DR. NATHANIEL D. PHILLIPS

YARRR! A PIRATE'S GUIDE TO R

PUBLISHED BY

HTTP://WWW.NATHANIELDPHILLIPS.COM

First printing, April 2015

Contents

This book is dedicated to my former statistics instructors Dr. Thomas Moore and Dr. Wei Lin who taught me everything I know about statistics, and my PhD colleagues Dr. Dirk Wulff and Dr. Stefan Herzog who taught me everything I know about R.

Introduction

Who am I?

TEST Testtest

I am a pirate on the Bodensee in Konstanz Germany. When I started pirate training, I discovered R and have been hooked ever since. I'm now on a mission to convince everyone I can to make the switch from SPSS (or Excel, Matlab, JMP...) to R.

This book is in progress..

If you haven't figured it out already, this book is very much a work in progress. I'm constantly experimenting with the material and the layout. If you have any recommendations for changes or spot any errors, please write me at YaRrr.Book@gmail.com or tweet me @YaRrrBook

Email me with comments, recommendations or typos at: YaRrr.Book@gmail.com or tweet me at @YaRrrBook

Who is this book for?

Anyone who wants to learn R can benefit from this book. I will assume that you have taken an introductory course in statistics, but have no substantial programming experience. While the techniques in this book apply to most data analysis problems, because my background is in experimental psychology I will cater the course to solving analysis problems commonly faced in psychological research.

Why is R so great?

1. R is 100% free and as a result, has a huge support community. Unlike SPSS, Matlab, Excel and JMP, R is, and always will be completely free. This doesn't just help your wallet - it means that a huge community of R programmers will constantly develop an distribute new R functionality and packages at a speed that leaves all those other packages in the dust! Unlike Fight Club, the first rule of R is "Do talk about R!" The size of the R programming community is staggering. If you ever have a question about how to implement something in R, a quick Poogle[1] search will lead you to

[1] I am in the process of creating Poogle - Google for Pirates. Kickstarter page coming soon...

your answer virtually every single time.

2. R is incredibly versatile. You can use R to do everything from calculating simple summary statistics, to performing complex simulations to creating gorgeous plots like the chord diagram in Figure 1. If you can imagine an analytical task, you can almost certainly implement it in R.

3. Using RStudio, You can easily and seamlessly combine R code, analyses, plots, and written text into elegant documents all in one place using Sweave (R and Latex) or RMarkdown. Instead of trying to manage two or three programs, say Excel, Word and (sigh) SPSS, where you find yourself spending half your time copying, pasting and formatting data, images and test, R combined with RStudio allows you to do everything in one place so nothing gets misread, mistyped, or forgotten.

4. Analyses conducted in R are transparent, easily shareable, and reproducible. If you ask an SPSS user how they conducted a specific analyses, they will either A) Not remember, B) Try (nervously) to construct an analysis procedure on the spot that makes sense - which may or may not correspond to what they actually did months or years ago, or C) Ask you what you are doing in their kitchen[2]. I used to primarily use SPSS, so I speak from experience on this. If you ask an R user (who uses good programming techniques!) how they conducted an analysis, they should always be able to show you the exact code they used. Of course, this doesn't mean that they used the appropriate analysis or interpreted it correctly, but with all the original code, any problems should be completely transparent!

5. And most importantly of all, R is the programming language of choice for pirates, (who prefer the "YaRrr!" pronunciation)

```
require("circlize")


## Loading required package:  circlize


mat = matrix(sample(1:100, 18, replace = TRUE), 3, 6)
rownames(mat) = letters[1:3]
colnames(mat) = LETTERS[1:6]
chordDiagram(mat)
```

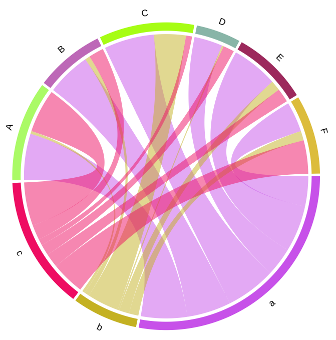


Figure 1: This is a **chordDiagram** plot that comes with the R package **circlize**.

[2] Get used to the bad jokes people. Lots more where that came from.

How this book is formatted

In this book, R code is (almost) always presented in a separate gray box like this one:

```
a <- 1 + 2 + 3 + 4 + 5
a

## [1] 15
```

This is called a *code chunk*. You should always be able to directly copy and paste code chunks directly into R. If you copy a chunk and it does not work for you, it is most likely because the code refers to a package, function, or object that I defined in a previous chunk. If so, read back and look for a previous chunk that contains the missing definition. As you'll soon learn, lines that begin with are either comments or output from prior code so R will ignore these lines if you paste them into R.

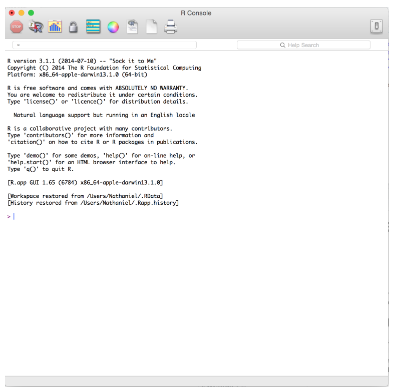
As you'll notice, I'll include code chunks before all plots in the book. In early chapters, the code might not make sense just yet. However, I elected to always include plotting code so you have the option of re-creating (and tweaking) any plot in the book.

1: Installing R and RStudio

Now that I've convinced you to use R, let's get started! First, you'll need to install the base R software.

1. Download and install the base R software (around 50mb) + Windows <http://cran.r-project.org/bin/windows/base/> + Mac <http://cran.r-project.org/bin/macosx/>

See Figure 2 Here's how the base R software looks (on Mac). As you can see, it's very much a bare-bones software - just how we want it! No extra gimmicks or flashy bloatware needed!



```
R version 3.1.1 (2014-07-10) -- "Sock it to Me"
Copyright (C) 2014 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin13.1.0 (64-bit)

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

  Natural language support but running in an English locale

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

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

[R.app GUI 1.65 (6784) x86_64-apple-darwin13.1.0]

[Workspace restored from /Users/Nathaniel/.RData]
[History restored from /Users/Nathaniel/.Rapp.history]

> |
```

Figure 2: Here is how the standard R application looks. Not too exciting - just how we like it!

While you can do pretty much everything you want within base R, you'll find that most people these days do their R programming in an application called RStudio. RStudio is a graphical user interface (GUI)-like interface for R that makes programming in R a bit easier. To download and install RStudio (around 40mb), go to <http://www.rstudio.com/products/rstudio/download/>

Once you've installed RStudio, you'll never need to open the base R application. Let's go ahead and boot up RStudio and see how she looks!

The four RStudio windows

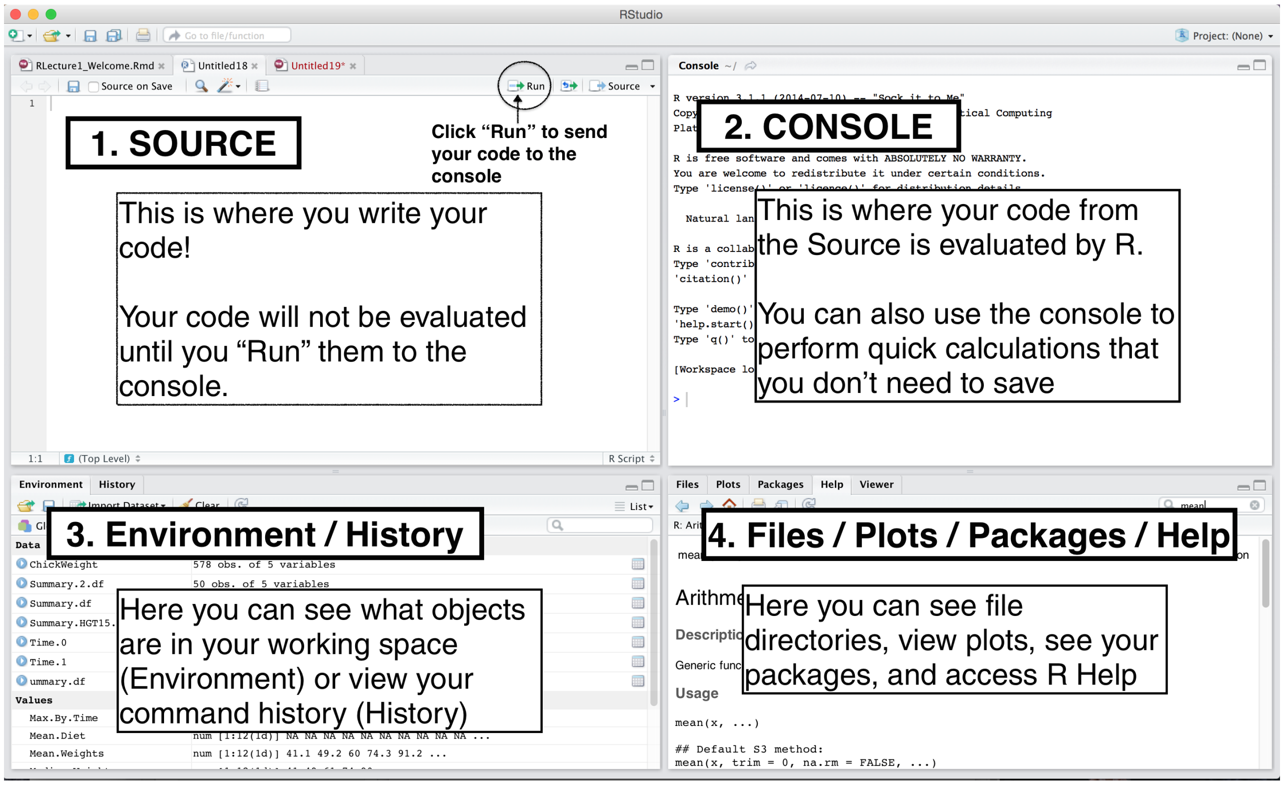
When you open RStudio, you'll see the following four windows (also called panes):

Note your windows might be in a different order. You can change the order of the windows under RStudio preferences.

Source - Your notepad for code

The source pane is where you create and edit R Scripts - which are just text files with the ".R" extension. When you open RStudio, it will automatically start a new Untitled script. You will write 99% of your R code in a script in the source panel. However, your R code will not be evaluated until you 'send' the code to the Console.

You can send your code from the source to the Console by highlighting the code you wish to evaluate and clicking on the "Run" button on the top right of the Source. Alternatively, you can use the hot-key



Click "Run" to send your code to the console

This is where you write your code!

Your code will not be evaluated until you "Run" them to the console.

This is where your code from the Source is evaluated by R.

You can also use the console to perform quick calculations that you don't need to save

Here you can see what objects are in your working space (Environment) or view your command history (History)

Here you can see file directories, view plots, see your packages, and access R Help

Figure 3: The four panes of RStudio.

"Command + Return" on Mac, or "Control + Enter" on PC to send code to the console.

Console: The calculator

The console is where R actually executes (calculates) code. You can type code directly into the console and get an immediate response. For example, if you type 1+1 into the console, you'll see that R immediately gives an output of 2

```
1+1

## [1] 2
```

Tip: Try to write most of your code in a document in the Source. Only type directly into the Console to de-bug or do quick analyses.

However, most of the time, you won't be typing directly into the console. Instead, you'll be writing code in the source and then "Running" it to the console. The reason for this is straightforward: If you type code into the console, it won't be saved (though you can look back on your command History). And if you make a mistake in typing code into the console, you'd have to re-type everything all over again. Instead, it's better to write all your code in the Source. When you are ready to execute some code, you can then send "Run" it to the console.

Environment / History

The Environment tab of this panel shows you the names of all the data objects (like vectors, matrices, and dataframes) that you've defined in your current R session. The tab also has a few clickable actions like importing a new dataset. However, I almost never look at this menu.

The History tab of this panel simply shows you a history of your R commands. I never look at this. In fact, I didn't realize it was even there until I started writing this tutorial.

Files / Plots / Packages / Help

This panel shows you file directories, plots, your current packages, and help menus.

1. Files - Gives you access to the file directory on your harddrive. One nice feature of the "Files" panel is that you can use it to set your working directory - once you navigate to a folder you want to read and save files to, click "More" and then "Set As Working Directory."

2. Plots - Shows your plots. There are buttons for opening the plot in a separate window and exporting the plot as a pdf or jpeg (though you can also do this with code using the `pdf()` or `jpeg()` functions.)

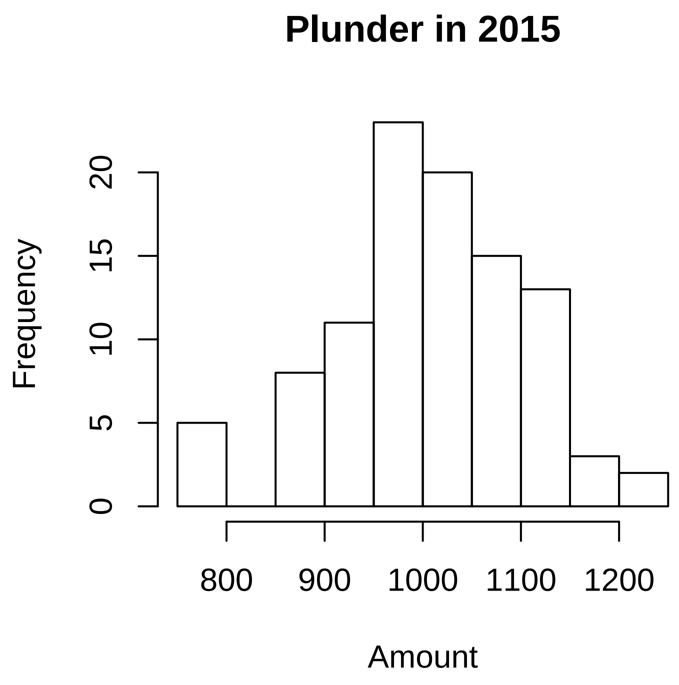
3. Packages - Shows a list of all the R packages installed on your harddrive and indicates whether or not they are currently loaded. Packages that are loaded in the current session are checked while those that are installed but not yet loaded are unchecked.

4. Help - Help menu for R functions. You can either type the name of a function in the search window, or use the code `?function.name` to search for a function with the name `function.name`

Most - if not all - of the time when you perform actions using your mouse by pointing and clicking in RStudio, RStudio will perform the function by sending the appropriate R Code to the console. You can then copy and paste this code into your documents to automate the process later.

To see how plots are displayed try the following command which should display a histogram of 100 values randomly drawn from a standard normal distribution.

```r
hist(rnorm(n = 100,
           mean = 1000,
           sd = 100),
     main = "Plunder in 2015",
     xlab = "Amount"
     )
```



Getting help

To get help and see documentation for a function, type `?fun`, where `fun` is the name of the function. For example, to get additional information on the histogram function, run the following code:

```
?hist
```

Tip: If you ever need to learn more about an R function: type `?functionname`, where `functionname` is the name of the function.

