Tutorials

by William B. King, Ph.D.
Coastal Carolina University

I think, therefore I R.

SIMPLE DATA ENTRY AND DESCRIPTION

A Couple Tips

One reason people don't like command line programs is because, if you make a mistake in typing a long command, you have to start all over from scratch. Not so in R. Suppose you were trying to set your working directory to "Rspace", and you accidently typed this...

```
> setwf("Rspace")                    # Type this into R.
```

There is no "setwf( )" function in R, and R will cheerfully tell you the function was not found. Go ahead and see for yourself. Now, instead of retyping the whole line, R will allow you to recall it to the command line and edit it. To recall the previously typed command, just press the up arrow key. You can then use the right and left arrow keys to move through the command line. The Backspace and Delete keys (on a Windows keyboard) can be used to erase the errors. Then make corrections and press Enter. Your cursor does not even need to be at the end of the line when you press Enter. Try it...

```
> ### Press up arrow key here...
> setwd("Rspace")                # Edit command using arrow keys, press Enter.
> getwd()
[1] "C:/Documents and Settings/kingw/My Documents/Rspace"
```

If you continue pressing the up arrow key, R will bring older and older commands to the command line. Thus, if you did something five commands ago, and you want to do it again, press the up arrow key five times to recall the command, then press Enter.

Here's another tip, and one you might be a bit miffed I didn't tell you earlier. You can copy and paste stuff into R. For example, suppose I told you to execute the following command...

```
> boxplot(log(islands), main="Boxplot of Islands", ylab="log(land area)")
```

You're saying to yourself, "Oh man! I don't want to type all that, and I'm gonna get commas in the wrong place, and come on!" You don't have to type it. With your mouse--yes, that's right, your mouse!--highlight the line on this webpage (not including the command prompt or > symbol). Then either go to the Edit menu of your browser and choose Copy, or press Ctrl-C on your keyboard (hold down the Ctrl key and press c and then release both). Now, go into the R Console window and either pull down the Edit menu (in Windows) and choose Paste, or (with the cursor at a command prompt) press Ctrl-V. Either one will paste the command at the command prompt. Then press Enter.

Note to my Mac friends: On the Mac keyboard the shortcuts for copy and paste are Command-C and Command-V, respectively. On older Mac keyboards, the Command key is the one to the left of the space bar with the little flowery thing on it.

Now you know. Of course, you will have to type your own command eventually.

Creating a Vector

Using built-in data objects is fine and dandy for demonstration purposes, but eventually you're going to want to enter and analyze your own data. If the data set is small, you can do this easily from within R. The following data were collected by a student doing his senior research project here at CCU. The numbers represent number of items recalled correctly on a digit span task, supposedly a measure of short term memory. The explanatory variable ("IV") was whether or not the subject admitted to regularly smoking marijuana.

```
smokers     16 20 14 21 20 18 13 15 17 18
nonsmokers  18 22 21 17 20 17 23 20 22 21
```

It might seem a little silly to go to the trouble of formally entering such a small data set into a data frame or a spreadsheet and then reading it into R, when the whole thing can be typed into an R console session in just a few seconds. The thing you

need to realize is that all these scores are ON THE SAME VARIABLE, the response variable, and therefore, they need to go into the same data object or vector. So...

```
> scores = c(16,20,14,21,20,18,13,15,17,18,18,22,21,17,20,17,23,20,22,21)
> scores
 [1] 16 20 14 21 20 18 13 15 17 18 18 22 21 17 20 17 23 20 22 21
> summary(scores)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  13.00   17.00   19.00   18.65   21.00   23.00
```

The scores have been entered into a vector using the `c( )` function. Since that was an assignment statement, it wrote nothing to the screen. Then we asked to see the scores, a good check, since (confession) it took me three tries to get the scores typed in correctly. (ALWAYS double check your data entry!) Then the `summary( )` function was used to produce a preliminary descriptive summary.

