

Introduction to R

Jarek Szlichta

<http://data.science.uoit.ca/>

Intro

- R is a programming language and software environment for statistical computing and graphics
 - Note: Many features of R now being incorporated into Python through packages like Pandas, SciPy, scikit, and others
 - Main data type is "data frame", similar to relational table
 - Play with R online or install environment:
 - http://www.tutorialspoint.com/execute_r_online.php

Input

Assignment

- Assignment is specified with the “<-” symbol
- A list is specified with the c command
 - c stands for “combine”
 - Another term used to describe the list of numbers is to call it a “vector”
- We can create a new variable, called “bubba” which will contain the numbers 3, 5, 7, and 9:

```
> bubba <- c(3, 5, 7, 9)
```

Access

- To see what numbers are included in bubba type “bubba” and press the enter key:

```
> bubba [1]
3 5 7 9
```

- If you wish to work with one of the numbers you can get access to it using the variable and then square brackets indicating which number:
 - Notice that the first entry is referred to as the number 1 entry

```
> bubba[2]
[1] 5
> bubba[0]
numeric(0)
```

Reading a CSV File

- It is rare to have just a few data points that you do not mind typing in at the prompt
- It is much more common to have a lot of data points with complicated relationships
- Here we will examine how to read a data set from a file using the `read.csv` function
 - We assume that the data file is in the format called “comma separated values” (csv)

CSV File

- We assume that the very first row contains a list of labels
- The idea is that the labels in the top row are used to refer to the different columns of values

trial	mass	velocity
A	10	12
A	11	14
B	5	8
B	6	10
A	10.5	13
B	7	11

Reading Data File

- The command to read the data file is *read.csv*
 - The first argument is the name of file
 - The second argument indicates whether or not the first row is a set of labels
 - The third argument indicates that there is a comma between each number of each line

Reading Data File

```
> heisenberg <- read.csv(file="simple.csv", head=TRUE, sep=",")
> heisenberg
trial mass velocity
1 A 10.0 12
2 A 11.0 14
3 B 5.0 8
4 B 6.0 10
5 A 10.5 13
6 B 7.0 11
> summary(heisenberg)
trial mass velocity
A:3 Min. : 5.00 Min. : 8.00
B:3 1st Qu.: 6.25 1st Qu.:10.25
     Median : 8.50 Median :11.50
     Mean : 8.25 Mean :11.33
     3rd Qu.:10.38 3rd Qu.:12.75
     Max. :11.00 Max. :14.00
```

Accessing Columns

- The variable “heisenberg” contains the three columns of data
 - Each column is assigned a name based on the header (the first line in the file)
 - You can now access each individual column using a “\$” to separate the two names:

```
> heisenberg$trial
```

```
[1] A A B B A B
```

```
Levels: A B
```

```
> heisenberg$mass
```

```
[1] 10.0 11.0 5.0 6.0 10.5 7.0
```

```
> heisenberg$velocity
```

```
[1] 12 14 8 10 13 11
```

Column Names

- The data can be read into a variable called “tree” in using the read.csv command:

```
> tree <- read.csv(file="trees91.csv",header=TRUE,sep=",");
```

- If you are not sure what columns are contained in the variable you can use the names command:

```
> names(tree)
[1] "C"      "N"      "CHBR"   "REP"    "LFBM"   "STBM"   "RTBM"   "LFNCC"
[9] "STNCC"  "RTNCC"  "LFBCC"  "STBCC"  "RTBCC"  "LFCACC" "STCACC" "RTCACC"
[17] "LFKCC"  "STKCC"  "RTKCC"  "LFMGCC" "STMGCC" "RTMGCC" "LFPCC"  "STPCC"
[25] "RTPCC"  "LFSCC"  "STSCC"  "RTSCC"
```

Data Types

Numbers

- The most basic way to store a number is to make an assignment of a single number:
 - The “<-” tells R to take the number to the right of the symbol and store it in a variable whose name is given on the left

```
> a <- 3
```

- This allows you to do all sorts of basic operations and save the numbers:

```
> b <- sqrt(a*a+3)
```

```
> b
```

```
[1] 3.464102
```

