Installing R

Go the R homepage at `http://cran.us.r-project.org/`. Click on the link to Linux, Mac OS X or Windows, depending on your computer.

Windows Click on the link `base` and then on the link `Download R 2.14.1 for Windows` (or a more recent version). then follow the installation instructions.

Mac OS X Click on the link to the latest version of the software (R-2.14.1.pkg as of this writing), download the file, double-click on the resulting icon, and follow onscreen instructions.

Basics

Prompts. When you start R, the first window that pops up is a console window with a prompt `>`. You type commands at the prompt, press return, and something happens. If you ever see a prompt `+`, this means that the previous command was incomplete and R is waiting for you to complete it. Most likely, your previous command included a left parenthesis '(' that was not matched by a right one ')'. Type something to complete the command, even if it results in a syntax error, and then continue. You can also press the Esc key one or more times to return to a new prompt.

Output. When R writes out an array of numbers to the screen, it labels each line with the position in the array of the first element of the array between square brackets (for example, [1]). This label is not part of the array.

```
> 100
[1] 100
> 1:100
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
[21] 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
[41] 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60
[61] 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80
[81] 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
```

Changing your workspace. R keeps all of the variables you keep in memory as it runs. When you end an R session, you may save your workspace. This allows you to have variables you have previously defined available without the need to create them all again from scratch. There will also be times when you will want to read in data sets or read some R code. To do these things more easily, it is often a good idea to change R's working directory.

By default, R will use as its working directory the folder in which the executable program exists. You will most likely want to change the working directory to a new folder where you might keep data from the textbook and your homework. You change the working directory for R under both Windows and Macintosh versions by using the `File` menu and selecting `Change Working Directory...` with your mouse. My advice is to have a folder where you keep work for this course and to lauch R from this folder. If you don't start R from this folder, you can still change the working directory to this folder.

Quitting R. To quit, you can type `q()` on a command line or you can quit through the `File` menu. R will prompt you if you want to save your workspace. Usually, say yes! Say no if, for example, you modified a variable when you didn't mean to, and you don't want this modification to be saved.

Useful shortcuts Try using the "up" and "down" arrows. This will recall previous commands. It is useful when you typed in a long command that included a small typo. You don't have to re-type the whole thing for correcting the typo.

Calculating with numbers. You can use R like a calculator. The `*` symbol stands for multiplication and the `^` symbol stands for exponentiation. The colon operator `:` creates an array of numbers from the first to the second. R has a number of built-in functions such as `mean()`, `sum()` `median()`, `sd()`, `sqrt()`, `log()` and `exp()` that have obvious meaning. Note that `log()` computes the natural (base e) logarithm. Use a second argument to compute the logarithm with a different base, such as `log(1000, 10)` for the logarithm of 1000 in base 10. Try these.

```
> 2 + 2
[1] 4
> 12 * 3 - 10/2 + sqrt(16)
[1] 35
> 3^2
[1] 9
> 1:10
[1] 1  2  3  4  5  6  7  8  9 10
> sum(1:10)
[1] 55
> mean(1:10)
[1] 5.5
> sd(1:10)
[1] 3.027650
```

Calculating with arrays. R can do arithmetic operations on arrays. If you multiply an array of numbers by a single number, the multiplication happens separately for each number. You can also add or multiply equal-sized arrays of numbers.

```
> 2 * (1:15)
[1] 2  4  6  8 10 12 14 16 18 20 22 24 26 28 30
> (1:10) + (10:1)
[1] 11 11 11 11 11 11 11 11 11 11
> (1:4)^2
[1] 1  4  9 16
```

Assigning variables. You can use the `=` sign to create new variables. Typing the name of a variable displays it.

```
> a = 1:10
```

```
> a
[1] 1  2  3  4  5  6  7  8  9 10
> mean(a)
[1] 5.5
```

(An alternative to the = syntax is to use the key combination <- which was created to look like an arrow. Older documentation may use this instead of the equal sign, but both are valid methods.)

Entering Data

Entering data directly. The easiest way to enter small data sets is with the function c that *concatenates* numbers (or vectors) together. For example, we could create an object named 'glucose' containing the 31 measures as follows. This data comes from Exercise 2.10 in the third edition of *Statistics for the Life Sciences* by Samuels and Witmer.

```
> glucose = c(81, 85, 93, 93, 99, 76, 75, 84, 78, 84, 81, 82, 89,
   81, 96, 82, 74, 70, 84, 86, 80, 70, 131, 75, 88, 102, 115,
   89, 82, 79, 106)
```

This is useful for very small data sets.

Entering data using a text file. The commands `read.table()` and `read.csv()` can be used to read data from a text file. For these commands to work, the files to be read should be in the working directory for R, or you will need to specify a full path name. It is simplest to change the working directory for R to where the data files are.