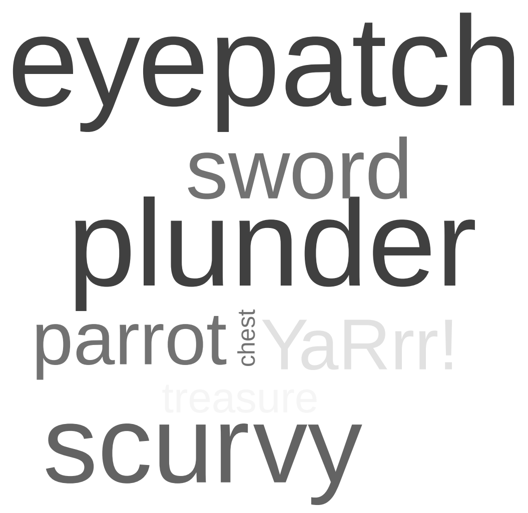
Installing and loading packages

When you download and install R for the first time, you are installing the Base R software. Base R will will contain most if not all the functions you need. However, one of the great things about R is that people are constantly writing and sharing new functions that you can use. When people share a new function, they usually do so in the form of an *R package* which contains anything from functions, to help menus, to vignettes (examples), to data. To install a new R package, you need to run the code `install.packages("package")`, where "package" is the name of the package. After you've installed the package, you need to *load* it into R by running the code `load("package")`. This will load the package into your current R session and allow you to use its contents.

Once you've installed a package on your computer, you never need to install it again. However, you do need to load the package every time you start a new R session.

For example, let's say you want to create a wordcloud - a graph that plots text in different sizes. You can certainly program this yourself in R, but thankfully someone has created a package called `wordcloud` with a function that will do this for you. Let's install the package, load it, and then use the `wordcloud` function:

```
install.packages("wordcloud") # Install the package

##
## The downloaded binary packages are in
##  /var/folders/yh/4fyk9h754h7fnmpgyc6p5whw0000gn/T//RtmpQiEAo8/downloaded_packages

library("wordcloud") # Load the package

## Loading required package:  RColorBrewer

par(mar = rep(0, 4))
wordcloud(words = c("sword", "YaRrr!", "eyepatch",
                    "parrot", "plunder", "treasure",
                    "chest", "scurvy"),
          freq = sample(50:1000, 8),
          colors = gray(runif(8, 0, 1)))
```



Finished!

That's it for this lecture! All you did was install the most powerful statistical package on the planet used by top universities and companies like Google. No big deal.

2: Coding Basics

Chapter Goals

1. Accept that learning R will take time (and promise you'll never go back to SPSS!)

2. Know how to use comments and spaces in R code.

3. Be able to define and manipulate scalers and vectors

4. Generate vectors using c(), :, rep(), and seq()

Before we get started, a word of warning...

So by now you've installed R and you're ready to get started. But first, let me give you a brief word of warning: Especially if this is your first experience programming, you are going to experience a *lot* of headaches when you get started. You will run into error after error and pound your fists against the table screaming: "WHY ISN'T MY CODE WORKING?!?!? There must be something wrong with this stupid software!!!" You will spend hours trying to find a bug in your code, only to find that - frustratingly enough, you had had an extra space or missed a comma somewhere. You'll then wonder why you ever decided to learn R when (::sigh::) SPSS was so "nice and easy."

Fun fact: SPSS stands for "Shitty Piece of Shitty Shit". True story.

This is perfectly normal! Don't get discouraged and DON'T GO BACK TO SPSS! Trust me, as you gain more programming experience, you'll experience fewer and fewer bugs (though they'll never go away completely). Once you get over the initial barriers, you'll find yourself conducting analyses much, much faster than you ever did before.

The basics of R programming

Ok, let's write some code! Again, we will write all our code in a script file in the Source pane of RStudio. When we want to execute it, we'll send it to the Console.

R as a calculator

At its heart, R is just a fancy calculator. Let's do some basic algebra, type the following command into the source, then highlight the text and click "Run" to execute it in the console:

Tip: To execute code from the source to the console, highlight it and use the hot-keys "Command-Return" on Mac or "Control-Enter" on PC.

```
1+1 # The result should be 2

## [1] 2
```

As you can see, R returns the (thankfully correct) value of 2. You'll notice that the console also returns the text [1]. This is just a 'prompt' that tells you the index of the value next to it. Don't worry about this for now, it will make more sense later.

Additionally, you'll notice that I included a comment in the code using the # sign. R will ignore everything on a line after the # sign. So why do we use comments? Mainly to explain to others, including your future self, what you are trying to do with your code.

Do your future self a favor and use comments to explain what you're doing with your code. Also, maybe go for a run once in a while.

Let's try some more:

```
2 * 3 - 1 # R ignores spaces

## [1] 5

2 * (3 - 1) # R observes order of operations

## [1] 4
```

As you can see, R ignores spaces in between arguments in code. I recommend using spaces to make your code easier to look at. Personally, I include spaces between arithmetic operators (like + and -) and after commas (which we'll get to later).

Defining objects with the "<-" assignment

So far so good, you can use R as a simple calculator. Now, let's do our first *object assignment*. Object assignment is our way of storing information, such as a number or a statistical test, into something we can easily refer to later. Let's start by creating the object "my.object" and assigning the outcome of 2^{10} to it:

Good object names strike a balance between being easy to type (i.e.; short names) and interpret. If you have several datasets, it's probably not a good idea to name them `a`, `b`, `c` because you'll forget which is which. However, using long names like `March2015Group1OnlyFemales` and `March2015Group1OnlyMales` will give you carpel tunnel syndrome.

```
# The symbol(s) "<-" mean "assign"
mateys <- 2 ^ 10 # Assign the value of 2^10 to mateys
```

As you can see in the example, we used the "<-" command to assign the value of 1 to my.object. When you assign a value to an object, R won't automatically print it. If you want to see the value, you need to call the object by just executing its name:

```
mateys # What is the value of mateys?

## [1] 1024
```

A few notes about defining objects: you can't start the name of an object with a number, and you can't have spaces or other 'weird' characters in the name. Here are some examples of *invalid* object names:

```
me mateys # Can't have spaces
5.mateys # Can't start a name with a number
YaRrr! # Can't have an "!" in the object name
```

R is case-sensitive. If you define an object with uppercase letters, you must keep referring to it with uppercase letters!

Avoid using too many capital letters in object names because they require you to hold the shift key. This may sound silly, but you'd be surprised how much easier it is to type `mydata` than `MyData` 100 times.

```
Plunder <- 1
plunder <- 100
Plunder

## [1] 1

plunder

## [1] 100
```

Once you've defined an object, you can use it in other commands:

```
prior.plunder <- 100
new.plunder <- 2
total.plunder <- prior.plunder + new.plunder
total.plunder

## [1] 102
```

If you want to change an object, you can just reassign it. You can even refer to the same object when reassigning it:

```
a <- 2
a <- a + a
a

## [1] 4
```

You can use `=` instead of `<-` for object assignment but I recommend you stick with <- because the direction of the assignment is clear.

Data object types in R

R stores everything as an object, and there are different types of objects. The first two objects we'll learn about are scalers and vectors. Later on, we'll talk about more complicated objects like matrices, dataframes, hypothesis tests, etc.

Two simple data objects: Scalers and Vectors

Two of the most common data objects in R are **scalers** and **vectors**. Let's discuss each in turn,

Scalers

A **scaler** is just a single value. A scaler can either be *numeric* or *character*. A numeric scaler is a number, while a character scaler is a letter. We denote character scalers by using quotation marks. Here are some examples:

```
a <- 1
b <- 3 * 40
ship <- "Black Pearl"
```

It is important to note that once you've defined an object, you refer to it without quotation marks, even if the object is a character. For example, to refer to the object `ship` that I defined above, you need to write `ship` without quotations marks

```
ship # Print the value of the object ship

## [1] "Black Pearl"

"ship" # R thinks this is a new string called "ship", not the object called ship

## [1] "ship"
```

```
# scaler v vector v matrix

par(mar = rep(1, 4))
plot(1, xlim = c(0, 4), ylim = c(-.5, 5),
     xlab = "", ylab = "",
     xaxt = "n", yaxt = "n",
     bty = "n", type = "n")

# Scaler
rect(rep(0, 1), rep(0, 1), rep(1, 1), rep(1, 1))
text(.5, -.5, "Scaler")

# Vector
rect(rep(2, 5), 0:4, rep(3, 5), 1:5)
text(2.5, -.5, "Vector")
```

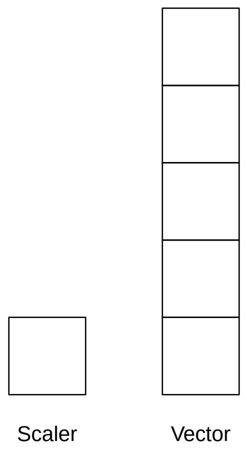


Figure 4: Visual depiction of a scaler and vector. Deep shit. Wait until we get to matrices - you're going to lose it.

Vectors

A **vector** is a combination of several scalers. For example, the numbers from one to ten could be a vector of length 10. Like scalers, you can also create character vectors that contain character scalers.

There are many ways to create vectors in R, here are the most common:

`c()`

The simplest way to create a vector is with the `c()` function. The c here stands for concatenate, which means "bring them together". When using c(), place a comma in between the scalers you want to combine:

`c(x, y, z)`: Create a vector with the c() command by separating elements with commas

```
c(1, 2, 3, 4, 5) # A vector of the integers 1 through 5

## [1] 1 2 3 4 5

a <- 9
b <- 10
c(a, b) # Combine the scalers a and b

## [1]  9 10

ship <- "Black Pearl"
captain <- "Jack Sparrow"

c(ship, captain)

## [1] "Black Pearl"  "Jack Sparrow"
```

A vector can only contain one type of scaler: either numeric or character. If you try to create a vector with numeric and character scalers, then R will convert all of the numeric scalers to characters:

```
movie <- "Pirates of the Carribean"
revenue <- 634954111
c(movie, revenue) # Result is just character scalers

## [1] "Pirates of the Carribean" "634954111"
```

Once you've created a vector, you can easily determine its length by using the `length()` function:

`length()`

```
length(c(1, 2, 3))

## [1] 3
```

Generating numeric vectors

While the c() operator is the most straightforward way to create a vector, it's also one of the most tedious. Let's say you wanted to create a vector of all integers from 1 to 100. You definitely don't want to have to type all the numbers into a c() operator. Instead, R has many simple built-in functions for generating numeric vectors. Let's start with three of them:

`a:b`

a:b - Creates a sequence from a to b in steps of 1.

The `a:b` function creates a vector of numbers from the starting point a to the ending point b in steps of 1:

```
1 : 10 # Integers from 1 to 10

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

10 : 1 # Integers from 10 to 1

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

20.1:30.1 # From 20.1 to 30.1

##  [1] 20.1 21.1 22.1 23.1 24.1 25.1 26.1 27.1 28.1 29.1 30.1
```

`seq(from, to, by)`

seq(from, to, by) - Creates a sequence between two numbers in steps that you specify.
`from`: The starting value
`to`: The ending value
`by`: The step size between `begin` and `end`

The `seq()` function allows you to create a sequence from a starting number to an ending number, in steps you specify. The function has three **arguments**, which are inputs to the function which changes how it works. There are three arguments to the seq() function:

- from: The starting number
- to: The ending number
- by: The steps between numbers

```
seq(from = 1, to = 10, by = 1)

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

seq(from = 0, to = 100, by = 10)

##  [1]   0  10  20  30  40  50  60  70  80  90 100
```

`rep(x, times, each)`

rep(x, times, each) - Repeats the numbers in x in a manner you specify
`times`: The number of times the vector should be repeated
`each`: The number of times you want to repeat each element in the vector.

The `rep()` function rep allows you to repeat a number (or vector) a specified number of times. There are three arguments to the rep function:

- x: The numeric object (scaler or vector) you want to repeat
- times: The number of times you want to repeat the entire object
- each: The number of times you want to repeat each element within a vector

```
rep(1:5, times = 2) # Repeat integers 1 to 5 two times

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

rep(1:5, each = 2) # Repeat each integer from 1 to 5 two times

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

rep(1:5, each = 2, times = 2) # Do both!

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

Arithmetic operations on scalers and vectors

You can do basic arithmetic operations like +, -, * and / on scalers and vectors. If you do an operation on a vector with a scaler, R will apply the scaler to each element in the vector:

```
a <- 1:5
a * 10

## [1] 10 20 30 40 50

a - 1

## [1] 0 1 2 3 4

a ^ 2

## [1]  1  4  9 16 25
```

If you do an operation on two vectors, R will try to apply the operation between the vectors by each item:

```
1:10 + 21:30

##  [1] 22 24 26 28 30 32 34 36 38 40

(1:5) * (1:5)

## [1]  1  4  9 16 25

seq(10, 100, 10)

##  [1]  10  20  30  40  50  60  70  80  90 100

seq(10, 100, 10) ^ 2

##  [1]   100   400   900  1600  2500  3600  4900  6400  8100 10000
```

Additional Tips

1. If you need to enter a lot of numeric data into R by hand you might want to use the `scan()` function. This function allows you to easily enter data using 10-key typing on a number pad. To do this, run the code `scan()` and then enter the data number by number. When you are finished, R will then print the appropriate code to store the data into a vector.

2. You can run several lines of code in one line by separating the code with the ; key. For example, the following two chunks of code are the same:

   ```
   a <- 1
   b <- 14
   c <- 67
   ```

   ```
   a <- 1 ; b <- 14 ; c <- 67
   ```

   However, I recommend you use the ; key sparingly. If you get in the habit of trying to cram several lines of code in one line, your code will get cluttered and difficult to understand.

3: Sampling data and Descriptive Statistics

Chapter Goals

1. Download the priceless R reference card
2. Learn functions for generating data from probability distributions: rnorm(), runif()
3. Learn functions for basic descriptive statistics: mean(), median(), sd(), var(), min(), max()

The R Reference Card

Over the next few lessons, you will be learning *lots* of new functions. Wouldn't it be nice if someone created a Cheatsheet / Notecard of many common R functions? Yes it would, and thankfully Tom Short has done this in his creation of the R Reference Card. You can download a copy at <https://dl.dropboxusercontent.com/u/7618380/RReferenceCard.pdf>. I highly encourage you to print this out and start highlighting functions as you learn them!

Sampling data from distributions

By now you know how to generate sequences of numbers with the functions :, `seq()`, and `rep()`. However, these functions don't generate very interesting data. Instead, we can use R to generate randomly sampled data from specified probability distributions. Because the data are drawn randomly, you'll get different results every time you run the functions.

Let's start with the Normal and Uniform distributions.

