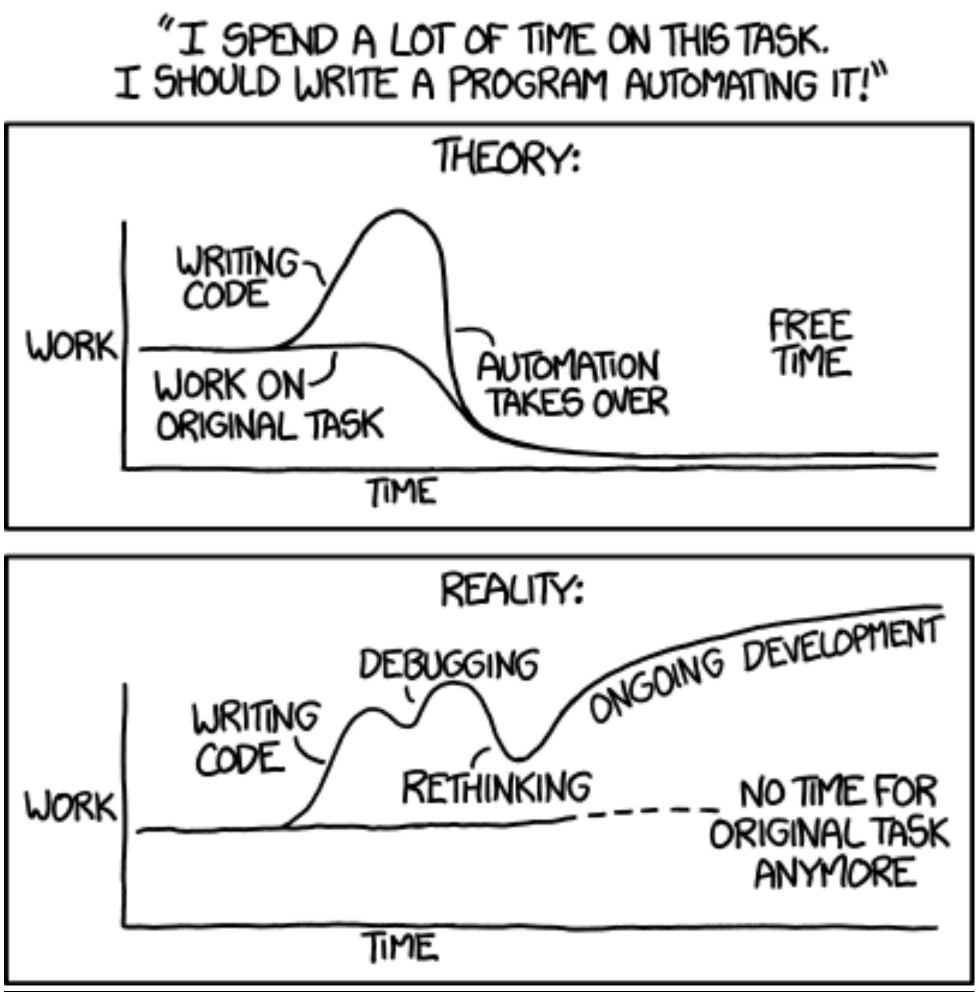
R Workshop, Session 1
A Basic Introduction to R

by Rebecca Clark, but only by standing on the shoulders of many other R aficionados



Session 1, Lesson 2. A Basic Introduction to R

R will not read anything preceded by #. When analyzing data, it is useful to use # to include a lot of comments so you can remember what you did.
R **input** is preceded by >
R **output** is preceded by [some number]

2.A. Using R as a calculator:

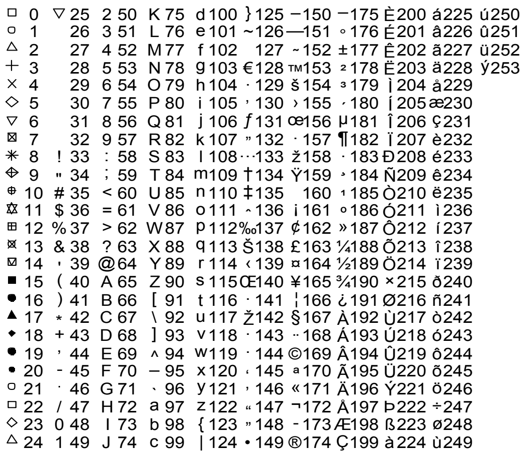
```
> 2+3+4+9
[1] 18

> 144^2
[1] 20736

> (144^0.5)/2*10
[1] 60

> sqrt(100)
[1] 10

> log(5)      # log in R equals the natural log (i.e. ln)
[1] 1.609438

> pi*4
[1] 12.56637
```

2.B. Handy functions

```
> data <- c(2,2,4,4,5,5,6,6,7,8,8,8,8,9,9,10,10,12,12,12,12)
# the c means "concatenate" which means individual numbers are
joined together in a vector. R really likes concatenated data.
> data
[1] 2  2  4  4  5  5  6  6  7  8  8  8  8  9  9 10 10 12 12 12 12
# yep it's there!
> mean(data)
[1] 7.571429
> min(data)
[1] 2
> max(data)
[1] 12
> range(data)
[1]  2 12
> sd(data)
[1] 3.1713
> var(data)
[1] 10.05714
> length(data)
[1] 21
```

2.C. Basic plotting

```
> hist(data) # you should see a histogram pop up in the graphics
device.

x <- c(1,3,4,6,8,9,12,14)
y <- c(5,6,8,10,9,13,12,15)
plot(x, y)

# add some x and y labels:
plot(x,y, xlab="Explanatory Variable", ylab="Response Variable")

# change the symbol color:
plot(x,y, xlab="Explanatory Variable", ylab="Response Variable",
col="red")
# for a full list of colors:
colors()  # test some of these out

# change the symbol type with pch (stands for "point character")
plot(x,y, xlab="Explanatory Variable", ylab="Response Variable",
col="red", pch=16)
plot(x,y, xlab="Explanatory Variable", ylab="Response Variable",
col="red", pch="m")

# to see the symbols, you need to install a package once you have
access to the internet.
> install.packages("Hmisc")  # select the closest CRAN mirror from
the list.  R will access the CRAN website and download this
package to your "libraries" folder in the R program file.  This
only has to be done once.

# when you start R, it will only start the base package.  If you
want to use a function in a package other than the base you must
access it from the library folder:
> library(Hmisc)  #now you can use the functions in this package
> show.pch()
```



2.D. Basic t-test: Getting started

```
# create a fake dataset
> length <- c(1.2, 1.3, 1.6, 1.4, 1.1, 2.0, 2.1, 2.8, 3.0, 2.6)
> gender <- rep(c("male", "female"), each=5)

> plot(length, gender)
# this may generate an error message because R may not know the
correct class to assign to "gender".  Check the class:
> class(gender)
# How to reassign it:
> gender <- as.factor(gender)
# Test this
> gender

> plot(gender, length)

# Summarize the data
> mean(length)
> sd(length) # Standard deviation

# What if we want the mean and standard deviation for each gender?
We need to "subset" the data:
length[gender == "male"]
```

# R translation: "give me the length data for the males". Brackets are used for subsetting data in R (more on this later). Subsetting is a critical, and often annoyingly difficult process in R. Notice that double = signs are required. Notice that "male" is in quotes.

```
> mean(length[gender == "male"])
> sd(length[gender == "male"])
> mean(length[gender == "female"])
> sd(length[gender == "female"])
# Or, how about a loop?
tapply(length, gender, mean)
```

# tapply is one of the "apply" functions. This suite of functions performs different types of loops. In this case, the loop goes through each category of "gender" and applies the function "mean" to the "length" data.

2.E. Obtaining more information about functions

Our goal is to analyze the data using a t-test to determine whether males and females have significantly different lengths. First, we need to figure out which function to use to accomplish this. Sometimes (often) you won't know the name of the function for a particular analysis. From within R, there are a few ways to look:

```
help.search("t-test")
```

The problem is that you will often get zero, or way too many hits. If help.search() fails, you may want to search online instead. Once you know the name of a function, you can use the ? or help to find out more:

```
?t.test
help(t.test)
```

This should open a help window with documentation for the function. At first, these windows may seem cryptic and useless, but after some time, you will probably start to rely on them heavily (I sure do!). I find the See Also and Examples sections especially helpful. Note that the Usage section tells you exactly how to execute the command (expected arguments, etc.), and details of the required arguments are given in the Arguments section.

