STATISTICAL HORIZONS

Statistics With R

Andrew Miles, Ph.D.

Upcoming Seminar:

June 6-7, 2019,Chicago, Illinois

Statistics with R

Andrew Miles
University of Toronto
www.andrewamiles.com

Introduction

R is a freely available program for doing statistics. It is minimalist in the sense that the basic installation is, well, basic, but you can add on countless packages containing functions to perform just about any task you wish. This make R extremely flexible.

This course is designed as an introduction to R for those who are looking to use R for applied statistical tasks. However, there is no way that we can cover all the possible uses of R in one short course, so an important theme will be helping you understand how R works so that you can use it for whatever kind of analyses you need to run. For that reason, I will typically focus on showing you how to accomplish tasks using basic R functions, which will help solidify the fundamentals of the R language in your mind. My goal is to get you read to be able to use R independently, so I'll focus on practical issues like finding what you need, interpreting output, and getting help. After this course, you should be well-equipped to tailor R to the sort of work you do.

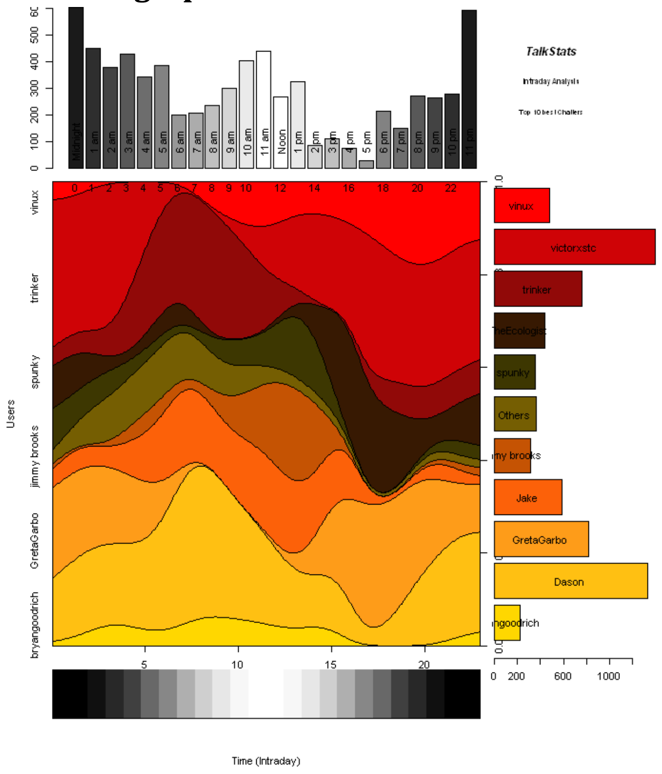
This document contains the R code used in the course, along with the associated output. However, I strongly recommend working through this material as I do it during the course, rather than simply observing. Learning R is largely a matter of acquiring new skills rather than learning new concepts, and there simply is no better way to acquire skills than rolling up your sleeves and doing the work. You can think of it this way: if you ever want to use R, you will have to spend some familiarizing yourself with the code anyhow, so why not do it now, when you have an instructor around to help you through any hang-ups?

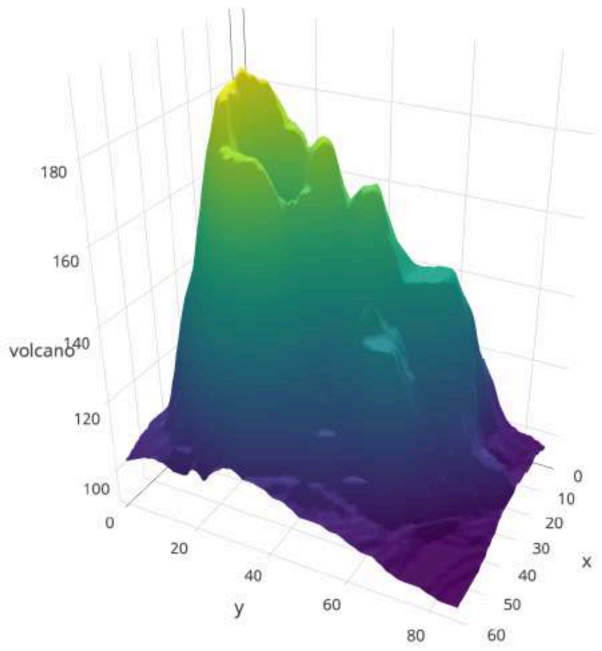
Why Use R?