Numbers

- If you want to get a list of the variables that you have defined you can use `ls` command:

```
> ls()  
[1] "a" "b"
```

- To initialize a list of numbers the *numeric* command can be used

```
> a <- numeric(10)  
> a  
[1] 0 0 0 0 0 0 0 0 0 0
```

- If you wish to determine the data type used for a variable the *typeof* command:

```
> typeof(a) [1] "double"
```

Strings

- A string is specified by using quotes.
- Both single and double quotes will work:

```
> a <- "hello"
> a
[1] "hello"
> b <- c("hello", "there")
> b [1] "hello" "there"
> b[1]
[1] "hello"
```

- The name of the type given to strings is *character*

Factors

- Factors express different levels of some variable
- For example, column labeled “C,” is a factor
 - Each trees was grown in an environment with one of four different possible levels of carbon dioxide
 - The researchers quite sensibly labeled these four environments as 1, 2, 3, and 4.

Factors

- Unfortunately, R cannot determine that these are factors and must assume that they are regular numbers
- There is a way to tell R to treat the “C” column as a set of factors
 - You specify that a variable is a factor using the factor command
 - In the following example we convert tree\$C into a factor.

Factors Example

```
> tree$C
[1] 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3
[39] 3 3 3 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
> summary(tree$C)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 1.000  2.000  2.000  2.519  3.000  4.000
> tree$C <- factor(tree$C)
> tree$C
[1] 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3
[39] 3 3 3 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
Levels: 1 2 3 4
> summary(tree$C)
 1  2  3  4
 8 23 10 13
> levels(tree$C)
[1] "1" "2" "3" "4"
```

Data Frames

- This is a way to take many vectors of different types and store them in the same variable.
 - The vectors can be of all different types
 - For example, a data frame may contain many lists, and each list might be a list of factors, strings, or numbers

Example how to Create Data Frame

```
> a <- c(1,2,3,4)
> b <- c(2,4,6,8)
> levels <- factor(c("A","B","A","B"))
> bubba <- data.frame(first=a,
                      second=b,
                      f=levels)

> bubba
  first second f
1     1     2 A
2     2     4 B
3     3     6 A
4     4     8 B

> summary(bubba)
      first      second      f
Min.   :1.00  Min.   :2.0  A:2
1st Qu.:1.75  1st Qu.:3.5  B:2
Median :2.50  Median :5.0
Mean   :2.50  Mean   :5.0
3rd Qu.:3.25  3rd Qu.:6.5
Max.   :4.00  Max.   :8.0

> bubba$first
[1] 1 2 3 4
> bubba$second
[1] 2 4 6 8
> bubba$f
[1] A B A B
Levels: A B
```

Logical

- Another important data type is the logical type
- There are two predefined variables, *TRUE* and *FALSE*:

```
> a = TRUE
> typeof(a)
[1] "logical"
> b = FALSE
> typeof(b)
[1] "logical"
```

Standard Logical Operators

<	less than
>	great than
<=	less than or equal
>=	greater than or equal
==	equal to
!=	not equal to
	entry wise or
	or
!	not
&	entry wise and
&&	and
xor(a,b)	exclusive or

```
> a = c(TRUE,FALSE)
> b = c(FALSE,FALSE)
> a|b
[1] TRUE FALSE
> a||b
[1] TRUE
> xor(a,b)
[1] TRUE FALSE
```

```
> a = c(1,2,3)
> is.numeric(a)
[1] TRUE
> is.factor(a)
[1] FALSE
```

One Way Tables

- One way to create a table is with the table command

```
> a <- factor(c("A", "A", "B", "A", "B", "B", "C", "A", "C"))
> results <- table(a)
> results
a
A B C
4 3 2
```

- Another way is to create a matrix
 - The *byrow=TRUE* option indicates that the numbers are filled in across the rows first, and the *ncols=3* indicates that there are three columns

```
> occur <- matrix(c(4,3,2),ncol=3,byrow=TRUE)
> occur
      [,1] [,2] [,3]
[1,]    4    3    2
```

