Tutorials

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

I think, therefore I R.

OBJECTS

Data.

Your data is the information upon which you wish to do a statistical analysis. By the way, the word "data" is plural, so ordinarily you would not say "data is" or "data was." Correct are "date are" and "data were." I'm not the grammar police, but I will nail you on that one!

Maintaining a data set is one of the most important things a statistician needs to know how to do. Most statistical software requires that the data set be in a very specific format, called a data table or, in R, a data frame (one word or two, take your pick). Data frames will be covered in detail in a future tutorial.

This is where R truly shines. R is much more flexible in that it does not require that you use the data frame format for your data. If it is more convenient to keep your data in a contingency table, or a list, or a matrix, or a single vector, you can do so. This flexibility has a price--more to learn. In the end, however, it makes R a much more convenient way to analyze data sets, especially simple ones.

In the behavioral and social sciences, the unit of analysis is usually a subject, human or animal. In the more general case, subjects are called "cases" or "observations" or "experimental units." I prefer cases. There will actually come a time when we have to distinguish between subjects and cases, so you should not think of these two terms as being exactly equivalent.

Let's say you've collected data from five subjects: Bob, Fred, Barb, Sue, and Jeff. From each subject you have collected information about age, height, weight, race, year in school (they are all college students), and SAT score. Your cases are Bob, Fred, Barb, Sue, and Jeff. Age, height, weight, race, year in school, and SAT score are called variables. You would ordinarily put this information into a data frame as follows:

```
name     age  hgt  wgt  race year   SAT
Bob       21   70  180  Cauc   Jr  1080
Fred      18   67  156 Af.Am   Fr  1210
Barb      18   64  128 Af.Am   Fr   840
Sue       24   66  118  Cauc   Sr  1340
Jeff      20   72  202 Asian   So   880
```

Notice that the cases, or subjects, go into rows in this table, and each variable has it's own column. This is the standard form for maintaining a data table (data frame). It looks a lot like a spreadsheet, and in fact, using spreadsheet software is a very good way to manage data. The first row in this table is called the header. It contains the variable names. Having a header row is optional but usually a good idea.

I should call your attention to the fact that we have two fundamentally different kinds of variables in this data frame. Some are numbers, like age and weight. These are called numerical variables. Other variables contain just the names of categories that the subject falls into. Race is an example of such a variable, called a categorical variable. It's absolutely essential that you be able to distinquish these two types of variables. You can't do statistics otherwise! Categorical variables are often called factors in R. Just to make matters a bit more confusing, examine the "year" variable. What would you call it, numerical or categorical? If those were your only choices, you'd have to call it categorical. In fact, in this variable the categories have a natural order to them: Fr, So, Jr, Sr. Sometimes such a categorical variable is called an ordered factor in R.

You may be more familiar with the terms nominal, ordinal, interval, and ratio variables. Nominal variables and categorical variables are roughly the same thing. Factors are usually nominal. However, ordered factors are ordinal. Numerical variables are either interval or ratio variables, and it usually doesn't matter which. One more catch to all this--examine the column labeled "name" in the table above. Is this a variable? I suppose it is since its value is different for everyone. Usually when we think of categorical variables or factors, we are thinking of variables that have relatively few possible values. These values are called levels. The levels of year, for example, are Fr, So, Jr, Sr. When a variable has a different value for everyone, like the subject's name or address for example, it's often called a character variable. You will see R make this distinction, and it's a useful one, so remember it.

You get data into R by creating data objects, so let's see how that is done.

Assignment.

In R you create things, called "objects", by a process called assignment. Start an R session and set the working directory to Rspace. Also, clear the workspace...

```
> setwd("Rspace")        # There is a menu item for this in the GUI, btw.
> rm(list=ls())          # Or use the menus to do this.
```

If you don't know what this means or have forgotten to create the Rspace directory, you can find out how in the tutorial called Preliminaries.

There are three ways to assign data to an object name in R (actually four, but one is rarely used). Here is one way...

```
> x = 7
```

This SHOULD NOT be read as "x equals 7", which will result in confusion later. Instead, the single equals sign means "takes the value" or "is assigned the value." R is not usually picky about spacing, so all of the following are equivalent...

```
> x=7
> x = 7
> x=           7
> x            =            7
> x =                                     # Press Enter here.
+ 7                                       # Press Enter again.
```

Use spacing to make your typed input look "pretty." Or not. It's (generally) up to you. There are a few situations where R will get uppity about spacing, but usually it is not an issue. DON'T, however, be so silly as to put a space in the middle of the name of something. That would be bad!