That's probably the most annoying way to get data into a vector--all those commas! So here is a more convenient way when typing data at the keyboard. First, remove the "scores" vector. Then recreate it using `scan( )`. The `scan( )` function allows you to type in numbers one at a time, hitting Enter after each one, rather than putting commas between them...

```
> rm(scores)
> scores <- scan()
1: 16          # press Enter
2: 20          # press Enter
3: 14          # etc.
4: 21
5: 20
6: 18
7: 13
8: 15
9: 17
10: 18
11: 18
12: 22
13: 21
14: 17
15: 20
16: 17
17: 23
18: 20
19: 22
20: 21
21:            # press Enter here to end data input
Read 20 items
> scores
 [1] 16 20 14 21 20 18 13 15 17 18 18 22 21 17 20 17 23 20 22 21
```

This is handy when you're using a numeric keypad. But it gets better. You don't have to hit the Enter key between each data value. You only have to leave some white space...

```
> rm(scores)
> scan() -> scores
1: 16 20 14 21 20 18 13 15
9: 17 18 18 22 21 17 20 17 23 20
19:  22 21
21:
Read 20 items
> scores
 [1] 16 20 14 21 20 18 13 15 17 18 18 22 21 17 20 17 23 20 22 21
```

The Enter key can be hit at any time to start a new line. Items entered into `scan( )` must be separated by white space: a space or spaces, a tab, a newline, a carriage return. Notice also that it doesn't matter whether left or right arrow assignment is used. Better still, you can copy and paste the numbers from this webpage...

```
> rm(scores)
> scores = scan()                     # The = assignment can also be used.
1: 16 20 14 21 20 18 13 15 17 18      # Copied and pasted from above.
11: 18 22 21 17 20 17 23 20 22 21     # Copied and pasted from above.
21:                                   # Remember to hit Enter to end entry.
Read 20 items
> scores
 [1] 16 20 14 21 20 18 13 15 17 18 18 22 21 17 20 17 23 20 22 21
```

You can also copy and paste comma separated values, but not into the `scan( )` function. Copy comma separated values into `c( )`. However, you can copy and paste a spreadsheet column (but not a row) into the `scan( )` function.

Now, about that summary--what we want, of course, is a summary by groups, and not of all the scores at once. You can probably think of one way to this...

```
> summary(scores[1:10])                # Summarize scores 1 to 10.
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  13.00   15.25   17.50   17.20   19.50   21.00
> summary(scores[11:20])               # Summarize scores 11 to 20.
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  17.00   18.50   20.50   20.10   21.75   23.00
```

Another way to do it is to create a second vector with group names (i.e., values of the explanatory variable) in it and to use that to extract scores by group...

```
> rep(c("smoker","nonsmoker"),c(10,10)) -> groups
> tapply(X=scores, IND=groups, FUN=summary)      # Similar to by() function.
$nonsmoker
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  17.00   18.50   20.50   20.10   21.75   23.00

$smoker
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  13.00   15.25   17.50   17.20   19.50   21.00
```

The syntax of the `tapply( )` function can be put into words like this: "Apply the summary function to scores by groups." The `by( )` function does something similar, but the output format is a bit different...

```
> by(data=scores, IND=groups, FUN=summary)
groups: nonsmoker
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  17.00   18.50   20.50   20.10   21.75   23.00
------------------------------------------------------------
groups: smoker
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  13.00   15.25   17.50   17.20   19.50   21.00
```

Now might be a good time to mention this. The `summary( )` function is very versatile, and it's output will depend upon what you are asking for a summary of, as we will have ample opportunity to see. When a numerical vector is summarized, the output is the minimum, 1st quartile, median, mean, 3rd quartile, and maximum. There is a qualification. The quartiles are calculated assuming the vector contains a continuous numerical variable. The variable in this example is not continuous. Therefore, the quartiles may not come out to have the same values as if you'd used the method you were taught in elementary statistics to calculate them. We will return to this in a future tutorial. For now, I'll simply say that R can use any of nine different methods to calculate these values.

```
> rm(list=ls())                        # Clean up.
```

Entering Categorical Data

There is no way to get around it. Entering categorical data, or character values, is a pain in the posterior! However, once they are entered, R handles them in a much more versatile way than any other statistical software I have ever used. For example, if you are going to use a categorical variable (entered as character values) in a regression analysis, you do not have to recode. R will do the appropriate recoding for you.

