R Users Guide

to accompany

Statistics: Unlocking the Power of Data

by Lock, Lock, Lock, Lock, and Lock

About R

R is a freely available environment for statistical computing. R works with a command-line interface, meaning you type in commands telling R what to do. For more information and to download R, visit cran.r-project.org.

Using This Manual

A "Quick Reference Guide" at the end of this manual summarizes all the commands you will need to know for this course by chapter. More detailed information and examples are given for each chapter. If this is your first exposure to R, we recommend reading through the detailed chapter descriptions as you come to each chapter in the book.

Commands are given using color coding. Code in `red` represents commands and punctuation that always need to be entered exactly as is. Code in `blue` represents names that will change depending on the context of the problem (such as dataset names and variable names). Text in `green` following # is either optional code or comments. This often includes optional arguments that you may want to include with a function, but do not always need. In R anything following a # is read as a comment, and is not actually evaluated

For example, the command `mean` is used to compute the mean of a set of numbers. The information for this command is given in this manual as

```
mean(y)     #for missing data: na.rm=TRUE
```

Whenever you are computing a mean, you always need to type the parts in red, `mean( )`. Whatever you type inside the parentheses (the code in blue) will depend on what you have called the set of numbers you want to compute the mean of, so if you want to calculate the mean body mass index for data stored in a variable called BMI , you would type `mean(BMI)`. The code in green represents an optional argument needed only if there are missing values. If there were missing (NA) values in `BMI`, you would compute the mean with `mean(BMI, na.rm=TRUE)`.

Getting Started with R

Entering Commands

Commands can be entered directly into the R console (the window that opens when you start R), following the red > prompt, and sent to the computer by pressing enter. For example, typing 1 + 2 and pressing enter will output the result 3:

```
> 1+2
[1] 3
```

Your entered code always follows the > prompt, and output always follows a number in square brackets. Each command should take its own line of code, or else a line of code should be continued with { } (see examples in Chapters 3 and 4).

It is possible to press enter before the line of code is completed, and often R will recognize this. For example, if you were to type `1 +` but then press enter before typing `2`, R knows that `1+` by itself doesn't make any sense, so prompts for you to continue the line with a + sign. At this point you could continue the line by pressing 2 then enter. This commonly occurs if you forget to close parentheses or brackets. If you keep pressing enter and keep seeing a + sign rather than the regular > prompt that allows you to type new code, and if you can't figure out why, often the easiest option is to simply press ESC, which will get you back to the normal > prompt and allow you to enter a new line of code.

Capitalization and punctuation need to be exact in R, but spacing doesn't matter. If you get errors when entering code, you may want to check for these common mistakes:

- Did you start your line of code with a fresh prompt (>)? If not, press ESC.
- Are your capitalization and punctuation correct?
- Are all your parentheses and brackets closed? For every forward (, {, or [, make sure there is a corresponding backwards), }, or].

R Script

Rather than entering commands into the console directly however, we recommend creating and using an R Script, basically a text editor for your code. A new script can be created by File -> New Script. Code (commands) can be typed here, and then entered into the console in one of three ways:

1) Copy the code in the R script and paste in the console
2) Right-click on a line or highlighted group of lines and choose "Run line or selection"
3) Place your cursor on a line or highlight a group of lines and press CTRL+R.

Using a separate R script is nice because you can save only the code that works, making it easy to rerun and edit in the future, as opposed to the R console in which you would also have to save all your mistakes and all the output. We recommend always saving your R Scripts so you have the commands easily accessible and editable for future use.

Basic Commands

Basic Arithmetic	
Addition	`+`
Subtraction	`-`
Multiplication	`*`
Division	`/`
Exponentiation	`^`
Other	
Naming objects	`=`
Open help for a command	`?`
Creating a set of numbers	`c(1, 2, 3)`

The basic arithmetic commands are pretty straightforward. For example, 1 + (2*3) would return 7.

You can also name the result of any command with a name of your choosing with =. For example, if you type

```
x = 3*4
```

you are setting x to equal the result of 3*4, or equivalently setting x = 12. If you type in x to the console now you will see 12 as the output:

```
> x
[1] 12
```

The choice of x here is completely arbitrary, and you could have named it whatever you wanted.

Naming objects and arithmetic works not just with numbers, but with more complex objects like variables. To get a little fancier, suppose you have variables called `Weight` (measured in pounds) and `Height` (measured in inches), and want to create a new variable for body mass index, which you decide to name `BMI`. You can do this with the following code:

```
BMI = Weight/(Height^2) * 703
```

If you want to create your own variable or set of numbers, you can collect numbers together into one object with `c( )` and the numbers separated by commas inside the parentheses. For example, to create your own variable `Weight` out of the weights 125, 160, 183, and 137, you would type

```
Weight = c(125, 160, 183, 137)
```

To get more information on any built-in R commands, simply type `?` followed by the command name, and this will bring up a separate help page.