Here is another way to do assignment...

```
> x <- 7
```

And here is one place where R insists on the correct spacing. The "arrow" assignment operator is actually two symbols, a less than sign and a dash or minus (not an underline character no matter what it might look like in your brower). THERE CANNOT BE A SPACE BETWEEN THEM! Why would anyone want to use two symbols instead of one if they do the same thing? You'll see!

Now look at the object called "x" in your workspace...

```
> ls()                                  # the "show me" function
[1] "x"
> x                                     # print out the value of x
[1] 7
```

We will use the third kind of assignment to overwrite this value...

```
> 9 -> x                                # arrow points to the variable name
> x
[1] 9
```

Two things to note here. First, R is perfectly willing to let you be stupid and overwrite things you have in your workspace. There is no warning. If you assign something to an object name that already exists, the old object is gone! Second, the arrow assignment works from either direction. The equal sign does not! When using =, you must give the object name first followed by the value you wish to give it.

Objects.

The following data objects exist in R:

- vectors
- lists
- arrays
- matrices
- tables
- data frames

Some of these are more important than others. And there are more, but these are the ones we need to know about for now. Let's begin at the beginning.

Object and Variable Names.

R doesn't care much what you name things,whether they are variables or complete data objects. As noted in the last tutorial, however, DO NOT put spaces or dashes in your names. Thus, all of these are acceptable (and different) object or variable names:

- x
- X
- x2
- x.2
- x_2
- myData
- MyData
- my_data
- my.data
- my.data.from.the.learning.experiment
- fred
- Fred
- FRED
- Rutherford.B.Hayes

Be creative! But if you make your object names too long, you'll be sorry, because you'll be typing them a lot! Another warning: It is generally safest to confine yourself to letters, numbers, dots, and underline characters and to start your variable names with letters. Also, try to avoid using names that are also functions in R, like "mean" for example, although R will usually work around this. The only names I would warn you against are T and F. Avoid those as variable names because, as we will see later, R uses them to mean true and false. If you assign them another value, that could cause trouble.

Where The Heck Did That Come From?

Remember, R has a large number of built-in data objects. Some of them will be used below to illustrate the various kinds of R data objects. For example, here is a data object containing the lengths of major North American rivers (in miles)...

```
> rivers
  [1]  735  320  325  392  524  450 1459  135  465  600  330  336  280  315
 [15]  870  906  202  329  290 1000  600  505 1450  840 1243  890  350  407
 [29]  286  280  525  720  390  250  327  230  265  850  210  630  260  230
 [43]  360  730  600  306  390  420  291  710  340  217  281  352  259  250
 [57]  470  680  570  350  300  560  900  625  332 2348 1171 3710 2315 2533
 [71]  780  280  410  460  260  255  431  350  760  618  338  981 1306  500
 [85]  696  605  250  411 1054  735  233  435  490  310  460  383  375 1270
 [99]  545  445 1885  380  300  380  377  425  276  210  800  420  350  360
[113]  538 1100 1205  314  237  610  360  540 1038  424  310  300  444  301
[127]  268  620  215  652  900  525  246  360  529  500  720  270  430  671
[141] 1770
```

(The output on your screen may be slightly different, depending upon how wide you have your R Console window set to.)

In this R output, everything is numbered, but only the number of the first item on each output line is printed. Thus, the value 1205 (third line from the bottom three items in--may be different on your screen) is item number 115 in this output. These index numbers are NOT PART OF THE DATA ITSELF! This will be made clearer in the following section. The object "rivers" is a vector, so...

Vectors.

One kind of vector consists of numbers, as was the case just above for the vector "rivers". This is called a numerical vector, cleverly enough. Any item in this vector can be addressed by using its index number...

```
> rivers[115]                          # "show item 115 in vector rivers"
[1] 1205
```

The index number must be enclosed within square brackets. Notice R prints it out as item [1], but within the "rivers" vector it is item [115]. Don't get hung up over this. It happens because R considers this printout also to be a new vector. This can be very useful, as we'll see. It means that, unlike other statistical software, R will allow you to use the output of a command as input for further calculations. (If this isn't working for you, by the way, it probably means that you are using a very old version of R. Try putting a copy of the "rivers" vector in your workspace first: `data(rivers)`. This should make the vector available no matter what.)