Also try the example function to see the function in action, if it has an example:

```
example(t.test)
```

2.F. Conducting the t-test

```
t.test(length~gender)
```

# Almost all models are defined in R with this syntax. It translates to: "length" as a function of "gender", or Y(response)~X(predictor/s).
# Note that the returned results are for a "Welch Two Sample t-test." If you look at the function description, you'll see that the default analysis assumes the variances between the two groups are not equal (var.equal=FALSE). The output also returns 95% confidence intervals. These settings can be changed:

```
t.test(length~gender, var.equal=TRUE, conf.level=0.90)
```

2.G. Closing R

R will ask:

`Save workspace image? [ y / n / c ]` #Generally, choose `n`.

Session 1, Lesson 3. Loading Data

3.A. The Working Directory

What is the working directory? It is the location on your computer that R is working from – where it will read in files, and where it will write files. To determine the current working directory on your computer:

```
getwd()
[1] "/Users/username"
```

It may help to imagine this as the room that R is in. It can see everything in the room, so you can refer directly to the things in the room. Otherwise, you have to tell R how to get to things that are outside of the room.

```
dir()
[1] "Applications"
[2] "Documents"
[3] "Downloads"
[4] "Movies"
[5] "Pictures"
[6] "chicken painting.jpg"
…
```

To change the working directory:

```
setwd("~/Documents/RebeccaClark/R Manuals/OSOS Workshop
Materials/")
dir()
```

Now R will look in this directory for files and will also save to this directory.

Important things to notice:

1. I checked to make sure the function worked by using `dir()` to see if the correct files were listed.
2. If you are on a PC, try to get in the habit of using forward slashes "/" instead of back slashes "\" because back slashes are used as an escape character in R. See the documentation for `?Quotes` for more details.
3. It's usually simplest to copy and paste in the file pathway to avoid typing errors.

3.B. Types of Data

Use `class()` to check

R class	Other terms	Examples
Numeric	Real, continuous, quantitative	1.34, 2.87, 1,000,005
Integer	Count data	1, 2, 3, 184
Factor (ordered)	Ordinal, categorical, discrete	Good, better, best; Large, extra large, grande
Factor (unordered)	Categorical, nominal, discrete	Red, green, blue; Alps, Rockies, Rainier
Date		11/12/2010*
Character		"accidentally smooshed", "fell asleep"

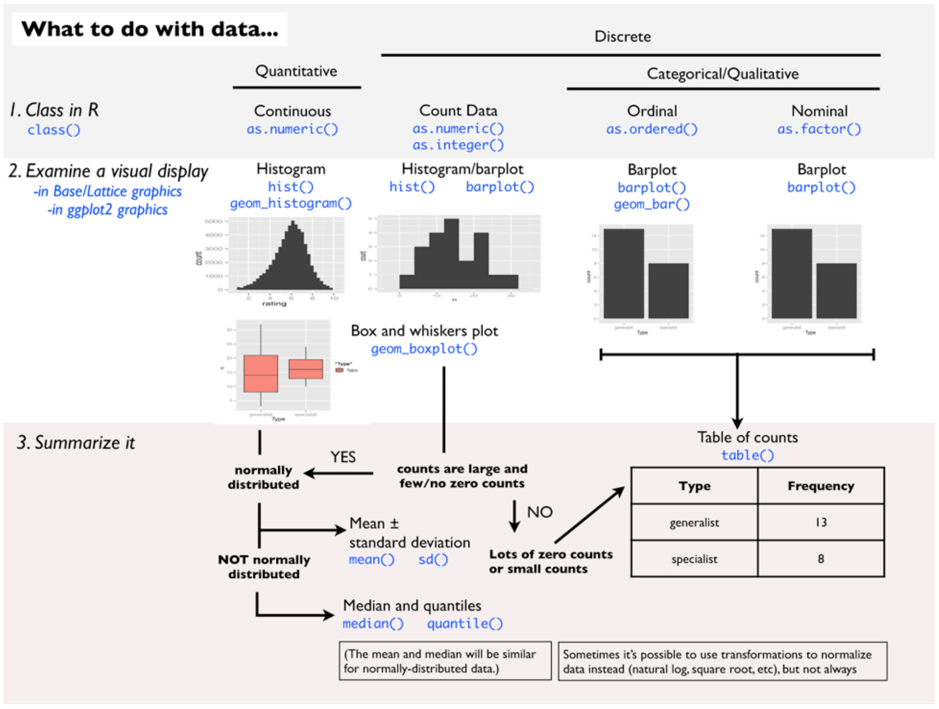
*Time-based data can get complicated in terms of formatting. See *Data Manipulation With R*, by Phil Spector, for an in-depth treatment of the topic, if applicable.

Because R is an *object-oriented programming language*, the *class* of the data often determines how it is handled by a particular function. For practice, make a note of how R should interpret the class of each column in this spreadsheet:

Family	Insect	Life Stage	Feeding Type	Sample size	Insect %P	Insect N (%)	Plant %P	Plant %N
Drosophilidae	Drosophila arizonae	adult	generalist	1	0.995	8.800	0.576	3.720
Drosophilidae	Drosophila hydei	adult	generalist	1	0.880	7.650	0.576	3.720
Drosophilidae	Drosophila nigrospiracula	adult	generalist	1	0.850	7.900	0.020	0.829
Drosophilidae	Drosophila pseudoobscura	adult	generalist	1	1.000	8.850	0.576	3.720
Drosophilidae	Drosophila simulans	adult	generalist	1	1.090	9.500	0.576	3.720
Acrididae	Melanoplus bilituratus	adult	generalist	2	0.700	-	0.250	4.000
Acrididae	Hesperotettix speciosus	multiple instars	generalist	1	0.631	11.080	0.249	1.643
Acrididae	Melanoplus bivittatus	multiple instars	generalist	2	0.560	10.775	0.238	2.424
Acrididae	Melanoplus keeleri	multiple instars	generalist	1	0.633	10.680	0.258	1.995
Acrididae	Mermiria bivittata	-	generalist	1	0.470	10.000	0.194	1.124
Acrididae	Schistocerca americana	-	generalist	2	0.693	9.792	0.234	2.426
Acrididae	Melanoplus packardii	multiple instars	generalist (33 spp)	2	0.628	10.980	0.190	2.099
Acrididae	Schistocerca gregaria	-	generalist/ polyphagous	1	0.903	9.430	0.260	2.555
Tortricidae	Choristoneura fumiferana	-	specialist	1	0.855	8.700	0.220	1.422
Drosophilidae	Drosophila mojavensis	adult	specialist	1	0.840	6.700	0.190	0.800
Drosophilidae	Drosophila pachea	adult	specialist	1	0.815	6.700	0.180	1.440
Sphingidae	Manduca sexta	larvae	specialist	2	1.124	9.500	0.209	4.500
Noctuidae	Spodoptera exempta	pupa	specialist	3	1.086	8.765	0.433	2.990
Diprionidae	Neodiprion sertifer	-	specialist	3	0.640	7.270	0.117	1.303
Chrysomelidae	Paropsis atomaria	larvae	specialist	1	0.929	6.693	0.236	1.175
Curculionidae	Sabinia setosa	adult	specialist	1	0.557	-	0.220	2.750

3.C. Data Analysis Overview

From a data analysis standpoint, here is an overview of ways to handle different kinds of data. More details on this will follow later on.



What to do with data...

Quantitative | Discrete | Categorical/Qualitative

1. Class in R `class()`
- Continuous: `as.numeric()`
- Count Data: `as.numeric()` `as.integer()`
- Ordinal: `as.ordered()`
- Nominal: `as.factor()`

2. Examine a visual display *-in Base/Lattice graphics* *-in ggplot2 graphics*
- Histogram: `hist()` `geom_histogram()`
- Histogram/barplot: `hist()` `barplot()`
- Barplot: `barplot()` `geom_bar()`
- Barplot: `barplot()`