- **flexibility** - it is a programming language as much as a statistical package, which means it can be adapted to do whatever it is that you need
  - it can be adapted to your preferred programming style (e.g., loops, vector operations)
  - flexibility means it is well-suited for analyses in many disciplines
  - flexibility also often means more complex coding
- **open source**
  - it often advances quickly, and contains cutting-edge techniques

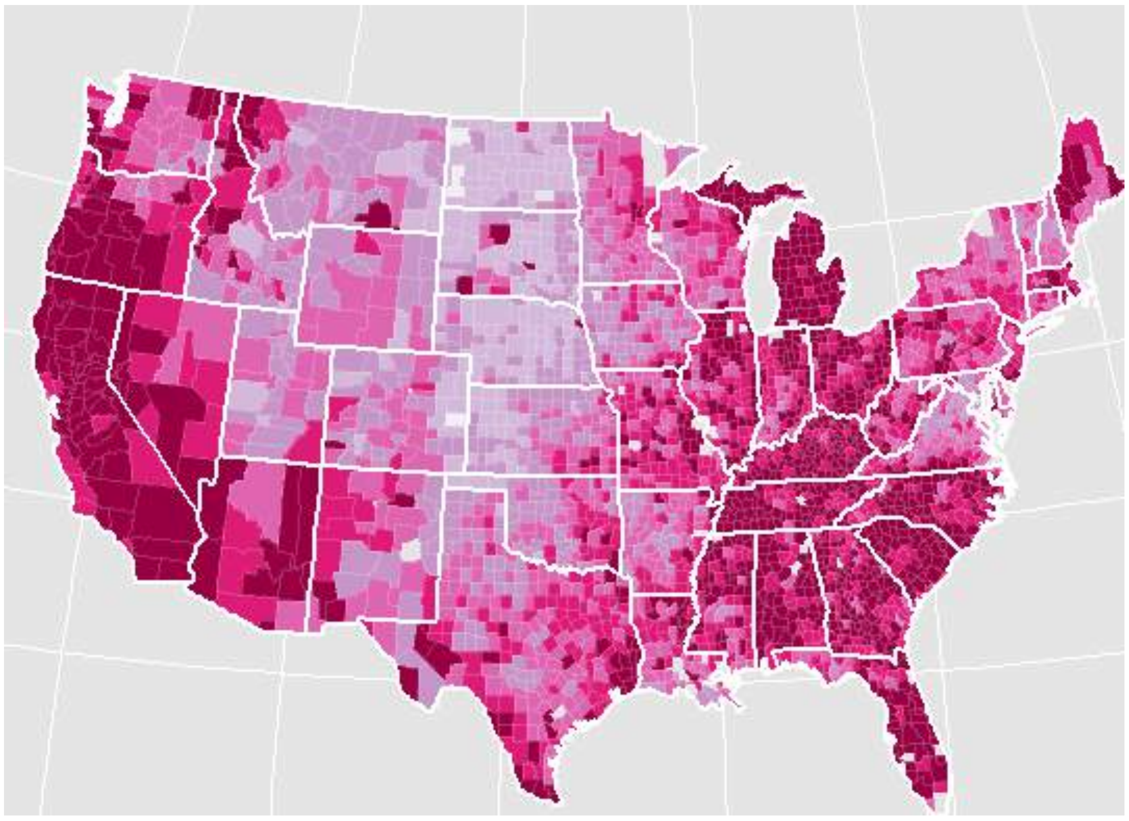
- there is often duplication, so you can find several functions that do the same thing and pick the one that most suits your preferences or the norms of your discipline
- the downside is there is not dedicated team of programmers making sure everything works well with everything else, or making sure help files are as user-friendly as they could be