Two Way Tables

- With table command

```
> a <- c("Sometimes", "Sometimes", "Never", "Always", "Always", "Sometimes", "Sometimes", "Never")
> b <- c("Maybe", "Maybe", "Yes", "Maybe", "Maybe", "No", "Yes", "No")
> results <- table(a,b)
> results
```

a	b		
	Maybe	No	Yes
Always	2	0	0
Never	0	1	1
Sometimes	2	1	1

- With matrix command

```
> sexsmoke <- matrix(c(70,120,65,140), ncol=2, byrow=TRUE)
> rownames(sexsmoke) <- c("male", "female")
> colnames(sexsmoke) <- c("smoke", "nosmoke")
> sexsmoke <- as.table(sexsmoke)
> sexsmoke
```

	smoke	nosmoke
male	70	120
female	65	140

Operations and Numerical Descriptions

Operations

- First, the vector will contain the numbers 1, 2, 3, and 4
 - We then see how to add 5 to each of the numbers, subtract 10 from each of the numbers, multiply each number by 4, and divide each number by 5

```
> a <- c(1,2,3,4)
> a
[1] 1 2 3 4
> a + 5
[1] 6 7 8 9
> a - 10
[1] -9 -8 -7 -6
> a*4
[1] 4 8 12 16
> a/5
[1] 0.2 0.4 0.6 0.8
```

SQRT, EXP and LOG

- If you want to take the square root, find e raised to each number, the logarithm, etc., then the usual commands can be used:

```
> sqrt(a)
[1] 1.000000 1.414214 1.732051 2.000000
> exp(a)
[1] 2.718282 7.389056 20.085537 54.598150
> log(a)
[1] 0.0000000 0.6931472 1.0986123 1.3862944
> exp(log(a))
[1] 1 2 3 4
```

- By combining operations and using parentheses you can make more complicated expressions:

```
> c <- (a + sqrt(a))/(exp(2)+1)
> c
[1] 0.2384058 0.4069842 0.5640743 0.7152175
```

MIN and PMIN

- Using the min command you will get the minimum of all of the numbers
- Using the pmin command you will get the minimum of the numbers at each position

```
> a <- c(1,-2,3,-4)
> b <- c(-1,2,-3,4)
> min(a,b)
[1] -4
> pmin(a,b)
[1] -1 -2 -3 -4
```

Numerical Operations

- The following commands are used to get the mean, median, quantiles, minimum, maximum, variance, and standard deviation of a set of numbers:

```
> mean(tree$LFBM)
[1] 0.7649074
> median(tree$LFBM)
[1] 0.72
> quantile(tree$LFBM)
      0%    25%    50%    75%   100%
0.1300 0.4800 0.7200 1.0075 1.7600
> min(tree$LFBM)
[1] 0.13
> max(tree$LFBM)
[1] 1.76
> var(tree$LFBM)
[1] 0.1429382
> sd(tree$LFBM)
[1] 0.3780717
```

Operations on Vectors

- The *sort* command can sort the given vector in either ascending or descending order:
- The *min* and the *max* commands find the minimum and the maximum numbers in the vector:
- The *sum* command adds up the numbers in the vector:

```
> a = c(2,4,6,3,1,5)
> b = sort(a)
> c = sort(a,decreasing= TRUE)
> a
[1] 2 4 6 3 1 5
> b
[1] 1 2 3 4 5 6
> c
[1] 6 5 4 3 2 1
> min(a)
[1] 1
> max(a)
[1] 6
> sum(a)
[1] 21
```