Box and whiskers plot `geom_boxplot()`

3. Summarize it

counts are large and few/no zero counts → YES → normally distributed

normally distributed → Mean ± standard deviation `mean()` `sd()`

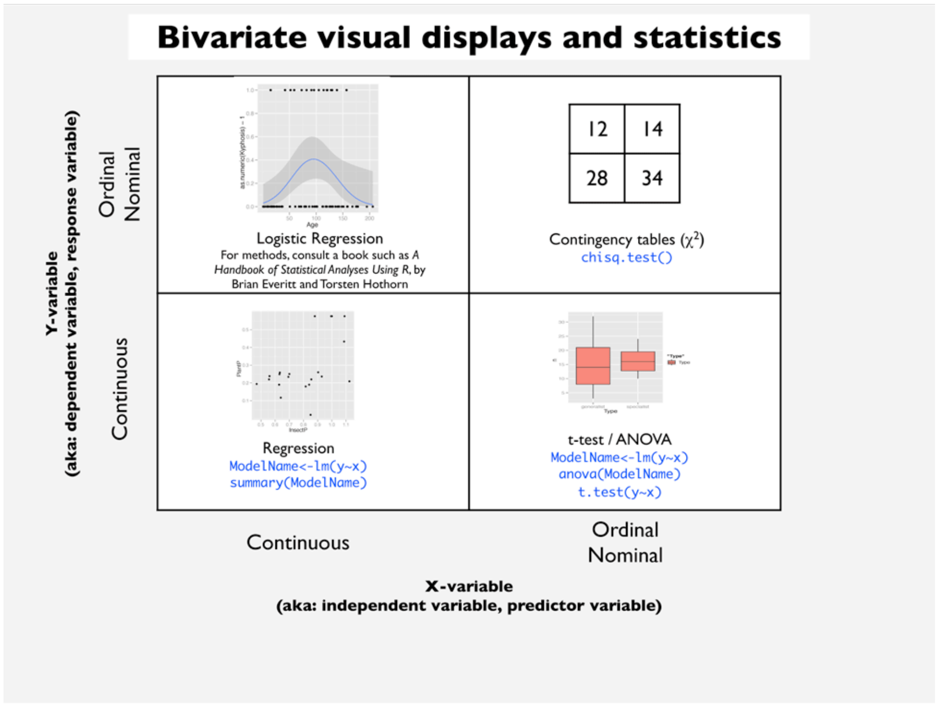
NOT normally distributed → Median and quantiles `median()` `quantile()`

counts are large and few/no zero counts → NO → Lots of zero counts or small counts → Table of counts `table()`

Type	Frequency
generalist	13
specialist	8

(The mean and median will be similar for normally-distributed data.)

Sometimes it's possible to use transformations to normalize data instead (natural log, square root, etc), but not always



Bivariate visual displays and statistics

Y-variable (aka: dependent variable, response variable)

X-variable (aka: independent variable, predictor variable)

	Continuous	Ordinal / Nominal
Ordinal / Nominal	Logistic Regression. For methods, consult a book such as *A Handbook of Statistical Analyses Using R*, by Brian Everitt and Torsten Hothorn	Contingency tables (χ^2) `chisq.test()`
Continuous	Regression `ModelName<-lm(y~x)` `summary(ModelName)`	t-test / ANOVA `ModelName<-lm(y~x)` `anova(ModelName)` `t.test(y~x)`

12	14
28	34

3.D. Worksheet: Importing Data into R

Open the file **Excel.xls**, and try to identify several issues that might be a problem when this dataset is opened in R.

Now, save **Excel.xls** as a tab-delimited text file (.txt).

File > Save As

File name: **ExcelAsText.txt**

Save As Type: Text [Tab delimited]

Open ExcelAsText.txt in R

```
setwd(“C:/YourPathHere/”)
data <- read.table(“ExcelAsText.txt”, header=TRUE, sep=”\t”)
```

Check out the data:

```
data
lapply(data, class) #find out the class of each column
```

What kinds of problems do you see? Are these the same ones you thought would be a problem?

Now, go back and try to fix any problems you have identified, repeating the process of saving the file as a new tab-delimited text file (**ExcelAsText_Fixed.txt**) and reading it into R (`data_tab`). Did that work?

Now, save the file as a comma-delimited text file (.csv), and open this in R to examine it.

File > Save As

File name: **ExcelAsText_Fixed.csv**

Save As Type: CSV [comma delimited]

```
data_csv <- read.table(“ExcelAsText_Fixed.csv”, header=TRUE, sep=”,”)
lapply(data_csv, class)
```

Are there any new problems?

3.E. Importing Data into R: Workflow

```
# set the working directory
setwd("C:/YourPathHere/")
# get the data - remember to designate the separator
data <- read.table("FileName.txt", header=TRUE, sep="\t")
# look at your data
data
```

lapply(data, class) # check that the classes of the columns match with your expectations

Things to check before converting your Excel file to text:

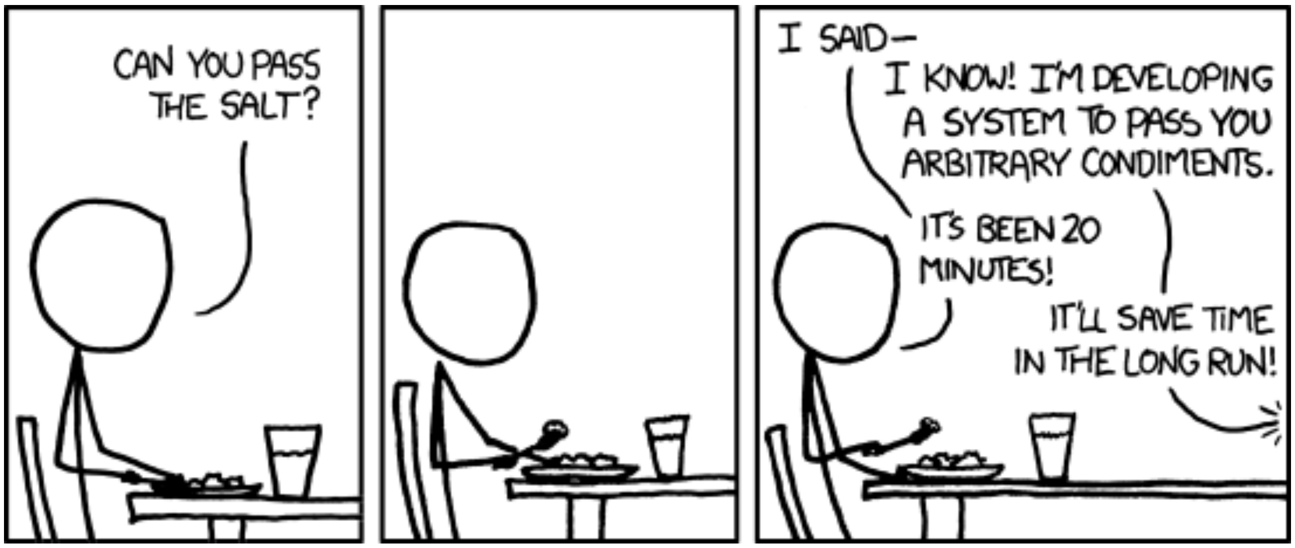
1. You can only have one row of headers
2. Use short descriptive headers without spaces
    a. Use "_" or caps between words (i.e. first_name or FirstName)
    b. Headers are typed over, and over, and over, so keep them short
3. Don't start a header with a number
    a. R will accept it, but will put an X in front of it.
4. Remove all summary data
    a. Average, sum, max, min, etc.
5. Check that all values in a column are of the same type
    a. Number, date, character, etc.
6. Remove commas and # symbols throughout
    a. Commas will screw up comma-delimited files
    b. # indicates a comment in R
7. Missing data are okay, but keep an eye on it.
8. If R throws this error:

    ```
    Error in scan(file, what, nmax, sep, dec, quote,
    skip, nlines, na.strings,  :
      line 29 did not have 8 elements
    ```

    You probably have extra lines at the end of your .txt file. Open your text file in notepad. Move the cursor to the bottom of the file and backspace (delete) all of the extra lines. Save and try again (remember to send the read.table line again).

Handy shortcuts to try

```
read.table(file.choose()) # type this in and see what happens

setwd("~") # resets working directory to your computer's home
directory (escape to home directory)

setwd("./FolderInCurrentDirectory/") # allows you to move from the
current directory into a subdirectory
```


CAN YOU PASS THE SALT?
I SAID—
I KNOW! I'M DEVELOPING A SYSTEM TO PASS YOU ARBITRARY CONDIMENTS.
IT'S BEEN 20 MINUTES!
IT'LL SAVE TIME IN THE LONG RUN!

Session 1, Lesson 4. Manipulating Data

Once you've read data into R, you can begin using some of the powerful workhorse aspects of the program to get your data organized and ready for analysis. This is where R can really shine…but also where it's easy to get lost and frustrated and give up. Here, we will walk through modified exercises that were originally designed by Jack Weiss at UNC-Chapel Hill.