```
# normal distribution
curve(dnorm,
      from = -3,
      to = 3,
      xlab = "x",
      lwd = 2,
      main = "Normal\nmean = 0, sd = 1")
```

The Normal (Gaussian) distribution: rnorm(n, mean, sd)

Let's start with the most famous distribution in statistics: the Normal (or if you want to sound pretentious, the Gaussian) distribution. From our intro stats class, we know that the Normal distribution is bell-shaped, and has two parameters: a mean and a standard deviation.

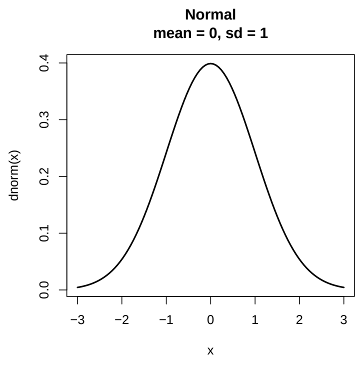


Figure 5: The standard normal

`rnorm(n, mean, sd)`

`rnorm(n, mean, sd)`: Generate n samples from a normal distribution with specified mean and standard deviation

To generate samples from a normal distribution, we use the function `rnorm()` this function has three arguments:

- n: The number of observations
- mean: The mean of the distribution
- sd: the standard deviation of the distribution

```
rnorm(10, mean = 0, sd = 1) # 10 samples from standard normal

##  [1]  0.62659985 -0.02861356 -2.15681939  0.40465106 -0.58383008
##  [6] -0.85491237 -0.31662150  0.55560691  1.25051393 -0.50696434

rnorm(10, mean = 100, sd = 10) # 10 samples with mean = 100 and sd = 10

##  [1]  80.70109 121.46515  89.57401 102.53068  98.95072 108.40335  96.74799
##  [8]  97.48537 115.12498  95.78920
```

Again, because the samples are drawn randomly, you'll get different values from the ones I got above.

Next, let's move on to the *uniform* distribution. The uniform distribution gives equal probability to all values between the minimum and maximum values.

```
# uniform distribution
curve(dunif,
      from = 0, to = 1,
      xlim = c(-.5, 1.5),
      xlab = "x",
      lwd = 2,
      main = "Uniform\nmin = 0, max = 1")
```

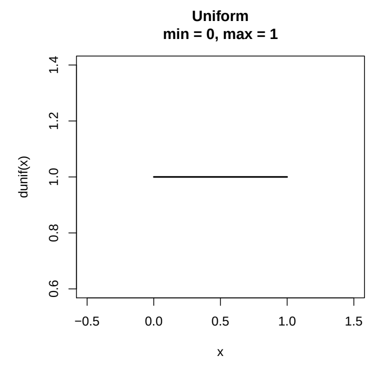


Figure 6: The Uniform distribution - known colloquially as the Anthony Davis distribution.

`runif(n, min, max)`

To generate samples from a uniform distribution, we use the function `runif()`, the function has 3 arguments:

- n: The number of observations
- min: The lower bound of the distribution
- max: The upper bound of the distribution

```
runif(10, min = 0, max = 1) # 10 samples from U[0, 1]

##  [1] 0.0643151312 0.4323487021 0.1792980442 0.7628376074 0.8990009876
##  [6] 0.5213774694 0.4634869648 0.0002581549 0.9550466693 0.5461913459

runif(10, min = -100, max = 100) # 10 samples from U[-100, 100]

##  [1]   3.770824  42.204283  92.723592  69.429819  97.776923  -7.240125
##  [7] -29.533086  89.086561  64.930196  51.851874
```

Sampling from a set of values: sample()

The next function we'll use is **sample()**. Sample allows you to draw values from a larger set with some probability.

```
sample(x, size, replace, prob)
```

`sample(x, size, replace, prob)`: Draw a sample of outcomes from a vector, each with a specified probability.

- x: A vector of outcomes you want to sample from
- size: The number of samples you want to draw
- replace: Should sampling be done with replacement? If T, then each individual sample will be replaced in the data vector. If F, then the same outcome will never be drawn more than once.
- prob: A vector of probabilities of the same length as `x` indicating how likely each outcome in "x" is. The first value corresponds to the first value of x and the second corresponds to the second value (etc.). The vector of probabilities you give as an argument should add up to one. However, if they don't, R will just rescale them so that they will sum to 1.

Replacement: Think about replacement like drawing different balls from a bag. Sampling with replacement (`replace = T`) means that each time you draw a ball, you return the ball back into the bag before drawing another ball. Sampling without replacement (`replace = F`) means that after you draw a ball, you remove that ball from the bag before drawing again.

Simulating Tinder Outcomes

Let's simulate some Tinder outcomes. For those who don't know, Tinder is (or "was" if it's not still around when you're reading this) an app that allows you to view profiles of potential dates. For each potential date, you can see their picture and either "like" them by swiping right, or "dislike" them by swiping left. If a person that you "liked" also "likes" you, then you've had a successful match and will be able to start chatting. let's say you "swipe right" on 20 Tinder profiles and the probability you get a match is 20%. We can simulate this using the sample function

```
sample(x = c(":)", "-"),
       size = 20,
       replace = T, # Replace each sample back to the set
       prob = c(.2, .8)  # Probability of Match! is .2, and No Match :( is .8)
       )

##  [1] ":)" "-"  "-"  "-"  "-"  "-"  "-"  ":)" "-"  "-"  "-"  "-"  "-"  "-"
## [15] "-"  "-"  "-"  ":)" "-"  "-"
```

Figure 7: Pinder. Tinder for Pirates.

In the example above I set `replace = T`. If I didn't do this, then on the third sample R would run out of objects to draw from. However, there are cases where we would not want sampling with replacement.

Where should I go clubbing in Berlin...?

Let's say you planning a weekend of clubbing in Berlin and need to decide on just 2 out of 5 clubs to visit. Since we don't want to visit the same club twice, we'll set `replace = F`

```
clubs <- sample(x = c("Berghain", "Club der Visionare",
             "Goldengate", "Watergate", "Keller"),
       size = 2,
       replace = F,
       prob = rep(1/5, 5) # The probability of each is 1/5
       )
clubs

## [1] "Watergate" "Keller"
```

Looks like R has selected Watergate and Keller. Let's see if we even make it to Keller...

Descriptive statistics

Ok, now that we can generate some data, let's learn the basic descriptive statistics functions. We'll focus on the most common ones for numerical analyses.

Common descriptive statistics:
`mean(x)`: Mean
`median(x)`: Median
`sd(x)`: Standard Deviation
`var(x)`: Variance
`min(x)`: Minimum
`max(x)`: Maximum

- mean(x): The arithmetic mean of the vector x
- median(x): The median of the vector x
- sd(x): Standard deviation
- var(x): Variance
- min(x), max(x): Minimum and maximum
- quantile(x, probs): Sample quantiles of a vector x corresponding to certain probabilities.

Each of these functions takes a vector as an argument, and returns a scaler as a result. Let's start by creating a vector of 50 samples from a normal distribution with mean 10 and standard deviation 10. We'll then calculate some descriptive statistics from that vector:

```
a <- rnorm(50, mean = 10, sd = 10) # 100 samples from standard normal
mean(a)

## [1] 8.743332

median(a)

## [1] 8.513639

sd(a)
```

```
## [1] 9.854695

var(a)

## [1] 97.11501

min(a)

## [1] -24.37957

max(a)

## [1] 30.14919

quantile(a, probs = c(.25, .75))

##       25%       75%
##  2.007304 14.747340
```

Let's say you want to get many summary statistics from a vector, you can do this by using the **summary()** function which gives you several key statistics:

`summary(x)`: Gives you some distributional information about a vector

```
summary(a)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -24.380   2.007   8.514   8.743  14.750  30.150
```

A quick test of the law of large numbers

According to the law of large numbers, the larger our sample size, the closer our sample mean should be to the population mean. Let's test this by drawing either a small (N = 5) or a large (N = 1,000,000) number of observations from a Normal distribution with mean = 100 and sd = 20:

Tip: You can easily write large powers of 10 by using the notation `1eN`, where N is the power of 10. For example: `1e6` is the same as 1,000,000

```
small <- rnorm(10, mean = 100, sd = 20) # 10 observations
large <- rnorm(1e6, mean = 100, sd = 20) # One million observations

mean(small) # What is the mean of the small sample?

## [1] 85.77618

mean(large) # What is the mean of the large sample?

## [1] 100.0036

mean(small) - 100 # How far is the mean of Small from 100?

## [1] -14.22382

mean(large) - 100 # How far is the mean of Large from 100?

## [1] 0.003564461
```

If our test worked, then the difference for the small sample should be larger than the large sample - which it is!

Additional Tips

1. Many functions for descriptive statistics, like `mean()` and `median()` have a logical argument `na.rm` which tells the function whether or not to remove `NA` values. If you do not specify `na.rm = T`, then the functions will return `NA` if there are any missing values in the vector.

```
mean(c(1, 2, 3, NA))

## [1] NA

mean(c(1, 2, 3, NA), na.rm = T)

## [1] 2
```

4: Indexing and comparing vectors

Chapter Goals:

1. Use brackets [] and logical vectors to index vectors
2. Combine indexing with descriptive statistics
3. Learn indexing functions which(), sort()
4. Vector discrete summary functions table() and unique()
5. Set functions: intersect(), union(), setdiff(),

Indexing vectors with brackets

Indexing with brackets [ ] To get the ith value of a vector called `vec`, use the bracket notation vec[i]

When we have a vector, we will frequently want to access specific values of a vector. These might be values in a specific location in the vector (i.e.; the fifth element) or based on some criteria (i.e.; all values greater than 0). We can accomplish this using indexing.

`[]`

Let's start with some data. Currently, I live in an apartment with large windows that open to a popular walking path. Imagine that 10 times over the course of an evening I pointed at a random person and asked/yelled "How much do you think I would pay for a date with Jennifer Lawrence?" I then recorded their answers in the vector `guesses`

```
guesses <- c(100, 20, 500, 50, 10000, 1, 0, 500, 100, 100)
```

Now, I can access specific values in the data vector by inserting a vector (or scaler) of index values between brackets:

```
guesses[1] # First value of guesses

## [1] 100

guesses[1:5] # Values 1 to 5 of guesses

## [1]   100    20   500    50 10000
```

```
guesses[c(1, 4, 10)] # 1st, 4th, and 10th value

## [1] 100  50 100

guesses[seq(2, 10, 2)] # Every 2nd value

## [1]  20  50   1 500 100
```

Indexing with logical vectors

Another way to index data vectors is with **logical vectors**. A logical vector is a vector that only contains TRUE and FALSE values. You can create a logical vector by using comparison operations such as ==, <, >, <=, >=

```
a <- 1:10
a > 5 # Are the values greater than 5?

##  [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE

a == 3 # Are the values equal to 3?

##  [1] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

a <= 9 # Are the values less than or equal to 9?

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

Once we have a logical vector, we can use that vector as an indexing vector. For example, let's say that I want to access values of the guess vector that are greater than 10

```
log.vec <- guesses > 100 # Create logical vector
log.vec # show me values of log.vec

##  [1] FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE

guesses[log.vec] # Index guesses by log.vec

## [1]   500 10000   500
```

In the examples above, we calculated a logical vector by comparing a data vector to a scaler. This told us, for example "Which values are greater than 0?" However, we can also do comparisons between two vectors.

Let's take the example of comparing some data on the changing quality of my jokes over time. In each of two semesters - winter and summer - I told 10 jokes. After telling each joke, I secretly recorded the number of laughs I got for that joke. I can represent the joke response data for each semester as a vector:

```
winter <- c(0, 0, 1, 2, 0, 1, 2, 10, 2, 1)
summer <- c(0, 1, 3, 0, 0, 5, 0, 1, 4, 3)
```

Ok, so let's see if my jokes improved from the winter to summer semester. I'll start by seeing which jokes had an increase in laughs between semesters:

```
improve.log <- summer > winter # create logical vector
improve.log # print values

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

which()

If I want to know which jokes improved, I can use the **which()** function. The which function will tell me the index of each TRUE value in a logical vector:

```
which(improve.log)

## [1]  2  3  6  9 10
```

Now I know that I did better with jokes 2, 3, 6, 7, 9 and 10. Not bad! Ok let's see which jokes actually resulted in fewer laughs

```
decrease.log <- summer < winter
decrease.log

##  [1] FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE  TRUE FALSE FALSE

which(decrease.log)

## [1] 4 7 8
```

Ok looks like I might need to work on my delivery of jokes 4, 7, and 8...

which(logical.vector): Tells you the index value of all **TRUE** values in the logical vector.
For example, the command **which(c(T, T, F))** will return the vector **[1, 2]**, telling you that the first and second values are true.

Combining indexing with descriptive statistics

Many (if not all) R functions that take numeric data as inputs will interpret TRUE values as 1 and FALSE values as 0. This allows us to easily answer questions like "How many values in a data vector are greater than 0?" or "What percentage of values are equal to 5?"

In the next example, I'll generate 10,000 values from a standard normal distribution and use logical indexing to figure out what percentage of values are positive in a vector x:

To see what percentage of values are TRUE in a logical vector, just take the mean of the vector. For example, the command **mean(c(-1, -2, 1, 1) > 0)** will return **0.50**, telling you that half of the values are positive.

```
values <- rnorm(n = 10000, mean = 0, sd = 1)
gt.0.log <- values > 0 # Logical vector indicating positive values
sum(gt.0.log) # How many values are positive?

## [1] 5026

mean(gt.0.log) # What percentage of values are positive?

## [1] 0.5026
```

The percentage is pretty close to 50%, which is exactly what we would expect given that the standard normal distribution is symmetric around 0.

Next, let's ask a slightly different question: What is the mean value for those values that are positive? In other words, given that a value is positive, how big do we expect it to be? We can answer this in 4 steps:

1. Create a vector called **values** generated from a standard normal distribution
2. Create a logical vector **log.vec** that indicates which values are positive.
3. Create a new vector called **pos.values** that only contains positive values from the vector **values**
4. Calculate the mean of the vector **pos.values**

```
values <- rnorm(n = 10000, mean = 0, sd = 1)  # Step 1
log.vec <- values > 0 # Step 2
pos.values <- values[log.vec] # Step 3
mean(pos.values) # Step 4

## [1] 0.7779074
```

The code above is easy to understand because each processing step is on a separate line. However, once you get better with programming in R, you may find it easier and more space-efficient to combine multiple steps into one line. For example, I can recreate steps 2 - 4 above with just one line of code:

```
mean(values[values > 0]) # Combining steps 2, 3, and 4 above in one line

## [1] 0.7779074
```

There is a fine balance between code that is space-efficient and code that is incomprehensible. When in doubt, I strongly recommend separating code into multiple lines (with comments when appropriate). This will not only help other people understand your code, but more importantly, it will help your future self figure out what the heck you were doing when you wrote it!

sort(x)

Once you have a vector of data, you may want to sort it in order to see, for example, the largest and smallest values. You can do this using the `sort()` function. Let's look back on my summer joke data and sort the results:

sort(x): Sort a vector x. If you set `decreasing = F`, it will sort the vector in ascending order.