- **excellent graphics**





volcano
180
160
140
120
100
0
20
40
60
80
y
0
10
20
30
40
50
60
x



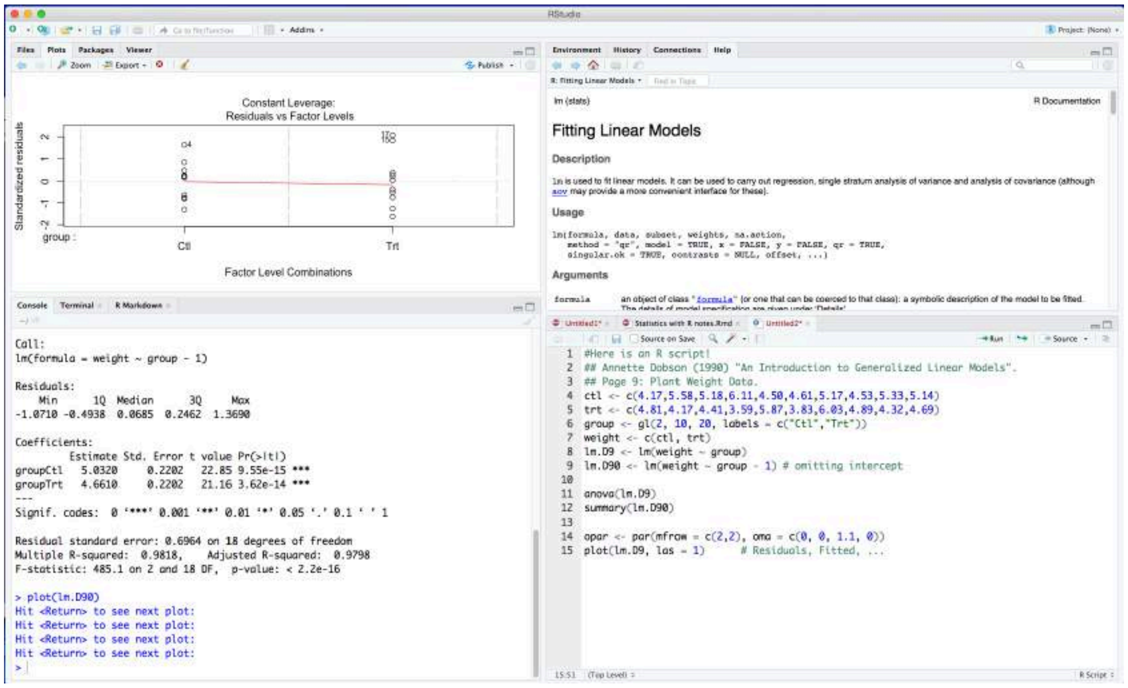
Outline of Course

- Basic Structure of R
- Data Basics
  - Basic Data Structures in R
  - Examining Objects
  - Importing and Exporting Data
  - Merging Data
  - Sorting Data
  - Cleaning Up the Workspace
  - Testing and Coercing Objects
- Interlude: Functions, Help Files, and Nesting
- Examining Data
  - Attaching Objects and R's Search Path
  - Descriptive Statistics
  - Tables for Categorical Variables
  - Exploratory Data Plots
- Data Coding
  - Logical Operators
  - Recoding Data
  - Coding Missing Values
- Bivariate Analyses
  - Analyses of Continuous Variables
  - Analyses of Categorical Variables
- Multivariate Analyses
  - ANOVA
  - Linear Models
  - Generalized Linear Models
  - Survival Analysis
- Additional Topics
  - Model Selection
  - Control Structures
  - Writing Functions
  - Exporting Results
- Addendum: Getting Help

Software

You can use R through the interface that comes with a basic installation, or with one of many front-end applications. We will use RStudio, a popular front-end that has a number of nice features for organizing your workspace and simplifying basic R tasks. However, I may occasionally show how to perform those tasks in R directly, both to aid understanding and in case you ever want to use something other than RStudio.

RStudio is laid out in four quadrants, as shown below.