4.A. R functions and commands demonstrated here

- apply is used to evaluate a function separately on the rows (second argument to apply is 1) or the columns (second argument to apply is 2) of a matrix or data frame
- attach adds a data frame to the default search path so that variables can be specified without reference to the data frame in which they reside
- boxplot produces a box plot or side-by-side box plots of a specified variable
- c the catenation function that turns the elements making up its arguments into a single vector
- colnames returns the column names of a data frame. Can also be used to assign column names to a data frame
- data.frame defines a data frame from its arguments which should be a set of vectors all of the same length
- detach undoes attach, removes a data frame from the search path
- dim returns the number of rows and columns of a data frame
- dimnames returns both the row names and the column names of a data frame in a list format
- edit invokes the R editor for modifying and viewing data frames
- expression creates an object of mode expression. We used it to create a mathematical expression in a graph axis label.
- function is not itself a function but is a key word to indicate that what follows is a function definition
- history opens up a history window in which you can view previously issued commands. For example, history(50) would display up to the last 50 commands issued. There is a 512 line default limit to what is saved, although this value can be changed.
- is.na is a logical function that returns TRUE if a value is missing (NA) and FALSE otherwise
- jitter randomly adds a small value to each element of its argument
- length returns the number of elements of a vector counting both non-missing and missing values
- mean calculates the mean of individual column entries of a data frame
- names displays names of variables in a data frame
- objects displays all created objects currently in workspace
- points adds individual points to the currently active plot
- read.table reads in text data from an external file
- rep is used to create patterned vectors of repeated units
- reshape interconverts 'wide' and 'long' data sets.
- rm is used to delete objects from the R workspace
- round rounds its argument to the number of decimals specified.
- rownames returns the row names of a data frame. Can also be used to assign row names to a data frame
- sd calculates the standard deviation of the individual column entries of a data frame

- sqrt is the square root function in R
- sum calculates the sum of all entries of a vector or matrix
- tapply stands for table apply. It applies a function (3rd argument) to a variable (1st argument) separately for each group specified by the second argument
- unlist unstacks the columns of a data frame into a vector
- # indicates a given line of code is a comment and should be ignored
- <- the assignment operator in R, a less than symbol followed by a dash, that is supposed to symbolize an arrow. The arrow points in the direction of assignment.
- [ ] used for specifying elements of vectors or portions of data frames and matrices
- [[ ]] denotes an element of a list
- $ list notation symbol that can be used to reference columns of a data frame
- ! is the logical not operator in R
- ^ denotes exponentiation
- ? followed by a function name brings up a help window on that function
- ~ symbol used in defining expressions in R for model fitting. We used it in the boxplot function

R function options

- cex= (argument to many graphics functions) specifies the character expansion for plotting symbols when used with the points function
- col= (argument to many graphics functions) specifies the color to use in plotting points and/or line segments
- header= (argument to read.table) takes on values TRUE or FALSE, indicates whether the first line of a text file contains the variables names (TRUE) or not (FALSE)
- pch= stands for print character and is used to designate the plotting symbol for use in various plotting functions: plot, points, etc.
- na.rm= (argument to mean, sum, and sd) take on values TRUE or FALSE, indicates whether missing values should be removed (TRUE) before performing calculations. If set to FALSE and there are missing value the function returns NA as its value.
- outline= (argument to boxplot) when set to FALSE it turns off the display of outliers in a box plot
- sep= (argument to read.table) specifies the character that was used to separate fields in the text file to be read into R. For example, sep=',' indicates that the entries are separated by commas while sep='\t' indicates that the entries are separated by tabs.
- skip= (argument to read.table) specifies the number of lines to skip in the text file before reading the first line of data
- xlab= (argument of boxplot) a user-specified value to be used as the label for the x-axis, e.g., xlab="WSSTA"
- ylab= (argument of boxplot) a user-specified value to be used as the label for the y-axis, e.g., ylab="Disease Prevalence "

4.B. Review: Data Entry

When R opens you are presented with Console window in which you can enter commands. In the Windows operating system to move back within a command line to correct mistakes you need to use the left and right arrow keys on the keyboard rather than the mouse. Previously issued commands can be recalled with the up arrow key.

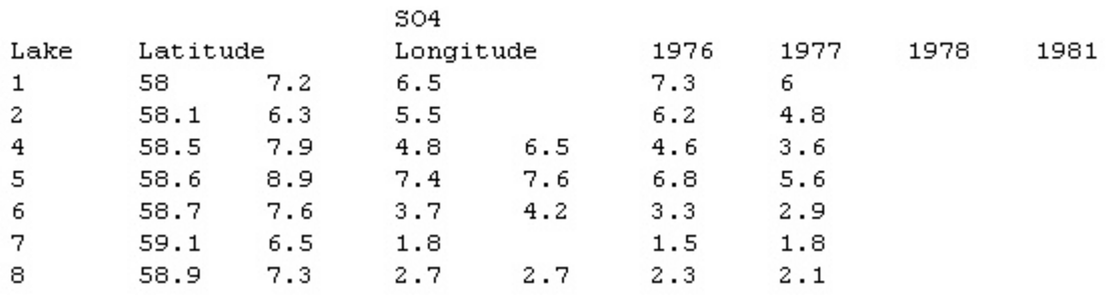
To read data into R from a text file, use the read.table function. For this exercise, we'll read in a file directly from a class web site. The data files are in the following folder.

http://www.unc.edu/courses/2007spring/enst/562/001/data/lab1/

There are three files in this folder. The files contain the same data but in different formats.

> **NorwaySO4.xls** is an Excel file. While a package exists that allows the direct importation of Excel files (the package is called RODBC), using it is far more trouble than it's worth. For a single worksheet it's far more convenient to first save the Excel file as a text file and then read the text file into R.
> **NorwaySO4.txt** is a tab-delimited text file
> **NorwaySO4.csv** is a comma-delimited text file

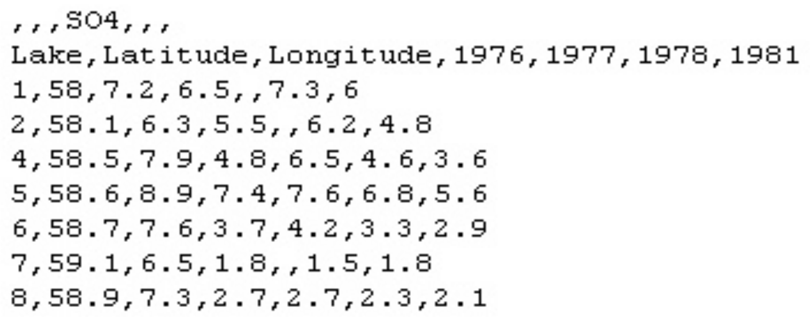
Each text file is set up in such a way that the first row of the file contains the word SO4, identifying the variable whose measurements are contained in the file. The second row contains the names of the variables and the subsequent rows contain the data values. The first few lines of the tab-delimited file are shown in **Fig. 1**, while **Fig. 2** shows the comma-delimited file. The tabs are not visible but they exist as special characters separating the different columns that correspond to the different fields in the Excel file.



```
                        SO4
Lake     Latitude       Longitude        1976     1977     1978     1981
1        58       7.2   6.5              7.3      6
2        58.1     6.3   5.5              6.2      4.8
4        58.5     7.9   4.8      6.5     4.6      3.6
5        58.6     8.9   7.4      7.6     6.8      5.6
6        58.7     7.6   3.7      4.2     3.3      2.9
7        59.1     6.5   1.8              1.5      1.8
8        58.9     7.3   2.7      2.7     2.3      2.1
```

Fig. 1 Tab-delimited text file

The use of delimiters is not always necessary, but is usually a good idea. We must use delimiters here because of the presence of missing data. Excel indicates missing data with blanks. The use of delimiters correctly identifies these missing values. Notice in Fig. 2 that for Lake 1 the SO_4 concentration in 1977 is missing and so we find two commas in succession. In Fig. 1 there are accordingly two tabs in succession (not visible).



```
,,,SO4,,,
Lake,Latitude,Longitude,1976,1977,1978,1981
1,58,7.2,6.5,,7.3,6
2,58.1,6.3,5.5,,6.2,4.8
4,58.5,7.9,4.8,6.5,4.6,3.6
5,58.6,8.9,7.4,7.6,6.8,5.6
6,58.7,7.6,3.7,4.2,3.3,2.9
7,59.1,6.5,1.8,,1.5,1.8
8,58.9,7.3,2.7,2.7,2.3,2.1
```

Fig. 2 Comma-delimited text file

read.table is a the name of an R function. R uses standard mathematical notation *f*(*x*,*y*,*z*) to specify functions and their arguments, so parentheses are always required with functions although sometimes it is not necessary to specify any arguments.

The arguments we need to specify here:

1. The location of this file on the web with the complete path enclosed in quotes. You may use single or double quotes, but you must not mix them as part of the same argument.
2. The argument **skip=1** to tell R to skip the first line of the file.
3. The argument **header=TRUE** to indicate that the variable names appear at the top of the file.
4. The argument **sep=','** to indicate a comma-delimited file or the argument **sep='\t'** to indicate a tab-delimited file.
5.