Using R in Chapter 1

Loading Data
Load a dataset from a .csv file
Load a dataset from the textbook
Type in a variable
	
	`dataname = read.csv(file.choose())`
	`data(dataname)`
	`variablename = c(3.2, 3.3, 3.1)`

Viewing Data
See the whole dataset, `dataname`
See the first 6 rows of a dataset
Finding the number of cases and variables
Get information about a textbook dataset
	
	`dataname`
	`head(dataname)`
	`dim(dataname)`
	`?dataname`

Variables
Seeing the variable names in a dataset
Extract a variable from a dataset
	
	`names(dataname)`
	`dataname$variablename`

Random Sample Generate n random integers up to max	 `sample(1:max, n)`

Loading Data

There are three different ways you may want to get data in R: loading data from a spreadsheet, loading datasets from the textbook, and manually typing in your own data.

Loading Data from a Spreadsheet

1. From your spreadsheet editing program (Excel, Google Docs, etc.) save your spreadsheet as a .csv (Comma Separated Values) file on your computer.
2. In R, decide on a name for your dataset. Usually a short name relevant to the particular dataset is best. For now, let's assume you picked the name `mydata`.
3. Type `mydata = read.csv(file.chose())` and press enter. A window will pop up asking you to locate the relevant .csv file on your computer.

Loading Data from the Textbook

1. Load the Lock5Data package[1]. Click on Packages at the top, then Install Packages. A window titled "CRAN mirror" will pop up – click on whatever location is closest to you and click OK. A window titled "Packages" will pop up – scroll down to click on Lock5Data, then click OK. (*Note: You only have to do this the first time you use textbook data.*)
2. Load this package by typing `library(Lock5Data)`. You'll have to do this every time you start a new R session.
3. Find the name of the dataset you want to access as it's written in bold in the textbook, for example, **AllCountries**, and type `data(AllCountries)`.

Manually Typing Data

If you survey people in your class asking for GPA, you could create a new variable called `gpa` (or whatever you want to call it) by entering the values as follows:

```
gpa = c(2.9, 3.0, 3.6, 3.2, 3.9, 3.4, 2.3, 2.8)
```

[1] If you can't install packages, you can access the datasets from the textbook as .csv files at www.wiley.com/college/lock

Viewing Data

Once you have a dataset loaded, you will want to explore different basic aspects of it, such as the structure, the names of the variables, and the number of cases. Let's work with the **AllCountries** data, loaded above. To view the dataset, simply type the dataset name

```
AllCountries
```

If there are a lot of cases, this may be awkward to see. Often it is useful to just view the first 6 rows of a dataset to a quick feel for the structure:

```
head(AllCountries)
```

If you want to find the number of cases and variables, type

```
dim(AllCountries)
```

The first number is the number of rows (cases) and the second is the number of columns (variables).

If the dataset comes from the textbook, you can type ? followed by the data name to pull up information about the data:

```
?AllCountries
```

Variables

If you want to see just the variable names, type

```
names(AllCountries)
```

If you want to extract a particular variable from a dataset, for example, Population, type

```
AllCountries$Population
```

If you will be doing a lot with one dataset, sometimes it gets cumbersome to always type the dataset name and a dollar sign before each variable name. To avoid this, you can type

```
attach(AllCountries)
```

Now you can access variables from the AllCountries data simply by typing the variable names directly. If you choose to use this option however, just remember to detach the dataset when you are done:

```
detach(AllCountries)
```

Taking a Random Sample

While you can sample directly from a list of cases in R, a more general way to generate a random sample is to randomly generate n (the sample size) numbers between 1 and the number of cases you want to sample from (max):

```
sample(1:max, n)
```

Once you have these random numbers, you can use this with either a dataset or a variable to create your random sample using square brackets.

A vector of numbers in square brackets after a variable says to only look at cases corresponding to the given numbers. For example, with our gpa variable, if we want only the 1st and 3rd cases, we could type:

```
gpa = c(2.9, 3.0, 3.6, 3.2, 3.9, 3.4, 2.3, 2.8)
gpa[c(1,3)]
```

to get a new variable of just 2.9 and 3.6. For example, if we wanted to take a random sample of 10 countries from all the 213 countries in the world, because Country within the dataset AllCountries lists the country names identifying each case, we could use

```
AllCountries$Country[sample(1:213, 10)]
```

This is useful if you have the case identifiers for the whole population, but not the data.

If you want to take a random sample from an entire dataset, indicate which rows and which columns you want within the square brackets, separated by a comma:

```
data[rows, columns]
```

So to take a random sample of 10 countries along with all the associated variables in the AllCountries dataset, we could use

```
AllCountries[sample(1:213, 10), ]
```

Notice the only difference when sampling a dataset versus a single column is the comma after the `sample()` command.

Randomized Experiment

If you want to randomize a sample into two different treatment groups for a randomized experiment, you can take a random sample from the whole sample to be the treatment group, and the rest of the sample would then go in the control group.