RStudio
Project: (None)
Files Plots Packages Viewer
Zoom Export Publish
Constant Leverage:
Residuals vs Factor Levels
Standardized residuals
group :
Ctl
Trt
Factor Level Combinations
Environment History Connections Help
R: Fitting Linear Models
Find in Topic
Console Terminal R Markdown
Untitled1* Statistics with R notes.Rmd Untitled2*
Source on Save
Run
Source
15:51 (Top Level)
R Script

lm (stats) R Documentation

Fitting Linear Models

Description

lm is used to fit linear models. It can be used to carry out regression, single stratum analysis of variance and analysis of covariance (although aov may provide a more convenient interface for these).

Usage

```
lm(formula, data, subset, weights, na.action,
   method = "qr", model = TRUE, x = FALSE, y = FALSE, qr = TRUE,
   singular.ok = TRUE, contrasts = NULL, offset, ...)
```

Arguments

formula   an object of class "formula" (or one that can be coerced to that class): a symbolic description of the model to be fitted.

```
Call:
lm(formula = weight ~ group - 1)

Residuals:
    Min      1Q  Median      3Q     Max 
-1.0710 -0.4938  0.0685  0.2462  1.3690 

Coefficients:
         Estimate Std. Error t value Pr(>|t|)    
groupCtl   5.0320     0.2202   22.85 9.55e-15 ***
groupTrt   4.6610     0.2202   21.16 3.62e-14 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.6964 on 18 degrees of freedom
Multiple R-squared:  0.9818,	Adjusted R-squared:  0.9798 
F-statistic: 485.1 on 2 and 18 DF,  p-value: < 2.2e-16

> plot(lm.D90)
Hit <Return> to see next plot:
Hit <Return> to see next plot:
Hit <Return> to see next plot:
Hit <Return> to see next plot:
> 
```

```r
#Here is an R script!
## Annette Dobson (1990) "An Introduction to Generalized Linear Models".
## Page 9: Plant Weight Data.
ctl <- c(4.17,5.58,5.18,6.11,4.50,4.61,5.17,4.53,5.33,5.14)
trt <- c(4.81,4.17,4.41,3.59,5.87,3.83,6.03,4.89,4.32,4.69)
group <- gl(2, 10, 20, labels = c("Ctl","Trt"))
weight <- c(ctl, trt)
lm.D9 <- lm(weight ~ group)
lm.D90 <- lm(weight ~ group - 1) # omitting intercept

anova(lm.D9)
summary(lm.D90)

opar <- par(mfrow = c(2,2), oma = c(0, 0, 1.1, 0))
plot(lm.D9, las = 1)      # Residuals, Fitted, ...
```

Here, the left hand side is dedicated to output. The top left quadrant shows a diagnostic plot from a linear model, while the bottom left shows the output on the R console. On the top right, you have a pane showing an entry from R's help database. The bottom right shows a code file (which in R is called a script). You can see as well that each quadrant has tabs that allow you to click through to different sorts of displays. For example, the top right corner allows you to display help entries, a history of commands that have been executed, and a few other things.

You can customize what is displayed in each quadrant via the View menu: View –> Panes –> Pane Layout.