```
summer <- c(0, 1, 3, 0, 0, 5, 0, 1, 4, 3)
sort(summer, decreasing = T) # Sort decreasing

##  [1] 5 4 3 3 1 1 0 0 0 0

sort(summer, decreasing = F) # Sort increasing

##  [1] 0 0 0 0 1 1 3 3 4 5
```

You'll notice that the sort function has an argument `decreasing` which you can set to TRUE or FALSE.

Getting quick summary information of vectors

When you have a vector that contains discrete values, you may want to get a quick idea of what values are in the vector. Here are two helpful functions to do that:

unique(x)

The function `unique(x)` will tell you all the unique values in the vector, but won't tell you anything about how often each value occurs:

unique(x): Gives you all unique values in a vector, ignoring the number of times each value occurs.

```
unique(summer)

## [1] 0 1 3 5 4

unique(c("a", "A", "A", "A", "b", "b", "b", "c"))

## [1] "a" "A" "b" "c"
```

table(x)

The function `table()` does the same thing as `unique()`, but goes a step further in telling you how often each of the unique values occurs:

table(x): Gives you all unique values in a vector and tells you how often each value occurs.

```
table(summer)

## summer
## 0 1 3 4 5
## 4 2 2 1 1

table(c("a", "A", "A", "A", "b", "b", "b", "c"))

##
## a A b c
## 1 3 3 1
```

Set functions

Set functions allow you to compare two vectors and see which values are common (or uncommon) between them. For example, let's say you are comparing your top 5 favorite bands with that of a friend. We'll create two vectors, your.favs and there.favs, that contain the data:

```
your.favs <- c("Green Day", "Mars Volta", "Tool", "Smashing Pumpkins", "Nada Surf")
their.favs <- c("Maroon 5", "Avril Lavigne", "Nada Surf", "Chopin", "Tool")
```

`union(x, y)`

To see all unique values between both sets, we'll use the `union` function:

union(x, y): Tells you all unique values across two vectors

```
union(your.favs, their.favs)

## [1] "Green Day"         "Mars Volta"        "Tool"
## [4] "Smashing Pumpkins" "Nada Surf"         "Maroon 5"
## [7] "Avril Lavigne"     "Chopin"
```

`intersect(x, y)`

To see which values are in both sets, we'll use the `intersect()` function:

intersect(x, y): Tells you all values that are present in multiple vectors.

```
intersect(your.favs, their.favs)

## [1] "Tool"      "Nada Surf"
```

`setdiff(x, y)`

To see which values are *different* between the two sets, we'll use the setdiff() function. This will give us the values that are in one set but *not* in the other. As you'll see, this function is directional as it gives different results depending which vector you put first:

setdiff(x, y): Tells you which values are in x but *not* in y. Keep in mind that `setdiff(x, y)` is *not* the same as `setdiff(y, x)`!

```
setdiff(your.favs, their.favs) # Which bands do YOU like but they don't like?

## [1] "Green Day"         "Mars Volta"        "Smashing Pumpkins"

setdiff(their.favs, your.favs) # Which bands do THEY like but not you?

## [1] "Maroon 5"      "Avril Lavigne" "Chopin"
```

`setequal(x, y)`

To see if two sets are identical, we use the **setequal()** function. This function simply tests if two sets are identical. We'll run this on our existing data, plus two vectors that we know are equal.

setequal(x, y): Simply tells you whether or not two sets are identical (but ignoring order).

```
setequal(your.favs, their.favs)

## [1] FALSE

setequal(c("Green Day", "Tool"), c("Tool", "Green Day"))

## [1] TRUE
```

x %in% y

Finally, to see if a specific value is in a vector, you can use the %in% function. This function looks a bit different from other functions because it doesn't follow the typical format of function(x, y). Instead, you place the function %in% between its arguments:

x %in% y: Tells you if the values in x are in y.

```
"Green Day" %in% your.favs

## [1] TRUE

"Green Day" %in% their.favs

## [1] FALSE
```

5: Plotting Basics

Chapter Goals

1. High-level plotting commands: plot(), hist(), boxplot, barplot()

2. Main plotting parameters: main, xlab, ylab, xlim, ylim

3. Low-level plotting functions: abline(), points(), text(), legend()

4. Saving plots with pdf() and jpg()

Plotting in R works like putting paint on a canvas. You start by creating a canvas and drawing a basic outline with a *high-level* plotting command, then add sequentially add individual elements with *low-level* plotting commands. And just like painting on a canvas, you can't 'erase' any elements after you've put them on a plot - once they're there they're there for good!

High-level plotting functions

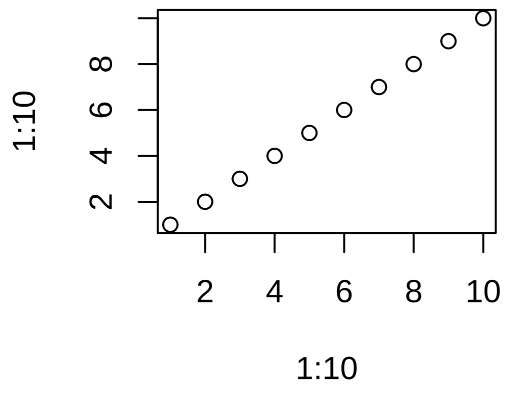
Next, we'll cover some of the key high-level plotting functions in R.

`plot(x ,y)`

`plot(x, y)`: Create a scatterplot from two vectors x and y.

Let's start by creating a simple scatterplot using the `plot()` command. The plot function has many, many optional arguments, but for now we'll just look at the two main arguments:
- x: A data vector for the x-coordinates. - y: A data vector for the y-coordinates.
Let's make a simple plot with 10 data points: (1, 1), (2, 2), (3, 3), ... (10, 10). I'll set both the x and y values to be the integers from 1 to 10.

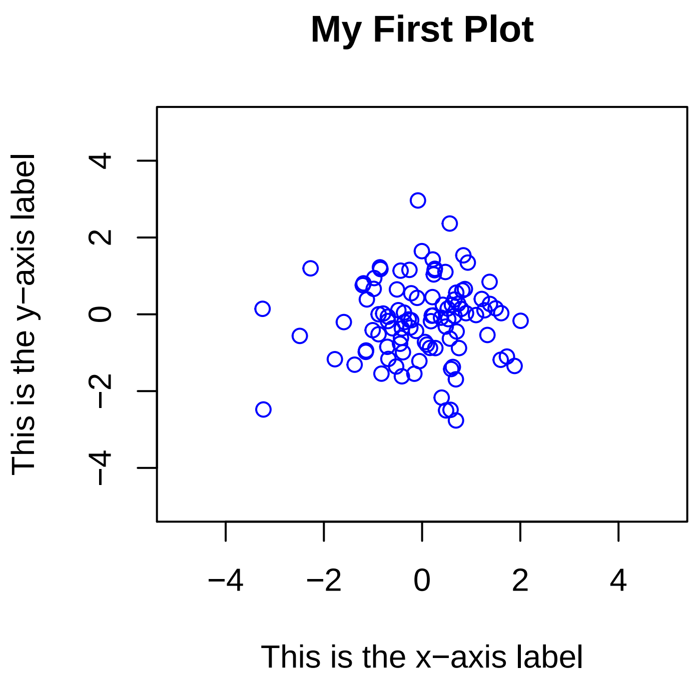
```
plot(x = 1:10,
     y = 1:10)
```



As you can see, R has plotted 10 points. The x and y labels are both just "1:10" and the plot has no title. Not too exciting. Now let's jazz up the plot a bit by putting in some more interesting data and adding several parameters:

Common plotting parameters
`main`: Title of plot
`xlab, ylab`: axes labels
`xlim, ylim`: Limits of axes
`xaxt, yaxt`: Set to `"n"` to remove the axes
`cex`: Size of the plotting points
`pch`: Type of plotting points (see `?points`)

```
plot(x = rnorm(n = 100, mean = 0, sd = 1),
     y = rnorm(n = 100, mean = 0, sd = 1),
     main = "My First Plot",
     xlab = "This is the x-axis label",
     ylab = "This is the y-axis label",
     xlim = c(-5, 5), # Min and max values for x-axis
     ylim = c(-5, 5), # Min and max values for y-axis
     col = "blue" # Color of the points
     )
```

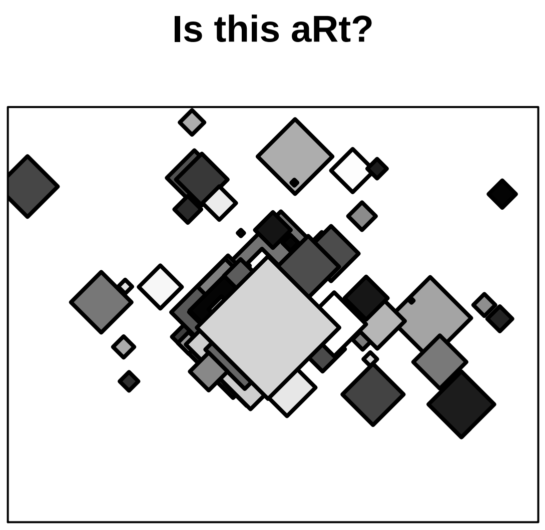


Now things are looking better. We've specified several of the most common plotting parameters including the plot title (main), axis labels (xlab and ylab), the axis limits (xlim and ylim) and the color of the plotting symbols (col).

There are many, many more parameters than change the look of the plotting region. To see all of these, type ?par in R to bring up the help menu on plotting parameters.

Here's another plot with some additional changes:

```
x.data <- rnorm(n = 100, mean = 0, sd = 1) # Generate the x data
y.data <- rnorm(n = 100, mean = 0, sd = 1) # Generate the y data

plot(x = x.data,
     y = y.data,
     main = "Is this aRt?",
     xlab = "",
     ylab = "",
     xaxt = "n",
     yaxt = "n",
     pch = 23, # Use filled diamond points
     bg = gray(runif(100, 0, 1)), # Color of the inside of the points,
     col = "black", # Color of the border of the points
     cex = rnorm(n = 100, mean = 1, sd = 3), # Size of the points
     lwd = 2 # Widths of the point borders
     )
```



Let's go through the new parameters that we specified:

- `pch`: This dictates the point type you want to use. You indicate which point type you want with a number. To see what each of the point types are, look at the points help menu with ?points. There, you'll see that the value of 23 is a filled diamond (like the ones in the plot above).
- `bg`: The color of the filling of a point. This only applies if you are using filled point type (that is, a type that has a filling in addition to a border). Notice that in the example above I specified this using the `gray()` function, which gives you different shades of gray. Check out `?gray` to see more on this useful color function.
- `cex`: The size of the points. The default value is 1 and can range anywhere from 0 to...I guess infinity.
- `lwd`: The width of the border of points. Again, default is 1 but can be any positive number.
- `xaxt` and `yaxt`: Set to "n" to hide the x or y axes. I did this in the example above.

You'll notice that you can input a vector for many of these parameters. When you do this, R will apply the ith value of the vector to the ith point. For example, in the plot above I set cex = rnorm(n = 100, mean = 0, sd = 1), 100 samples from a standard normal distribution. This means that the size of each of the 100 points will be determined by each of these 100 standard normal values. For this plot, the size of the points was meaningless, but in other plots you can change the size of the point based on a meaningful third variable in your data (for example, the number of instances of that point).

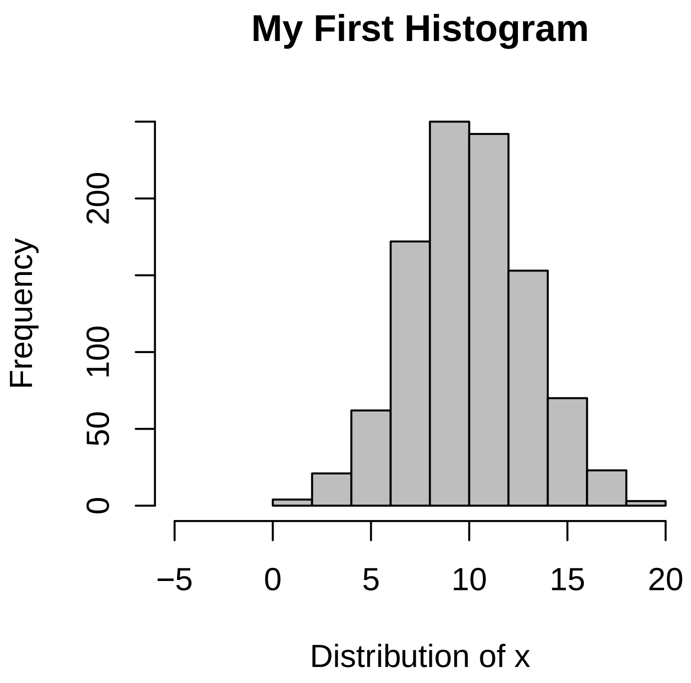
The plot() command creates scatterplots, which require two vectors of data (for the x and y values). For univariate data, we may want to create a different type of plot. Here are three common ones:

```
hist(x, main, xlab, ylab)
```

The function `hist()` creates a histogram based on a vector of data. Let's create a histogram of 1,000 values drawn from a Normal distribution with μ = 10 and σ = 3:

```
hist(x = rnorm(n = 1000, mean = 10, sd = 3),
     main = "My First Histogram",
     xlab = "Distribution of x",
     xlim = c(-5, 20),
     col = "gray" # Color of the bars
     )
```

`hist(x)`: Create a histogram of a vector x
Optional Parameters
`breaks`: Defines how bins are set (see `?hist`)
`border`: Color of bar borders
`col`: Color of bar fillings
`freq`: Set to `FALSE` to plot probability densities

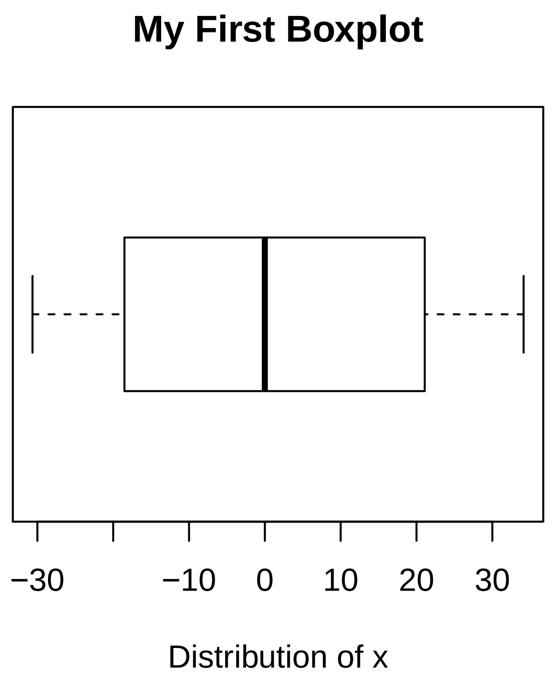


The histogram function has a few special arguments you may want to vary (including the number and width of the bars). Look at the help menu for histograms (?hist) to see these.

`boxplot(x, main, xlab, ylab)`

The function `boxplot()` creates a boxplot based on a vector of data. A boxplot shows you several *quantiles* of a dataset. Let's create a boxplot of a *bi-modal* set of data that contains 50 samples from a normal distribution with μ = 20 and σ = 5, and 50 samples from another normal distribution with μ = -20 and σ = 5.