Plots

Plots

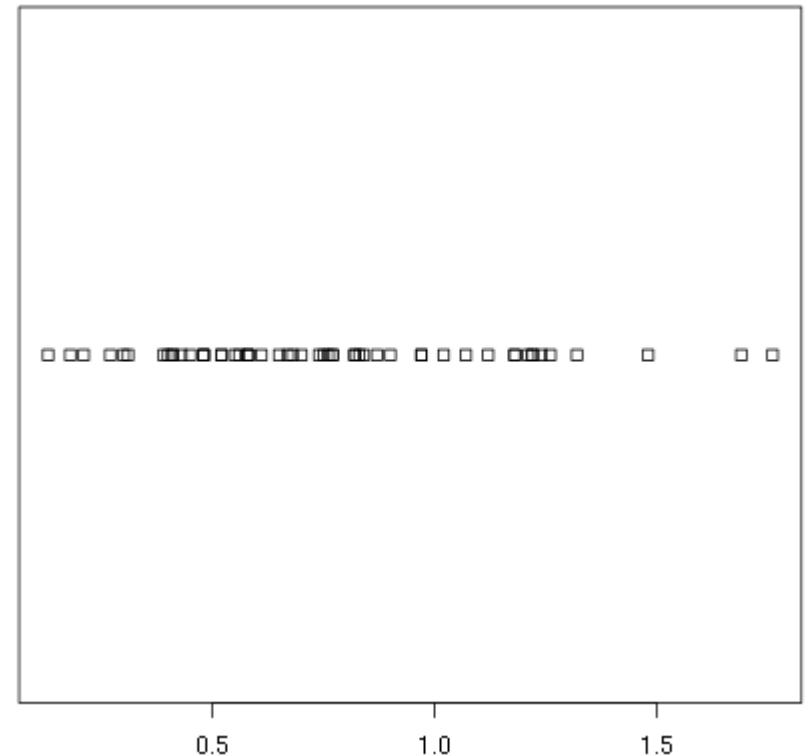
- It is assumed that two different data sets, w1.dat and trees91.csv have been read

```
> w1 <- read.csv(file="w1.dat",sep=";",head=TRUE)
> names(w1)
[1] "vals"
> tree <- read.csv(file="trees91.csv",sep=";",head=TRUE)
> names(tree)
 [1] "C"      "N"      "CHBR"   "REP"    "LFBM"   "STBM"   "RTBM"   "LFNCC"
 [9] "STNCC"  "RTNCC"  "LFBCC"  "STBCC"  "RTBCC"  "LFCACC" "STCACC" "RTCACC"
[17] "LFKCC"  "STKCC"  "RTKCC"  "LFMGCC" "STMGCC" "RTMGCC" "LFPCC"  "STPCC"
[25] "RTPCC"  "LFSCC"  "STSCC"  "RTSCC"
```


Strip Charts

- Strip chart plots the data in order along a line with each data point represented as a box
- To create a strip chart of this data use the strip chart command:

```
> help(stripchart)
> stripchart(w1$vals)
```



Organizing Plots

- If you prefer to see which points are repeated you can specify that repeated points be stacked:

```
> stripchart(w1$vals,method="stack")
```

- A variation on this is to have the boxes moved up and down so that there is more separation between them:

```
> stripchart(w1$vals,method="jitter")
```

- If you do not want the boxes plotting in the horizontal direction you can plot them in the vertical direction:

```
> stripchart(w1$vals,vertical=TRUE)
> stripchart(w1$vals,vertical=TRUE,method="jitter")
```

Annotating Plots

- Annotating Plots can be done within strip chart command

```
> stripchart(w1$vals,method="stack",  
             main='Leaf BioMass in High CO2 Environment',  
             xlab='BioMass of Leaves')
```

- If you have a plot already and want to add a title, you can use the title command:

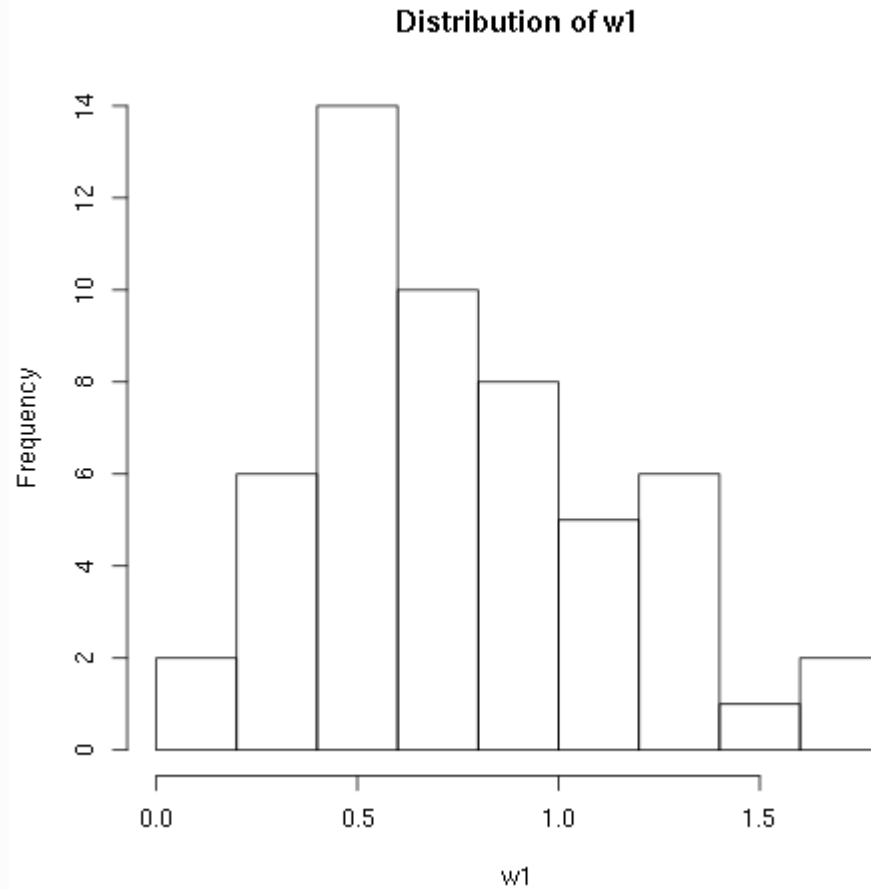
```
> title('Leaf BioMass in High CO2 Environment',xlab='BioMass of Leaves')
```

Histograms

- Histogram plots the frequencies that data appears within certain ranges
- To plot a histogram of the data use the “hist” command:

```
> hist(w1$vals)
> hist(w1$vals,main="Distribution of w1",xlab="w1")
```

Histograms



Intervals

- By default R calculates automatically the intervals to use.
- You can specify the number of breaks to use using the breaks option

```
> hist(w1$vals,breaks=2)
> hist(w1$vals,breaks=4)
> hist(w1$vals,breaks=6)
> hist(w1$vals,breaks=8)
> hist(w1$vals,breaks=12)
>
```

Xlim option

- You can also vary the size of the domain using the xlim option
 - This option takes a vector with two entries in it, the left value and the right value:

```
> hist(w1$vals,breaks=12,xlim=c(0,10))
> hist(w1$vals,breaks=12,xlim=c(-1,2))
> hist(w1$vals,breaks=12,xlim=c(0,2))
> hist(w1$vals,breaks=12,xlim=c(1,1.3))
> hist(w1$vals,breaks=12,xlim=c(0.9,1.3))
>
```

Adding Titles and Labels

- The options for adding titles and labels are exactly the same as for strip charts

- One way is within the hist command itself:

```
> hist(w1$vals,  
      main='Leaf BioMass in High CO2 Environment',  
      xlab='BioMass of Leaves')
```

- If you have a plot already and want to change or add a title, you can use the title command:

- Note that this simply adds the title and labels and will write over the top of any titles or labels you already have.

```
> title('Leaf BioMass in High CO2 Environment',xlab='BioMass of Leaves')
```


Multiple Plots

- It is not uncommon to add other kinds of plots to a histogram
 - For example, you might want to have a histogram with the strip chart drawn across the top
 - The addition of the strip chart might give you a better idea of the density of the data

```
> hist(w1$vals,main='Leaf BioMass in High CO2 Environment',xlab='BioMass of Leaves',ylim=c(0,16))  
> stripchart(w1$vals,add=TRUE,at=15.5)
```

Scatter Plots

- A scatter plot provides a graphical view of the relationship between two sets of numbers
 - we look at the relationship between the stem biomass (“tree\$STBM”) and the leaf biomass (“tree\$LFBM”)
 - The command to plot each pair of points as an x-coordinate and a y-coordinate is “plot:”

```
> plot(tree$STBM,tree$LFBM)
```

Correlation

- It appears that there is a strong positive association between the biomass in the stems of a tree and the leaves of the tree
 - It appears to be a linear relationship
 - The correlation between these two sets of observations is quite high:

```
> cor(tree$STBM,tree$LFBM)
[1] 0.911595
```