There are some cautions about entering character values that it will be very healthy to know about right up front. Suppose we enter the following vector into R...

```
> gender = c("male","female","female","male ","Male","female","female","mail")
> summary(gender)
   Length     Class      Mode
        8 character character
> table(gender)                    # summary(factor(gender)) will work; try it!
gender
female   mail   male   Male  male
     4      1      1      1      1
```

First we learn that `summary( )` is not so useful for summarizing a character vector. So I used `table( )` instead, which gives me a freqency table. Notice I got what appears to be an unintended result. First, there is a misspelling. R doesn't know you can't spell, so it assumes this is what you (I) intended. There is also a case where "Male" was capitalized, and R, being

case sensitive, counted that as a different value from the uncapitalized "male"s. The one that can really puzzle you is the difference between "male" and "male ". This can be a real mystery, in fact, when you've entered data using another program, like a spreadsheet, and then read it into R. Moral of the story: BE CAREFUL TYPING CHARACTER DATA! If you put a space on the beginning or end of a value, R will assume you mean it to be that way.

Here is another way you can go wrong entering character data...

```
> country = c("England","Russia","United States","England","England")
> table(country)
country
      England         Russia United States
            3              1             1
```

What I am attempting to illustrate is that some data entry methods in R assume that white space separates variable values. So suppose you have a value like United States. There are some cases in which R will read that as two values, "United" and "States". If you are typing values into a vector, the necessary quotes will take care of it. However, it might be a good idea not to put spaces inside data values. You can type a period into what would otherwise be a space, "United.States", and that will never cause a problem.

Now let's use `scan( )` to enter the same values...

```
> rm(country)
> country = scan(what="character")
1: England
2: Russia
3: United States
5: England
6: England
7:
Read 6 items
> table(country)
country
England  Russia  States  United
      3       1       1       1
```

And there it is! Now you see the problem. Let's do it right...

```
> rm(country)
> country = scan(what="character")
1: England
2: Russia
3: United.States
4: England
5: England
6:
Read 5 items
> table(country)
country
      England         Russia United.States
            3              1             1
```

The default data type for `scan( )` is numeric. Using `scan( )` to enter character data is very convenient because you can avoid typing commas and quotes, but you do have to remember to specify that you are entering character data by using the what= option.

Large data sets, however, will probably be typed into a spreadsheet and then read into R. In this case, you will have to be careful how you tell R the file is formatted. More about that when we get to reading and writing external files.

One more thing about character data...

```
> summary(country)
   Length     Class      Mode
        5 character character
> country = factor(country)
> summary(country)
      England         Russia United.States
            3              1             1
```

Until you declare your entered vector to be a factor, R will consider it character data. Sometimes that is what you want, but usually not. If you mean it to be a factor, use `factor( )` to declare it as such.

```
> rm(list=ls())                       # Clean up.
```

Entering Tabled Data

Sometimes you have data that someone has already done the work of putting into a table for you. (This happens especially with problems out of a textbook.) The following data occur in "*A Handbook of Small Data Sets*" by Hand et al. (1994)...

```
24. Snoring and heart disease (on page 18 of Hand et al.)

Norton, P.G. and Dunn, E.V. (1985) Snoring as a risk factor for disease: an
epidemiological survey. British Medical Journal, 291, 630-632.
```

Heart disease	Non-snorers	Occasional snorers	Snore nearly every night	Snore every night
yes	24	35	21	30
no	1355	603	192	224

These data can be entered into a matrix, an array, or a table. I prefer to enter them into a table, so that's what I'm going to illustrate here, along with a few pointers for making things look a little neater when R prints it out...

```
> row1 = c(24,35,21,30)
> row2 = c(1355,603,192,224)
> rbind(row1,row2) -> snoring.table
> snoring.table
     [,1] [,2] [,3] [,4]
row1   24   35   21   30
row2 1355  603  192  224
> dimnames(snoring.table) = list("heart.disease" = c("yes","no"),
+                           "snore.status" = c("nonsnorer","occasional",
+                                  "nearly.every.night","every.night"))
> snoring.table
             snore.status
heart.disease nonsnorer occasional nearly.every.night every.night
          yes        24         35                21          30
          no       1355        603               192         224
```