To save the output of the **read.table** function, use the assignment operator, **<-** , to assign the output to an object in R
Read the tab-delimited file into R as follows. The name I choose for the result in R is **so4**. R is case-sensitive so **so4**, **So4**, **SO4** all represent different objects.

```
# Read in tab-delimited data from Web
> so4<-
read.table('http://www.unc.edu/courses/2007spring/enst/562/001/dat
a/lab1/NorwaySO4.txt', skip=1, header=TRUE, sep='\t')
# Read in comma-delimited data from the Web
> so4<-
read.table('http://www.unc.edu/courses/2007spring/enst/562/001/dat
a/lab1/NorwaySO4.csv', skip=1, header=TRUE, sep=',')
```

4.C. Getting Information about Data Frames

The object **so4** that we've created in R is called a data frame. Data frames look like matrices but the elements of data frames can be mixtures of character and numeric data. More formally data frames are tightly coupled collections of variables that share many of the properties of matrices and of lists. They are the fundamental data structures for most of **R**'s modeling functions.

The **dim** function is used to determine the dimensions of a data frame.

```
> dim(so4)
[1] 48  7
```

From the output we see that there are 48 rows and 7 columns. The output is a vector and we can access the individual entries using standard vector notation.

```
> dim(so4)[1]
[1] 48
> dim(so4)[2]
[1] 7
```

The functions **colnames** and **rownames** return the column names and row names of the data frame.

```
> colnames(so4)
[1] "Lake"      "Latitude"  "Longitude" "X1976"     "X1977"
"X1978"     "X1981"

> rownames(so4)
[1]  "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "11" "12"
[13] "13" "14" "15" "16" "17" "18" "19" "20" "21" "22" "23" "24"
[25] "25" "26" "27" "28" "29" "30" "31" "32" "33" "34" "35" "36"
[37] "37" "38" "39" "40" "41" "42" "43" "44" "45" "46" "47" "48"
```

What’s returned are vectors of character data. If you use just the **names** function you get the column names only.

```
> names(so4)
[1] "Lake"      "Latitude"  "Longitude" "X1976"     "X1977"
"X1978"     "X1981"
```

Observe that R has appended an X in front of the names of the years. Variable names cannot start with a number.

In S-Plus, the commercial cousin of R, the **rownames** and **colnames** functions don’t exist. Instead use the **dimnames** function.

```
> dimnames(so4)
[[1]]
[1]  "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "11" "12"
[13] "13" "14" "15" "16" "17" "18" "19" "20" "21" "22" "23" "24"
[25] "25" "26" "27" "28" "29" "30" "31" "32" "33" "34" "35" "36"
[37] "37" "38" "39" "40" "41" "42" "43" "44" "45" "46" "47" "48"
[[2]]
[1] "Lake"      "Latitude"  "Longitude" "X1976"     "X1977"
"X1978"     "X1981"
```

The **dimnames** function returns an object called a list. Notice the double bracket notation. Lists are useful when elements are of different lengths (as they are here) or of different types. To access the second element of this list, use the double bracket notation:

```
> dimnames(so4)[[2]]
[1] "Lake"      "Latitude"  "Longitude" "X1976"     "X1977"
"X1978"     "X1981"
```

To access the third element of the second element of the list:

```
> dimnames(so4)[[2]][3]
[1] "Longitude"
```

4.D. Accessing elements of data frames

Because a data frame is both a matrix and a list, we can use either notation to access its elements. Let's examine matrix notation first using row and column numbers.

`so4[2,3]` returns the element in row 2 and column 3 of the data frame so4.

```
> so4[2,3]
[1] 6.3
```

Specifying a row number followed by a comma and then no column number inside the brackets returns the entire row. Below I get the second row. The NA that appears in the output is the missing code in R. It stands for "not applicable".

```
> so4[2,]
  Lake Latitude Longitude X1976 X1977 X1978 X1981
2    2     58.1       6.3   5.5    NA   6.2   4.8
```

Specifying a comma followed by a number inside the brackets returns an entire column. Below I get the third column.

```
> so4[,3]
 [1]  7.2  6.3  7.9  8.9  7.6  6.5  7.3  8.5  9.3  6.4  7.5  7.6  9.8
[14] 11.8  6.2  7.3  8.3  8.9 12.0  5.9 10.2 12.2  5.5  7.3 10.0 12.2
[27]  5.0  5.6  6.9  9.7 10.8  4.9  5.5  4.9  5.8  7.1  6.4  6.7  8.0
[40]  7.1  6.1 11.3  9.4  7.6  7.3  6.3 11.5  4.6
```

Get two adjacent rows. The colon notation is used to generate a sequence of numbers. Thus 2:5 yields the vector (2, 3, 4, 5).

```
> so4[2:3,]
  Lake Latitude Longitude X1976 X1977 X1978 X1981
2    2     58.1       6.3   5.5    NA   6.2   4.8
3    4     58.5       7.9   4.8   6.5   4.6   3.6
```

Get 3 rows not all adjacent. Here we use the **c** function of R to concatenate the terms into a vector.

```
> so4[c(2:3,5),]
  Lake Latitude Longitude X1976 X1977 X1978 X1981
2    2     58.1       6.3   5.5    NA   6.2   4.8
3    4     58.5       7.9   4.8   6.5   4.6   3.6
5    6     58.7       7.6   3.7   4.2   3.3   2.9
```

We can also specify column elements by name (row elements too if they had names). Here I select the 1976 SO4 values. Notice that the variable name appears in quotes and is used instead of the column number.

```
> so4[,"X1976"]
 [1] 6.5 5.5 4.8 7.4 3.7 1.8 2.7  3.8 8.4 1.6 2.5 3.2 4.6 7.6 1.6 1.5 1.4
[18] 4.6 5.8 1.5 4.0 5.1  NA 1.4  3.8 5.1 2.8 1.6 1.5 3.2 2.8 3.0 0.7 3.1
[35] 2.1 3.9 1.9 5.2 5.3 2.9 1.6 13.0 5.5 2.8 1.6 2.0 5.8  NA
```

An alternative way of obtaining the same column is with list notation. In list notation we specify the data frame name followed by a $ sign followed by the variable name (unquoted).

```
> so4$X1976
 [1] 6.5 5.5 4.8 7.4 3.7 1.8 2.7  3.8 8.4 1.6 2.5 3.2 4.6 7.6 1.6 1.5 1.4
[18] 4.6 5.8 1.5 4.0 5.1  NA 1.4  3.8 5.1 2.8 1.6 1.5 3.2 2.8 3.0 0.7 3.1
[35] 2.1 3.9 1.9 5.2 5.3 2.9 1.6 13.0 5.5 2.8 1.6 2.0 5.8  NA
```

4.E. Attaching a data frame

Notice that to access a variable in a data frame we had to also specify the name of the data frame. Using the name of a variable all by itself does not work.

```
> X1976
Error: object "X1976" not found
> "X1976"
[1] "X1976"
```

The reason for this is that the data frame is currently not part of the R search path. R doesn't try to look in the data frame for the variables. To add a data frame to the search path use the **attach** function.

```
> attach(so4)
```

Now R can see the variable.

```
> X1976
 [1] 6.5 5.5 4.8 7.4 3.7 1.8 2.7  3.8 8.4 1.6 2.5 3.2 4.6 7.6 1.6 1.5 1.4
[18] 4.6 5.8 1.5 4.0 5.1  NA 1.4  3.8 5.1 2.8 1.6 1.5 3.2 2.8 3.0 0.7 3.1
[35] 2.1 3.9 1.9 5.2 5.3 2.9 1.6 13.0 5.5 2.8 1.6 2.0 5.8  NA
```

Attaching data frames can lead to confusion. If you change an entry in a variable from a data frame that has been attached, e.g., `X1976[5]<-10`, R makes another copy of the variable in the workspace with the changed entry, but doesn't change the variable in the data frame itself. A worse situation can arise if a variable name in a data frame matches a variable name for an object that already existed in the workspace. When this happens the latest variable shadows the first. If you're not aware that this has happened you may end up referencing the wrong variable. Because of the confusion that can result I recommend not attaching data sets as a general practice, but you will see this used in vignettes and examples. To undo the attachment, use the detach function.

```
> detach(so4)
```

Now the variable X1976 is again invisible to R.