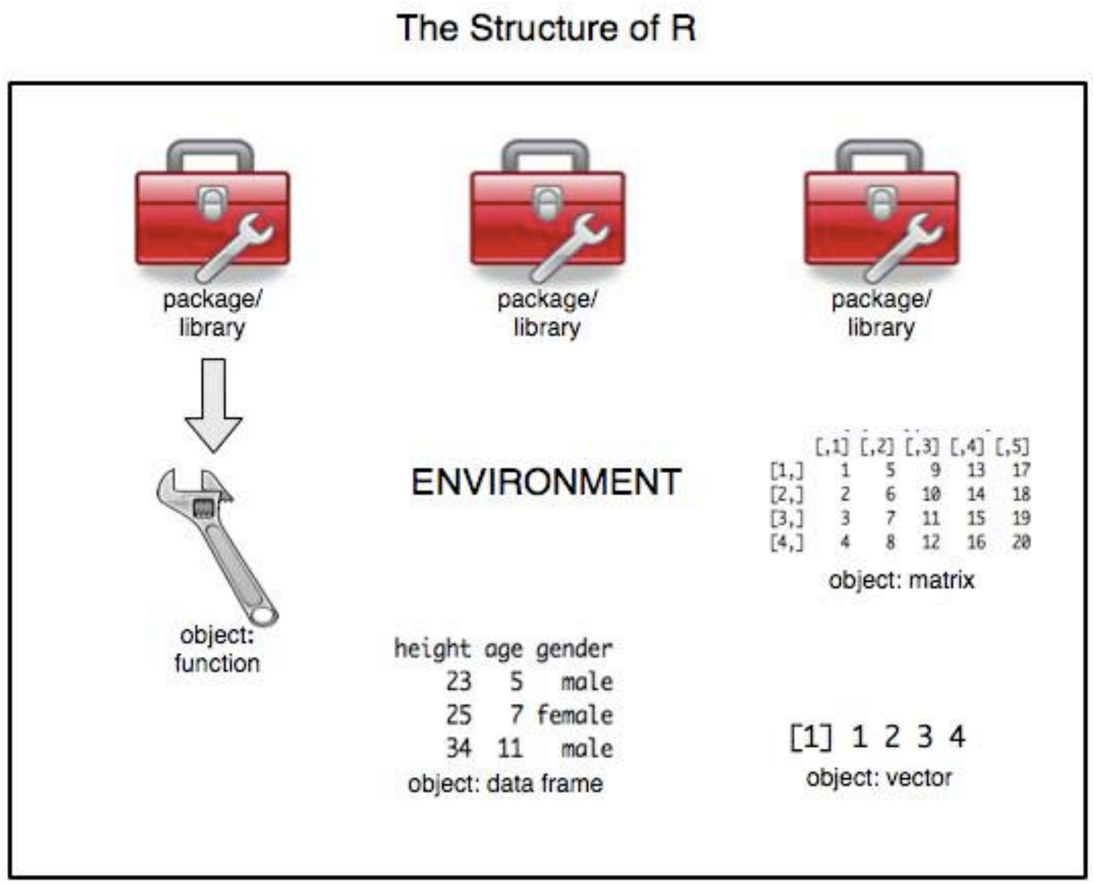
The Basic Structure of R

R is an object-oriented language. That means everything you use - the data structures, the functions, etc. - are stored in objects, and you do things by finding and manipulating those objects. You can think of R like a big room.

You can put just about anything in the room, but you can only use it if

- it is in the room
- you can find it in the room

The diagram below illustrates this idea. The room in which you do most of your work is the global environment. You can think of it as your user workspace.



You can bring in a number of different things into your workspace. Two of the most common are:

packages/libraries – pre-packaged collections of objects, usually functions (tools) that let you perform tasks (e.g., structural equation modeling, multilevel modeling)

objects – things in the room that you work on, or places that you store the things you are working with – these can be of various types

For instance, take a look at the packages that R comes pre-loaded with:

```
search()

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

But, there are no objects in your room to work on.

```
objects()

character(0)
```

The "character(0)" is just R's way of saying something doesn't exist.

Note an important implication of R's structure. As the figure above illustrates, there are a number of different data structures in R. You can have a single vector of numbers, or a matrix or data frame. All of these structures are stored as individual objects. That means you can have many different data structures in your workspace simultaneously, including multiple data sets. This can be extremely useful in many circumstances, but this flexibility comes with a cost - it can sometimes be difficult to know which data structures R is accessing when we give it a command. Consequently, it becomes very important to know how R searches for objects, so we know what we are getting. We'll delve into this issue later on.

Let's turn now to how we get data into R (i.e., getting objects to work on).

Script Files

You can enter commands directly at the prompt in the R Console, but you typically want to do most of your coding work in a script file. This will as this will allow you to save your work, write notes, and run large amounts of code at once. You can create a script file like this:

RStudio: File –> New File –> R Script

R: File –> "New Document"" or "New Script"

You can use the # sign to write notes in an R script file, like so:

```
#this line will not be executed, nor will the command on the following line
#mean(1)
```

```r
#but the line below will be, because it is not commented out
mean(1)

[1] 1
```

You can use as many #'s and spaces as you need to make your code easy to read. You can also include a comment on the same line as a code snippet that you want to execute.

