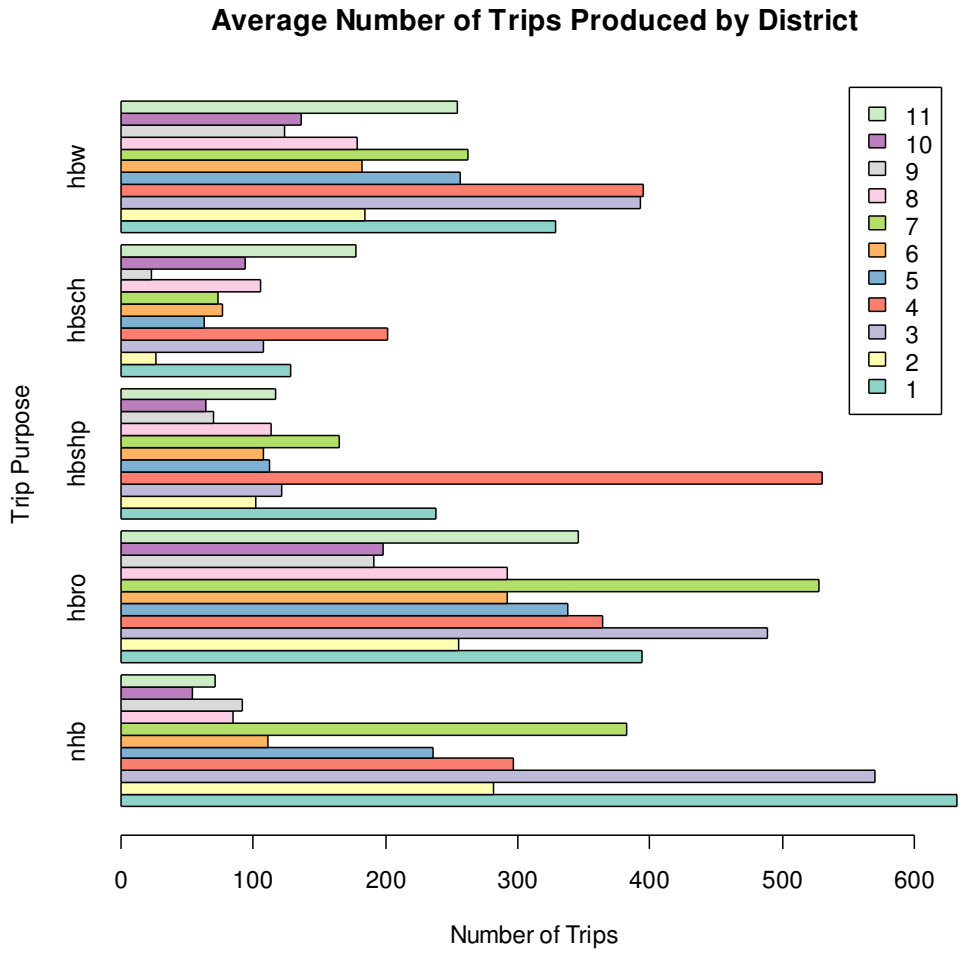
R for Data Analysis and Modeling



Brian Gregor
brian.j.gregor@odot.state.or.us

Ben Stabler
benjamin.stabler@odot.state.or.us

Spring 2003

From: GREGOR Brian J
Sent: Friday, April 18, 2003 10:37 AM
Subject: R Training Course: Week 1, Lesson 1

Hello,

Welcome to the first installment of this email class on learning to use R for data analysis and modeling. Ben Stabler and I will be your instructors. You've either requested or were chosen to receive these emails. Over the next seven weeks, you will receive an email each working day with a short lesson. You should be able to read through each lesson and run a few examples in about fifteen minutes (at least that's our goal).