4.F. Descriptive Statistics For Data Frames

The **mean** function calculates the mean of variables. If we try to use it to obtain the mean of the 1976 sulfate concentrations we get a surprise.

```
> mean(so4[,4])
X1976
NA
```

The problem is that there are missing values in this variable and arithmetic on missing values is undefined. We need to remove them first. The **mean** function has an optional argument that will do this for us. Recall that, to get help on a function you can enter a **?** followed by the name of the function. This brings up the help window.

```
> ?mean
```

From the help window you should see that there is an argument, **na.rm**, which can be used to remove missing values. We just need to set its value to TRUE.

```
> mean(so4[, 4], na.rm=TRUE)
X1976
3.743478
```

The help screen for mean tells us that "there are methods for numeric data frames, numeric vectors and dates". R is an object-oriented language meaning functions are written in such a way that they behave differently for different kinds of objects. If we apply mean to the four columns containing sulfate concentrations in different years we get the mean for each year.

```
> mean(so4[, 4:7], na.rm=TRUE)
   X1976    X1977    X1978    X1981
3.743478 3.978125 3.715217 3.334091
```

You can also round the values. The second argument to the round function is the number of decimals to display.

```
> round(mean(so4[,4:7],na.rm=TRUE),2)
X1976 X1977 X1978 X1981
 3.74  3.98  3.72  3.33
```

Not all functions behave this way. The function **sum** returns a single number when given a matrix because it is not vectorized the way mean is.

```
> sum(so4[,4:7],na.rm=TRUE)
[1] 617.1
```

To force the sum function to return separate sums for each column, use the **apply** function. The **apply** function has three required arguments:

1. The first argument is a matrix to operate on.
2. The second argument is the number 1 if we wish to operate on the rows of the matrix or the number 2 if we wish to operate on columns.
3. The last argument is the function we wish to apply to the matrix.

The **sum** function behaves like the **mean** function when it encounters missing values.

```
> apply(so4[,4:7],2,sum)
X1976 X1977 X1978 X1981
   NA    NA    NA    NA
```

We need to pass the **na.rm=TRUE** argument to sum in order to first remove the missing values. This can be done by specifying it as an additional argument to **apply**.

```
> apply(so4[,4:7],2,sum,na.rm=TRUE)
X1976 X1977 X1978 X1981
172.2 127.3 170.9 146.7
```

An alternative approach is to create what's called a generic function. This is a function that's created on the fly. A generic function begins with the key word function followed by parentheses with the variable for the function inside the parentheses. This is followed by a formula using that variable. Here's how we would write a generic function within the **apply** function that takes the sums of non-missing values.

```
> apply(so4[,4:7],2,function(x) sum(x,na.rm=TRUE))
X1976 X1977 X1978 X1981
172.2 127.3 170.9 146.7
```

4.G. Changing the Form of a Data Set from Wide to Long

Currently the so4 data frame has the sulfate concentrations spread across four columns. Each column corresponds to a different year of sampling. While this was a convenient way to enter the data, it is not a format that is useful for statistical analysis. For example, we might want to plot the sulfate concentrations for each year, which means we want to treat the concentration as the *y*-variable and year as the *x*-variable. For this we need the four columns of sulfate concentrations stacked in a single column and there should be a second column that records the year in which the sulfate concentration was measured. The lake, latitude, and longitude columns would need to be adjusted accordingly.

As you might expect, because moving between the two formats is a fairly common task, there is an R function that does this. It's called **reshape**. Rather than illustrate this function I'm going to show you how this can be done using the more primitive functions **unlist** and **rep** because these functions are very useful in their own right.
Stacking the four columns of sulfate concentrations in a single column is easily done with the **unlist** function.

```
> unlist(so4[,4:7])
 X19761  X19762  X19763  X19764  X19765  X19766  X19767  X19768 
    6.5     5.5     4.8     7.4     3.7     1.8     2.7     3.8 
 X19769 X197610 X197611 X197612 X197613 X197614 X197615 X197616 
    8.4     1.6     2.5     3.2     4.6     7.6     1.6     1.5 
X197617 X197618 X197619 X197620 X197621 X197622 X197623 X197624 
    1.4     4.6     5.8     1.5     4.0     5.1      NA     1.4 
X197625 X197626 X197627 X197628 X197629 X197630 X197631 X197632 
    3.8     5.1     2.8     1.6     1.5     3.2     2.8     3.0 
X197633 X197634 X197635 X197636 X197637 X197638 X197639 X197640 
    0.7     3.1     2.1     3.9     1.9     5.2     5.3     2.9 
X197641 X197642 X197643 X197644 X197645 X197646 X197647 X197648 
    1.6    13.0     5.5     2.8     1.6     2.0     5.8      NA 
 X19771  X19772  X19773  X19774  X19775  X19776  X19777  X19778 
     NA      NA     6.5     7.6     4.2      NA     2.7     3.7 
 X19779 X197710 X197711 X197712 X197713 X197714 X197715 X197716 
    9.1     2.6     2.7      NA      NA     9.1     2.4     1.3 
X197717 X197718 X197719 X197720 X197721 X197722 X197723 X197724 
    1.6      NA     6.2     1.6     3.9     5.7      NA     1.0 
X197725 X197726 X197727 X197728 X197729 X197730 X197731 X197732 
    3.3     5.8     3.2      NA     1.5      NA     1.7     1.9 
X197733 X197734 X197735 X197736 X197737 X197738 X197739 X197740 
    1.8      NA     1.9     1.5     1.9      NA      NA      NA 
X197741 X197742 X197743 X197744 X197745 X197746 X197747 X197748 
    1.5    15.0     5.9      NA     1.6      NA     6.9      NA 
 X19781  X19782  X19783  X19784  X19785  X19786  X19787  X19788 
    7.3     6.2     4.6     6.8     3.3     1.5     2.3     3.6 
 X19789 X197810 X197811 X197812 X197813 X197814 X197815 X197816 
    8.8     1.8     2.8     2.7     4.9     9.6     2.6     1.9 
X197817 X197818 X197819 X197820 X197821 X197822 X197823 X197824 
    1.8     5.3     5.9      NA     4.9     5.4     1.4     1.1 
X197825 X197826 X197827 X197828 X197829 X197830 X197831 X197832 
    3.1     5.0     1.6      NA     1.4     2.6     1.9     1.5 
X197833 X197834 X197835 X197836 X197837 X197838 X197839 X197840 
```

```
    1.5     2.4     1.3     1.7     1.5     5.6     5.4     2.9 
X197841 X197842 X197843 X197844 X197845 X197846 X197847 X197848 
    1.7    13.0     5.7     2.6     1.4     2.4     5.9     2.3 
 X19811  X19812  X19813  X19814  X19815  X19816  X19817  X19818 
    6.0     4.8     3.6     5.6     2.9     1.8     2.1     3.8 
 X19819 X198110 X198111 X198112 X198113 X198114 X198115 X198116 
    8.7     1.5     2.9     2.9     4.9     7.6     2.0     1.7 
X198117 X198118 X198119 X198120 X198121 X198122 X198123 X198124 
    1.8     4.2     5.4      NA     4.3     4.3     1.3     1.2 
X198125 X198126 X198127 X198128 X198129 X198130 X198131 X198132 
     NA     4.2      NA     1.6     1.6     2.3     1.8     1.7 
X198133 X198134 X198135 X198136 X198137 X198138 X198139 X198140 
    1.5     2.2     1.6      NA     1.7     3.9     4.2     2.2 
X198141 X198142 X198143 X198144 X198145 X198146 X198147 X198148 
    1.9    10.0     4.8     3.0     1.8     2.0     5.8     1.6 
```

From the output and the labels R has created we see that the values are arranged by year with all the 1976 values coming first, followed by the 1977 values, etc.

To generate the year labels and to reorganize the lake, latitude, and longitude columns we use the **rep** function. I start by illustrating how **rep** works for some simple examples. The **rep** function takes two arguments.

1. The first argument is the object to replicate. It can be a scalar or a vector.
2. The second argument describes how many times it should be repeated. This value can be a scalar or a vector and the choice yields very different consequences, as I'll now illustrate.