```r
###############
## Example 1 ##
###############

# take the mean of a constant
mean(2)

[1] 2

# the mean of the sum of two constants
mean(2+1)  # this is still just a constant!

[1] 3
```

To run code from a script file, select the code you wish to run and then use the following key combinations:

Mac: Command+Enter

PC: Control+R

RStudio: Control+Enter

Data Basics

Basic Data Structures in R

Vectors

The primary function for creating a vector is the concatenate function c().

```r
vec = c(1,2,3,4,5)
```

Here we created an object called *vec* and stored the numbers 1 through 5 in it. The = tells R to set the value of whatever is on the left to whatever is on the right. The technical name for = is an assignment operator.

We could also have created the same vector in either of these two ways:

```r
vec <- c(1,2,3,4,5)
vec = 1:5
```

In the first example, we use the <- symbol to represent assignment. Some people prefer the <- symbol because it visually looks like an arrow and shows you which way the assignment is operating - that is, the numbers 1 through 5 are being put into an object called *vec*. I don't prefer the <- operator, so I'll use the = operator throughout this course, but feel free to use <- if it makes more sense to you, or brings you more joy. The second example uses the : operator, which is a shortcut in R for generating sequences that increase or decrease by 1.

Let's take a look at our workspace to see what objects we have.

```
objects()

[1] "vec"
```

We only have a single object names *vec*, even though we created *vec* three times. The reason is that each time we assigned values into *vec*, it overwrote the original contents. So we didn't create three objects called *vec* (which would be confusing), but created a single object and then changed its contents three times. Unlike some programs, R does not warn you when you are about to overwrite existing data. This can make coding less cumbersome, but also can lead to erasing valuable data if you aren't careful.

Let's take a look at the contents of *vec*. To do this, type "vec" and press enter.

```
vec

[1] 1 2 3 4 5
```

We see that *vec* contains the numbers 1 through 5.

In R, there is an important difference between **assigning** values to an object, and asking R to **display** the contents of an object. Assignment always uses an assignment operator like = or <-. Displaying the contents simply requires typing the name of the object and pressing enter. To return to the room analogy, assignment is like putting something into a storage crate, while displaying is like peeking inside. The importance of this difference will become apparent in a few minutes.

Now let's see what type of object *vec* is using the class() function.

```
class(vec)

[1] "integer"
```

We see that we have numeric data. All objects in R have a classification, and this determines how they are handled by R's functions. Because it is numeric, *vec* can be used in functions that accept numerical data, but not ones that require, say, text data. c() can also be used to create vectors of non-numerical data.

```
char.vec = c("a", "b", "c", "d", "e")
log.vec = c(TRUE, TRUE, FALSE, TRUE, FALSE)
class(char.vec)

[1] "character"
```

```
class(log.vec)

[1] "logical"
```

Here we have created a vector of character/string data, and a vector of logical data. Note that TRUE and FALSE (all caps) are reserved words in R that represent, well, the logical values of true and false. Note, too, that in both object names I used a dot (.), which in many statistical programs is not allowed. In R, however, dots are fair game.

We can access individual elements of any vector using index subscripts.

```
vec[1]; char.vec[3]; log.vec[5]

[1] 1

[1] "c"

[1] FALSE
```

Here I pulled out the 1st element of *vec*, the 3rd element of *char.vec*, and the 5th element of *log.vec*. I could have written all three variable names on separate lines, but I opted to save space by placing them on the same line separated by semicolons. R treats a semi-colon like a line break.

If I try to access an element using an index that does not exist, I get NA, R's value for missing.

```
vec[6]

[1] NA
```

You can access multiple values by placing a vector of subscripts inside the index brackets.

```
vec[c(1,4,5)]

[1] 1 4 5
```

Another way to access values is to supply a vector of TRUE's and FALSE's. R will return the values marked with TRUE.

```
vec[c(TRUE, FALSE, FALSE, TRUE, TRUE)]

[1] 1 4 5
```

If your TRUE/FALSE values are stored in another object, you can place that object in the brackets instead.

```
vec[log.vec]

[1] 1 2 4
```

R evaluates the value of *log.vec* first, and obtains c(TRUE, TRUE, FALSE, TRUE, FALSE). It then uses that sequence of logical values to figure out which elements of *vec* to return.

We can also request that certain elements be removed by using a negative subscript.