If you want to see items 10 through 20 in "rivers" do this...

```
> rivers[10:20]
 [1]  600  330  336  280  315  870  906  202  329  290 1000
```

In R, a colon has two meanings. This is one of them. When two numbers are separated by a colon, it means "to" as in "10 to 20". Try this...

```
> 10:20                               # Output not shown.
```

Since no function is specified to operate on these numbers, R assumes you meant `print(10:20)`. If you want to see items 18, 104, and 168, do this...

```
> rivers[c(18,104,168)]
[1] 329 380  NA
```

"NA" means not available, or missing. The "rivers" vector is only 141 items long, so you just asked for something that doesn't exist. The point is, to see specific items within a vector, enter a vector of index numbers inside the square brackets. You can also use relational operators (about which more later) to pick out certain items from a vector. If you just want to see the data values greater than 500, do this...

```
> rivers[rivers > 500]
 [1]  735  524 1459  600  870  906 1000  600  505 1450  840 1243  890  525  720
[16]  850  630  730  600  710  680  570  560  900  625 2348 1171 3710 2315 2533
[31]  780  760  618  981 1306  696  605 1054  735 1270  545 1885  800  538 1100
[46] 1205  610  540 1038  620  652  900  525  529  720  671 1770
```

I will tell you how to find out which rivers those are in a later tutorial.

To create a vector, use the `c( )` function (short for concatenate, or combine)...

```
> x = c(12, 14, 15, 17, 19, 8, 10)
> x
[1] 12 14 15 17 19  8 10
```

Once again, R isn't picky about spacing. None of the spaces in the above command need to be there. Or you can put more in if you like. I won't mention this again. I assume if you get curious about some special case, you will experiment and find the answer for yourself.

If the values you wish to enter into a vector are consecutive, then this is sufficient:

```
> x = 100:200     # x = c(100:200) also works (but not in older versions of R)
> x
  [1] 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117
 [19] 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135
 [37] 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153
 [55] 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171
 [73] 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189
 [91] 190 191 192 193 194 195 196 197 198 199 200
```

And remember (also the last time I'll mention this), the old "x" has been overwritten, gone, history, is no more, irretrievable!

Vectors can also contain words or character values. When you enter these values, they must be in double or single quotes...

```
> x = c("Bob","Carol","Ted","Alice")
> x
[1] "Bob"   "Carol" "Ted"   "Alice"
```

Two vectors can also be concatenated into one with the concatenate function as follows...

```
> y = c("John","Joy","Fred","Frances")
> z = c(x, y)
> z
[1] "Bob"     "Carol"   "Ted"     "Alice"   "John"    "Joy"     "Fred"
[8] "Frances"
```

What would have happened if, instead, you had done this?

```
> z = c("x", "y")
```

```
> z
```

It's worth finding out, so don't just sit there wondering. Type! One thing I had a bit of trouble getting used to in R is when to put things in quotes and when not to. The basic rule is: If it's an already defined object, don't quote it. If you want to refer to the values inside already existing x and y vectors, don't quote. If it's a new character value (i.e., a string--someone's or something's name), use quotes. R assumes anything not in quotes is an object name (an already defined vector, list, dataframe, etc.), and it will hunt for that object in the search path. If it doesn't find it, you will be told so...

```
> Joy                        # Print out object Joy.
Error: object "Joy" not found
> "Joy"                      # Print out "Joy".
[1] "Joy"
> y[2]                       # Print out the second value in vector y.
[1] "Joy"
> Joy = 6                    # Create a new object named Joy.
> Joy
[1] 6
```

Now do this...

```
> islands                    # Only first four lines of output shown.
          Africa       Antarctica            Asia        Australia
           11506             5500           16988             2968
     Axel Heiberg           Baffin           Banks           Borneo
              16              184              23              280
...
```

This is called a named vector. Here is how to create one.

```
> x = c("Robert Culp","Natalie Wood","Elliott Gould","Dyan Cannon")
> x                          # The values are not named yet.
[1] "Robert Culp"   "Natalie Wood"  "Elliott Gould" "Dyan Cannon"
> names(x) = c("Bob","Carol","Ted","Alice")
> x                          # And now they are.
            Bob           Carol             Ted           Alice
  "Robert Culp"  "Natalie Wood" "Elliott Gould"   "Dyan Cannon"
> x[Alice]                   # This is not correct! Why not?
Error: object "Alice" not found
> x["Alice"]
        Alice
"Dyan Cannon"
> Alice = 4
> x[Alice]                   # Same thing as x[4].
        Alice
"Dyan Cannon"
```

Confusing, right? You'll get used to it. This is a helpful example to study and play around with.