Using R in Chapter 2

One Categorical (x)
Frequency table
Proportion in group A
Pie chart
Bar chart
	
	`table(x)`
	`mean(x == "A")`
	`pie(table(x))`
	`barplot(table(x))`

Two Categorical (x1, x2)
Two-way table
Difference in proportions of x1 in group A by x2
Segmented bar chart
Side-by-side bar chart
	
	`table(x1, x2)`
	`diff(by(x1,x2,function(o) mean(o=="A")))`
	
	`barplot(table(x1, x2), legend=TRUE)`
	`barplot(table(x1,x2),legend=TRUE,beside=TRUE)`

One Quantitative (y)
Mean
Median
Standard deviation
5-Number summary
Percentile
Histogram
Boxplot
	
	`mean(y)  #for missing data: na.rm=TRUE`
	`median(y)  #for missing data: na.rm=TRUE`
	`sd(y)  #for missing data: na.rm=TRUE`
	`summary(y)`
	`quantile(y, 0.05)`
	`hist(y)`
	`boxplot(y) #ylab="y-axis label"`

One Quantitative (y) and One Categorical (x)
Means by group
Difference in means
S.D. by group
Side-by-side boxplots
	
	`by(y, x, mean) #for missing data: na.rm=TRUE`
	`diff(by(y, x, mean))`
	`by(y, x, sd)`
	`boxplot(y ~ x) #ylab="y-axis label"`

Two Quantitative (y1, y2)
Scatterplot
Correlation
Linear Regression
	
	`plot(y1, y2)`
	`cor(y1, y2) #missing data: use="complete.obs"`
	`lm(response ~ explanatory)`

Time Series Plot y1 = time variable	`plot(y1, y2, type="o")`

Additional Variables
Scatterplot of y1, y2
Point size: y3
Points colored by x1
Point symbol: x2
Broken down by: x3
	`#uses ggplot2 package (much easier)`
	`install.packages(ggplot2)`
	`library(ggplot2)`
	`qplot(y1, y2, size=y3, colour=x1, shape=x2,`
	`facet=~x3)`

Example – Student Survey

To illustrate these commands, we'll explore the **StudentSurvey** data. We load the data, attach it, and use head() to see what the data looks like:

```
library(Lock5Data)
data(StudentSurvey)
attach(StudentSurvey)
```

```
head(StudentSurvey)
```

The following are commands we could use to explore each of the following variables or pairs of variables. They are not the only commands we could use, but illustrate some possibilities.

Award preferences (one categorical variable):

```
table(Award)
barplot(table(Award))
```

Award preferences by gender (two categorical variables):

```
table(Award, Gender)
barplot(table(Award, Gender), legend=TRUE)
```

Pulse rate (one quantitative variable):

```
summary(Pulse)
hist(Pulse)
```

Hours of exercise per week by award preference (one quantitative and one categorical variable):

```
by(Pulse, Award, mean)
boxplot(Pulse~Award)
```

Pulse rate and SAT score (two quantitative variables):

```
plot(Pulse, SAT)
cor(Pulse, SAT)
lm(SAT~Pulse)
```

Missing Data

You may notice that if you try to do some of these commands on certain variables, you get NA for a result. This often means there are some missing values in the data, which R codes as NA. To calculate the average avoiding missing values, use the argument `na.rm=TRUE`:

```
mean(Exercise, na.rm=TRUE)
by(Exercise, Award, mean, na.rm=TRUE)
```

For correlation a similar problem exists, but the fix just takes a different argument. To calculate the correlation between SAT score and GPA (for which there are missing values), use

```
cor(SAT, GPA, use = "complete.obs")
```

More Details for Plots

If you want to get a bit fancier, you can add axis labels and titles to your plots. This is especially useful for including units, or if your variable names are not self-explanatory. You can specify the x-

axis label with `xlab`, the y-axis label with `ylab`, and a title for the plot with `main`. For example, below would produce a labeled scatterplot of height versus weight:

```
plot(Height, Weight, xlab = "Height (in inches)", ylab = "Weight
(pounds)", main = "Scatterplot")
```

These optional labeling arguments work for any graph produced.

Additional Variables (Section 2.7)

Let's add additional variables to the basic scatterplot. This can be done in base R, but is a bit tricky (particularly to get the legend correct), so we instead use the ggplot2 package. We first need to install and load the package:

```
install.packages(ggplot2)
library(ggplot2)
```

Below we can create a scatterplot of `Height` and `Weight`, with points colored by `Gender`, sized by hours of `Exercise`, symbol determined by `Award`, and broken down by `Year`:

```
qplot(Height, Weight, colour=Gender, shape=Award, size=Exercise,
facets=~Year, data=StudentSurvey)
```

Data over Time

Data over time can be plotted just as you would plot a normal scatterplot, but with the extra argument type = "o" for overlaid points and a line connecting them, or type = "l" for just the line.

The dataset CarbonDioxide gives Carbon Dioxide concentration in the atmosphere every year from 1960-2010. We can create a time series plot with

```
data(CarbonDioxide) #loads the data
attach(CarbonDioxide)
plot(Year, CO2, type="o")
```

Using R in Chapter 3

Generating a Bootstrap Distribution	`b = 10000 #number of bootstrap statistics`
	`boot.dist = rep(NA, b)`
	`for (i in 1:b) {`
	`  boot.sample = sample(n, replace=TRUE)`
	`  boot.dist[i] = statistic(y[boot.sample])`
	`}`

Using a Bootstrap Distribution	`hist(boot.dist)`
	`quantile(boot.dist, c(0.025, 0.975))`
	`sd(boot.dist)`

To generate a bootstrap confidence interval we first learn how to generate one bootstrap statistic, then how to repeat this procedure many times to generate an entire bootstrap distribution, and then how to use the bootstrap distribution to calculate a confidence interval.

