1.4211  4.5513

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 49.80842    3.60516  13.816 5.01e-14 ***
hp          -0.12010    0.02470  -4.863 4.04e-05 ***
wt          -8.21662    1.26971  -6.471 5.20e-07 ***
hp:wt        0.02785    0.00742   3.753 0.000811 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 2.153 on 28 degrees of freedom
Multiple R-squared:  0.8848,Adjusted R-squared:  0.8724
F-statistic: 71.66 on 3 and 28 DF,  p-value: 2.981e-13
```

11.7 Selecting the Best Regression Variables

- Backward stepwise regression

```
> library(MASS)
> states = as.data.frame(state.x77[,c("Murder","Population","Illiteracy","Income","Frost")])
> full.model = lm(Murder ~ Population + Illiteracy + Income + Frost,data=states)
> reduced.model = step(full.model, direction = "backward")
```

11.7 Selecting the Best Regression Variables

- Backward stepwise regression

```
Start:  AIC=97.75
Murder ~ Population + Illiteracy + Income + Frost

             Df Sum of Sq    RSS     AIC
- Frost       1     0.021 289.19  95.753
- Income      1     0.057 289.22  95.759
<none>                   289.17  97.749
- Population  1    39.238 328.41 102.111
- Illiteracy  1   144.264 433.43 115.986


Step:  AIC=95.75
Murder ~ Population + Illiteracy + Income

             Df Sum of Sq    RSS     AIC
- Income      1     0.057 289.25  93.763
<none>                   289.19  95.753
- Population  1    43.658 332.85 100.783
- Illiteracy  1   236.196 525.38 123.605


Step:  AIC=93.76
Murder ~ Population + Illiteracy

             Df Sum of Sq    RSS     AIC
<none>                   289.25  93.763
- Population  1    48.517 337.76  99.516
- Illiteracy  1   299.646 588.89 127.311
```

11.7 Selecting the Best Regression Variables

- Forward stepwise regression

```
> min.model = lm(Murder ~ 1,data=states)
> fw.model = step(min.model, direction="forward",
scope = (~ Population + Illiteracy + Income + Frost))
```

11.7 Selecting the Best Regression Variables

- Forward stepwise regression

```
Start:  AIC=131.59
Murder ~ 1

             Df Sum of Sq    RSS     AIC
+ Illiteracy  1    329.98 337.76  99.516
+ Frost       1    193.91 473.84 116.442
+ Population  1     78.85 588.89 127.311
+ Income      1     35.35 632.40 130.875
<none>                    667.75 131.594

Step:  AIC=99.52
Murder ~ Illiteracy

             Df Sum of Sq    RSS     AIC
+ Population  1    48.517 289.25  93.763
<none>                    337.76  99.516
+ Frost       1     5.387 332.38 100.712
+ Income      1     4.916 332.85 100.783

Step:  AIC=93.76
Murder ~ Illiteracy + Population

             Df Sum of Sq    RSS    AIC
<none>                    289.25 93.763
+ Income  1  0.057022 289.19 95.753
+ Frost   1  0.021447 289.22 95.759
```

11.8 Regressing on a Subset of Your Data

You want to fit a linear model to a subset of your data, not to the entire dataset.

- ```
  > fit2 = lm(mpg~ hp + wt + hp*wt ,subset = 1:30, data = mtcars) #use only data[1:30]
  > summary(fit2)

  Call:
  lm(formula = mpg ~ hp + wt + hp * wt, data = mtcars, subset = 1:30)

  Residuals:
  Min      1Q  Median      3Q     Max
  -3.1320 -1.5636 -0.6691  1.4594  4.4220

  Coefficients:
  Estimate Std. Error t value Pr(>|t|)
  (Intercept) 49.380844   3.703071  13.335 3.91e-13 ***
  hp          -0.121860   0.025383  -4.801 5.68e-05 ***
  wt          -7.835597   1.356251  -5.777 4.38e-06 ***
  hp:wt        0.026710   0.007657   3.488  0.00175 **
  ---
  Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

  Residual standard error: 2.195 on 26 degrees of freedom
  Multiple R-squared:  0.8859,Adjusted R-squared:  0.8728
  F-statistic: 67.31 on 3 and 26 DF,  p-value: 2.191e-12
  ```
  ```

11.9 Regressing on a Polynomial

- Use the poly(x,n) function in your regression formula to regress on an n-degree polynomial of x.