The vector "x" now contains the names of the actors in the movie "Bob and Carol, Ted and Alice." The `names( )` function was used to label these values with the names of the characters they played in the movie. Then we used the name of the character to retrieve the name of the actor. Dyan Cannon could also have been referred to as x[4]. Try it. (I have a very funny story about this movie, but this is not the place for it!)

In the "islands" vector, the data values are the size of the land mass in thousands of square miles. Each data value is named with the name of the land mass. Thus, to retrieve the area of Cuba, we do not need to know which of the data values is Cuba. We can retrieve the value by name. The name is put inside of square brackets just as it if were an index number, and it is quoted...

```
> islands["Cuba"]
Cuba
  43
```

Cuba has a land area of 43,000 square miles. Suppose you wanted to work with this data vector, but you wanted the land areas in square kilometers instead of square miles. The following procedure will allow this. First, use the `data( )` function to write a copy of "islands" to your workspace. Then do the conversion. The converted values can either be stored back into the "islands" vector, in which case the old values are overwritten, or it can be stored into a new vector with a new name...

```
> data(islands)                        # writes a copy to your workspace
> ls()
[1] "Alice"   "islands" "Joy"     "x"       "y"       "z"
> km_islands <- islands * 2.59         # probably the best way
> km_islands["Cuba"]
  Cuba
111.37
> islands <- islands * 2.59            # overwrites the original data values
```

```
> islands["Cuba"]
  Cuba
111.37
```

And finally...

```
> ls()
[1] "Alice"      "islands"    "Joy"        "km_islands" "x"
[6] "y"          "z"
> rm(list=ls())                        # clean up!
> ls()
character(0)
```

Vectors are used a lot in R. You should take some time to understand them.

Lists.

Lists are collections of other R objects collected into one place. To create a list, use the `list( )` function...

```
> x=1:10                          # a vector
> y=matrix(1:12,nrow=3)           # a matrix
> z="Bill"                        # a character variable
> my.list=list(x,y,z)             # create the list
> my.list                         # view the list
[[1]]
 [1]  1  2  3  4  5  6  7  8  9 10

[[2]]
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

[[3]]
[1] "Bill"
```

The output of a lot of R functions is actually composed of lists. Notice that items in a list are indexed by values inside double brackets. Thus...

```
> my.list[[3]]                    # The third item in my.list.
[1] "Bill"
```

To name the items in the list...

```
> names(my.list) = c("my.vector","my.matrix","my.name")
> my.list
$my.vector
 [1]  1  2  3  4  5  6  7  8  9 10

$my.matrix
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

$my.name
[1] "Bill"
```

In R, the $ is used for list indexing. That is, it allows you to pull elements out of lists by name. First type the name of the list, followed by $, followed by the name of the item in the list. For example...

```
> my.list$my.name
[1] "Bill"
```

Kinda trivial in this case, but it won't be when you have a much longer list. That's enough on lists for now.

```
> ls()
[1] "my.list" "x"       "y"       "z"
> rm(my.list,x,y,z)                      # Don't forget to clean up!
```

There is one more thing you should remember about lists. Data frames are actually lists. In fact, this is probably the most important thing you need to remember about lists!

Matrices and Arrays.

Essentially, these are both table-like objects. You saw how to create a matrix in the last section on lists. That's really enough for now. Except maybe for extracting values from one. The syntax is my.matrix[row,col], as follows...

```
> y = matrix(1:16, nrow=4)        # First we need a matrix! With 4 rows.
> class(y)                        # y is an object of class "matrix"
[1] "matrix"
> y
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16
> y[3,2]
[1] 7
```

Remember this! Always put the row index first followed by the column index, and always put the indexes inside of square brackets. The `matrix( )` function, which is used to create a matrix, takes a vector as its argument, and then the option "nrow=" tells how many rows to break the vector into. The matrix is filled "down the columns" first, although there is another option that will change this behavior. Notice our matrix has no row names or column names. The notation [1,] means "row one, all columns". To recall an entire row or an entire column of a matrix (or an array or a table), do this...

```
> y[1,]                                # all values in row 1
[1]  1  5  9 13
> y[,3]                                # all values in column 3
[1]  9 10 11 12
```

More later on matrices, including how to name the rows and columns.