```
data <- c(rnorm(n = 50, mean = 20, sd = 5),
          rnorm(n = 50, mean = -20, sd = 5)
          )

boxplot(x = data,
        main = "My First Boxplot",
        horizontal = T,
        xlab = "Distribution of x"
        )
```



Boxplots are a good way to visualize key percentiles in a data set. However, they can hide multiple modes in a dataset.

Looking at the boxplot, we can see that the median falls around 0. However, the boxplot completely hides the bi-modal nature of the data. To see this, we need to use a different kind of plot. One great candidate is the *beanplot*:

```
beanplot(x)
```

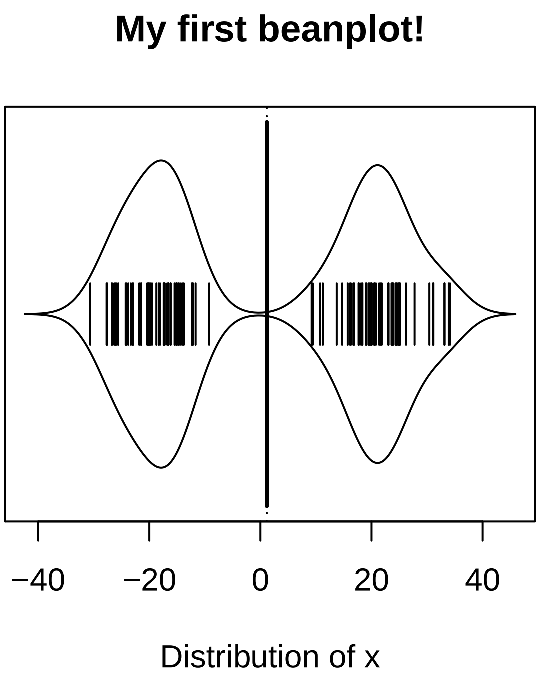
Beanplots are a really cool way to picture a distribution. In fact, beanplots are so cool that when I use them in a conference poster - 9 times out of 10 people ignore my research and ask "How did you make those cool plots?!" The beanplot function is not part of the base R software. To use it, you first need to install the R Package "beanplot". To do this run the following code:

```
install.packages("beanplot") # Install the beanplot library
```

Once you've installed the package on your machine, you need to load the package using the function library. Once you've done that, you can use the beanplot function to make beautiful beanplots!

`beanplot(x)`: Create a beanplot. This function has many optional arguments. See `?beanplot` to learn about them. Also, be warned that because they're so darn pretty, they may distract conference-attendees from your real hard work!

```
require("beanplot")

## Loading required package:  beanplot

beanplot(x = data,
         main = "My first beanplot!",
         xlab = "Distribution of x",
         col = "white",
         horizontal = T
         )
```



Beanplots are great because they show you all the original data, represented as small lines, the average value, shown by a long line, and a smoothed distribution of the data with curved lines. To see how you can customize the look of beanplots further, check the help menu (?beanplot)

Low-level plotting functions

Once you've created a plot with a high-level plotting function, you can add elements to it using low-level plotting functions. Here are some of the most common low-level plotting functions:

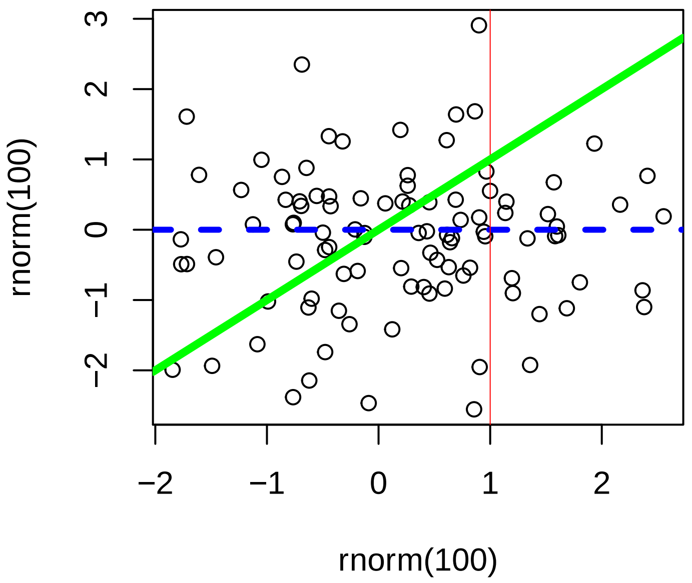
`abline(a, b, h, v, lty, lwd)`

With `abline()` you can add one or more lines to a plot. You may want to add lines for regression equations, reference points, or to indicate group means. Here are some of the optional parameters:

`abline(a, b, h, v)`: Add straight lines to an existing plot
`a, b`: Intercept and slope of lines
`h`: y-values of horizontal lines
`v`: x-values of vertical lines

- a, b: The intercept and slope
- h: the y-value(s) for horizontal line(s)
- v: the x-value(s) for vertical line(s)
- lty: The type of line(s)
- lwd: The width of the line(s)
- col: The color(s) of the line(s)

```
plot(x = rnorm(100), y = rnorm(100))

abline(h = 0, lty = 2, lwd = 3, col = "blue") # Add a horizontal blue line
abline(v = 1, lty = 1, lwd = .5, col = "red") # Add a thin vertical red line
abline(a = 0, b = 1, lty = 1, lwd = 4, col = "green") # Add a diagonal green line
```

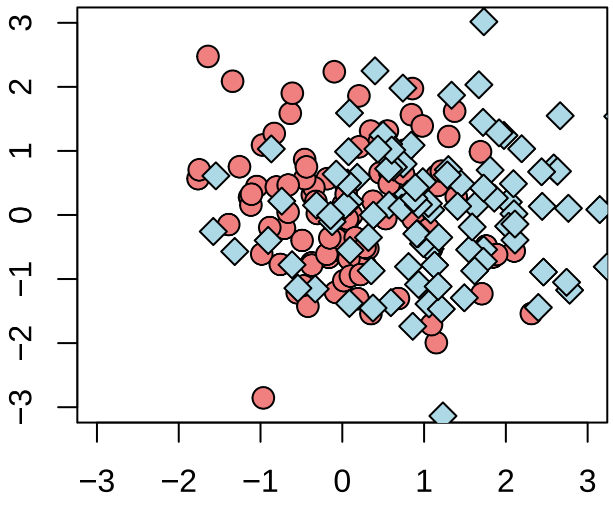


`points(x, y, pch, cex)`

With texttpoints, you can add points to an existing plot. The points() function works a lot like plot(), except that it adds points to an existing plot. You would want to do this if, for example, you want to plot different colored points for different data sets:

`points(x, y)`: Add points to an existing plot
`pch`: The plotting symbol (entered as a number). You can also plot characters as symbols (e.g.; "x")
`bg`: Background color of symbols (only when symbol has a background)
`col`: Color of symbols
`cex`: Size of symbols

```
plot(x = rnorm(100),
     y = rnorm(100),
     xlim = c(-3, 3),
     ylim = c(-3, 3),
     bg = "lightcoral",
     pch = 21,
     cex = 1.5,
     xlab = "", # Don't show xlabel
     ylab = "" # Don't show ylabel
     )

points(x = rnorm(100, mean = 1, sd = 1),
       y = rnorm(100),
       pch = 23,
       bg = "lightblue",
       cex = 1.5
       )
```



Of course, because the points() command is a low-level plotting command that only adds to an existing plot, you don't need to specify things like the axis limits or titles of the plot (that's all done in the high-level command)

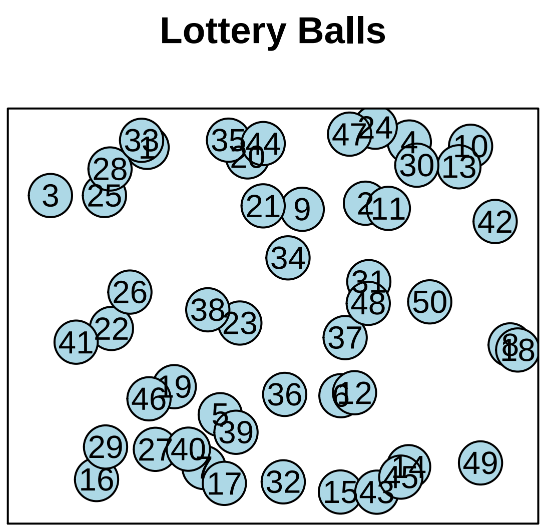
`text(x, y, labels, cex)`

With `text()`, you can add text to a plot. You can use text() to highlight specific points of interest in the plot, or to add information (like a third variable) for every point in a plot

- x, y: Coordinates of the text
- labels: The text you're plotting
- cex: The size of the text

`text(x, y, labels)`: Add text to an existing plot at specified points
`x, y`: x and y coordinates of the text
`labels`: The text you want to plot
`cex`: Size of text

```
x.vals <- runif(50, min = -20, max = 20)
y.vals <- runif(50, min = -20, max = 20)

plot(x = x.vals,
     y = y.vals,
     xlim = c(-20, 20),
     ylim = c(-20, 20),
     bg = "lightblue",
     col = "black",
     pch = 21,
     cex = 3,
     main = "Lottery Balls",
     xlab = "", ylab = "", xaxt = "n", yaxt = "n"
     )

# Add numbers to the balls
text(x = x.vals,
     y = y.vals,
     labels = 1:50,
     cex = 1
     )
```



Formatting text for plotting

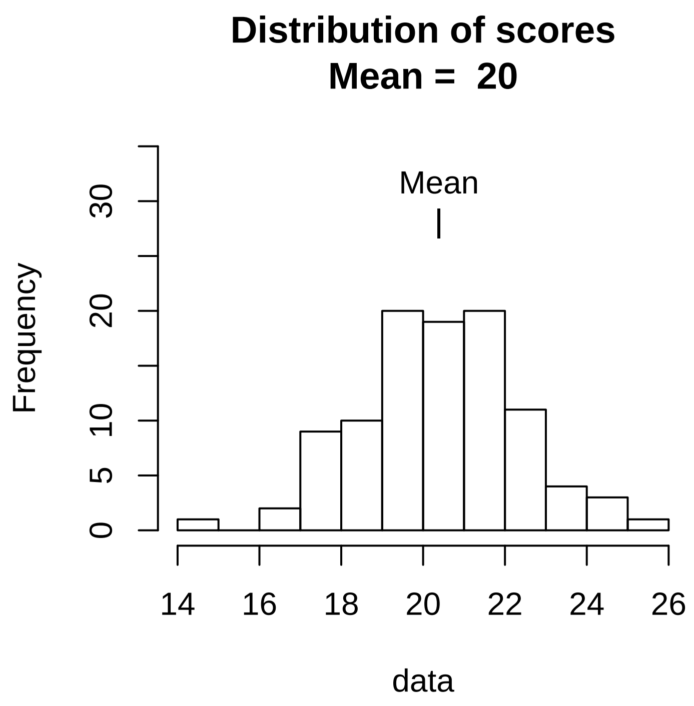
A common way to use text in a plot, either in the main title of a plot or using the `text()` function, is to combine text with numerical data. For example, you may want to include the text "Mean = 3.14" in a plot to show that the mean of the data is 3.14. But how can we combine numerical data with text? In R, we can do this with the `paste()` function:

```
paste(x, y, sep)
```

- x, y, ...: One or more arguments to be converted to characters
- sep: A character string that separates the arguments. Set to `""` for no separation

When you include descriptive statistics in a plot, you will almost always want to use the `round(x, digits)` function to reduce the number of digits in the statistic.

```
data <- rnorm(100, mean = 20, sd = 2)

hist(data,
     ylim = c(0, 35),
     main = paste("Distribution of scores\nMean = ", round(mean(data, 2)))
     )

text(x = mean(data), y = 30,
     labels = "Mean\n|")
```



The benefit of using `paste()` over hard-coding the text (for example by typing `labels = "Mean = 20"`) is that the code will automatically change the value of the mean when the data changes. In later chapters, when use loops to create multiple graphs over different sets of data, this will become extremely helpful!

`legend(x, y, labels)`

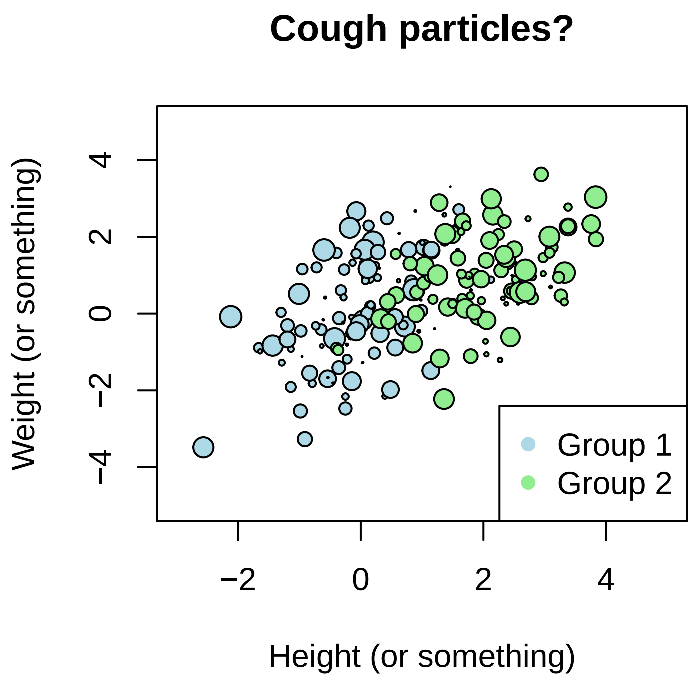
With `legend()`, you can add a legend to the plot. Here are a few common arguments, though you can see additional ones in the help menu (?legend)

- x, y: Coordinates of the legend. Alternatively, you can enter a string indicating where to put the legend (i.e.; ”topright”, ”topleft”)
- legend: Text in the legend
- lty: Line type for lines in the legend (optional)
- pch: Point types in the legend (optional)

`legend(x, y, labels)`: Add a legend to a plot
`x, y`: Either two separate x and y coordinates, or a string indicating where to put the legend (e.g.; ”topright”)
`legend`: The text in the legend
`lty`: The line types
`pch`: The symbol types

```
g1.x <- rnorm(100)
g1.y <- g1.x + rnorm(100, sd = 1.5)

plot(x = g1.x,
     y = g1.y,
     bg = "lightblue",
     pch = 21,
     xlab = "Height (or something)", ylab = "Weight (or something)",
     main = "Cough particles?",
     xlim = c(-3, 5),
     ylim = c(-5, 5),
     cex = runif(100, min = 0, max = 1.5)
     )

g2.x <- rnorm(100, mean = 2)
```

```
g2.y <- g2.x * .5  - rnorm(100, sd = 1)

points(x = g2.x,
       y = g2.y,
       bg = "lightgreen",
       pch = 21,
     cex = runif(100, min = 0, max = 1.5)
       )

legend("bottomright",
       pch = c(16, 16),
       col = c("lightblue", "lightgreen"),
       legend = c("Group 1", "Group 2")
       )
```



Additional low-level plotting functions

There are many more low-level plotting functions that can add additional elements to your plots. Here are some I use:

Drawing elements

- `rect()`: Add rectangles.
- `polygon()`: Add polygons (pretty much any shape you want)
- `segments()`: Add segments (lines with fixed endings).
- `arrows()`: Add arrows

Plotting elements

- `axis()`: Add an additional axis to a plot (or add fully customizable x and y axes.
- `mtext()`: Add text to the margins of a plot

Saving plots to a file

Once you've created a plot in R, you may wish to save it to a file so you can use it in another document. To do this, you'll use either the `pdf()` or `jpeg()` functions. These functions will save your plot to either a .pdf of jpeg file.

To use these functions to save files, you need to follow 3 steps

`pdf(file, width, height)`, `jpeg(file, width, height)`: Save a plot as a pdf or jpeg
`file`: The directory you want to save the file in
`height, width`: The height and width of the plot in inches.

1. Entering the function and specifying a few parameters, like where you want the file to be saved and how big you want the plot to be.
2. Execute all your plotting code.
3. Close the function by entering the closing command **dev.off()**

Here's an example of the three steps.

