RStudio Users Guide

to accompany

Statistics: Unlocking the Power of Data

by Lock, Lock, Lock, Lock, and Lock

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)
```

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)`.

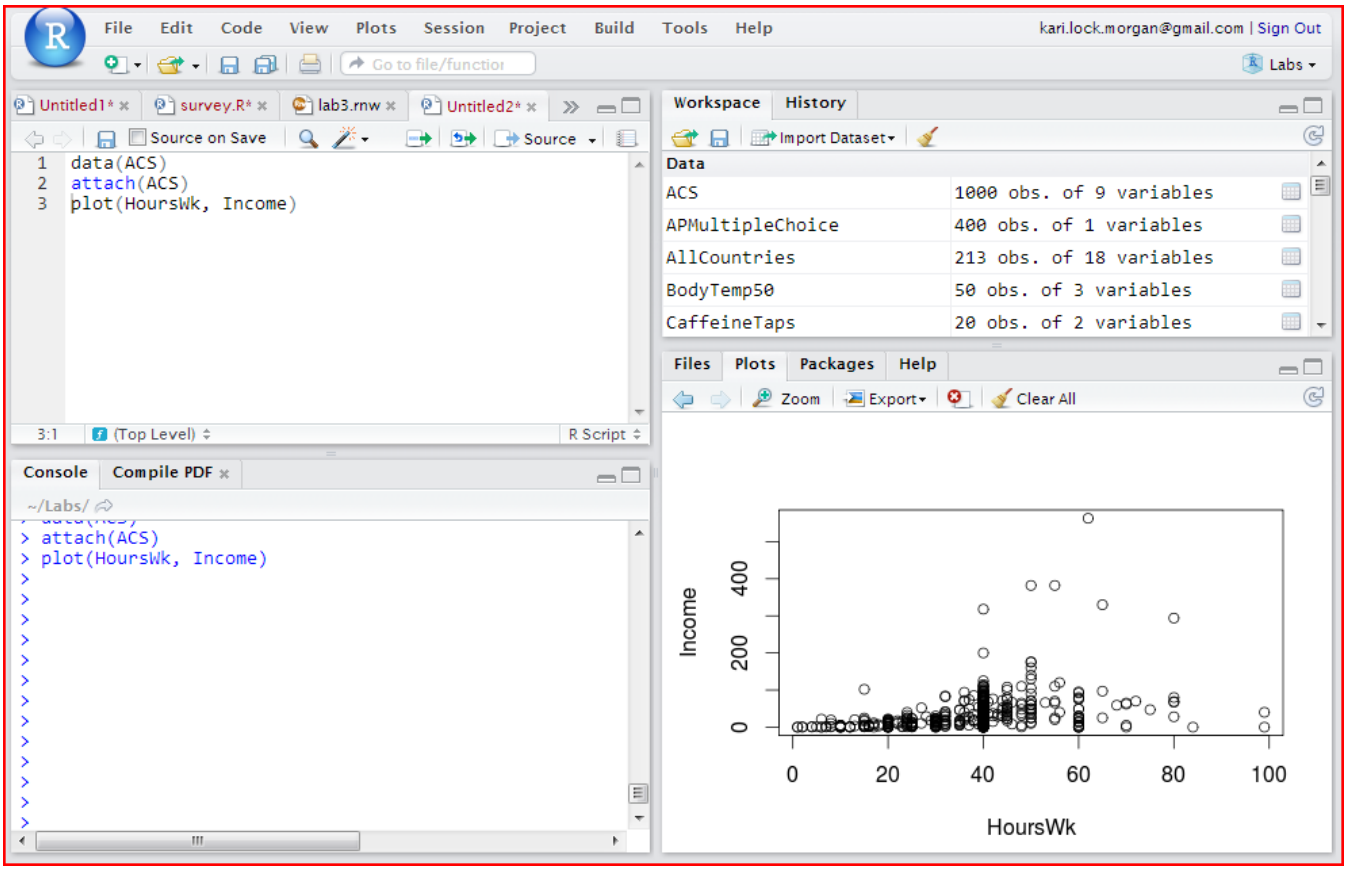
Text after # represents a comment - this is only for you, and R will ignore this code if it is typed.

IMPORTANT: Many commands in this manual require installation of the Lock5 package, which includes all datasets from the textbook, as well as many commands designed to make R coding easier for introductory students. This package only needs to be installed once, and can be installed with the following command:

```
source("/shared/kari.lock.morgan@gmail.com/Lock5.R")
```

About R and RStudio

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. RStudio is a convenient interface for using R, which can either be accessed online (http://beta.rstudio.org/) or downloaded to your computer. For more information about RStudio, go to http://www.rstudio.com/.



The bottom left panel is the console. Here you can type code directly to be sent to R.

The top left is called the RScript, and is basically a text editor that color codes for you and sends commands easily to 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. Code can be sent from the RScript to the console either by highlighting and clicking this icon: or else by typing CTRL+ENTER at the end of the line. Different RScripts can be saved in different tabs.

The top right is your Workspace and is where you will see objects (such as datasets and variables). Clicking on the name of a dataset in your workspace will bring up a spreadsheet of the data.

The bottom right serves many purposes. It is where plots will be appear, where you manage your files (including importing files from your computer), where you install packages, and where the help information appears. Use the tabs to toggle back and forth between these screens as needed.

Getting Started with RStudio

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)`

Entering Commands

Commands can be entered directly into the R console (bottom left), following the > 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.

You can also enter this code into the RScript and run it from there. Create a new RScript by File - New - R Script. Now you can type in the R Script (top left), and then send your code to the console either by pressing  or CTRL+ENTER. Try typing 1+2 in the R Script and sending it to the console.

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]. When working in the RScript if you click next to (, the corresponding) will be highlighted.

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

```
x = 3*4
```

sets x to equal the result of 3*4, or equivalently sets x = 12. The choice of x is arbitrary - you can name it whatever you want. If you type x into the console now you will see 12 as the output:

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

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.

Loading Data

There are several different ways you may want to get data in RStudio:

Loading Data from a Google Doc

1. From within the google spreadsheet, click File -> Publish to Web -> Start Publishing.
2. Type `google.doc("key")`, where key should be replaced with everything in between key= and # in the link for the google doc.

Loading Data from the Textbook

1. 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)`.

Loading Data from a Spreadsheet on your Computer

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 the bottom right panel, click the Files tab, then Upload. Choose the .csv file and click OK.
3. In the top right panel, click Import Dataset, From Text File, then choose the dataset you just uploaded. If needed adjust the options until the dataset looks correct, then click Import.

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)
```

Viewing Data

Once you have your dataset loaded, it should appear in your workspace (top right). Click on the name of the dataset to view the dataset as a spreadsheet in the top left panel. Click the tabs of that panel to get back to your RScript.

Using R in Chapter 1

Loading Data
Load a dataset from a google doc[1]
Load a dataset from the textbook
Help for textbook datasets
Type in a variable
	
	`google.doc("key")#key: between key= and # in url`
	`data(dataname)`
	`?dataname`
	`variablename = c(3.2, 3.3, 3.1)`

Variables
Extract a variable from a dataset
Attach a dataset
Detach a dataset
	
	`dataname$variablename`
	`attach(dataname)`
	`detach(dataname)`

Subsetting Data Take a subset of a dataset	 `subset(dataname, condition)`

Random Sample
Taking a random sample of size n
n random integers 1 to max
	
	`sample(dataname, n) #use for data or variable`
	`sample(1:max, n)`

Loading and Viewing Data

Let's load in the AllCountries data from the textbook with the following command:

```
data(AllCountries)
```

This loads the dataset, and you should see it appear in your workspace. To view the dataset, simply click on the name of the dataset in your Workspace and a spreadsheet of the data will appear in the top left. Scroll down to see all the cases and right to see all the 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 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)
```

[1] For your own google spreadsheet, within the google spreadsheet you first have to do File -> Publish to Web -> Start Publishing.

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)
```

Subsetting a Dataset

To take a subset from a dataset, say all countries with a population greater than 1 million (units are in millions), you can create a new dataset and use the subset command:

```
newdata = subset(AllCountries,  Population < 1)
```

You could have named it anything, I just choose newdata. If you didn't attach AllCountries or already detached it, then you need to use AllCountries$Population, not just Population.