One Bootstrap Statistic

To generate a bootstrap distribution we first have to be comfortable generating a single bootstrap statistic. To do this, we sample with replacement from the original sample, using a sample size equal to the original sample, and then compute the statistic of interest on this bootstrap sample. We create `boot.sample` to be a random sample of n (the sample size) integers between 1 and n, sampled with replacement:

```
boot.sample = sample(n, replace=TRUE)
```

For example,

```
sample(4, replace = TRUE)
```

could yield `2, 2, 1, 4`. To use this to get a bootstrap sample from our variable, we use square brackets, [ ] to select those cases from the variable. For example, if we wanted to create a bootstrap sample of Atlanta commute times (`Time`), which has 500 values originally, we would use

```
boot.sample = sample(500, replace=TRUE)
Time[boot.sample]
```

Lastly, we compute our statistic of interest on this bootstrap sample. For example, for the mean Atlanta commute time we would use

```
mean(Time[boot.sample])
```

If we instead were doing a correlation between `Distance` and `Time`, we would use

```
cor(Distance[boot.sample], Time[boot.sample])
```

For Loop

A *for loop* is a convenient way to repeat a procedure many times, without having to type it over and over again. The code

```
for (i  in 1:5) { }
```

tells the computer to do whatever is inside of the brackets 5 times, once with i = 1, once with i = 2, etc. up to i = 5. We use `for (i in 1:b)`, where `b` is the number of bootstrap statistics we want (usually 10,000 is sufficient).

We want to save the bootstrap statistic from every iteration of the for loop (for every i). To do this we first need to create a new object to save these results in:

```
boot.dist = rep(NA, b)
```

This creates a set of b NA values which will become the bootstrap distribution. Within the for loop,

```
boot.dist[i] = statistic(boot.y, boot.x)
```

tells each iteration of the for loop to fill in the i^{th} value of boot.dist with the bootstrap statistic from that iteration. When the for loop has finished, `boot.dist` has `b` different bootstrap statistics.

Before continuing, let's look at a simple for loop example:

```
result = rep(NA, 5)
for (i in 1:5) {
  result[i] = i + 2
}
```

After the for loop runs, `result` would be `3, 4, 5, 6, 7`, because the for loop when i = 1 would fill in the 1^{st} value of result with i + 2 = 1 + 2 = 3, when i = 2 it would fill in the 2^{nd} value of result with i + 2 = 2 + 2 = 4, etc., up to i = 5.

Using the Bootstrap Distribution

Once you have the bootstrap distribution, you should graph it to make sure it is approximately symmetric. Assuming it is, you have two options for computing a 95% confidence interval:

1. Compute the standard error of the statistic by taking the standard deviation of the bootstrap statistics and then use statistic $\pm$ 2·SE.

2. Find the endpoints that correspond to the middle 95% of the bootstrap distribution. For the middle 95%, we want the 2.5% in each tail, so the 0.025 and 0.975 percentiles (for other confidence levels, just replace 0.025 and 0.975 with the appropriate percentiles):

   ```
   quantile(boot.dist, c(0.025, 0.975))
   ```

Example: Bootstrap CI for a Correlation

Let's compute a 95% confidence interval for the correlation between `Distance` and `Time` for **AtlantaCommutes**. Load and attach the data:

```
library(Lock5Data)
data(CommuteAtlanta)
attach(CommuteAtlanta)
```

Create a bootstrap distribution:

```
b = 10000
boot.dist = rep(NA, b)
for (i in 1:b) {
    boot.sample = sample(500, replace=TRUE)
    boot.dist[i] = cor(Distance[boot.sample], Time[boot.sample])
}
```

Graph the bootstrap distribution to check for symmetry:

```
hist(boot.dist)
```

There is possibly a slight left skew, but nothing too concerning.

Create a 95% confidence interval using the standard error:

```
se = sd(boot.dist)
cor(Distance, Time) - 2*se
cor(Distance, Time) + 2*se
```

Although it's redundant to do both, for illustration we also use the percentile method:

```
quantile(boot.dist, c(0.025, 0.975))
```

Using R in Chapter 4

Generating a Randomization Distribution	`b = 10000 #number of randomization statistics`
	`rand.dist = rep(NA, b)`
	`for (i in 1:b) rand.dist[i] = rand.statistic`

Randomization Statistic: Proportion	`rbinom(1, n, 0.5) #change 0.5 to null value`
Mean	`boot.samp = sample(n, replace=TRUE)` `mean(y[boot.samp] + shift) #shift is amount to shift original data to make null true`
Shuffle one variable (x)	`sample(x) #randomly permutes x`

Finding p-value:
Proportion ≤ statistic
Proportion ≥ statistic
	`#double for two-sided` H_a
	`mean(rand.dist <= statistic)`
	`mean(rand.dist >= statistic)`

The general idea with the `for` loop is the same as we learned in Chapter 3 with bootstrapping, although if there is only one line of code within the `for` loop, we put it on one line for simplicity. For `b` iterations we calculate a randomization statistic, each time storing it as a value in `rand.dist`.