WEEK #1, LESSON #1: INSTALLING AND SETTING UP R

~~~~~~~ Step 1: R Gateway on the Internet ~~~~~~~
The first thing you should do is go to and bookmark the R Project web site (https://www.r-project.org/). This is your main gateway to R resources. Along the left border of the web page you will see various links about R. The main ones to pay attention to are the download and the documentation links.

The download link is named CRAN. This stands for Comprehensive R Archive Network. If you click on it, you will be shown a list of CRAN mirrors, which are basically servers which have all of the R files. You can click on any one of these, but the idea is to use one that is close by. When you've done this, you will be shown a page that has two sections at the top, one saying "Precompiled Binary Distributions" and the other saying "Source Code for all Platforms". You will want to get the precompiled binary distribution. Before you click on the link, see the notes just below.

****Note**** I apologize to our Linux friends at Metro. I don't know enough about installing Linux binaries to describe the process. I believe you have all the help you need there though.
****Note**** The folks at TPAU don't need to download anything. We have put a copy of the "rw1070.exe" file in the R directory of the 6420only drive.

~~~~~~~ Step 2: Download Binary File ~~~~~~~
When you click on the "Windows (95 and later)" link, you will be shown a page with three links. Click on "base". That will get you to the page with a link to download the base R distribution. If you click on "rw1070.exe", you will then get a dialog box asking whether you want to open or save the file. Click on "Save" and then use the file browser dialog box to point to a convenient location to save the file to (e.g. C:\Temp).

~~~~~~~ Step 3: Install the Program ~~~~~~~
The file you've downloaded is an installer program. It will begin installing R if you double-click its icon (or right-click, open). Note that you can do this yourself because the program does not write to the system registry. (**** Note to TPAUers **** If you have any questions about this, talk to Ben or me.) Here are some notes on the installation dialogs you will see:
1. License: The license you are asked to accept is the GNU General Public License. This is good. It basically says that you are free to use and modify the program as you wish and to distribute it to whomever you wish to. However, if you distribute it, you must also distribute the source code, and if you modify it and distribute it, you must include the source code for your modifications.
~~~~~~~

2. Installation Directory: By default, it will install in the "C:\Program Files\R\rw1062" directory. I suggest you change this to "C:\Program Files\R\rw". I'll explain the reason for this at the end.
3. Component Selection: You are safe going with the default component selection. You could also install the "Source Package Installation Files" if you envision yourself developing packages in the future.
4. Start Menu Folder: I accept the default.
5. Select Additional Tasks: I accept the default.

~~~~~~~ Step 4: Final Setup Steps to Make R Most Usable ~~~~~~~
You can start an R session by double-clicking on the R icon on your desktop or by using the start menu. Before you start using R, though, I have a few suggestions that will make it easier for you to use R.

The first thing to do is to change some of the startup parameters for R. You can do this in the desktop shortcut that was created. If you right-click on the shortcut and then highlight and click on the "properties" menu item, you will get a tabbed dialog box with the properties for the shortcut. Click on the "Shortcut" tab and note the "Target" input box. It should say "C:\Program Files\R\rw\bin\Rgui.exe" if you installed the program according to my suggestions. Put the cursor at the end of this line, add a space and then type "--internet2" (without the quotes). Add another space and then type "--max-mem-size=400000000" (without the quotes). The first addition allows R to connect to the internet. This is necessary if you want to get additional R packages over the internet. The second addition increases the amount of memory that is available to R from the default of 256Mb. This is the setting I use on my computer (400Mb on a 512Mb computer).

The second change involves the "Start in" dialog box. Don't make any changes just yet to this. For now, click the "OK" button. To understand the point of the second change, there's some basic things you should know about working with R. As you work in R, it keeps track of all the objects that you are working with. (Objects are data, functions, etc.) It also keeps track of all the commands you type into R. When you quit R, it saves these as two files named ".RData" and ".RHistory" in the working directory. If you start R now, and type "getwd()" (without the quotes) at the prompt and then press the Enter key, it should reply "C:\\Program Files\\R\\rw". (**** Note **** The reason why the double backslashes has to do with the fact that DOS uses the "escape" symbol to separate directory names. This symbol is used for other purposes as well. For example, "\t" means a tab. So in order to distinguish the use of "\" as a directory separator, it is necessary to type "\\")

I find it helpful to set up a separate folder for each analysis project I'm working on. I paste a copy of the desktop shortcut into that folder. Then I open properties dialog for that copy and type in the path for that folder in the "Start in" input line. Then when I start R using that shortcut, it will set the working directory to that folder. When I quit working, it will save the ".RData" and ".RHistory" files there.