Annotations

- You should always annotate your graphs!
 - The title and labels can be specified in exactly the same way as with the other plotting commands:

```
> plot(tree$STBM,tree$LFBM,  
       main="Relationship Between Stem and Leaf Biomass",  
       xlab="Stem Biomass",  
       ylab="Leaf Biomass")
```

Indexing Vectors

Indexing Into Vectors

- Given a vector of data one common task is to isolate particular entries or censor items that meet some criteria
- Here it will be shown how to use R's indexing notation to pick out specific items within a vector
 - Indexing With Logicals
 - Not Available or Missing Values
 - Indices With Logical Expression

Indexing With Logicals

- The first step is to define a vector of data, and the second step is to define a vector made up of logical values
 - When the vector of logical values is used for the index into the vector of data values only the items corresponding to the variables that evaluate to *TRUE* are returned:

```
> a <- c(1,2,3,4,5)
> b <- c(TRUE,FALSE,FALSE,TRUE,FALSE)
> a[b]
[1] 1 4
> max(a[b])
[1] 4
> sum(a[b])
[1] 5
```

Not Available (Missing Values)

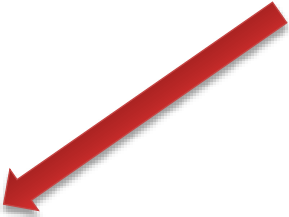
- One common problem is data entries that are marked *NA* (not available)
 - There is a predefined variable called *NA* that can be used to indicate missing information
 - The problem with this is that some functions throw an error if one of the entries in the data is *NA*

```
> a <- c(1,2,3,4,NA)
> a
[1] 1 2 3 4 NA
> sum(a)
[1] NA
```


Special Options

- Some functions allow you to ignore the missing values through special options:

```
> a <- c(1,2,3,4,NA)
> a
[1] 1 2 3 4 NA
> sum(a)
[1] NA
> sum(a,na.rm=TRUE)
[1] 10
```



is.na Function

- The *is.na* function can be used to determine which items are not available
- The logical “not” operator in R is the **!** Symbol
 - When used with the indexing notation the items within a vector that are *NA* can be easily removed:

```
> a <- c(1,2,3,4,NA)
> is.na(a)
[1] FALSE FALSE FALSE FALSE TRUE
> !is.na(a)
[1] TRUE TRUE TRUE TRUE FALSE
> a[!is.na(a)]
[1] 1 2 3 4
> b <- a[!is.na(a)]
> b
[1] 1 2 3 4
```

Indices With Logical Expression

- Any logical expression can be used as an index
 - For example, you can remove or focus on entries that match specific criteria:

```
> a = c(6,2,5,3,8,2)
> a
[1] 6 2 5 3 8 2
> b = a[a<6]
> b
[1] 2 5 3 2
```

Linear Regression

Linear Least Squares Regression

- Here we look at the most basic linear least squares regression
- We will examine the interest rate for four year car loans, and the data that we use comes from the U.S. Federal Reserve's mean rates

Consumer Credit

	2000	2001	2002	2003	2004

Percent change at annual rate 2,3					
Total	10.7	8.0	4.4	4.7	4.5
Revolving	11.6	6.5	2.2	3.2	4.3
Nonrevolving 4	10.1	9.0	5.9	5.6	4.7
Amount: billions of dollars					
Total	1704.3	1841.1	1922.8	2013.5	2104.9
Revolving	676.9	721.2	736.9	760.7	793.2
Nonrevolving 4	1027.4	1119.9	1185.9	1252.8	1311.8

TERMS OF CREDIT AT COMMERCIAL BANKS AND FINANCE COMPANIES 5
Percent except as noted: not seasonally adjusted

Institution, terms, and type of loan

Commercial banks					
Interest rates					
48-mo. new car	9.34	8.50	7.62	6.93	6.60
24-mo. personal	13.90	13.22	12.54	11.95	11.89
Credit card plan					
All accounts	15.78	14.87	13.40	12.30	12.71
Accounts assessed interest	14.92	14.46	13.11	12.73	13.21
New car loans at auto finance companies					
Interest rates	6.61	5.65	4.29	3.40	4.36
Maturity (months)	54.9	55.1	56.8	61.4	60.5
Loan-to-value ratio	92	91	94	95	89
Amount financed (dollars)	20,923	22,822	24,747	26,295	24,888