The file `cows.txt` contains the cow data. This file is in a plain text file, not a Word or rich-text formatted file, which can be created in Windows using Notepad or on a Mac using Text Edit (or with another program). The first row contains variable names, separated by white space, which are spaces or tabs. Subsequent rows contain the data. Each row must contain the same number of fields, but it is not necessary to line up all of the data into neat columns. The function that reads data into R from a text file in this format is `read.table()`. For historical reasons, the default is to not include a header line, so we add the argument `header=T` (T for true) to let R know that the first line of the file contains a header row with variable names.

```
> cows = read.table("cows.txt", header = T)
> str(cows)
'data.frame': 50 obs. of 11 variables:
$ treatment : Factor w/ 4 levels "control","high",..: 1 1 1 1 1 1 1 1 1 1 ...
$ level : num 0 0 0 0 0 0 0 0 0 0 ...
$ lactation : int 3 3 2 2 2 1 1 1 3 3 ...
$ age : int 49 47 36 33 31 22 34 21 65 61 ...
$ initial.weight: int 1360 1498 1265 1190 1145 1035 1090 960 1495 1439 ...
$ dry : num 15.4 18.8 17.9 18.3 17.3 ...
$ milk : num 45.6 66.2 63 68.4 59.7 ...
$ fat : num 3.88 3.4 3.44 3.42 3.01 2.97 2.99 3.54 2.65 4.04 ...
$ solids : num 8.96 8.44 8.7 8.3 9.04 8.6 8.46 8.78 9.04 8.51 ...
```

```
$ final.weight : int 1442 1565 1315 1285 1182 1043 1030 1057 1520 1300 ...
$ protein : num 3.67 3.03 3.4 3.37 3.61 3.03 3.31 3.48 3.42 3.27 ...
```

(If you see an error that the file is not found, it is probably the case that the file is not in your working directory. You may want to try the `file.choose` method described below.) The function `str()` shows the structure of the data we just read in. Notice that numerical variables and categorical variables (factors) are distinguished. If levels of a categorical variable had been stored as numbers, we would have needed to tell R to reclassify the variable as a factor. R calls a rectangular array of data where rows are observations and columns are variables a *data frame*. The name `cows` for the data fram is arbitrary. You may use any valid variable name (which does not begin with a digit or use characters with other meaning). A data frame is a data matrix, but can include both categorical and numerical variables.

Entering data from an Excel worksheet. Many of us use Excel rather than a plain text editor for entering/manipulating data. To enter data into an Excel spreadsheet for subsequent entry into R, use the first row as a header row with variable names and put the values of each variable in a column. After the data is entered, save the file as a comma-separated-variable file (CSV file, for short). Excel will ask if you really mean to do this and warn you of all of the things you will lose of you do so, but disregard the warning and save the data in this format nevertheless. The resulting file is a plain text file where each field is separated by a comma rather than white space. This file can be read into R using the function `read.csv()`. There is no need with this function to specify `header=T`.

```
> cows = read.csv("cows.csv")
> str(cows)
```

Working with data frames

Access to variables

The operator `$` is used to specify variables within a data frame. For example, we can work with the variable milk by typing `cows$milk`.

```
> cows$milk
[1] 45.552 66.221 63.032 68.421 59.671 44.045 55.153 46.957 63.948 65.994 57.603
[12] 63.254 57.053 69.699 71.337 68.276 74.573 66.672 72.237 58.168 48.063 60.412
[23] 45.128 53.759 52.799 76.604 64.536 71.771 59.323 62.484 70.178 48.013 60.140
[34] 56.506 40.245 45.791 59.373 54.281 71.558 56.226 49.543 55.351 64.509 74.430
[45] 68.030 46.888 53.164 53.096 50.471 66.619
> mean(cows$milk)
[1] 59.54314
```

Assuming that this variable is measured in kg/day and that the density of milk is 1.03 kg/liter, we could add a new variable volume to the cow data set equal to the number of liters of milk produced on average each day.

```
> cows$volume = cows$milk/1.03
> str(cows)
```

```
'data.frame': 50 obs. of 12 variables:
$ treatment : Factor w/ 4 levels "control","high",..: 1 1 1 1 1 1 1 1 1 1 ...
$ level : num 0 0 0 0 0 0 0 0 0 0 ...
$ lactation : int 3 3 2 2 2 1 1 1 3 3 ...
$ age : int 49 47 36 33 31 22 34 21 65 61 ...
$ initial.weight: int 1360 1498 1265 1190 1145 1035 1090 960 1495 1439 ...
$ dry : num 15.4 18.8 17.9 18.3 17.3 ...
$ milk : num 45.6 66.2 63 68.4 59.7 ...
$ fat : num 3.88 3.4 3.44 3.42 3.01 2.97 2.99 3.54 2.65 4.04 ...
$ solids : num 8.96 8.44 8.7 8.3 9.04 8.6 8.46 8.78 9.04 8.51 ...
$ final.weight : int 1442 1565 1315 1285 1182 1043 1030 1057 1520 1300 ...
$ protein : num 3.67 3.03 3.4 3.37 3.61 3.03 3.31 3.48 3.42 3.27 ...
$ volume : num 44.2 64.3 61.2 66.4 57.9 ...
```