~~~~~~~ Step 5: Goodbye ~~~~~~~
Now here's the reason why I suggested that you rename the installation directory. By renaming the directory, you've made your shortcuts more generic. If you install a new R version in this same directory, all of the shortcuts you've created will continue to work. If you had used the default, then at best the shortcuts would start an older version of R and at worst would not start anything if you deleted the old version.

That should get you up and running for now. On Monday, Ben will lead you around the R interface and explain how you use this program.

Brian Gregor, P.E.
Transportation Planning Analysis Unit
Oregon Department of Transportation
Brian.J.GREGOR@odot.state.or.us
(503) 986-4120

From: GREGOR Brian J
Sent: Friday, April 18, 2003 11:31 AM
Subject: R Lesson

Peter brought to my attention that not everyone in the unit has 512Mb of memory in their computers. If you edit the shortcut as I suggested to increase the memory that R can use, your computer will crash when you start R. (At least that's what happened to Peter.) You can check how much memory you have by right-clicking on "My Computer" and selecting the "Properties" menu item. The amount of memory you have in bytes is at the bottom of the "General" tab. If you don't change the memory allocation in the shortcut, this will not be an issue. Unless you are working with very large datasets, you won't need to allocate more memory to R anyway.

Brian

From: GREGOR Brian J
Sent: Friday, April 18, 2003 12:07 PM
Subject: Lesson Correction

Here is a correction to today's lesson.

I had recommended that you add "--max-mem-size=400000000". You do not need to do this. I was doing this with an earlier version of R and did not realize that newer versions now default to 1Gb or the maximum amount of memory available. Thanks to Peter for discovering this problem.

Brian

From: STABLER Benjamin
Sent: Monday, April 21, 2003 8:40 AM
Subject: R Training Course: Week 2, Lesson 1

Welcome to lesson 2 of the R class on learning to use R for data analysis and modeling.

WEEK 2, LESSON #1: USING R

~~~~~~~ Step 1: R Programs ~~~~~~~

R is a "programming environment for data analysis and graphics," which basically means that it is a programming language in addition to an application such as Excel or SPSS. There are three interfaces to R:

1) RGui - The most "Windows" like interface to R. It has some menus and allows for easy copying and pasting and printing. This is the interface that Brian and I most often use. There are a few limitations to RGui but those are mostly in its interaction with other programs.
2) Rterm - The "DOS" or "UNIX" interface to R. It is strictly command line and is not very integrated with Windows. This is also the interface for the Linux version of R.
3) Rcmd - The additional tools interface to R. It is used to run various advanced tools that can be used with R. These includes such tools as BATCH (to run R in batch mode), COMPILE (to compile files for use with R), SHLIB (to build shared libraries (code) for dynamic loading), INSTALL (to install add-on packages), and REMOVE (to remove add-on packages). Rcmd is for advanced R sessions and is not currently in the lesson plan.

All three (two for Linux users) of these interfaces should be in your R installation folder. Windows users should start with RGui.

~~~~~~~ Step 2: Entering Expressions ~~~~~~~

Start up R by clicking the R shortcut on your desktop, the shortcut in your project folder, or by typing R at the Linux prompt. You will see the R start-up message followed by the ">" character. The ">" character is the R prompt that signifies that R is ready for input. The prompt is where you enter expressions for R to evaluate. Entering expressions is easy in R. In addition to all the great stuff R can do, R can be a simple calculator. Type 5 + 3 and then press enter. R should return [1] 8. Ignore the [1] for now. R is really smart though, so if you type c(5, 6) + 3 you get [1] 8 9. R knows to add 3 to each number in the vector. The c( ) function takes any number of numbers or strings and combines or concatenates them into a vector.