Shuffling One Variable

In many hypothesis tests, we generate a randomization sample by shuffling one of the two variables (the explanatory variable if the data comes from a randomized experiment). For example, suppose we want to do a hypothesis test to see if caffeine increases finger tapping rate, based on Data 4.6.

We load and attach the dataset **CaffeineTaps**, and see that the explanatory variable is `Group` and the response variable is `Taps`. To randomly shuffle the explanatory variable, we again use `sample`. If you sample a variable without specifying the sample size and without replacement, R randomly shuffles (permutes) the variable. So to shuffle `Group` we use

```
sample(Group)
```

For this problem we want to calculate the difference in mean tap rate for the shuffled groups:

```
diff(by(Tap, sample(Group), mean))
```

If we instead wanted a difference in proportions or correlation, we would calculate the statistic as always, just using the shuffled group instead of the actual group variable.

Now that we know how to calculate one randomization statistic, and know how to do a `for` loop from Chapter 3, creating a randomization distribution is easy!

```
b = 10000
rand.dist = rep(NA, b)
for (i in 1:b) rand.dist[i] = diff(by(Tap, sample(Group), mean))
```

To compute a p-value from this randomization distribution, we want the proportion of randomization statistics *above* the observed statistic (because we want to see if caffeine *increases* tap rate):

```
originalstat = diff(by(Tap, Group, mean))
mean(rand.dist >= originalstat)
```

Test for a Single Variable

Doing tests for a single variable is a bit different, because there is not an explanatory variable to shuffle. Here are two examples, one for a proportion, and one for a mean.

1. *Dogs and Owners*. 16 out of 25 dogs were correctly paired with their owners, is this evidence that the true proportion is greater than 0.5? In R, you can simulate flipping 25 coins and counting the number of heads with

   ```
   rbinom(1, 25, 0.5)
   ```

   (for null proportions other than 0.5, just change the 0.5 above accordingly). Therefore, we can create a randomization distribution with

   ```
   b = 10000
   rand.dist = rep(NA, b)
   for (i in 1:b) rand.dist[i] = rbinom(1, 25, 0.5)
   ```

   The alternative is upper-tailed, so we compute a p-value as the proportion above 16/25:

   ```
   mean(rand.dist >= 16/25)
   ```

2. *Body Temperature.* Is a sample mean of 98.26°F based on 50 people evidence that the true average body temperature in the population differs from 98.6°F? To answer this we create a randomization distribution by bootstrapping from a sample that has been shifted to make the null true, so we add 0.34 to each value. We can create the corresponding randomization distribution with

   ```
   b = 10000
   rand.dist = rep(NA, b)
   NullTemp=BodyTemp+0.34
   for (i in 1:b) {
      boot.samp = sample(50, replace=TRUE)
      rand.dist[i] = mean(NullTemp[boot.samp])
   }
   ```

   In this case we have a two-sided H_a, so calculate the p-value as the proportion in the smaller tail beyond the observed statistic (in this case ≤ because 98.26 is less than 98.6), and double it:

   ```
   2*mean(rand.dist <= 98.26)
   ```

Using R in Chapter 5

Standard Normal Distribution:	
Find a percentile	`qnorm(0.10)`
Find the area below z	`pnorm(z)`
Find the area above z	`pnorm(z, lower.tail=FALSE)`
Any Normal Distribution	
Find a percentile	`qnorm(0.10,mean,stddev)`
Find the area below x	`pnorm(x,mean,stddev)`
Find the area above x	`pnorm(x,mean,stddev,lower.tail=FALSE)`

Example 1: Find z^* for a 90% confidence interval.

We want the middle 90% of the normal distribution, so want 5% in each tail, so need to find the 5th and 95th percentiles:

```
qnorm(0.05)
qnorm(0.95)
```

Example 2: Find a p-value when z = 1.5, and the alternative is upper-tailed.

Because H_a is upper tailed, we find the area in the standard normal distribution above 1.5:

```
pnorm(1.5, lower.tail=FALSE)
```

Example 3: Find the area below 60 for a normal distribution with mean 75 and standard deviation 12.

```
pnorm(60,75,12)
```

Using R in Chapter 6

Normal Distribution:	
Find a percentile	`qnorm(0.10)`
Find the area below z	`pnorm(z)`
Find the area above z	`pnorm(z, lower.tail=FALSE)`
t-Distribution:	
Find a percentile	`qt(0.10, df)`
Find the area below t	`pt(t, df, lower.tail=TRUE)`
Find the area above t	`pt(t, df, lower.tail=FALSE)`
Inference for Proportions:	
Single proportion	`prop.test(count, n, p0) #delete p0 for intervals`
Difference in proportions	`prop.test(c(count1, count2), c(n1, n2))`
Inference for Means:	
Single mean	`t.test(y, mu = mu0) #delete mu0 for intervals`
Difference in means	`t.test(y ~ x)`
Using prop.test or t.test	
For p-values	`#alternative = "two.sided", "less", "greater"`
For confidence intervals	`#conf.level = 0.95 or desired confidence level`

There are two ways of using R to compute confidence intervals and p-values using the normal and t-distributions:

1. Use the formulas in the book and `qnorm, qt, pnorm,` and `pt`.
2. Use `prop.test` and `t.test` on the raw data without using any formulas

The two methods should give very similar answers, but may not match exactly because `prop.test` and `t.test` do things slightly more complicated than what you have learned (continuity correction for proportions, and a more complicated algorithm for degrees of freedom for difference in means).