$$y_i = \beta_0 + \beta_0 x_i + \beta_2 (x_i)^2 + \beta_3 (x_i)^3 + \varepsilon_i$$

```
> lm(y.vec~poly(x.vec,3,raw=TRUE))
```

11.10 Regressing on Transformed Data

- You can embed the needed transformation inside the regression formula. If, for example, y must be transformed into log(y), then the regression formula becomes as follows.

```
> lm(log(y) ~ x)
```

```
> Example
> x.vec = c(1.5,2.1,3.4,6.4,7.2,16.1,20.8,30.7,32.1,42.1)
> y.vec = c(0.2,0.3,0.5,1.7,1.2,2.7,3.3,3.9,4.1,4.9)
> lm(log(y.vec)~x.vec)

Call:
lm(formula = log(y.vec) ~ x.vec)

Coefficients:
(Intercept)        x.vec
-0.73993      0.06868
```

11.11 Finding the Best Power Transformation (Boxox Procedure)

- **boxcox**

```
library(MASS)
x.vec = c(1.5,2.1,3.4,6.4,7.2,16.1,20.8,30.7,32.1,42.1)
y.vec = c(0.2,0.3,0.5,1.7,1.2,2.7,3.3,3.9,4.1,4.9)
m = lm(y.vec ~ x.vec)
boxcox(m)
```

11.12 Forming Confidence Intervals for Regression Coefficients

- **confint** function

```
> m = lm(y.vec ~ x.vec)
> confint(m)
2.5 %    97.5 %
(Intercept) -0.04511280 0.8380322
x.vec        0.09527616 0.1366869
```

11.13 Plotting Regression Residuals

You want a visual display of your regression residuals.

- ```
  m = lm(y.vec~x.vec)
  plot(m,which=1)
  ```
  ```

11.14 Diagnosing a Linear Regression

You have performed a linear regression.
Now you want to verify the model quality by running diagnostic checks.

- Start by plotting the model object,
  which will produce several diagnostic plots.

```
> m <- lm(y.vec ~ x.vec)
> plot(m)
```

11.15 Identifying Influential Observations

- The influence.measures function reports several useful statistics for identifying influential observations, and it flags the significant ones with an asterisk (*).
  Its main argument is the model object from your regression.

```
> influence.measures(m)
```

11.16 Testing Residuals for Autocorrelation (Durbinatson Test)

- **dwtest** function

```
> library(lmtest)
> m <- lm(y.vec ~ x.vec) # Create a model object
> dwtest(m) # Test the model residuals
```

11.16 Testing Residuals for Autocorrelation (Durbinatson Test)

- **acf** function

```
> acf(m) # Plot the ACF of the model residuals
```

11.17 Predicting New Values

- **predict** function

```
> m = lm(y.vec ~ x.vec)
> preds = data.frame(x.vec = 8)
> predict(m,newdata=preds)
    1
1.324312
```

11.18 Forming Prediction Intervals

- Use the predict function and specify interval="prediction"

```
> predict(m, newdata=preds, interval="prediction")
```

11.19 Logistic Regression

Model

$$\begin{aligned} y_i =& \beta_0 + \beta_1 x_{1i} + \epsilon_i \\ & i = 1, 2, \cdots, n \\ & y_i = 1 \text{ or } 0 \end{aligned}$$

$$E(y) = \frac{exp(\beta_0 + \beta_1 x)}{1 + exp(\beta_0 + \beta_1 x)}$$

$$log\left(\frac{E(y)}{1 - E(y)}\right) = \beta_0 + \beta_1 x$$

11.19 Logistic Regression

- Example

```
> mydata = read.csv("https://stats.idre.ucla.edu/stat/data/binary.csv")
> mydata = mydata[,-4]
> head(mydata)
      admit  gre   gpa
1     0   380   3.61
2     1   660   3.67
3     1   800   4.00
4     1   640   3.19
5     0   520   2.93
6     1   760   3.00
```

```
fit = glm(admit~gre+gpa,data=mydata,family="binomial")
summary(fit)
```

```
Coefficients:
Estimate Std. Error   z value    Pr(>|z|)
(Intercept) -4.949378   1.075093  -4.604     4.15e-06 ***
gre          0.002691   0.001057   2.544     0.0109 *
gpa          0.754687   0.319586   2.361     0.0182 *
```

```
hy.vec = predict(fit,mydata[,-1],type = "response")
hy.vec = ifelse(hy.vec>0.5,1,0)
t = table(mydata$admit,hy.vec)
1-sum(diag(t))/sum(t)
[1] 0.32
```