You can also assign the results of an expression to a named object. The assignment operator "<-" will assign the result of the calculation to an object, which you refer to by name. For example, type x <- 5 and then press enter. Then type x and press enter. R returns [1] 5. So we have assigned the number 5 to the R object named "x." You can enter multiple operators on one line as well. For example, type y <- 3 * 4 + x and then press enter. Then type y and press enter and R returns [1] 17. You now have an object named x and an object name y in your workspace, which R will ask to save when you close R. You also have the option of saving the commands you entered at the prompt (such as y <- 3 * 4 + x).

~~~~~~~ Step 3: The Workspace ~~~~~~~
~~~~~~~

All the objects (such as a vector consisting of 5 numbers - say 1,2,3,4,5) that you create will be stored in the workspace by default. After you create R objects you can recall them since they are stored in the workspace. When you exit R, it prompts you to save the workspace. If you enter "yes" then R creates two files in the directory that you defined as the "Start In" folder for the R shortcut. The .Rdata file is a collection of R objects that were in the workspace and the .Rhistory file is a text file of every line you typed at the ">" prompt. When you start R again, it automatically loads the .Rdata file and the .Rhistory file so that all the objects you had in the workspace and all the commands you typed earlier are available. To see your history of commands simply press the up arrow and R will display the next one with each press of the key. You can also type history(x) where x is equal to number of commands to display.

One other important function to note is the ls( ) function. The ls( ) function returns a listing of all the objects in the workspace. So in our example above ls( ) returns [1] "x" "y". This is a nice way to get the name of an object you created.

~~~~~~~ Step 4: Menu Commands ~~~~~~~

In addition to the standard Windows menu commands, RGui has the load/save workspace and history commands, the packages commands, and the help commands. The load/save workspace and history commands are similar to what R does when it asks you if you want to save your workspace and when it loads the workspace upon startup. These commands can be very helpful in saving your workspace as a backup to another location or loading that backup from a location other than that specified by your "Start In" location.

The packages commands allow you to install packages (such as collections of R code, functions, data, and faster C++ functions) that others have written. Since R is an open source project, there are lots of additional packages available that extend R. These can be downloaded from CRAN (Comprehensive R Archive Network) by using the packages menu in RGui. For details on the packages see the CRAN website.

The help menu consists of many different routes for obtaining help. I recommend the "HTML Help" option since it links to an HTML index page that has a search function and hyperlinks to related R commands.

~~~~~~~ Step 6: Using a Text Editor ~~~~~~~

Finally, we thought it would be a good idea to mention a few tips on using a text editor in conjunction with R. Soon you will be writing multiple line R expressions that are not easy to recall. Thus you might want to have a working text document open in another program where you write your R code. Then when you are satisfied with the code, copy and paste it into RGui and R will execute it. A number of text editors will do, but one with syntax highlighting for the R language is preferred. At http://cran.us.r-project.org/other-software.html is a few links to editors with R syntax highlighting. Syntax highlighting is when a text editor colors your code in a certain way depending on categorized words. For example, I created an Ultra Edit syntax highlighting file that you can get on the link above that colors functions blue, strings red, comments green and braces bright blue. Brian likes j-edit, which is open source and written in Java. There is also Emacs, WinEdt, and Kate. TPAU people should probably install Ultra Edit as their R text editor.

That is all for this lesson, I hope it is not too long. Tomorrow Brian will write about the R environment.

Benjamin Stabler
Transportation Planning Analysis Unit
Oregon Department of Transportation
555 13th Street NE, Suite 2
Salem, OR 97301 Ph: 503-986-4104

From: GREGOR Brian J
Sent: Tuesday, April 22, 2003 9:37 AM
Subject: R Lesson, Week 2, Lesson2

WEEK 2, LESSON #2: THE R ENVIRONMENT

We've already acquainted you with some aspects of the R environment. To recap:

There are several programs that you can use to run R, but most likely you will use Rgui.

When you type something at the prompt and then press the enter key, R will evaluate what you have typed.

You create objects in the R workspace by assigning them to a name with the '<-' operator. These objects are now in memory and can be summoned by typing their names.