Loading Data

- The first thing to do is to specify the data
- Here there are only five pairs of numbers so we can enter them in manually
- Each of the five pairs consists of a year and the mean interest rate:

```
> year <- c(2000 , 2001 , 2002 , 2003 , 2004)
> rate <- c(9.34 , 8.50 , 7.62 , 6.93 , 6.60)
```

Linear Trend

- The next thing we do is take a look at the data
- If we plot the data using a scatter plot, we can notice that it looks linear
 - To confirm our suspicions we then find the correlation between the year and the mean interest rates:

```
> plot(year,rate,  
       main="Commercial Banks Interest Rate for 4 Year Car Loan",  
       sub="http://www.federalreserve.gov/releases/g19/20050805/")  
> cor(year,rate)  
[1] -0.9880813
```


Regression

- The next question is what straight line comes “closest” to the data?
- we will use least squares regression as one way to determine the line
 - we have to decide which is the explanatory and which is the response variable
 - we arbitrarily pick the explanatory variable to be the year, and the response variable is the interest rate
 - This was chosen because it seems like the interest rate might change in time rather than time changing as the interest rate changes
 - We could be wrong, finance is very confusing

lm Command

- The command to perform the least square regression is the *lm* command
- If you are interested use the *help(lm)* command to learn more
- Instead the only option we examine is the one necessary argument which specifies the relationship
 - Since we specified that the interest rate is the response variable and the year is the explanatory variable this means that the regression line can be written in slope-intercept form:

$$rate = (slope)year + (intercept)$$

lm Command

- In the *lm* command you write the vector containing the response variable, a tilde (“~”), and a vector containing the explanatory variable:
 - When you make the call to *lm* it returns a variable with a lot of information in it.
 - If you are just learning about least squares regression you are probably only interested in two things at this point, the slope and the y-intercept

lm Command

- If you just type the name of the variable returned by *lm* it will print out this minimal information to the screen.

```
> fit <- lm(rate ~ year)
> fit
Call:
lm(formula = rate ~ year)

Coefficients:
(Intercept)      year
  1419.208      -0.705
```

Two Way Tables

Smoker Data

- For each person it was determined whether or not they are current smokers, former smokers, or have never smoked
 - smoking status (Smoke) and their socioeconomic status (SES)
 - The data file contains only two columns, and when read R interprets them both as factors:

```
> smokerData <- read.csv(file='smoker.csv',sep=',',header=T)
> summary(smokerData)
      Smoke      SES
current:116  High  :211
former  :141  Low   : 93
never   : 99  Middle: 52
```

Creating Table

■ Table commend

```
> smoke <- table(smokerData$Smoke,smokerData$SES)
> smoke
```

```
      High Low Middle
current  51  43    22
former   92  28    21
never    68  22     9
```

■ Creating Table Directly

```
> smoke <- matrix(c(51,43,22,92,28,21,68,22,9),ncol=3,byrow=TRUE)
> colnames(o) <- c("High","Low","Middle")
> rownames(o) <- c("current","former","never")
> smoke <- as.table(smoke)
> smoke
```

```
      High Low Middle
current  51  43    22
former   92  28    21
never    68  22     9
```

Tools for Working with Tables

- First, there are a couple of ways to get graphical views of the data:

- Barplot

```
> smokerData <- read.csv(file='smoker.csv',sep=',',header=T)
> smoke <- table(smokerData$Smoke,smokerData$SES)
> mosaicplot(smoke)
> help(mosaicplot)
>
```

- The *plot* command will automatically produce a mosaic plot if its primary argument is a table

Mosaicplot

- The *mosaicplot* command takes many of the same arguments for annotating a plot:

```
> mosaicplot(smoke,main="Smokers",xlab="Status",ylab="Economic Class")  
>
```

Introduction to Programming

If Statements

- Conditional execution is available using the *if* statement

```
> x = 0.1
> if( x < 0.2)
{
  x <- x + 1
  cat("increment that number!\n")
}
increment that number!
> x
[1] 1.1
```