First, I entered the table row by row into separate vectors. Then I used the `rbind( )`, or "row bind", function to bind the rows into a table. (There is also a `cbind( )` function, if you prefer to enter your tables column by column.) Then I added names to the various dimensions of the table, making liberal use of the Enter key and space bar so the screen did not scroll as I was typing. Notice the row names were entered first followed by the column names. The same method would be used to name the dimensions in an array or a matrix. It's worth taking a few minutes to examine the syntax of the `dimnames( )` function. Notice it takes a list of the variable names, and the individual levels of each variable are assigned via vectors typed within the list. Tricky!

I don't like this table, and the reason I don't is because it's customary to put the explanatory variable in the rows and the response variable in the columns of a contingency table (but not required). So I'm going to flip it using the `t( )`, for "transpose", function...

```
> snoring.table = t(snoring.table)
> snoring.table
                    heart.disease
snore.status          yes   no
  nonsnorer            24 1355
  occasional           35  603
  nearly.every.night   21  192
  every.night          30  224
```

Better! Notice also I avoided putting spaces into my variable names. This is a good practice, although since the names had to be quoted anyway in the dimnames command, it is not strictly necessary. Also, you should ignore the fact that I am sometimes using arrow assignment and sometimes = assignment. I'm doing it to illustrate that it usually does not matter which you use.

Now let's look at a few functions for extracting information from this table...

```
> dim(snoring.table)              # no. of rows by no. of columns
```

```
[1] 4 2

> dimnames(snoring.table)         # We already know this, but what the heck?
$snore.status
[1] "nonsnorer"          "occasional"         "nearly.every.night"
[4] "every.night"

$heart.disease
[1] "yes" "no"

> snoring.table[1,]               # Look at row 1.
 yes   no
  24 1355
> snoring.table[,2]               # Look at column 2.
        nonsnorer         occasional nearly.every.night        every.night
             1355                603                192                224
> snoring.table[3,2]              # Look at the entry in row 3 and column 2.
[1] 192

> addmargins(snoring.table)       # Show row and column sums.
                    heart.disease
snore.status         yes   no  Sum
  nonsnorer           24 1355 1379
  occasional          35  603  638
  nearly.every.night  21  192  213
  every.night         30  224  254
  Sum                110 2374 2484

> prop.table(snoring.table, margin=1)  # Get proportions relative to row sums.
                    heart.disease
snore.status                 yes        no
  nonsnorer           0.01740392 0.9825961
  occasional          0.05485893 0.9451411
  nearly.every.night 0.09859155 0.9014085
  every.night         0.11811024 0.8818898

> prop.table(snoring.table, margin=2)  # Get proportions relative to column sums.
                    heart.disease
snore.status                yes         no
  nonsnorer           0.2181818 0.57076664
  occasional          0.3181818 0.25400168
  nearly.every.night 0.1909091 0.08087616
  every.night         0.2727273 0.09435552

> prop.table(snoring.table)            # Get proportions relative to overall sum.
                    heart.disease
snore.status                  yes         no
  nonsnorer           0.009661836 0.54549114
  occasional          0.014090177 0.24275362
  nearly.every.night 0.008454106 0.07729469
  every.night         0.012077295 0.09017713

> chisq.test(snoring.table)            # You were wondering, weren't you?

        Pearson's Chi-squared test

data:  snoring.table
X-squared = 72.7821, df = 3, p-value = 1.082e-15
```

It's also easy enough to turn those proportions into percentages...

```
> prop.table(snoring.table, margin=1)*100
                    heart.disease
snore.status                 yes       no
  nonsnorer           1.740392 98.25961
  occasional          5.485893 94.51411
  nearly.every.night  9.859155 90.14085
```

```
  every.night        11.811024 88.18898
```

Just multiply the entire prop.table by 100. And finally...

```
> rm(list=ls())                       # clean up
```

---

revised 2010 July 29

Return to the Table of Contents