The commands `prop.test` and `t.test` give both confidence intervals and p-values. For confidence intervals, the default level is 95%, but other levels can be specified with `conf.level`. For p-values, the default is a two-tailed test, but the alternative can be changed by specifying either `alternative = "less"` or `alternative = "greater"`.

Using Option 1 directly parallels the code in Chapter 5, so we refer you to the Chapter 5 examples. Here we just illustrate the use of `prop.test` and `t.test`.

Example 1: In a recent survey of 800 Quebec residents, 224 thought that Quebec should separate from Canada. Give a 90% confidence interval for the proportion of Quebecers who would like Quebec to separate from Canada.

```
prop.test(224, 800, conf.level=0.90)
```

Example 2: Test whether caffeine increases tap rate (based on CaffeineTaps data).

```
t.test(Tap~Group, alternative = "greater")
```

Using R in Chapter 7

Chi-Square Distribution Find the area above χ^2	`pchisq(chisquare, df, lower.tail=FALSE)`
Chi-Square Test Goodness-of-fit Test for association	`chisq.test(table(x)) #if null probabilities not equal, use p = c(p1, p2, p3) to specify` `chisq.test(table(x1, x2))`
Randomization Test Goodness-of-fit Test for association	`chisq.test(table(x), simulate.p.value=TRUE)` `chisq.test(table(x1, x2), simulate.p.value=TRUE)`

Option 1: Use formula to calculate chi-square statistic and use `pchisq`

If we get $\chi^2 = 3.1$ and the degrees of freedom are 2, we would calculate the p-value with

```
pchisq(3.1, 2, lower.tail=FALSE)
```

Option 2: Use `chisq.test` on raw data

1. *Goodness of Fit.* Use the data in **APMultipleChoice** to see if all five choices (A, B, C, D, E) are equally likely:

   ```
   chisq.test(table(Answer))
   ```

2. *Test for Association.* Use the data in **StudentSurvey** to see if type of award preference is associated with gender:

   ```
   chisq.test(table(Award, Gender))
   ```

Randomization Test

If the expected counts within any cell are too small, you should not use the chi-square distribution, but instead do a randomization test. If you use `chisq.test` with small expected counts cell, R helps you out by giving a warning message saying the chi-square approximation may be incorrect.

If the sample sizes are too small to use a chi-squared distribution, you can do a randomization test with the optional argument `simulate.p.value` within the command `chisq.test`. This tells R to calculate the p-value by simulating the distribution of the χ^2 statistic, assuming the null is true, rather than compare it to the theoretical chi-square distribution.

For example, for a randomization test for an association between Award and Gender:

```
chisq.test(table(Award, Gender), simulate.p.value=TRUE)
```

Using R in Chapter 8

F Distribution Find the area above F	`pf(F, df, lower.tail=FALSE)`
Analysis of Variance	`summary(aov(y ~ x))`
Pairwise Comparisons	`pairwise.t.test(y, x, p.adjust="none")`

As with t-tests and chi-square tests, one option is to calculate the F-statistic by hand, and then compare it to the F distribution using `pf`. The (much easier) option is to use R's built in analysis of variable function, `aov`.

Analysis of Variance

Let's test whether the average number of ants that feed on a sandwich differs by type of filling, using data from **SandwichAnts** (Data 8.1).

We can calculate the sample means in each group, visualize the data, and check the conditions for ANOVA with

```
by(Ants, Filling, mean)
boxplot(Ants ~ Filling)
by(Ants, Filling, sd)
table(Filling)
```

In the sample, we see that the most ants came to the ham & pickles sandwich, and the least to the vegemite. The sample standard deviations within each group are close enough to proceed. The sample sizes are very small, so we should proceed with caution, but looking at the boxplots we see the data appear to be at least symmetrically distributed within each group, without any outliers, so we proceed.

We can calculate the entire ANOVA table directly with

```
summary(aov(Ants ~ Filling))
```

Pairwise Comparisons

Finding the overall ANOVA significant, we may want to test individual pairwise comparisons. We can test all pairwise comparisons with

```
pairwise.t.test(Ants, Filling, p.adjust = "none")
```

This gives us the p-value corresponding to each pairwise comparison. The optional argument `p.adjust = "none"` tells R to give the raw p-values and not adjust for multiple comparisons. If you leave off this argument R will increase the p-values to account for multiple comparisons, but the details here are beyond the scope of this text.