```
# Step 1: Call the pdf command
pdf(file = "/Users/Nathaniel/Desktop/My Plot.pdf",   # The directory you want to save the file in
    width = 4, # The width of the plot in inches
    height = 4 # The height of the plot in inches
    )

# Step 2: Create the plot
hist(rnorm(100))

# Step 3: Close the pdf function and create the file
dev.off()
```

You'll notice that after you close the plot with `dev.off()`, you'll see a message in the prompt like "null device".

Using the command `pdf()` will save the file as a pdf. If you use `jpeg()`, it will be saved as a jpeg.

6: Matrices and Data Frames

Chapter Goals

1. Learn about the matrix and dataframe data objects
2. Create matrices with matrix(), cbind(), and data.frame()
3. Index matrices/dataframes with brackets [], and $
4. Use matrix/dataframe functions dim(), nrow(), ncol(), head(), tail()
5. Subset dataframes with subset()

Creating matrices and dataframes

By now, you should be comfortable with scalers and vectors. Next, we'll cover the next two most common data objects in R, **matrices** and **dataframes**

Matrices and dataframes are both two dimensional objects that contain rows and columns. Really, they're just like spreadsheets in Excel. Each matrix or dataframe contains a certain number of rows (call that number m) and columns (n). You can think of a matrix as a combination of n vectors, where each vector has a length of m. See Figure 8 to see the difference.

I use one of the following three functions to create matrices:

```
# scaler v vector v matrix

par(mar = rep(1, 4))
plot(1, xlim = c(0, 10), ylim = c(-.5, 5),
     xlab = "", ylab = "",
     xaxt = "n", yaxt = "n",
     bty = "n", type = "n")

# Scaler
rect(rep(0, 1), rep(0, 1), rep(1, 1), rep(1, 1))
text(.5, -.5, "Scaler")

# Vector
rect(rep(2, 5), 0:4, rep(3, 5), 1:5)
text(2.5, -.5, "Vector")

# Matrix
rect(rep(4:8, each = 5),
     rep(0:4, times = 5),
     rep(5:9, each = 5),
     rep(1:5, times = 5))
text(6.5, -.5, "Matrix / Data Frame"
     )
```

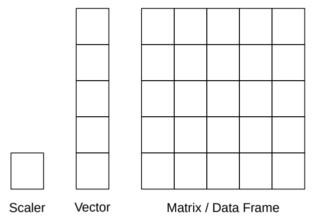


Figure 8: Scaler, Vector, Matrix... ::drops mike::

`cbind(x, y), rbind(x, y)`: Combine two (or more) vectors into a matrix by column or row.

cbind() and rbind()

`cbind(x, y), rbind(x, y)`

`cbind()` and `rbind()` joins several vectors together to create a matrix. cbind() combines vectors as columns in the matrix, while rbind() combines them as rows:

```
x <- 1:10
y <- 11:20
z <- 21:30

matrix.1 <- rbind(x, y, z)
matrix.1

##   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## x    1    2    3    4    5    6    7    8    9    10
## y   11   12   13   14   15   16   17   18   19    20
## z   21   22   23   24   25   26   27   28   29    30

matrix.2 <- cbind(x, y, z)
matrix.2

##         x  y  z
##  [1,]   1 11 21
##  [2,]   2 12 22
##  [3,]   3 13 23
##  [4,]  4 14 24
##  [5,]  5 15 25
##  [6,]  6 16 26
##  [7,]  7 17 27
##  [8,]  8 18 28
##  [9,]  9 19 29
## [10,] 10 20 30
```

matrix(data, nrow, ncol, byrow)

`matrix()` will create a matrix from a single vector of data with dimensions that you designate.

- data: A vector of data
- nrow: The number of rows in the resulting matrix
- ncol: The number of columns in the resulting matrix
- byrow: A logical value indicating whether to fill the matrix by row or column

```
matrix.1 <- matrix(data = 1:30,
                   nrow = 10,
                   ncol = 3)
matrix.1

##       [,1] [,2] [,3]
##  [1,]    1   11   21
##  [2,]    2   12   22
##  [3,]    3   13   23
##  [4,]    4   14   24
##  [5,]    5   15   25
##  [6,]    6   16   26
##  [7,]    7   17   27
##  [8,]    8   18   28
##  [9,]    9   19   29
## [10,]   10   20   30

matrix.2 <- matrix(data = 1:30,
                   nrow = 3,
                   ncol = 10)
matrix.2

##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    1    4    7   10   13   16   19   22   25    28
## [2,]    2    5    8   11   14   17   20   23   26    29
## [3,]    3    6    9   12   15   18   21   24   27    30
```

Keep in mind that matrices can either contain numbers or characters. If you try to create a matrix with both numbers and characters, it will turn all the numbers into characters:

```
cbind(1:5, c("a", "b", "c", "d", "e"))

##      [,1] [,2]
## [1,] "1"  "a"
## [2,] "2"  "b"
## [3,] "3"  "c"
## [4,] "4"  "d"
## [5,] "5"  "e"
```

Indexing matrices with brackets [rows, columns]

Just like vectors, you can access specific data in matrices using brackets. But now, instead of just using one index vector, we use two index vectors: one for the rows and one for the columns. To do this, use the notation `matrix[rows, columns]` like this:

```
matrix.1 <- matrix(data = 1:30, nrow = 10, ncol = 3)

matrix.1[1:5, 1] # Give me rows 1 through 5 in column 1

## [1] 1 2 3 4 5

matrix.1[2:6, 2:3] # Give me rows 2 through 6 in columns 2 and 3

##      [,1] [,2]
## [1,]   12   22
## [2,]   13   23
## [3,]   14   24
## [4,]   15   25
## [5,]   16   26

matrix.1[,1] # Give me column 1

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

matrix.1[3,] # Give me row 3

## [1]  3 13 23
```

As you can see, if you leave one of the entries blank in the brackets [,], R will assume that you want all values in that index. For example, as you see above, the code matrix.1[,1] gives you the data in all rows in column 1.

Dataframe: An m x n object containing numbers, strings and factors

A dataframe looks a lot like a matrix at first: it is also rectangular and has m rows and n columns. However, unlike matrices, dataframes can contain *both* string vectors and numeric vectors within the same object. For this reason, most large datasets in R, for example, a survey including numeric data and text data, will be stored as dataframes.

A dataframe is just a more flexible matrix that allows you to combine both character and numeric vectors into the same data object. Because dataframes are more flexible than matrices, Most datafiles you use will be stored as dataframes.

`data.frame()`

To create a dataframe, you can use the `data.frame()` function. Let's create a dataframe of fictional survey data. I'll create 5 entries for Males and 5 entries for Females. I'll then generate 10 heights from a normal distribution with mean 150 and standard deviation 10.

```
survey <- data.frame("gender" = rep(c("Female", "Male"), each = 10),
                     "height" = rnorm(20, mean = 150, sd = 10),
                     stringsAsFactors = F # don't convert strings to factors
                    )
survey # Print the dataframe

##    gender   height
## 1  Female 151.9214
## 2  Female 141.9948
## 3  Female 157.7751
## 4  Female 151.8093
## 5  Female 136.6475
## 6  Female 139.1870
## 7  Female 153.9135
## 8  Female 162.4014
## 9  Female 157.3316
## 10 Female 144.6161
## 11   Male 163.0944
## 12   Male 148.8995
## 13   Male 140.9836
## 14   Male 137.8854
## 15   Male 151.3401
## 16   Male 154.7951
## 17   Male 141.6672
## 18   Male 143.8541
## 19   Male 161.9026
## 20   Male 154.7752
```

You'll notice I included the argument **`stringsAsFactors = F`**, this tells R to NOT convert the strings (the Gender column) to a factor datatype. For now, don't worry about what factors are. Just know that you don't want to use them just yet!

Accessing columns in a dataframe with $

You'll notice that I included a name for each column. This not only helps to remind you which data are in each column, but also allows you to access the columns by name using the **`$`** operator:

```
survey$height

##  [1] 151.9214 141.9948 157.7751 151.8093 136.6475 139.1870 153.9135
##  [8] 162.4014 157.3316 144.6161 163.0944 148.8995 140.9836 137.8854
## [15] 151.3401 154.7951 141.6672 143.8541 161.9026 154.7752

survey$gender

##  [1] "Female" "Female" "Female" "Female" "Female" "Female" "Female"
##  [8] "Female" "Female" "Female" "Male"   "Male"   "Male"   "Male"
## [15] "Male"   "Male"   "Male"   "Male"   "Male"   "Male"
```

Accessing names of dataframe columns

Once you have a dataframe, you can always get the names of the dataframe by using the function **`names()`**

```
names(survey)

## [1] "gender" "height"
```

If you want to change the names of columns in a dataframe, you can do this using indexing and assignments:

```
names(survey)[1:2] <- c("GENDER", "HEIGHT")
survey

##    GENDER   HEIGHT
## 1  Female 151.9214
## 2  Female 141.9948
## 3  Female 157.7751
## 4  Female 151.8093
## 5  Female 136.6475
## 6  Female 139.1870
## 7  Female 153.9135
## 8  Female 162.4014
## 9  Female 157.3316
## 10 Female 144.6161
## 11   Male 163.0944
## 12   Male 148.8995
## 13   Male 140.9836
## 14   Male 137.8854
## 15   Male 151.3401
## 16   Male 154.7951
## 17   Male 141.6672
## 18   Male 143.8541
## 19   Male 161.9026
## 20   Male 154.7752
```

Adding columns to a dataframe with andassignment

You can easily add columns to a dataframe using the $ and assignment <- operators:

```
survey$age <- round(rnorm(20, 20, 2), 1)
survey$siblings <- sample(1:5, 20, replace = T)
survey

##    GENDER   HEIGHT  age siblings
## 1  Female 151.9214 21.3        1
## 2  Female 141.9948 21.7        5
## 3  Female 157.7751 20.5        2
## 4  Female 151.8093 18.1        1
## 5  Female 136.6475 22.5        3
## 6  Female 139.1870 18.5        3
## 7  Female 153.9135 17.9        4
## 8  Female 162.4014 17.6        2
## 9  Female 157.3316 22.5        1
## 10 Female 144.6161 13.9        5
## 11   Male 163.0944 20.2        3
## 12   Male 148.8995 23.2        5
## 13   Male 140.9836 16.6        3
## 14   Male 137.8854 16.6        2
## 15   Male 151.3401 19.7        5
## 16   Male 154.7951 20.5        5
## 17   Male 141.6672 17.6        3
## 18   Male 143.8541 20.2        1
## 19   Male 161.9026 19.3        2
## 20   Male 154.7752 17.3        2
```

Getting information about matrices and dataframes

Once you have a matrix or dataframe, you can use many functions to get information about them. Here are some common ones:

- head: Print the first few rows (this is especially usefull when the matrix is very large)
- tail: Print the last few rows
- dim: How many rows and columns are there?

- nrow: How many rows are there?
- ncol: How many columns are there?

```
head(survey) # Print the first few rows

##   GENDER   HEIGHT  age siblings
## 1 Female 151.9214 21.3        1
## 2 Female 141.9948 21.7        5
## 3 Female 157.7751 20.5        2
## 4 Female 151.8093 18.1        1
## 5 Female 136.6475 22.5        3
## 6 Female 139.1870 18.5        3

tail(survey) # Print the last few rows

##    GENDER   HEIGHT  age siblings
## 15   Male 151.3401 19.7        5
## 16   Male 154.7951 20.5        5
## 17   Male 141.6672 17.6        3
## 18   Male 143.8541 20.2        1
## 19   Male 161.9026 19.3        2
## 20   Male 154.7752 17.3        2

dim(survey) # How many rows and columns?

## [1] 20  4

nrow(survey) # How many rows?

## [1] 20

ncol(survey) # How many columns?

## [1] 4
```

Subsetting dataframes with indexing and subset()

Frequently you will want to get access to subsets of data in a dataframe based on some criteria. For example, we may want to look just at the data from females in our survey data. To do this, we can use one of two methods: indexing with logical vectors, or the `subset()` function.

Indexing dataframes with logical vectors is very similar to indexing data vectors. First, we create a logical vector. Next, we index the dataframe using that logical vector:

```
females.log <- survey$GENDER == "Female"
females.log

##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
## [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

survey.females <- survey[females.log,] # Give me the survey data for females
survey.females

##    GENDER   HEIGHT  age siblings
## 1  Female 151.9214 21.3        1
## 2  Female 141.9948 21.7        5
## 3  Female 157.7751 20.5        2
## 4  Female 151.8093 18.1        1
## 5  Female 136.6475 22.5        3
## 6  Female 139.1870 18.5        3
## 7  Female 153.9135 17.9        4
## 8  Female 162.4014 17.6        2
## 9  Female 157.3316 22.5        1
## 10 Female 144.6161 13.9        5
```

`subset(x, subset, select)`

Alternatively we can use the `subset()` function which usually looks a bit nicer than logical indexing. The subset() function has the following arguments:

- x: The data (usually a dataframe)
- subset: A logical vector indicating which rows you want to select
- select: An optional vector of the columns you want to select

`subset(x, subset, select)` allows you to select certain rows and columns of a dataframe based on criteria you set.
`x`: The data (usually a dataframe)
`subset`: A logical vector indicating which rows you want to select. For example: **`gender == "female"`**.
`select`: An optional vector of the names of columns you want to select.