4.H. Examples on Use of the rep function

Repeat the number 5 ten times:

```
> rep(5,10)
 [1] 5 5 5 5 5 5 5 5 5 5
```

Create the sequence 2,5,2,5, ..., 2,5 where 2,5 appears ten times:

```
> rep(c(2,5),10)
 [1] 2 5 2 5 2 5 2 5 2 5 2 5 2 5 2 5 2 5 2 5
```

Create a sequence of ten 2s followed by ten 5s:

```
> rep(c(2,5),c(10,10))
 [1] 2 2 2 2 2 2 2 2 2 2 5 5 5 5 5 5 5 5 5 5
```

This can also be accomplished by nesting a rep within a rep:

```
> rep(c(2,5),rep(10,2))
 [1] 2 2 2 2 2 2 2 2 2 2 5 5 5 5 5 5 5 5 5 5
```

4.I. …Back to Creating the Long Data Set

For years we need 48 values of 1976, followed by 48 copies of 1977, etc. This is the third example of using rep given above.

```
> rep(c(1976:1978, 1981), rep(48, 4))
  [1] 1976 1976 1976 1976 1976 1976 1976 1976 1976 1976 1976 1976 1976
 [14] 1976 1976 1976 1976 1976 1976 1976 1976 1976 1976 1976 1976 1976
 [27] 1976 1976 1976 1976 1976 1976 1976 1976 1976 1976 1976 1976 1976
 [40] 1976 1976 1976 1976 1976 1976 1976 1976 1976 1977 1977 1977 1977
 [53] 1977 1977 1977 1977 1977 1977 1977 1977 1977 1977 1977 1977 1977
 [66] 1977 1977 1977 1977 1977 1977 1977 1977 1977 1977 1977 1977 1977
 [79] 1977 1977 1977 1977 1977 1977 1977 1977 1977 1977 1977 1977 1977
 [92] 1977 1977 1977 1977 1977 1978 1978 1978 1978 1978 1978 1978 1978
[105] 1978 1978 1978 1978 1978 1978 1978 1978 1978 1978 1978 1978 1978
[118] 1978 1978 1978 1978 1978 1978 1978 1978 1978 1978 1978 1978 1978
[131] 1978 1978 1978 1978 1978 1978 1978 1978 1978 1978 1978 1978 1978
[144] 1978 1981 1981 1981 1981 1981 1981 1981 1981 1981 1981 1981 1981
[157] 1981 1981 1981 1981 1981 1981 1981 1981 1981 1981 1981 1981 1981
[170] 1981 1981 1981 1981 1981 1981 1981 1981 1981 1981 1981 1981 1981
[183] 1981 1981 1981 1981 1981 1981 1981 1981 1981 1981
```

For lakes we need the entire sequence of lake values repeated four times. (Latitude and longitude should be handled the same way.) This is the second example illustrating the use of rep above.

```
> rep(so4$Lake,4)
  [1] 1    2    4    5    6    7    8    9    10   11   12   13   15
 [14] 17   18   19   20   21   24   26   30   32   34-1 36   38   40
 [27] 41   42   43   46   47   49   50   57   58   59   65   80   81
 [40] 82   83   85   86   87   88   89   94   95-1 1    2    4    5
 [53] 6    7    8    9    10   11   12   13   15   17   18   19   20
 [66] 21   24   26   30   32   34-1 36   38   40   41   42   43   46
 [79] 47   49   50   57   58   59   65   80   81   82   83   85   86
 [92] 87   88   89   94   95-1 1    2    4    5    6    7    8    9
[105] 10   11   12   13   15   17   18   19   20   21   24   26   30
[118] 32   34-1 36   38   40   41   42   43   46   47   49   50   57
[131] 58   59   65   80   81   82   83   85   86   87   88   89   94
[144] 95-1 1    2    4    5    6    7    8    9    10   11   12   13
[157] 15   17   18   19   20   21   24   26   30   32   34-1 36   38
[170] 40   41   42   43   46   47   49   50   57   58   59   65   80
[183] 81   82   83   85   86   87   88   89   94   95-1
48 Levels: 1 10 11 12 13 15 17 18 19 2 20 21 24 26 30 32 34-1 ... 95-1
```

To assemble the individual columns into a new data frame we use the **data.frame** function. I enter things in the order concentrations, years, lakes, latitude, longitude.

```
> temp2 <- data.frame(unlist(so4[,4:7]), rep(c(1976:1978, 1981),
rep(48,4)), rep(so4$Lake, 4), rep(so4$Latitude, 4),
rep(so4$Longitude, 4))
```

By default the columns are named with the formulas we used to create each column. To change the names to something sensible use the **colnames** function again but this time by assigning new values for the column names. Also change the row names to just the sequential row numbers.

```
> colnames(temp2)<-c('SO4','Year','Lake','Latitude','Longitude')
> rownames(temp2)<-1:dim(temp2)[1]
```

Examine the first ten rows of the new data frame we've created:

```
> temp2[1:10,]
   SO4 Year Lake Latitude Longitude
1  6.5 1976    1     58.0       7.2
2  5.5 1976    2     58.1       6.3
3  4.8 1976    4     58.5       7.9
4  7.4 1976    5     58.6       8.9
5  3.7 1976    6     58.7       7.6
6  1.8 1976    7     59.1       6.5
7  2.7 1976    8     58.9       7.3
8  3.8 1976    9     59.1       8.5
9  8.4 1976   10     58.9       9.3
10 1.6 1976   11     59.4       6.4
```

Suppose we wanted again to obtain the mean sulfate concentration for all lakes in each year, but using the new data frame in which concentrations and years are in separate columns. Once again a member of the "apply" family comes to the rescue. The **tapply** function is designed to do what's called subset analysis. It requires three arguments.

1. The name of the variable to which a function will be applied.
2. A categorical variable (or list of categorical variables) defining the subsets for which we want separate summaries.
3. The function to use, either a name of a function or a formula.

If we are using a named function that has additional arguments, these arguments can appear as additional arguments to **tapply** and are listed after the function name. The following use of **tapply** attempts to get the mean sulfate concentration in each year.

```
> tapply(temp2$SO4,temp2$Year,mean)
1976 1977 1978 1981
  NA   NA   NA   NA
```

We get a missing mean for each year because there are missing values in each year. We need to strip out the missing values using the **na.rm=TRUE** option of the mean function. Here are two correct ways of getting the results.
Specify **na.rm=TRUE** as a fourth argument to **tapply**.

```
> tapply(temp2$SO4,temp2$Year,mean,na.rm=T)
    1976     1977     1978     1981
3.743478 3.978125 3.715217 3.334091
```

Write a generic function that uses **mean** but in which we specify **na.rm=TRUE** explicitly as an argument.

```
> tapply(temp2$SO4,temp2$Year,function(x) mean(x,na.rm=T))
    1976     1977     1978     1981 
3.743478 3.978125 3.715217 3.334091
```

4.J. Graphing the Data

To compare the distributions of the samples in the four years a nice graphical device is to produce side-by-side boxplots. This can be done with the **boxplot** function in R. The basic syntax is `boxplot(y~x)` where *y* is the variable to be plotted and *x* is the grouping variable. I elect to use list notation (the $ notation) to reference the variables in the data frame. **Fig. 5** shows the result.

```
> boxplot(temp2$SO4~temp2$Year)
```

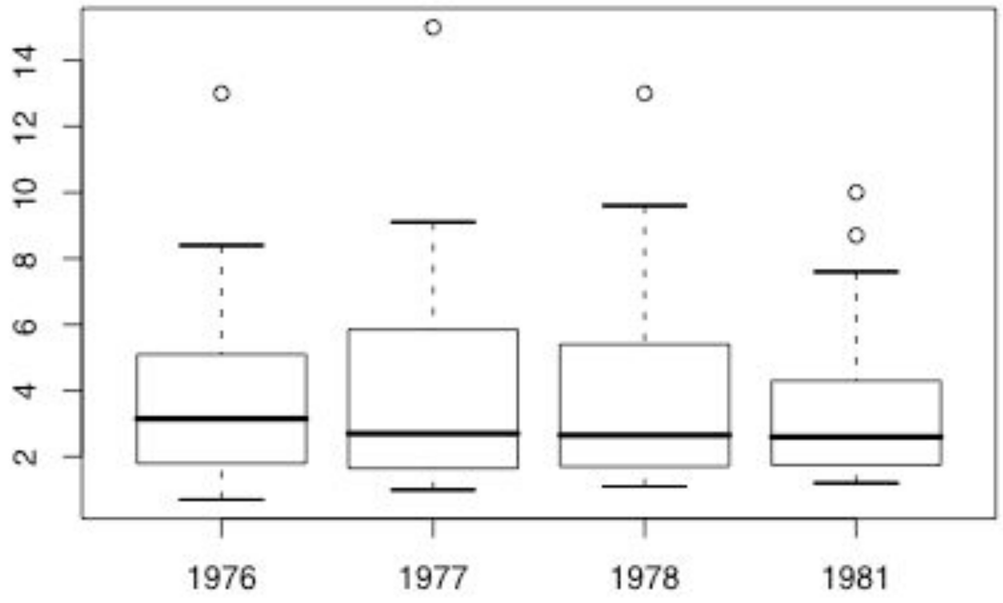


Fig. 5 Box plot of sulfate concentrations by year

The box locates the middle 50% of the data. The bottom edge of the box denotes the 1st quartile, the top edge locates the 3rd quartile, and the horizontal line inside the box corresponds to the median. The distance between the quartiles is called the inter-quartile range (IQR). The "whiskers" run out to the smallest and largest observations inside the inner fences that are located 1.5 times the IQR beyond each quartile. Observations beyond the inner fences are plotted individually and are outliers.