Subsets

It is frequently useful to partition data into smaller groups, often on the basis of the levels of a categorical variable. For example, with the cows data, we may want to calculate the mean protein level for cows by treatment group. In R, we can get subsets of a data frame using the square brackets [and]. It may help you to think of the square brackets as a verbal such that. For example, to display the protein numbers for all cows in the control group, we can do the following

```
> cows$protein[cows$treatment == "control"]
[1] 3.67 3.03 3.40 3.37 3.61 3.03 3.31 3.48 3.42 3.27 3.31 3.32
```

which you can think of as listing the protein data for all cows such that the treatment group is control. Note that two equal signs without a space == is a comparison operator (answer True or False for each comparison) and that a single equal sign will not work. Use = for variable assignment and when specifying arguments in functions and == when asking if two items are equal to each other. The array `cows$treatment=="control"` has length 50 (the length of the `cows$treatment` variable) and the values are True and False. Inside the square brackets, only those elements corresponding to True are retained.

```
> cows$treatment == "control"
[1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE FALSE
[14] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[27] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[40] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

We could find the mean of each group in turn.

```
> mean(cows$protein[cows$treatment == "control"])
[1] 3.351667
> mean(cows$protein[cows$treatment == "low"])
[1] 3.378462
> mean(cows$protein[cows$treatment == "medium"])
[1] 3.242308
> mean(cows$protein[cows$treatment == "high"])
[1] 3.341667
```

There is a shortcut using the functions `split()` which partitions a variable into a list for each level of a factor and `sapply()` which applies a function to each element of a list.

```
> sapply(split(cows$protein, cows$treatment), mean)
 control     high      low   medium
3.351667 3.341667 3.378462 3.242308
```

Notice that the ordering of the levels of treatment is alphabetical. Here, it makes sense to order by level. The `reorder()` function in the lattice package can be used for this purpose.

```
> cows$treatment = reorder(cows$treatment, cows$level)
> sapply(split(cows$protein, cows$treatment), mean)
 control      low   medium     high
3.351667 3.378462 3.242308 3.341667
```

The first argument to `reorder()` is the factor whose levels should be reordered, the second argument is a quantitative variable of the same length as the first argument. The new order is from lowest to highest mean value for each level of the factor. The square brackets can also be used to find subsets of a data frame. Here, the command has the form data frame[row subset,column subset]. For example, to show columns 1, 7, and 11 for the first five cows, we could do the following.

```
> cows[1:5, c(1, 7, 11)]
treatment milk protein
1 control 45.552 3.67
2 control 66.221 3.03
3 control 63.032 3.40
4 control 68.421 3.37
5 control 59.671 3.61
```

Plotting data

To produce **barplots** and **mosaic** plots, use function `plot()` when applied to a table, or `barplot()` or `mosaicplot()`. See examples below on the recombination data.

```
recomb = matrix( c(114,226,202,102),  2, 2)
colnames(recomb) = c("normal","miniature")
rownames(recomb) = c("red","white")
barplot(recomb, beside=TRUE, legend.text = rownames(recomb), col=c("orangered", "white"), ylim
barplot(recomb, beside=FALSE, col=c("orangered","white"))
mosaicplot(t(recomb), col=c("orangered", "white"), dir=c("h","v"), main="")
mosaicplot(t(recomb), col=c("orangered", "white"), main="")
```

Now plot the cow's treatment, additive level and lactation variables using barplots and mosaic plots.

To produce **boxplots**, use the command `boxplot()`. It can be used in several ways to get a single boxplot or a number of parallel boxplots.

```
boxplot(cows$fat)
boxplot(cows$fat, horizontal=T)

boxplot(fat ~ treatment, data=cows)
boxplot(fat ~ treatment, data=cows, horizontal=T)
boxplot(fat ~ treatment, data=cows, horizontal=T, las=1)
boxplot(fat ~ treatment, data=cows, horizontal=T, las=1,
        col=c("azure1","azure2","azure3","azure4"))
```

Now use boxplots to display the distribution of protein percent from each treatment group, and to display the distribution of age (in months) from each lactation group.

Scatter plots are obtained with the function `plot()`, which can also be used in different ways.

```
layout(matrix(1:2,1,2))
plot(milk~initial.weight, data=cows)   # 2 first plots
plot(cows$initial.weight, cows$milk)   # are the same


plot(milk~initial.weight, data=cows, col=treatment)
legend("topleft",pch=1,col=1:4, legend=levels(cows$treatment))

# next plot: to experiment with different points and
# different shades of grey.
plot(0:20, 0:20, col=gray( (0:20) /20), pch=16)
plot(1:21, 1:21, col=gray( (0:20) /20), pch=1:21)
```