Using R in Chapter 9

Simple Linear Regression	
Plot the data	`plot(y ~ x) # y is the response (vertical)`
Fit the model	`lm(y ~ x)  # y is the response)`
Give model output	`summary(model)`
Add regression line to plot	`abline(model)`
Inference for Correlation	`cor.test(x, y)` `#alternative = "two.sided", "less", "greater"`
Prediction	
Calculate predicted values	`predict(model)`
Calculate confidence intervals	`predict(model, interval = "confidence")`
Calculate prediction intervals	`predict(model, interval = "prediction")`
Prediction for new data	`predict(model, as.data.frame(cbind(x=1)))`

Let's load and attach the data from **RestaurantTips** to regress `Tip` on `Bill`. Before doing regression, we should plot the data to make sure using simple linear regression is reasonable:

```
plot(Tip~Bill)     #Note:  plot(Bill, Tip) does the same
```

The trend appears to be approximately linear. There are a few unusually large tips, but no extreme outliers, and variability appears to be constant as `Bill` increases, so we proceed. We fit the simple linear regression model, saving it under the name `mod` (short for model - you can call it anything you want). Once we fit the model, we use `summary` to see the output:

```
mod = lm(Tip ~ Bill)
summary(mod)
```

Results relevant to the intercept are in the (Intercept) row and results relevant to the slope are in the `Bill` (the explanatory variable) row. The estimate column gives the estimated coefficients, the std. error column gives the standard error for these estimates, the t value is simply estimate/SE, and the p-value is the result of a hypothesis test testing whether that coefficient is significantly different from 0.

We also see the standard error of the error as "Residual standard error" and R^2 as "Multiple R-squared". The last line of the regression output gives details relevant to the ANOVA table: the F-statistic, degrees of freedom, and p-value.

After creating a plot, we can add the regression line to see how the line fits the data:

```
abline(mod)
```

Suppose a waitress at this bistro is about to deliver a $20 bill, and wants to predict her tip. She can get a predicted value and 95% (this is the default level, change with `level`) prediction interval with

```
predict(mod,as.data.frame(cbind(Bill = 20)),interval = "prediction")
```

Lastly, we can do inference for the correlation between `Bill` and `Tip`:

```
cor.test(Bill, Tip)
```

Using R in Chapter 10

Multiple Regression	
Fit the model	`lm(y ~ x1 + x2)`
Give model output	`summary(model)`
Residuals	
Calculate residuals	`model$residuals`
Residual plot	`plot(predict(model), model$residuals)`
Histogram of residuals	`hist(model$residuals)`
Prediction	
Calculate predicted values	`predict(model)`
Calculate confidence intervals	`predict(model, interval = "confidence")`
Calculate prediction intervals	`predict(model, interval = "prediction")`
Prediction for new data	`predict(model,as.data.frame(cbind(x1=1,x2=3)))`

Multiple Regression Model

We'll continue the **RestaurantTips** example, but include additional explanatory variables: number in party (`Guests`), and whether or not they pay with a credit card (`Credit` = 1 for yes, 0 for no).

We fit the multiple regression model with all three explanatory variables, call it `tip.mod`, and summarize the model:

```
tip.mod = lm(Tip ~ Bill + Guests + Credit)
summarize(tip.mod)
```

This output should look very similar to the output from Chapter 9, except now there is a row corresponding to each explanatory variable.

Conditions

To check the conditions, we need to calculate residuals, make a residual versus fitted values plot, and make a histogram of the residuals:

```
plot(tip.mod$fit, tip.mod$residuals)
hist(tip.mod$residuals)
```

Categorical Variables

While `Credit` was already coded with 0/1 here, this is not necessary for R. You can include any explanatory variable in a multiple regression model, and R will automatically create corresponding 0/1 variables. For example, if you were to include gender coded as male/female, R would create a variable `GenderMale` that is 1 for males and 0 for females. The only thing you should *not* do is include a categorical variable with more than two levels that are all coded with numbers, because R will treat this as a quantitative variable.

R Commands: Quick Reference Sheet

CHAPTER 1

Loading Data	
Load a dataset from a .csv file	`dataname = read.csv(file.choose())`
Load a dataset from the textbook	`data(dataname)`
Type in a variable	`variablename = c(3.2, 3.3, 3.1)`
Viewing Data	
See the whole dataset, `dataname`	`dataname`
See the first 6 rows of a dataset	`head(dataname)`
Finding the number of cases and variables	`dim(dataname)`
Variables	
Seeing the variable names in a dataset	`names(dataname)`
Extract a variable from a dataset	`dataname$variablename`
Random Sample	
Generate n random integers up to max	`sample(1:max, n)`

CHAPTER 2

One Categorical (x)	
Frequency table	`table(x)`
Proportion in group A	`mean(x == "A")`
Pie chart	`pie(table(x))`
Bar chart	`barplot(table(x))`
Two Categorical (x1, x2)	
Two-way table	`table(x1, x2)`
Difference in proportions of x1 in group A by x2	`diff(by(mean(x1,x2,function(o) mean(o=="A")))`
Segmented bar chart	`barplot(table(x1, x2), legend=TRUE)`
Side-by-side bar chartt	`barplot(table(x1,x2),legend=TRUE,beside=TRUE)`
One Quantitative (y)	
Mean	`mean(y)  #for missing data: na.rm=TRUE`
Median	`median(y)  #for missing data: na.rm=TRUE`
Standard deviation	`sd(y)  #for missing data: na.rm=TRUE`
5-Number summary	`summary(y)`
Percentile	`quantile(y, 0.05)`
Histogram	`hist(y)`
Boxplot	`boxplot(y) #ylab="y-axis label"`
One Quantitative (y) and One Categorical (x)	
Means by group	`by(y, x, mean) #for missing data: na.rm=TRUE`
Difference in means	`diff(by(y, x, mean))`
S.D. by group	`by(y, x, sd)`
Side-by-side boxplots	`boxplot(y ~ x) #ylab="y-axis label"`
Two Quantitative (y1, y2)	
Scatterplot	`plot(y1, y2)`
Correlation	`cor(y1, y2) #missing data: use="complete.obs"`
Linear Regression	`lm(response ~ explanatory)`