A nice addition to a box plot is to include the location of the mean for each group. The **boxplot** function is an example of a high-level graphics command in R. A high-level graphics function is one that clears the graph window and produces a new plot when it is used. There are many low-level graphics functions that will add elements to existing plots. One of these functions is **points**.

The **points** function does not use the ~ notation. Instead the first argument is a list of *x*-coordinates and the second argument is the corresponding list of *y*-coordinates. Additional arguments can follow, including the following:

1. **pch=** followed by a number that denotes the desired print character.

2. **col=** followed by the number 1 through 8 to denote one of R's base colors, or one of the more than 600 colors available in R that can be specified by name. To see a full list of colors go to http://research.stowers-institute.org/efg/R/Color/Chart/index.htm.

3. **cex=** followed by a number that indicates the character expansion ratio. A number like 0.5 would plot symbols at one half the default size.

The years are actually plotted at locations 1:4 in the plot (not the actual year values as is obvious from the fact that the years are equally spaced in Fig. 5). The means for each year were obtained using the **tapply** function above. I add a red asterisk to each box to denote the mean.

```
> points(1:4,tapply(temp2$SO4, temp2$Year, function(x)
mean(x,na.rm=T)), pch=8, col=2, cex=1.1)
```

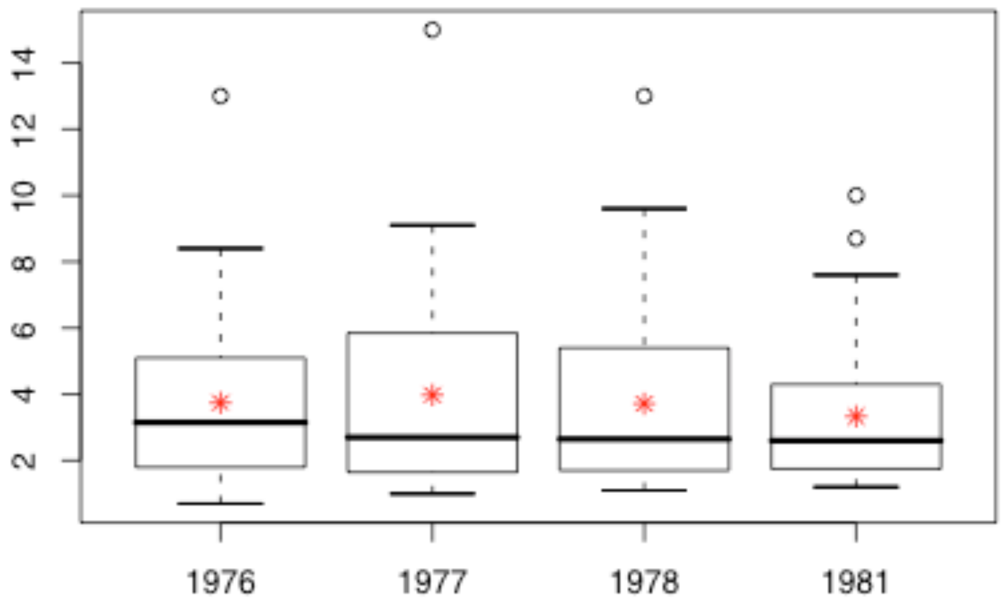


Fig. 6 Adding the group means to the box plots

If there aren't too many data values it can be useful to superimpose the raw data on top of the box plots. Here we have 48 observations per year, which may be a bit much, but I'll go ahead anyway.

The raw data can be added with another **points** call this time specifying the actual *x*-locations on the boxplot as the *x*-values and temp2$SO4 as the *y*-values. The locations are 1:4 and we will need to repeat each of them 48 times. The years are in consecutive order so `rep(1:4,rep(48,4))` will correctly plot the concentrations at the proper years. Here's what I have in mind.

```
> points(rep(1:4, rep(48,4)), temp2$SO4, pch=16, col='seagreen',
cex=.5)
```

A problem with the above command is that data values all end up along the midline of the boxes with many of them overlapping. We can get around this by jittering the points randomly in the *x*-direction by applying the jitter function to the *x*-coordinates as shown below.

```
> points(jitter(rep(1:4,rep(48,4))), temp2$SO4, pch=16,
col='seagreen', cex=.5)
```

One problem with this plot (left side of Fig. 7) is that the outliers are being plotted twice, once from the **boxplot** command and then a second time in the **points** command. Because they are jittered the second time the two versions don't coincide on the plot. There is an option in the boxplot function, **outline=FALSE**, that turns off the printing of the outliers. I rerun the **boxplot** command and the two points commands to produce the plot shown in the right half of Fig. 7.

```
> boxplot(temp2$SO4~temp2$Year, outline=FALSE)
> points(1:4,tapply(temp2$SO4,temp2$Year, function(x)
mean(x,na.rm=T)), pch=8, col=2, cex=1.1)
> points(jitter(rep(1:4,rep(48,4))), temp2$SO4, pch=16, col='seagreen', cex=.5)
```

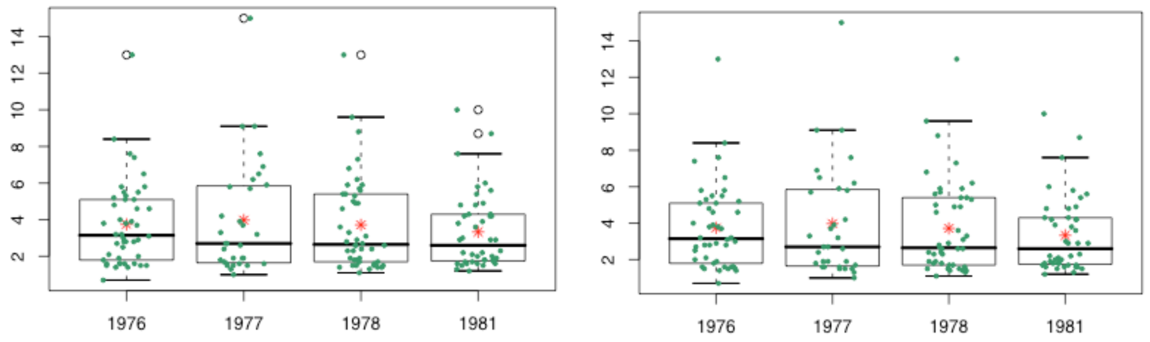


Fig. 7 Adding the jittered raw data to the plot. In the right graph the option outline=FALSE *was used in boxplot to suppress the plotting of outliers*

Finally it would be nice to have a label on the *y*-axis to indicate that SO_4 concentrations are being plotted. Labels for *y*- and *x*-axes are specified with the ylab and xlab arguments to boxplot. R supports mathematical typesetting and we can specify SO_4 in the label as follows: ylab=expression("SO"[4]) . Here's the final boxplot call.

```
> boxplot(temp2$SO4~ temp2$Year, outline=FALSE,
ylab=expression("SO"[4]))
> points(1:4,tapply(temp2$SO4, temp2$Year, function(x)
mean(x,na.rm=T)), pch=8, col=2, cex=1.1)
> points(jitter(rep(1:4, rep(48,4))), temp2$SO4, pch=16,
col='seagreen', cex=.5)
```

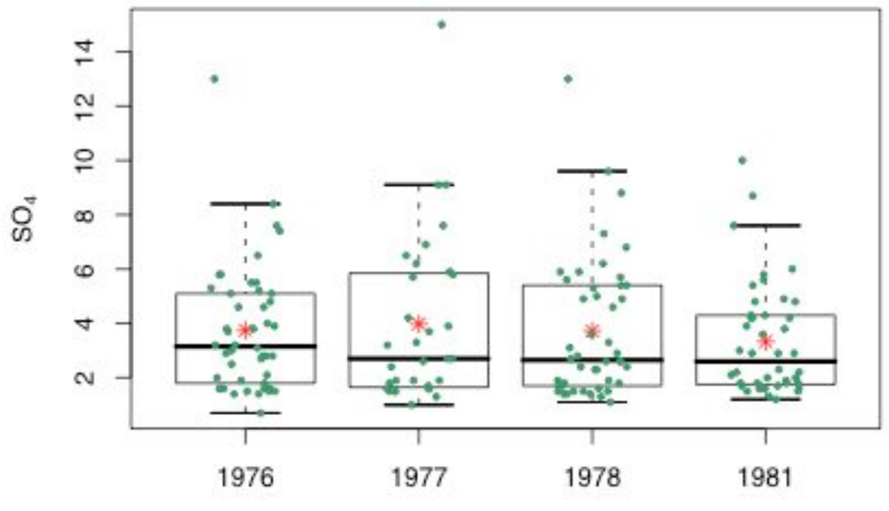


Fig. 8 Final box plot