Taking a Random Sample

You can take a random sample from either a variable or dataset with `sample`. Suppose we want to take a random sample of 10 countries from the Country variable in AllCountries, we could use

```
sample(Country, 10)
```

If we want to take a random sample of 10 rows from the AllCountries dataset, we could use

```
sample(AllCountries, 10)
```

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)
```

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	`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)`
Proportions by group	`mean(x1=="A"~x2)`
Difference in proportions	`diffProp(x1=="A"~x2)`
Segmented bar chart	`barplot(table(x1, x2), legend=TRUE)`
Side-by-side bar chart	`barplot(table(x1,x2),legend=TRUE,beside=TRUE)`
One Quantitative (y)	
Mean	`mean(y)`
Median	`median(y)`
Standard deviation	`sd(y)`
5-Number summary	`summary(y)`
Percentile	`percentile(y, 0.05)`
Histogram	`hist(y)`
Boxplot	`boxplot(y)`
One Quantitative (y) and One Categorical (x)	
Means by group	`mean(y ~ x)`
Difference in means	`diffMean(y ~ x)`
Standard deviation by group	`sd(y ~ x)`
Side-by-side boxplots	`boxplot(y ~ x)`
Two Quantitative (y1, y2)	
Scatterplot	`plot(y1, y2)`
Correlation	`cor(y1, y2)`
Labels	`#optional arguments for any plot:`
Add a title	`main = "title of plot"`
Label an axis	`xlab = "x-axis label", ylab = "y-axis label"`

Example – Student Survey

To illustrate these commands, we'll explore the **StudentSurvey** data. We load and attach the data:

```
data(StudentSurvey)
attach(StudentSurvey)
```

Click on the dataset name in the workspace to view the data and variable names.

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):

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

Pulse rate and SAT score (two quantitative variables):

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

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.

Using R in Chapter 3

Repeat Code 1000 Times	`do(1000)*`
Sampling Distribution for Mean	`do(1000)*mean(sample(y, n))`
Bootstrap Distribution for Mean	`do(1000)*mean(sample(y, n, replace=TRUE))`

Generating a Sampling Distribution for any Statistic	`do(1000)*{`
	`  samp = sample(pop.data, n)`
	`  statistic(samp$var1, samp$var2)`
	`}`

Manually Generating a Bootstrap Distribution for any Statistic	`do(1000)*{`
	`  boot.samp = sample(data, n, replace=TRUE)`
	`  statistic(boot.samp$var1, boot.samp$var2)`
	`}`

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

Generate a Bootstrap CI	`bootstrap.interval(var1, var2) #level = .95`

To create a sampling distribution or a bootstrap distribution, we need to first draw a random sample (with or without replacement), calculate the relevant statistic, and repeat this process many times.

As a review, we can select a random sample with the command `sample()`. We can then calculate the relevant statistic on this sample, as we learned how to do in Chapter 2. The new part is *do*ing this process many times.

do()

`do(1000)*` provides a convenient way to *do* something *1000* (or any desired number) times. R will repeat whatever follows the * 1000 times. If the code fits on one line then everything can be written after the * on the same line. If the code to be done multiple times takes up multiple lines, we can use {} as follows:

```
do(1000)*{
  code to be repeated
}
```

Example: Sampling Distribution for a Mean

We have population data on the gross income for all 2011 Hollywood movies stored in the dataset **HollywoodMovies2011** under the variable name *WorldGross*. Suppose we want to generate a sampling distribution for the mean gross income for samples of size 30. First, load and attach the data.

Let's first compute one statistic for the sampling distribution. We take a random sample of 30 values and call it *samp* (we could call it anything), and compute the sample mean:

```
samp = sample(WorldGross, 30)
mean(samp)
```

If preferred, we could have done this in all one line, by nesting commands:

```
mean(sample(WorldGross, 30))
```

Equivalently, we could take a random sample of 10 movies from the dataset (which is necessary for doing a sampling distribution for more than one variable), and compute the sample mean:

```
samp = sample(HollywoodMovies2011, 30)
mean(samp$WorldGross)
```

We now repeat this process many (say 1,000) times to form the sampling distribution, with do(). There are several options for how to do this:

```
do(1000)* mean(sample(WorldGross, 30))
```

or

```
do(1000)*{
   samp = sample(HollywoodMovies2011, 30)
   mean(samp$WorldGross)
}
```

If you want to, you can save this sampling distribution so you can do things like visualize it or take the standard deviation to compute the standard error:

```
samp.dist = do(1000)* mean(sample(WorldGross, 30))
hist(samp.dist)
sd(samp.dist)
```

Example: Bootstrap Confidence Interval for a Correlation

Let's create a bootstrap confidence interval for the correlation between time and distance for Atlanta commuters, based on a random sample of 500 Atlanta commuters stored in the dataset **CommuteAtlanta**. After loading and attaching the data, we can create one bootstrap sample of commuters by taking a sample of size 500 with replacement from the dataset:

```
sample(CommuteAtlanta, 500, replace=TRUE)
```

To compute a bootstrap statistic, we should give this bootstrap sample a name (we'll use `boot.samp`, although you could choose a different name if you like), and then compute the relevant statistic (correlation) on the variables taken from this bootstrap sample:

```
boot.samp = sample(CommuteAtlanta, 500, replace=TRUE)
cor(boot.samp$Time, boot.samp$Distance)
```

This gives us *one* bootstrap statistic. For an entire bootstrap distribution, we want to generate *thousands* of bootstrap statistics! We can use do() to repeat this process 1000 times:

```
boot.dist = do(1000)*{
     boot.samp = sample(CommuteAtlanta, 500, replace=TRUE)
     cor(boot.samp$Time, boot.samp$Distance)
}
```

We gave this distribution a name (`boot.dist`), so we are able to use it. We first visualize the distribution to make sure it is symmetric (and bell-shaped for the SE method):

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

It may be a little left-skewed, but in general doesn't look too bad, so we proceed. We can compute the standard error and use the standard error method for a 95% confidence interval:

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

We could also use the percentile method for a 90% confidence interval, chopping off 5% in each tail:

```
percentile(boot.dist, c(0.05, 0.95))
```

Example: bootstrap.interval()

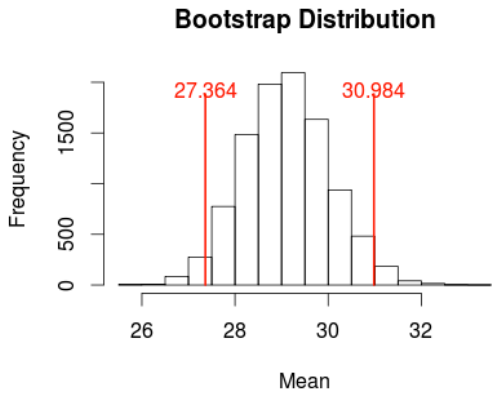
If manually coding bootstrap distributions isn't your thing, you could instead generate a bootstrap confidence interval with the built-in command `bootstrap.interval()`. Based on whether you have one or two variables and whether they are categorical or quantitative, resample automatically guesses at which parameter you are interested in, creates a bootstrap distribution and displays it for you, gives you the sample statistic and standard error, and gives you a confidence interval based on the percentile method for any desired level of confidence. (Life doesn't get much easier!)

For example, if you type have the **CommuteAtlanta** dataset loaded and attached, if you type

```
bootstrap.interval(Time)
```

R knows that Time is a quantitative variable, so you are most likely interested in a mean[2], and you will get the following output:

```
> bootstrap.interval(Time)
One quantitative variable
Observed Mean: 29.11
Resampling, please wait...
SE =  0.927
95% Confidence Interval:
2.5%   97.5%
27.3639 30.9840
```



If you want a 90% confidence interval for the difference in mean commute time by sex:

```
bootstrap.interval(Time, Sex, level = .90)
```

Or if you want a 99% confidence interval for the correlation between time and distance:

```
bootstrap.interval(Time, Distance, level=.99)
```

WARNING: While `bootstrap.interval()` makes generating a bootstrap confidence interval very easy, it also makes it easy to stop thinking about what you are doing. It is important to make sure you understand the process![3]

---

[2] If you are interested in a median instead, you can change this with the optional argument `stat = median`, although this bootstrap distribution would definitely not be smooth or symmetric.

[3] If you are a professor, you may want to teach `bootstrap.interval()` only after you are sure the students understand the process, or you may choose not to teach `bootstrap.interval()` at all, to encourage them to think through the process each time.

Using R in Chapter 4

Randomization Statistic:
Shuffle one variable (x)
Proportion
Mean
	
	`shuffle(x)`
	`coin.flips(n, p)`
	`mean(sample(y + shift, n, replace=TRUE))`

Randomization Distribution	`do(1000)*one randomization statistic`

Finding p-value from a randomization distribution:
Lower-tailed test
Upper-tailed test
Two-tailed test
	`#rand.dist = randomization distribution`
	`#obs.stat = observed sample statistic`
	`tail.p(rand.dist, obs.stat, tail="lower")`
	`tail.p(rand.dist, obs.stat, tail="upper")`
	`tail.p(rand.dist, obs.stat, tail="two")`

Randomization Test	`randomization.test(y, x) #null = for one var` `#tail="lower", "upper", "two"`

Example: Difference in Means

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), which is equivalent to rerandomizing the cases to treatment groups. For example, let's 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`. We first calculate the sample statistic, the observed difference in means:

```
diffMean(Tap~shuffle(Group))
```

We now want to see what kinds of statistics we would observe, just by random chance, if the null hypothesis were true, or equivalently, if the explanatory variable was randomly shuffled. To randomly shuffle the explanatory variable, we use

```
shuffle(Group)
```

We can then calculate the difference in mean tap rate for the shuffled groups:

```
diffMean(Tap~shuffle(Group))
```

Now that we know how to calculate one randomization statistic, and know how to use `do` to repeat something many times (see Chapter 3), creating a randomization distribution is easy!

```
rand.dist = do(1000)*diffMean(Tap~shuffle(Group))
```

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):

```
obs.stat = diffMean(Tap~Group)
tail.p(rand.dist, obs.stat, tail="upper")
```

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.

We also use the command `randomization.test`, which will do the randomization test for us. `randomization.test` creates the randomization distribution by shuffling the second variable if two variables, or shifting a bootstrap distribution to match the null if one variable. Thus we could conduct the same randomization test as above with

```
randomization.test(Tap, Group, tail="upper")
```

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

   ```
   coin.flips(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

   ```
   rand.dist = do(1000)*coin.flips(25, .5)
   ```

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

   ```
   tail.p(rand.dist,16/25, tail="upper")
   ```

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

   ```
   rand.dist = do(1000)*mean(sample(BodyTemp+0.34, 50,
   replace=TRUE))
   ```

   In this case we have a two-sided H_a:

   ```
   tail.p(rand.dist, 98.26, tail="two")
   ```

Using R in Chapter 5

Normal Distribution:
Find a percentile for N(0,1)
Find the area beyond z on N(0,1)
Find percentiles or area for any normal
	`#tail="lower", tail="upper", tail="two"`
	`percentile("normal", 0.10)`
	`tail.p("normal", z, tail="lower")`
	`#add the optional arguments mean=, sd=`

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

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

```
percentile("normal", 0.05)
percentile("normal", 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:

```
tail.p("normal", z, tail="upper")
```

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

```
tail.p("normal", 60, mean=75, sd=12, tail="lower")
```

Using R in Chapter 6

Normal Distribution: Find a percentile for N(0,1) Find the area beyond z on N(0,1)	`percentile("normal", 0.10)` `tail.p("normal", z, tail="lower")`
t-Distribution: Find a percentile for t Find the area beyond t	`percentile("t", df = 20, 0.10)` `tail.p("t", df = 20, t, tail="lower")`
Inference for Proportions: Single proportion Difference in proportions	`prop.test(count, n, p0) #delete p0 for CI` `prop.test(c(count1, count2), c(n1, n2))`
Inference for Means: Single mean Difference in means	`t.test(y, mu = mu0) #delete mu0 for CI` `t.test(y ~ x)`

Additional arguments
p-values using tail.p
p-values using prop.test or t.test
Intervals using prop.test or t.test
	`#tail="lower", "upper", "two"`
	`#alternative="two.sided", "less", "greater"`
	`#conf.level = 0.95 or 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 `percentile` and `tail.p`
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 stat	`tail.p("chisquare", df = 2, stat, tail="upper")`
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

```
percentile("chisquare", df=2, 3.1, tail="upper")
```

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-statistic	`tail.p("F", df1=3, df2=114, F, tail="upper")`
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 `tail.p`. 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

```
mean(Ants ~ Filling)
boxplot(Ants ~ Filling)
sd(Ants ~ Filling)
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
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)))`

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
Give model output
	
	`lm(y ~ x1 + x2)`
	`summary(model)`

Residuals
Calculate residuals
Residual plot
Histogram of residuals
	
	`model$residuals`
	`plot(predict(model), model$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)))`

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 google doc[4]
Load a dataset from the textbook
Help for textbook datasets
Type in a variable
	
	`google.doc("key")#key: between key= and # in url`
	`data(dataname)`
	`?dataname`
	`variablename = c(3.2, 3.3, 3.1)`