```
vec[-2]

[1] 1 3 4 5
```

Here, we returned the numbers stored in *vec* after removing the second element.

You can name the elements of your vector using names().

```
names(vec)=char.vec
vec

a b c d e
1 2 3 4 5
```

Now *vec* is a more complex object that includes both the data - the numbers 1 through 5 - and the associated names. That is, we have added an attribute - names - to *vec*. You can see the attributes of *vec* using attributes().

```
attributes(vec)

$names
[1] "a" "b" "c" "d" "e"
```

Having names gives us another way to access the individual values in *vec*. Instead of using the index position, we can use the name.

```
vec[c("b", "d")]

b d
2 4
```

Matrices

Another useful data structure is a matrix. When you have existing vectors of the same length, you can create a matrix using cbind(), which stands for column bind, or rbind(), which stands for row bind.

```
cbind(vec, log.vec)

  vec log.vec
a   1       1
b   2       1
c   3       0
d   4       1
e   5       0

rbind(vec, log.vec)

        a b c d e
vec     1 2 3 4 5
log.vec 1 1 0 1 0
```

There are a few things to note.

1. cbind() turns your vectors into columns in the matrix, while rbind() turns your vectors into rows.
2. The names from *vec* were adopted and used as column/row labels. If both *vec* and *log.vec* had had names, the functions would have adopted the names of whichever one was listed first.
3. *log.vec* is a vector of logical values, but in the matrix these were converted to 0's and 1's. Why is that? Well, a matrix is only allowed to have one type of data - all numeric, all character, and so on. When you try to create a matrix out of vectors containing different types of data, R changes vectors in whatever way will preserve the most information. In this case, you can convert TRUE's and FALSE's to numeric values and still have them retain their meaning (1 = TRUE, 0 = FALSE). But you can't convert numbers beyond 0 and 1 to true's and false's without destroying their meaning. Technically, what R did is known as coercion. That sounds pretty hardcore, but really it just means it forced one type of data to become another type.

Now let's go ahead and work with our matrix…

```
objects()

[1] "char.vec" "log.vec"  "vec"
```

Wait a moment, where is our matrix? We forgot to store the results of our command in an object! Remember, whenever we type a command without an assignment operator, it is like we are saying to R: "Hey R, why don't you show me what would happen if, you know, hypothetically speaking, I was to run this command?" Of course, we actually do run the command, but we don't store the results anywhere, because R does not store the results of its commands by default. It is up to us to do that. We can correct our error using an assignment operator.

```
my.matrix = cbind(vec, log.vec)
class(my.matrix)

[1] "matrix"

objects()

[1] "char.vec"  "log.vec"   "my.matrix" "vec"
```

Now we have a matrix stored in an object called *my.matrix*. We could also have created a matrix using the matrix() function, though in practice I typically find this less convenient.

```
my.matrix=matrix(c(vec, log.vec), ncol=2, byrow=FALSE, dimnames=list(names(ve
c), c("vec", "log.vec")))
my.matrix

  vec log.vec
a   1       1
b   2       1
```

```
c   3       0
d   4       1
e   5       0
```

Just like with vectors, we can access data in the matrix using indices/subscripts. Matrix subscripts are of the form:

matrix[row, column]

So if we wanted to display the data in the 1st row and 2nd column, we would type:

```
my.matrix[1,2]

[1] 1
```

If we wanted to access that value and store it in a vector, we could do that like this:

```
new.vec = my.matrix[1,2]
new.vec

[1] 1
```

As with vectors, you can also use names to access values in your matrix.

```
my.matrix["a", "log.vec"]

[1] 1
```

You can access multiple values just as you did with vectors.

```
my.matrix[1:3,1]

a b c
1 2 3

my.matrix[c(4,5), c(TRUE, FALSE)]

d e
4 5
```

By convention, an empty index spot means “all of.” So to see all rows of column 1, you would type:

```
my.matrix[,1]

a b c d e
1 2 3 4 5
```

Data Frames

Data frames are the equivalent of the data sets most often used in other statistical programs. Like matrices, they can contain any number of rows and columns, but can also accommodate a mixture of data types. All the objects used to create a data frame are placed in columns, and must have compatible dimensions.