An array is like a matrix, except it can have more than two dimensions. In other words, a matrix is just a two-dimensional array.

```
> y = array(1:16, dim=c(4,2,2))
> y
, , 1

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

, , 2

     [,1] [,2]
[1,]    9   13
[2,]   10   14
[3,]   11   15
[4,]   12   16
```

The `array( )` function creates arrays. The "dim" option gives the number of rows, columns, and layers, in that order. Of course, this would be more useful if we were putting real data into the array rather than just the numbers 1 to 16. It was just a quick example. To put real data into a matrix or an array, simply put the data into a vector, and replace "1:16" with the name of the vector in the `matrix( )` or `array( )` function.

```
> x = c(1.26, 3.89, 4.20, 0.76, 2.22, 6.01, 5.29, 1.93, 3.27)
> y = matric(x, nrow=3)           # Hey! Everybody makes mistakes!
Error: could not find function "matric"
> y = matrix(x, nrow=3)
> y
     [,1] [,2] [,3]
[1,] 1.26 0.76 5.29
[2,] 3.89 2.22 1.93
[3,] 4.20 6.01 3.27
```

Don't forget to clean up.

Tables.

If the function to create a matrix is `matrix( )`, and the function to create an array is `array( )`, I bet you can guess what function is used to create a table. It's used quite a bit differently, however. The `table( )` function is used to create frequency tables or crosstabulations from raw data contained in a vector or a data frame. The result is something that looks, in many cases, very much like a matrix or an array, and behaves very much like one as well. For now, we will confine ourselves to one relatively simple example. First, we have to create some raw data...

```
> y = rnorm(100, mean=100, sd=15)        # 100 normally distributed random nos.
> y = round(y, 0)                        # Rounded to zero decimal places.
```

Once again, don't worry about the syntax of these statements. I'm just using them to create some data to put into a table. Since the values in the y vector are random, everyone's results here will be different. To view a frequency table (badly formatted, but...small steps!), do this...

```
> table(y)
y
 64  69  73  74  77  79  80  81  82  84  85  86  87  88  89  90  91  92  93 
  1   1   1   1   4   4   2   1   1   2   1   1   1   3   1   1   1   2   1 
 94  95  96  97  98  99 100 101 102 103 104 105 106 107 109 110 111 112 113 
  4   4   3   3   5   2   6   3   1   5   4   2   2   2   1   2   1   4   3 
114 116 117 118 119 120 123 125 129 
  2   2   1   1   2   1   1   2   1 
```

The top row of numbers contains the data values, which we can see range from 64 to 129, and the bottom row of numbers gives the frequencies. The data value (i.e., y-value) of 100, for example, occurs 6 times in the data vector. (Once again, your result will be different.) Tables, of course, just like everything else in R, can be stored and then used for further analysis...

```
> table(y) -> myTable             # Store it.
> barplot(myTable)
> ls()
[1] "myTable" "y"
> rm(myTable, y)                  # And remember to clean up.
```

This table is (was!) one-dimensional. The "HairEyeColor" object we were playing with previously was a multidimensional table of frequencies, also called a crosstabulation.

Data Frames.

Data frames are so important that I will devote an entire tutorial just to them. For now, if you want to see one, try this...

```
> women
```

The basic structure of a data frame is illustrated here. It's basically a table (in fact, it's a list of column vectors) in which each variable goes in its own column and each case goes in its own row.

Usually, data frames are read into the R workspace from external files, which may have been created using a spreadsheet. Small ones can be typed in at the command line, however. Let's use the data at the beginning of this tutorial to see how that would work.

```
> myFirstDataframe = data.frame(       # Press Enter to start a new line.
+    name=c("Bob","Fred","Barb","Sue","Jeff"),
+    age=c(21,18,18,24,20), hgt=c(70,67,64,66,72),
+    wgt=c(180,156,128,118,202),
+    race=c("Cauc","Af.Am","Af.Am","Cauc","Asian"),
+    year=c("Jr","Fr","Fr","Sr","So"),
+    SAT=c(1080,1210,840,1340,880))    # End with double close parenthesis. Why?
> myFirstDataframe
  name age hgt wgt  race year  SAT
1  Bob  21  70 180  Cauc   Jr 1080
2 Fred  18  67 156 Af.Am   Fr 1210
3 Barb  18  64 128 Af.Am   Fr  840
4  Sue  24  66 118  Cauc   Sr 1340
5 Jeff  20  72 202 Asian   So  880
```

That's probably not something you're going to want to do too very often! In fact, I'd almost be willing to bet you got at least one comma, one quote, or one parenthesis out of place, and the whole thing failed because of that. There is an easier way.

Last Word.

Further details as needed on these data objects will be covered in future tutorials. For now, you should get the general idea.

---

revised 2010 July 27

Return to Table of Contents