Time Series Plot y1 = time variable	`plot(y1, y2, type="o")`

Additional Variables
Scatterplot of y1, y2
Point size: y3
Points colored by x1
Point symbol: x2
Broken down by: x3
	`#uses ggplot2 package (much easier)`
	`install.packages(ggplot2)`
	`library(ggplot2)`
	`qplot(y1, y2, size=y3, colour=x1, shape=x2, facet=~x3)`

CHAPTER 3

Generating a Bootstrap Distribution	`b = 10000 #number of bootstrap statistics`
	`boot.dist = rep(NA, b)`
	`for (i in 1:b) {`
	`  boot.sample = sample(n, replace=TRUE)`
	`  boot.dist[i] = statistic(y[boot.sample])`
	`}`

Using a Bootstrap Distribution	`hist(boot.dist)`
	`quantile(boot.dist, c(0.025, 0.975))`
	`sd(boot.dist)`

CHAPTER 4

Generating a Randomization Distribution	`b = 10000 #number of randomization statistics`
	`rand.dist = rep(NA, b)`
	`for (i in 1:b) rand.dist[i] = rand.statistic`

Randomization Statistic:
Proportion
 	
Mean
 	
Shuffle one variable (x)
	
	`rbinom(1, n, 0.5) #change 0.5 to null value`
	
	`boot.samp = sample(n, replace=TRUE)`
	`mean(y[boot.samp] + shift) #shift is amount to shift original data to make null true`
	
	`sample(x) #randomly permutes x`

Finding p-value:
Proportion ≤ statistic
Proportion ≥ statistic
	`#double for two-sided Ha`
	`mean(rand.dist <= statistic)`
	`mean(rand.dist >= statistic)`

CHAPTER 5

Standard Normal Distribution:
Find a percentile
Find the area below z
Find the area above z
	
	`qnorm(0.10)`
	`pnorm(z)`
	`pnorm(z, lower.tail=FALSE)`

CHAPTER 6

t-Distribution:
Find a percentile
Find the area below t
Find the area above t
	
	`qt(0.10, df)`
	`pt(t, df)`
	`pt(t, df, lower.tail=FALSE)`

Inference for Proportions:	

Single proportion Difference in proportions	`prop.test(count, n, p0) #delete p0 for intervals` `prop.test(c(count1, count2), c(n1, n2))`

Inference for Means:
Single mean
Difference in means
	
	`t.test(y, mu = mu0) #delete mu0 for intervals`
	`t.test(y ~ x)`

Using prop.test or t.test
For p-values
For confidence intervals
	
	`#alternative = "two.sided", "less", "greater"`
	`#conf.level = 0.95 or desired confidence level`

CHAPTER 7

Chi-Square Distribution Find the area above χ^2	 `pchisq(chisquare, df, lower.tail=FALSE)`

Chi-Square Test
Goodness-of-fit
 	
Test for association
	
	`chisq.test(table(x)) #if null probabilities not equal, use p = c(p1, p2, p3) to specify`
	`chisq.test(table(x1, x2))`

Randomization Test
Goodness-of-fit
Test for association
	
	`chisq.test(table(x), simulate.p.value=TRUE)`
	`chisq.test(table(x1, x2), simulate.p.value=TRUE)`

CHAPTER 8

F Distribution Find the area above F	 `pf(F, df, lower.tail=FALSE)`
Analysis of Variance	`summary(aov(y ~ x))`
Pairwise Comparisons	`pairwise.t.test(y, x, p.adjust="none")`

CHAPTER 9

Simple Linear Regression
Plot the data
Fit the model
Give model output
Add regression line to plot
	
	`plot(y ~ x) # y is the response (vertical)`
	`lm(y ~ x)  # y is the response)`
	`summary(model)`
	`abline(model)`

Inference for Correlation	`cor.test(x, y)` `#alternative = "two.sided", "less", "greater"`

Prediction
Calculate predicted values
Calculate confidence intervals
Calculate prediction intervals
Prediction for new data
	
	`predict(model)`
	`predict(model, interval = "confidence")`
	`predict(model, interval = "prediction")`
	`predict(model, as.data.frame(cbind(x=1)))`

CHAPTER 10

Multiple Regression
Fit the model
Give model output
	
	`lm(y ~ x1 + x2)`
	`summary(model)`

Residuals
Calculate residuals
	
	`model$residuals`
	`plot(predict(model), model$residuals)`

Residual plot Histogram of residuals	`hist(model$residuals)`

Prediction
Calculate predicted values
Calculate confidence intervals
Calculate prediction intervals
Prediction for new data
	
	`predict(model)`
	`predict(model, interval = "confidence")`
	`predict(model, interval = "prediction")`
	`predict(model,as.data.frame(cbind(x1=1,x2=3)))`