Else Statement

- The else statement can be used to specify an alternate option.

```
> x = 2.0
> if ( x < 0.2)
{
  x <- x + 1
  cat("increment that number!\n")
} else
{
  x <- x - 1
  cat("nah, make it smaller.\n");
}
nah, make it smaller.
> x
[1] 1
```

For Statements

- The *for* loop can be used to repeat a set of instructions
 - The basic format for the *for* loop is *for(var in seq) expr*

```
> for (lupe in seq(0,1,by=0.3))
  {
  cat(lupe,"\n");
  }
0
0.3
0.6
0.9
>
> x <- c(1,2,4,8,16)
> for (loop in x)
  {
  cat("value of loop: ",loop,"\n");
  }
value of loop: 1
value of loop: 2
value of loop: 4
value of loop: 8
value of loop: 16
```

While Statements

- The *while* loop can be used to repeat a set of instructions
 - The basic format for a *while* loop is *while(cond) expr*

```
>
> lupe <- 1;
> x <- 1
> while(x < 4)
{
  x <- rnorm(1,mean=2,sd=3)
  cat("trying this value: ",x," (",lupe," times in loop)\n");
  lupe <- lupe + 1
}
trying this value: -4.163169 ( 1 times in loop)
trying this value:  3.061946 ( 2 times in loop)
trying this value:  2.10693 ( 3 times in loop)
trying this value: -2.06527 ( 4 times in loop)
trying this value:  0.8873237 ( 5 times in loop)
trying this value:  3.145076 ( 6 times in loop)
trying this value:  4.504809 ( 7 times in loop)
```

Repeat Statements

- The *repeat* loop is similar to the *while* loop
 - But you need to execute the *break* statement to get out of the loop

```
> repeat
{
  x <- rnorm(1)
  if(x < -2.0) break
}
> x
[1] -2.300532
```

Break and Next Statements

- The *break* statement is used to stop the execution of the current loop
- The *next* statement is used to skip the statements that follow and restart the current loop

```
> x <- rnorm(5)
> x
[1] 1.41699338 2.28086759 -0.01571884 0.56578443 0.60400784
> for(lupe in x)
{
  if (lupe > 2.0)
    next

  if( (lupe<0.6) && (lupe > 0.5))
    break

  cat("The value of lupe is ",lupe,"\n");
}
The value of lupe is 1.416993
The value of lupe is -0.01571884
```


Switch Statement

- The *switch* takes an expression and returns a value in a list based on the value of the expression
 - Note that the expression is cast as an integer if it is not an integer

```
> x <- as.integer(2)
> x
[1] 2
> z = switch(x,1,2,3,4,5)
> z
[1] 2
> x <- 3.5
> z = switch(x,1,2,3,4,5)
> z
[1] 3
```

Scan Statement

- The command to read input from the keyboard is the *scan* statement
 - When using the command with no set number of lines the command will continue to read keyboard input until a blank line is entered.

```
> help(scan)
> a <- scan(what=double(0))
1: 3.5
2:
Read 1 item
> a
[1] 3.5
> typeof(a)
[1] "double"
>
> a <- scan(what=double(0))
1: yo!
1:
Error in scan(file, what, nmax, sep, dec, quote, skip, nlines, na.strings, :
scan() expected 'a real', got 'yo!'
```

Functions

- To define a function you assign it to a name, and the keyword *function* is used to denote the start of the function and its argument list

```
> newDef <- function(a,b)
{
  x = runif(10,a,b)
  mean(x)
}
> newDef(-1,1)
[1] 0.06177728
> newDef
function(a,b)
{
  x = runif(10,a,b)
  mean(x)
}
```

What is Returned

- In the example the sample mean of the numbers is returned

```
> x <- newDef(0,1)
> x
[1] 0.4800442
```

Passing Arguments

- The arguments that are passed are matched in order. They can be specified explicitly, though.

```
> newDef(b=10,a=1)
[1] 4.747509
> newDef(10,1)
[1] NaN
Warning message:
In runif(10, a, b) : NAs produced
```

Reading List

- **Recommended**

- Review Slides

- R Tutorial

- <http://www.cyclismo.org/tutorial/R/>

- **Optional**

- Quick-R: accessing the power of R

- <https://support.google.com/docs/answer/190718>