```
survey.males <- subset(x = survey,
                       subset = GENDER == "Male"
                       )
survey.males

##    GENDER   HEIGHT  age siblings
## 11   Male 163.0944 20.2        3
## 12   Male 148.8995 23.2        5
## 13   Male 140.9836 16.6        3
## 14   Male 137.8854 16.6        2
## 15   Male 151.3401 19.7        5
## 16   Male 154.7951 20.5        5
## 17   Male 141.6672 17.6        3
## 18   Male 143.8541 20.2        1
## 19   Male 161.9026 19.3        2
## 20   Male 154.7752 17.3        2
```

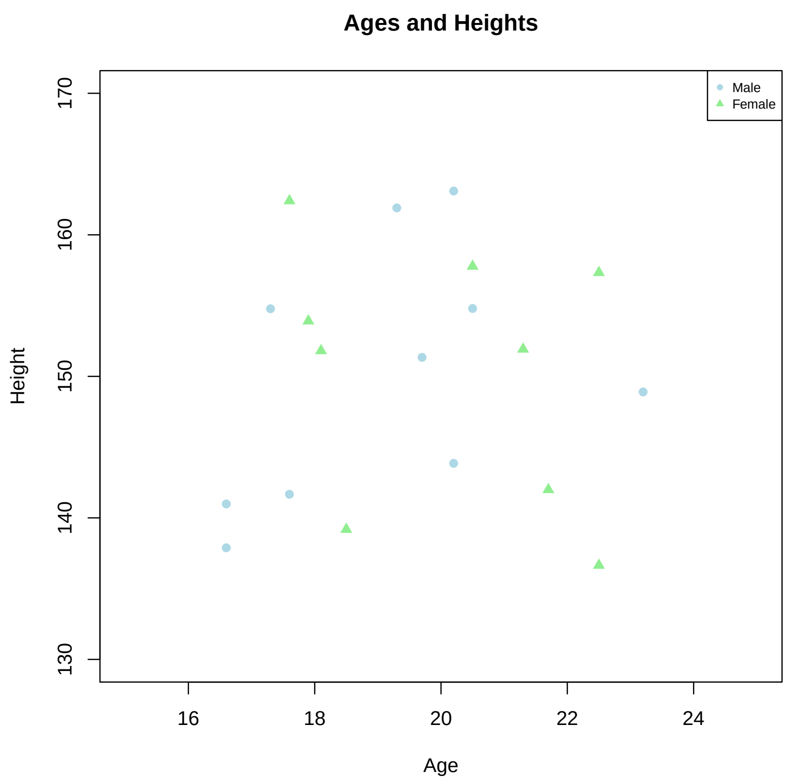
Combining indexing and descriptive statistics

Once you know how to index a dataframe to get the data vectors you want, you can then easily calculate descriptive statistics based on specific criteria. For example, we can separately calculate statistics for males and females:

```
mean(survey$HEIGHT[survey$GENDER == "Female"]) # Mean height of females

## [1] 149.7598

mean(survey$HEIGHT[survey$GENDER == "Male"]) # Mean height of males

## [1] 149.9197

max(subset(survey, GENDER == "Female")$HEIGHT) # Tallest female

## [1] 162.4014

max(subset(survey, GENDER == "Male")$HEIGHT) # Tallest male

## [1] 163.0944
```

Now we can also create plots with different elements for different groups:

```r
male.height <- survey$HEIGHT[survey$GENDER == "Male"]
male.age <- survey$age[survey$GENDER == "Male"]

female.height <- survey$HEIGHT[survey$GENDER == "Female"]
female.age <- survey$age[survey$GENDER == "Female"]

plot(x = male.age,
     y = male.height,
     xlab = "Age",
     ylab = "Height",
     xlim = c(15, 25),
     ylim = c(130, 170),
     pch = 16,
     col = "lightblue",
     main = "Ages and Heights"
     )

points(x = female.age,
       y = female.height,
       pch = 17,
       col = "lightgreen"
     )

legend("topright",
       pch = c(16, 17),
       col = c("lightblue", "lightgreen"),
       legend = c("Male", "Female"),
       bg = "white",
       cex = .7
       )
```


Ages and Heights
Male
Female
Height
Age
130
140
150
160
170
16
18
20
22
24

7: 1 and 2-sample Null-Hypothesis tests

Chapter Goals

1. Learn about hypothesis test objects in R
2. One and two sample tests: Correlations, t-tests and chi-square

Do we get more treasure from chests buried in the sand or at the bottom of the ocean? Is there a relationship between the number of scars a pirate has and how much grogg he can drink? Are pirates with nipple rings more likely to wear bandannas than those without nipple rings? Glad you asked, let's see how we can answer these questions some hypothesis tests.

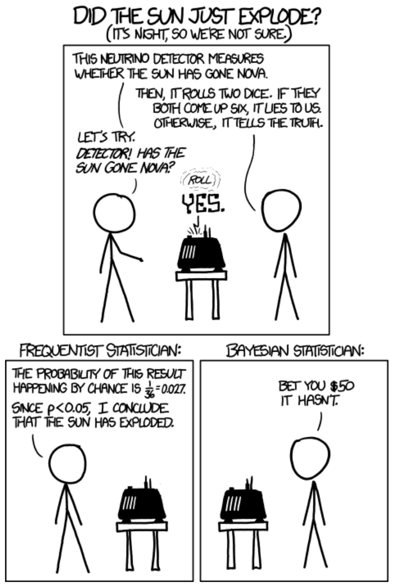

DID THE SUN JUST EXPLODE?
(IT'S NIGHT, SO WE'RE NOT SURE.)
THIS NEUTRINO DETECTOR MEASURES WHETHER THE SUN HAS GONE NOVA.
THEN, IT ROLLS TWO DICE. IF THEY BOTH COME UP SIX, IT LIES TO US. OTHERWISE, IT TELLS THE TRUTH.
LET'S TRY. DETECTOR! HAS THE SUN GONE NOVA?
(ROLL)
YES.
FREQUENTIST STATISTICIAN:
THE PROBABILITY OF THIS RESULT HAPPENING BY CHANCE IS 1/36=0.027. SINCE p<0.05, I CONCLUDE THAT THE SUN HAS EXPLODED.
BAYESIAN STATISTICIAN:
BET YOU $50 IT HASN'T.

Figure 9: xkcd comic. Currently used without any permission.

Warning about null-hypothesis tests with "frequentist" statistics

Until recently, null-hypothesis testing using frequentist statistics has been the most popular method of conducting inferential statistics. However, it has serious flaws. While I can't go into the details here, I can point out that the main flaw is that frequentist statistics don't give you the information you really want to know. For example, imagine that you are comparing the effectiveness of a cancer drug to a placebo. After conducting a double-blind study, where you give some patients the placebo and some patients the drug, you want to know the probability that that the drug is better than a placebo. Unfortunately frequentist statistics cannot give you this information. They can only tell you the probability of getting a specific result *given* that the null hypothesis (in this case, that the drug is equally as effective as the placebo) is true. If that sounds confusing, it's because it is. A better alternative is Bayesian statistics which *can* give you posterior probability information. Unfortunately, Bayesian statistics can be computationally demanding, so in the past we've lived with frequentist statistics and tried to ignore its fundamental flaws. However, given improvements in processing speed, we can now easily conduct Bayesian alternatives to frequentist tests on modern computers.

We will cover Bayesian statistics in Chapter X and I strongly encourage you to adopt them in your own analyses. However, for the purposes of completeness, I'll show you how to conduct most of the standard frequentist tests here.

T-test

The t-test function is easy to remember: `t.test()`. The `t.test()` function can take several arguments, here I'll emphasize a few of them. To see them all, check the help menu for t.test (`?t.test`).

`t.test(x, y)`: Conduct either a one sample t-test on a vector x, or a two-sample t-test on two vectors x and y.

```
t.test(x, y, alternative, mu, paired, var.equal)
```

- x: A vector of data.
- y: An (optional) second vector of data if you are conducting a two-sample test.
- alternative: A character string indicating whether the test is two-tailed or one-tailed (including the direction). Type "t" for two-tailed, "g" for a 'greater than' one-tailed test, or "l" for a "less than" one-tailed test.

- mu: The population mean under the null hypothesis.
- paired: A logical value (either T or F) indicating whether the test is paired (T) or unpaired (F).
- var.equal: A logical value indicating whether or not you treat the two variances as equal.

Let's do an example using the ChickWeight dataset in R (type ?ChickWeight for more info). This dataset shows the weights of several chickens over several time periods, where different chickens are on different diets.

First, let's do a one-sample t-test on the chicken weights at time 0. Let's see if it's significantly different from 30:

```
time.0 <- ChickWeight$weight[ChickWeight$Time == 0] # Vector of weights at time 0

test.result <- t.test(x = time.0,
                      mu = 30)
```

You'll notice that when you assign the t.test to an object (in this case `result`), you do not see any output. To see the output of the test, you need to tell R to print the object by executing the name of the test object:

```
test.result

##
##  One Sample t-test
##
## data:  time.0
## t = 69.07, df = 49, p-value < 2.2e-16
## alternative hypothesis: true mean is not equal to 30
## 95 percent confidence interval:
##  40.73821 41.38179
## sample estimates:
## mean of x
##     41.06
```

Now, you can see the main output of the test. Looks like we got a test statistic of 69.07 and a resulting p-value that's pretty darn small. But what if you want to access specific values like the test statistic or the p-value? Thankfully, this is easy in R. To see which information you can extract from the t-test object, apply the `names()` function to the test object:

If you want to see what information is in a test object, just apply `names()` to the object. You can then extract specific information with $

```
names(test.result)

## [1] "statistic"   "parameter"   "p.value"     "conf.int"    "estimate"
## [6] "null.value"  "alternative" "method"      "data.name"
```

From this vector of names, I see that I can extract the test statistic with the name `statistic` and the p-value with the name `p.value`. To get these from the test object, use the $ operator:

```
test.result$statistic

##        t
## 69.06996

test.result$p.value

## [1] 1.708683e-50
```

Being able to quickly extract key numerical information from a test object is huge. For one thing, it allows you to automate the process of running statistical tests over different datasets or simulations. In Chapter XX, we'll see how you can use loops to do this in a snap.

Next, let's do a two-sample t-test where we compare chicken's weights as a function of their diet. As you can see in the dataset, the chickens were fed one of three diets. Let's compare the weight at time 10 between chickens being fed the first and second diets:

```
x <- subset(ChickWeight, subset = Time == 10 & Diet == 1)$weight
y <- subset(ChickWeight, subset = Time == 10 & Diet == 2)$weight

test.result <- t.test(x, y)
test.result

##
##  Welch Two Sample t-test
##
## data:  x and y
## t = -1.6679, df = 17.234, p-value = 0.1134
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -34.966958   4.072221
## sample estimates:
## mean of x mean of y
##  93.05263 108.50000
```

Looks like we see a test statistic of -1.67 with a resulting p-value of 0.11. According to null-hypothesis test logic, we fail to reject the null hypothesis.

Correlation test

Next we'll cover two-sample correlation tests. Recall that in a correlation test, you are accessing the relationship between two variables on a ratio or interval scale.

To run a correlation test, use the `cor.test(x, y)` function. The test has the following arguments

`cor.test(x, y)`: Conduct a correlation test between two vectors x and y.

```
cor.test(x, y, alternative, method, conf.level)
```

- x, y: The two vectors of data
- alternative: A string indicating the direction of the test. You can use ”t” for two-sided, ”l”, for less than, and ”g” for greater than.
- method: A string indicating which correlation coefficient to test. You can use ”pearson”, ”kendall”, or ”spearman”
- conf.level: The confidence level for the Pearson correlation coefficient.

Let's conduct a correlation test on some data on the number of scars a pirate has and how long he can hold his breathe. I've got the data from 10 pirates below

```
scars <- c(8, 11, 12, 10, 15, 15, 10, 13, 9, 10)
breath <- c(39, 35.9, 37, 39.9, 57.9, 53.1, 33.6, 37.2, 35.4, 38.8)

test.result <- cor.test(x = scars,
                        y = breath
                        )

test.result

##
##  Pearson's product-moment correlation
##
```

```
## data:  scars and breath
## t = 3.3297, df = 8, p-value = 0.01039
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.2547787 0.9404709
## sample estimates:
##       cor
## 0.7621452
```

Looks like we have a positive correlation of 0.76! To see what information we can extract for this test, let's run the command **names()** on the test object:

```
names(test.result)

## [1] "statistic"   "parameter"   "p.value"     "estimate"    "null.value"
## [6] "alternative" "method"      "data.name"   "conf.int"
```

Looks like we've got a lot of information in this test object. As an example, let's look at the confidence interval:

```
test.result$conf.int

## [1] 0.2547787 0.9404709
## attr(,"conf.level")
## [1] 0.95
```

You'll notice that when we tried to access the confidence interval, we got an additional piece of information called **attr(,"conf.level")**. This means that the result of the command **test.result$conf.int** not only contains the bounds of the confidence interval, but also the level of confidence. This is a good thing because the confidence interval only makes sense in terms of the level of confidence used to calculate the interval.

Chi-square test

Next, we'll cover chi-square tests. Remember that a chi-square test tests the relationship between two variables on a nominal scale. For example, are pirates with nipple rings more likely to wear bandannas than those without? To answer this, I collected the following data from 20 pirates

```
nipple.rings <- c(1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1)
bandanas <- c(1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1)
```

Let's use these data and see if there is any relationship between the two variables. To calculate a chi-square test, we'll use the **chisq.test()** function:

```
chisq.test(x, y)
```

chisq.test(x, y): A chi-square test on two nominal vectors x and y.

```
test.result <- chisq.test(nipple.rings, bandanas)

## Warning in chisq.test(nipple.rings, bandanas):  Chi-squared approximation
may be incorrect

test.result

##
##  Pearson's Chi-squared test with Yates' continuity correction
##
## data:  nipple.rings and bandanas
## X-squared = 1.7162, df = 1, p-value = 0.1902
```

Looks like we got a test-statistic of 1.72 and a p-value of 0.19. According to this test, we do not have sufficient data to reject the null hypothesis that there is no relationship between the two variables.

Let's see what other information we can get from the chi-squre test object:

I encourage you to run the `names()` function on statistical objects. You never know what interesting things you'll discover!

```
names(test.result)

## [1] "statistic" "parameter" "p.value"   "method"    "data.name" "observed"
## [7] "expected"  "residuals" "stdres"
```

We've got some interesting new options here. Let's look at the value of `observed` and `expected`

```
test.result$observed

##             bandanas
## nipple.rings 0 1
##            0 6 3
##            1 3 8

test.result$expected

##             bandanas
## nipple.rings    0    1
##            0 4.05 4.95
##            1 4.95 6.05
```

Cool. It looks like R stores a table of the observed frequencies and the expected frequencies under the null-hypothesis. Thanks R!

8: Regression and ANOVA

Chapter Goals

1. Learn about regression
2. ANOVA

The Linear Model

The linear model is easily the most famous and widely used model in all of statistics. Why? Because it can apply to so many interesting research questions where you are trying to predict a variable of interest (the *response* or *dependent variable*) on the basis of one or more other variables (the *predictor* or *independent variables*).

The linear model takes the following form:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ...\beta_n x_n$$

where the x values represent the predictors, while the beta values represent weights.

To use the linear model in R, we use the `lm()` function:

```
lm(y ~ x1 + x2 + ..., data)
```

- y: The name of the response (dependent) variable.
- x1, x2, ... xn: The names of one or more predictor (independent) variables.
- data: The name of the dataframe containing the data.

Let's try an example with some made-up data. For this example, I'll create three independent variables x1, x2 and x3 from normal distributions. I'll then create a dependent variable `y` as a linear function of the three independent variables (with a little error thrown in)[3]. Finally, I'll run the linear model and see if we can recover the true beta values:

[3] If there was no error in the model, then the response variable would be a perfect linear combination of the predictor variables. When this happens, the model will always perfectly fit the data and no stats are necessary (or even possible!)