Variables
Extract a variable from a dataset
Attach a dataset
Detach a dataset
	
	`dataname$variablename`
	`attach(dataname)`
	`detach(dataname)`

Subsetting Data Take a subset of a dataset	 `subset(dataname, condition)`

Random Sample
Taking a random sample of size n
n random integers 1 to max
	
	`sample(dataname, n) #use for data or variable`
	`sample(1:max, n)`

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
Proportions by group
Difference in proportions
Segmented bar chart
Side-by-side bar chart
	
	`table(x1, x2)`
	`mean(x1=="A"~x2)`
	`diffProp(x1=="A"~x2)`
	`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)`
	`median(y)`
	`sd(y)`
	`summary(y)`
	`percentile(y, 0.05)`
	`hist(y)`
	`boxplot(y)`

One Quantitative (y) and One Categorical (x)
Means by group
Difference in means
Standard deviation by group
Side-by-side boxplots
	
	`mean(y ~ x)`
	`diffMean(y ~ x)`
	`sd(y ~ x)`
	`boxplot(y ~ x)`

[4] For your own google spreadsheet, within the google spreadsheet you first have to do File -> Publish to Web -> Start Publishing.

Two Quantitative (y1, y2)
Scatterplot
Correlation
	
	`plot(y1, y2)`
	`cor(y1, y2)`

Labels
Add a title
Label an axis
	`#optional arguments for any plot:`
	`main = "title of plot"`
	`xlab = "x-axis label", ylab = "y-axis label"`

CHAPTER 3

Repeat Code 1000 Times	`do(1000)*`
Sampling Distribution for Mean	`do(1000)*mean(sample(y, n))`
Bootstrap Distribution for Mean	`do(1000)*mean(sample(y, n, replace=TRUE))`

Generating a Sampling Distribution for any Statistic	`do(1000)*{`
	`  samp = sample(pop.data, n)`
	`  statistic(samp$var1, samp$var2)`
	`}`

Manually Generating a Bootstrap Distribution for any Statistic	`do(1000)*{`
	`  boot.samp = sample(data, n, replace=TRUE)`
	`  statistic(boot.samp$var1, boot.samp$var2)`
	`}`

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

Generate a Bootstrap CI	`resample(var1, var2) #level = .95`

CHAPTER 4

Randomization Statistic:
Shuffle one variable (x)
Proportion
Mean
	
	`shuffle(x)`
	`coin.flips(n, p)`
	`mean(sample(y + shift, n, replace=TRUE))`

Randomization Distribution	`do(1000)*one randomization statistic`

Finding p-value from a randomization distribution:
Lower-tailed test
Upper-tailed test
Two-tailed test
	`#rand.dist = randomization distribution`
	`#obs.stat = observed sample statistic`
	`tail.p(rand.dist, obs.stat, tail="lower")`
	`tail.p(rand.dist, obs.stat, tail="upper")`
	`tail.p(rand.dist, obs.stat, tail="two")`

Randomization Test via Reallocating	`reallocate(y, x) #tail="lower", "upper", "two"`

CHAPTER 5

Normal Distribution:
Find a percentile for N(0,1)
Find the area beyond z on N(0,1)
Find percentiles or area for any normal
	`#tail="lower", tail="upper", tail="two"`
	`percentile("normal", 0.10)`
	`tail.p("normal", z, tail="lower")`
	`#add the optional arguments mean=, sd=`

CHAPTER 6

Normal Distribution: Find a percentile for N(0,1) Find the area beyond z on N(0,1)	`percentile("normal", 0.10)` `tail.p("normal", z, tail="lower")`
t-Distribution: Find a percentile for t Find the area beyond t	`percentile("t", df = 20, 0.10)` `tail.p("t", df = 20, t, tail="lower")`
Inference for Proportions: Single proportion Difference in proportions	`prop.test(count, n, p0) #delete p0 for CI` `prop.test(c(count1, count2), c(n1, n2))`
Inference for Means: Single mean Difference in means	`t.test(y, mu = mu0) #delete mu0 for CI` `t.test(y ~ x)`

Additional arguments
p-values using tail.p
p-values using prop.test or t.test
Intervals using prop.test or t.test
	`#tail="lower", "upper", "two"`
	`#alternative="two.sided", "less", "greater"`
	`#conf.level = 0.95 or confidence level`

CHAPTER 7

Chi-Square Distribution Find the area above χ^2 stat	`tail.p("chisquare", df = 2, stat, tail="upper")`
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-statistic	`tail.p("F", df1=3, df2=114, F, tail="upper")`
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	`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)))`

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)))`