```
df = data.frame(my.matrix, char.vec, log.vec)
df

  vec log.vec char.vec log.vec.1
a   1       1        a      TRUE
b   2       1        b      TRUE
c   3       0        c     FALSE
d   4       1        d      TRUE
e   5       0        e     FALSE
```

Here, we have created a data frame from our objects *my.matrix*, *char.vec*, and *log.vec*. The matrix has 5 rows, and each of the vectors has 5 elements, so they have compatible dimensions which made the combination possible. We now have two columns of numbers, one column of characters, and one column of logical data. Note that the logical vector was renamed "log.vec.1" because the name "log.vec" was already in use, and names must uniquely identify columns. Importantly, the data frame maintains different data types for different columns.

```
class(df[,1]); class(df[,3]); class(df[,4])

[1] "integer"

[1] "factor"

[1] "logical"
```

The "char.vec" column has been converted from character data to a factor. Factors are R's name for categorical data. By default, the command data.frame() converts all character data to factors, but you can override this by setting the argument stringsAsFactors to FALSE.

```
df = data.frame(my.matrix, char.vec, log.vec, stringsAsFactors = FALSE)
class(df[,"char.vec"])

[1] "character"
```

You can add variables (columns) to data frames using the $ operator.

```
df$gender = c("male", "female", "female", "male", "male")
df

  vec log.vec char.vec log.vec.1 gender
a   1       1        a      TRUE   male
b   2       1        b      TRUE female
c   3       0        c     FALSE female
d   4       1        d      TRUE   male
e   5       0        e     FALSE   male
```

The same operator can be used to access specific variables.

```
df$gender

[1] "male"   "female" "female" "male"   "male"
```

It can also be used to remove variables.

```
df$gender2 = df$gender
df

  vec log.vec char.vec log.vec.1 gender gender2
a   1       1        a      TRUE   male    male
b   2       1        b      TRUE female  female
c   3       0        c     FALSE female  female
d   4       1        d      TRUE   male    male
e   5       0        e     FALSE   male    male

df$gender2 = NULL
df

  vec log.vec char.vec log.vec.1 gender
a   1       1        a      TRUE   male
b   2       1        b      TRUE female
c   3       0        c     FALSE female
d   4       1        d      TRUE   male
e   5       0        e     FALSE   male
```

Arrays

Arrays are multi-dimensional generalizations of matrices. As such, they can only contain one type of data.

```
my.array = array(1:27, dim=c(3,3,3))
my.array

, , 1

     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

, , 2

     [,1] [,2] [,3]
[1,]   10   13   16
[2,]   11   14   17
[3,]   12   15   18

, , 3

     [,1] [,2] [,3]
[1,]   19   22   25
[2,]   20   23   26
[3,]   21   24   27
```

Here we have an array of three dimensions, each containing numeric data. The first and second dimensions create two-dimensional matrices, which are arranged along the third dimension. Use can use indices to access data just like in matrices or data frames, except that you now have three indices to work with. For example, if I want the value stored in the first row of the first column in the second matrix, I would type:

```
my.array[1,1,2]

[1] 10
```

Likewise, if I wanted the entire first column of the second matrix, I would type

```
my.array[,1,2]

[1] 10 11 12
```

I won't spend a lot of time on arrays, as they tend not to come up in most of the basic statistical tasks in R.

Lists

Lists are hands down the most flexible data structure in R. A list is simply a collection of other R objects of any type or size. For example, I can create a list that contains a numeric vector, a matrix, and a data frame.

```
my.list = list(vec, my.matrix, df)
my.list

[[1]]
a b c d e
1 2 3 4 5

[[2]]
  vec log.vec
a   1       1
b   2       1
c   3       0
d   4       1
e   5       0

[[3]]
  vec log.vec char.vec log.vec.1 gender
a   1       1        a      TRUE   male
b   2       1        b      TRUE female
c   3       0        c     FALSE female
d   4       1        d      TRUE   male
e   5       0        e     FALSE   male
```

Here we have a list with three elements. The first one is the vector *vec*, the second is the matrix *my.matrix*, and the third is the data frame *df*. The remarkable thing is that all of