```
# Step 1: Create a dataframe of predictors

random.data <- data.frame(
              "x1" = rnorm(100, mean = 0, sd = 1),
              "x2" = rnorm(100, mean = 4, sd = 5),
              "x3" = rnorm(100, mean = -2, sd = 2)
              )

# Step 2: Create the DV with beta values 0, 1, 2, 3
random.data$y <- with(random.data, 0 + 1 * x1 + 2 * x2 + 3 * x3)

# Step 3: Add some random noise to the DV
```

```
random.data$y <- random.data$y + rnorm(100, mean = 0, sd = 1)

# Step 4: Run the model then print the result
result <- lm(y ~ x1 + x2 + x3, data = random.data)
result

##
## Call:
## lm(formula = y ~ x1 + x2 + x3, data = random.data)
##
## Coefficients:
## (Intercept)           x1           x2           x3
##      -0.214        1.076        2.026        2.966
```

Looks like our estimated beta values of −0.214, 1.0762, 2.0262 and 2.9655 are pretty close to the true values of 0, 1, 2, 3! This means the model did a pretty good job of estimating the true beta values from the (noisy) data.

We can get lots of other information from the linear model object. Here are three of them (to see all of them, run **`names(result)`**):

- **`coefficients`**: A vector of the estimated beta values
- **`residuals`**: A vector of the differences between the true response values and the fitted response values.
- **`fitted.values`**: A vector of the fitted values.

These attributes let us easily calculate some interesting model diagnostics. For example, let's see how far the model fits are on average from the true values:

```
abs.resid <- abs(result$residuals) # Calculate the absolute value of the resid-
uals
mean(abs.resid) # Calculate the mean

## [1] 0.7216105
```

```
plot(x = result$fitted.values,
     y = result$residuals,
     main = "Model Diagnostics")
```

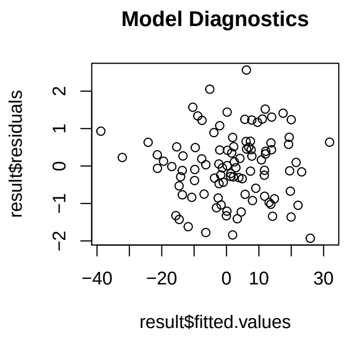


Figure 10: We see no evidence for a relationship between predicted values and residuals. This is good!

So it looks like, on average, our model fits are 0.72 off from the true values (This value is driven by the standard deviation in errors, which we set to 1).

The linear model assumes that there should be no relationship between the predicted values and the distribution of errors. We can easily check this with a scatterplot with the predicted values on the x-axis, and residuals on the y-axis(see Figure 10 in the margin).

ANOVA

Once you've calculated a regression object, you can easily create an *ANOVA* table based on the regression analysis using the **`anova()`** function.

`anova(mod)`

To use the **`anova()`** function, you apply it to an existing linear model object. For example, let's apply it to the previous model:

```
anova(result)

## Analysis of Variance Table
##
## Response: y
##           Df  Sum Sq Mean Sq  F value    Pr(>F)
## x1         1   252.0   252.0   296.32 < 2.2e-16 ***
## x2         1 12389.6 12389.6 14569.92 < 2.2e-16 ***
## x3         1  3649.6  3649.6  4291.78 < 2.2e-16 ***
## Residuals 96    81.6     0.9
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Looks like we have significant effects for all predictors

Generalized Linear Model (GLM)

In the Generalized Linear Model (GLM), we take the original linear model, but apply a link function that wraps around the linear combination of predictors

$$\texttt{glm(y} \sim \texttt{x1 + x2 + ..., family, data)}$$

The key new argument in `glm()` compared to `lm()` is the `family` argument. This argument tells R which link function to use. To see more information about the families, look at help under `?family`.

```r
# Logit
logit.fun <- function(x) {1 / (1 + exp(-x))}

curve(logit.fun,
      from = -3,
      to = 3,
      lwd = 2,
      main = "Logit",
      ylab = "p(y = 1)",
      xlab = expres-
sion("b_{0} + b_{1}x_{1} + b_{2}x_{2} + ... b_{n}x_{n}")
      )

abline(h = .5, lty = 2)
abline(v = 0, lty = 1)
```

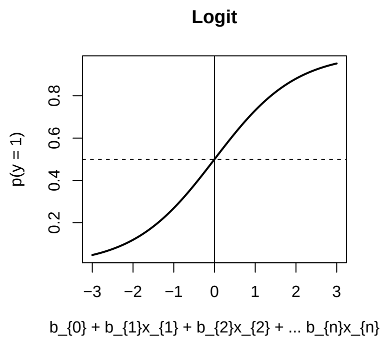


Figure 11: The logit function used in binary logistic regression

Binary Logistic Regression

Probably the most common non-Normal family you will use is binomial which corresponds to binary logistic regression. In binary logistic regression, we predict a binary outcome variable (containing 0s and 1s) as the logit transformation of a linear combination of a set of predictors. Formally:

$$p(y = 1) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots \beta_n x_n)}}$$

To conduct binary logistic regression, we use the `family = "binomial"` argument. Let's try an example with some more made-up data. We'll set the true beta values to 0, 1, -2 and 3

```r
# Step 1: Create a dataframe of predictors
random.data <- data.frame(
             "x1" = rnorm(500, mean = 0, sd = 2),
             "x2" = rnorm(500, mean = 4, sd = 2),
             "x3" = rnorm(500, mean = -2, sd = 2)
             )

# Step 2: Create the DV with beta values 0, 1, -2, 3
random.data$y <- with(random.data, 0 + 1 * x1 - 2 * x2 + 0 * x3)

# Step 3: Add some random noise to the DV
random.data$y <- random.data$y + rnorm(500, mean = 0, sd = 1)

# Step 4: Convert to probability
random.data$y.prob <- with(random.data, 1 / (1 + exp(-y)))

# Step 5: Create binary response
random.data$y.bin <- round(random.data$y.prob, 0)

# Step 4: Run the model then print the result
result <- glm(y.bin ~ x1 + x2 + x3,
              data = random.data, family = "binomial")

## Warning:  glm.fit:  fitted probabilities numerically 0 or 1 occurred

result
```

```
##
## Call:  glm(formula = y.bin ~ x1 + x2 + x3, family = "binomial", data = random.data)
##
## Coefficients:
## (Intercept)           x1           x2           x3
##      0.5074       1.7908      -3.3381       0.3341
##
## Degrees of Freedom: 499 Total (i.e. Null);  496 Residual
## Null Deviance:     204.4
## Residual Deviance: 44.58  AIC: 52.58

summary(result)

##
## Call:
## glm(formula = y.bin ~ x1 + x2 + x3, family = "binomial", data = random.data)
##
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -2.14494  -0.01383  -0.00133  -0.00009   2.10477
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)   0.5074     0.6808   0.745    0.456
## x1            1.7908     0.4244   4.220 2.44e-05 ***
## x2           -3.3381     0.7147  -4.670 3.00e-06 ***
## x3            0.3341     0.2160   1.546    0.122
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 204.363  on 499  degrees of freedom
## Residual deviance:  44.581  on 496  degrees of freedom
## AIC: 52.581
##
## Number of Fisher Scoring iterations: 10
```

R Reference Card

R Reference Card

by Tom Short, EPRI PEAC, tshort@epri-peac.com 2004-11-07

Getting help

Most R functions have online documentation.

`help(topic)` documentation on `topic`
`?topic` id.
`help.search("topic")` search the help system
`apropos("topic")` the names of all objects in the search list matching the regular expression "topic"
`help.start()` start the HTML version of help
`str(a)` display the internal *str*ucture of an R object
`summary(a)` gives a "summary" of `a`, usually a statistical summary but it is *generic* meaning it has different operations for different classes of `a`
`ls()` show objects in the search path; specify `pat="pat"` to search on a pattern
`ls.str()` str() for each variable in the search path
`dir()` show files in the current directory
`methods(a)` shows S3 methods of `a`
`methods(class=class(a))` lists all the methods to handle objects of class a

Input and output

`load()` load the datasets written with `save`
`data(x)` loads specified data sets
`library(x)` load add-on packages
`read.table(file)` reads a file in table format and creates a data frame from it; the default separator `sep=""` is any whitespace; use `header=TRUE` to read the first line as a header of column names; use `as.is=TRUE` to prevent character vectors from being converted to factors; use `comment.char=""` to prevent `"#"` from being interpreted as a comment; use `skip=n` to skip `n` lines before reading data; see the help for options on row naming, NA treatment, and others
`read.csv("filename",header=TRUE)` id. but with defaults set for reading comma-delimited files
`read.delim("filename",header=TRUE)` id. but with defaults set for reading tab-delimited files
`read.fwf(file,widths,header=FALSE,sep="",as.is=FALSE)` read a table of *f*ixed *w*idth *f*ormatted data into a 'data.frame'; `widths` is an integer vector, giving the widths of the fixed-width fields
`save(file,...)` saves the specified objects (...) in the XDR platform-independent binary format
`save.image(file)` saves all objects
`cat(..., file="", sep=" ")` prints the arguments after coercing to character; `sep` is the character separator between arguments
`print(a, ...)` prints its arguments; generic, meaning it can have different methods for different objects
`format(x,...)` format an R object for pretty printing
`write.table(x,file="",row.names=TRUE,col.names=TRUE, sep=" ")` prints `x` after converting to a data frame; if `quote` is `TRUE`, character or factor columns are surrounded by quotes (`"`); `sep` is the field separator; `eol` is the end-of-line separator; `na` is the string for missing values; use `col.names=NA` to add a blank column header to get the column headers aligned correctly for spreadsheet input
`sink(file)` output to `file`, until `sink()`

Most of the I/O functions have a `file` argument. This can often be a character string naming a file or a connection. `file=""` means the standard input or output. Connections can include files, pipes, zipped files, and R variables.

On windows, the file connection can also be used with `description = "clipboard"`. To read a table copied from Excel, use
`x <- read.delim("clipboard")`
To write a table to the clipboard for Excel, use
`write.table(x,"clipboard",sep="\t",col.names=NA)`
For database interaction, see packages `RODBC`, `DBI`, `RMySQL`, `RPgSQL`, and `ROracle`. See packages `XML`, `hdf5`, `netCDF` for reading other file formats.

Data creation

`c(...)` generic function to combine arguments with the default forming a vector; with `recursive=TRUE` descends through lists combining all elements into one vector
`from:to` generates a sequence; ":" has operator priority; 1:4 + 1 is "2,3,4,5"
`seq(from,to)` generates a sequence `by=` specifies increment; `length=` specifies desired length
`seq(along=x)` generates `1, 2, ..., length(along)`; useful for `for` loops
`rep(x,times)` replicate `x times`; use `each=` to repeat "each" element of `x each` times; `rep(c(1,2,3),2)` is 1 2 3 1 2 3; `rep(c(1,2,3),each=2)` is 1 1 2 2 3 3
`data.frame(...)` create a data frame of the named or unnamed arguments; `data.frame(v=1:4,ch=c("a","B","c","d"),n=10)`; shorter vectors are recycled to the length of the longest
`list(...)` create a list of the named or unnamed arguments; `list(a=c(1,2),b="hi",c=3i);`
`array(x,dim=)` array with data `x`; specify dimensions like `dim=c(3,4,2)`; elements of `x` recycle if `x` is not long enough
`matrix(x,nrow=,ncol=)` matrix; elements of `x` recycle
`factor(x,levels=)` encodes a vector `x` as a factor
`gl(n,k,length=n*k,labels=1:n)` generate levels (factors) by specifying the pattern of their levels; `k` is the number of levels, and `n` is the number of replications
`expand.grid()` a data frame from all combinations of the supplied vectors or factors
`rbind(...)` combine arguments by rows for matrices, data frames, and others
`cbind(...)` id. by columns

Slicing and extracting data

Indexing vectors

`x[n]`	`n`th element
`x[-n]`	all *but* the `n`th element
`x[1:n]`	first `n` elements
`x[-(1:n)]`	elements from `n+1` to the end
`x[c(1,4,2)]`	specific elements
`x["name"]`	element named `"name"`
`x[x > 3]`	all elements greater than 3
`x[x > 3 & x < 5]`	all elements between 3 and 5
`x[x %in% c("a","and","the")]`	elements in the given set

Indexing lists

`x[n]`	list with elements `n`
`x[[n]]`	`n`th element of the list
`x[["name"]]`	element of the list named `"name"`
`x$name`	id.

Indexing matrices

`x[i,j]`	element at row `i`, column `j`
`x[i,]`	row `i`
`x[,j]`	column `j`
`x[,c(1,3)]`	columns 1 and 3
`x["name",]`	row named `"name"`

Indexing data frames (matrix indexing plus the following)

`x[["name"]]`	column named `"name"`
`x$name`	id.

Variable conversion

`as.array(x), as.data.frame(x), as.numeric(x), as.logical(x), as.complex(x), as.character(x), ...` convert type; for a complete list, use `methods(as)`

Variable information

`is.na(x), is.null(x), is.array(x), is.data.frame(x), is.numeric(x), is.complex(x), is.character(x), ...` test for type; for a complete list, use `methods(is)`
`length(x)` number of elements in `x`
`dim(x)` Retrieve or set the dimension of an object; `dim(x) <- c(3,2)`
`dimnames(x)` Retrieve or set the dimension names of an object
`nrow(x)` number of rows; `NROW(x)` is the same but treats a vector as a one-row matrix
`ncol(x)` and **`NCOL(x)`** id. for columns
`class(x)` get or set the class of `x`; `class(x) <- "myclass"`
`unclass(x)` remove the class attribute of `x`
`attr(x,which)` get or set the attribute `which` of `x`
`attributes(obj)` get or set the list of attributes of `obj`

Data selection and manipulation

`which.max(x)` returns the index of the greatest element of **`x`**
`which.min(x)` returns the index of the smallest element of **`x`**
`rev(x)` reverses the elements of `x`
`sort(x)` sorts the elements of `x` in increasing order; to sort in decreasing order: `rev(sort(x))`
`cut(x,breaks)` divides `x` into intervals (factors); `breaks` is the number of cut intervals or a vector of cut points
`match(x, y)` returns a vector of the same length than `x` with the elements of `x` which are in `y` (`NA` otherwise)
`which(x == a)` returns a vector of the indices of `x` if the comparison operation is true (`TRUE`), in this example the values of `i` for which `x[i] == a` (the argument of this function must be a variable of mode logical)
`choose(n, k)` computes the combinations of k events among n repetitions $= n!/[(n-k)!k!]$
`na.omit(x)` suppresses the observations with missing data (`NA`) (suppresses the corresponding line if `x` is a matrix or a data frame)
`na.fail(x)` returns an error message if `x` contains at least one `NA`

Index