You can list all of the objects in the workspace by typing ls().

You can save all of the objects in the workspace by using the menus command 'File\Save Workspace'. These objects are saved in a file called '.RData' in the current directory. When you quit R, it will automatically ask you whether you want to save the workspace.

R also keeps track of all of the commands that you type. This command history is saved in the '.Rhistory' when you exit. You can also save it at any time using the 'File\Save History' menu item. When you are working, you can use the up and down arrows to scroll through recent history.

Now here are a few more aspects of the R environment to be aware of.

When you start up R, it will automatically reload the '.Rdata' file into the workspace. All of the objects you created in your last session should be there providing that you saved it on exiting, and you are in the same directory where you saved them. (If you put a shortcut to Rgui in the folder where you are doing your analysis and change the shortcut 'Start in' property to that folder, you should have no problems with misplacing data.)

Your workspace is the first place that R will look for objects. If R does not find an object in the workspace, it proceeds through a search path of places to look for it. You can see these places by typing 'search()' at the prompt. Here's what I get when I type this.

```
".GlobalEnv"      "package:methods" "package:ctest"   "package:mva"
"package:modreg" "package:nls"     "package:ts"      "Autoloads"
"package:base"
```

The first item you see is '.GlobalEnv', which is another name for the workspace. What you mostly see are a bunch of names starting with 'package'. Packages are libraries of objects that are loaded into memory. These are most often functions and data. 'Autoloads' is a reference to on demand loading of a library. This is a way to specify that a library is to be loaded if a certain object is requested.

When you refer to an object, R will work its way through the search path in the order listed to find it. If same name is used for two different objects, R will

give you the first object with that name. This means you could unintentionally hide an existing object if you create an object with the same name. For example, if you type 'pi' at the command prompt, you will get the value 3.141593. Now if you type 'pi <- 3' and then type 'pi' at the command prompt, you will get the value 3. The real value of pi is still there, but to get it, you have to skip your workspace (".GlobalEnv"). This can be done with the 'get()' function. Type 'get("pi", pos=2)' and you will get the correct value because you told 'get()' to start the search at the second spot in the search path. (Note that 'get()' expects the object name to be in quotes.) It's best, though, to try not to use the name of an object in any loaded package.

You can find out what is in a loaded package with the 'ls()' function. For example, if you type 'ls("package:mva")', you will get the following:

```
"as.dendrogram"      "as.dist"            "as.hclust"          "biplot"
"cancor"             "cmdscale"           "cophenetic"         "cutree"
"dist"               "factanal"           "format.dist"        "hclust"
"heatmap"            "kmeans"             "loadings"
"order.dendrogram"   "plclust"            "plot.hclust"        "prcomp"
"princomp"
"princomp.formula"   "promax"             "rect.hclust"        "reorder"
"reorder.dendrogram" "screeplot"          "varimax"
```

You'd get the same result if you typed 'ls(pos=4)' because "package:mva" is in the fourth position in the search path. When you type 'ls()' without anything in the parentheses, R gives you the contents of the first position in the search path which is your workspace.

Many libraries of useful functions and data came with your R installation. These can be loaded with the 'library(some_package_name)' command. If you just type 'library()' at the command prompt, a window will pop up which lists the available libraries and short description of each. If for example, you wanted to read in an SPSS dataset, you would type 'library(foreign)' at the command prompt. If you then type 'search()' you would find that the foreign package was loaded at the second position in the search path.

Additional libraries are available over the internet. You can view what's available and install them with the menu command 'Packages\Install package(s) from CRAN ..'. This will connect to CRAN and produce a window containing a list of all the libraries that are available. You then can use standard windows selection methods to choose one or more of them to load. Then click on OK and they will be installed. Finally you may answer 'y' to the question at the command prompt "Delete downloaded files (y/N)?".

There are a few additional useful functions to cover this lesson.

You can remove an object from the workspace with the 'rm(some_object_name)' function. So if you type 'ls()' now you should see the name of the object 'pi' which we created earlier. Typing 'rm(pi)' or 'rm("pi")' will remove it. The real pi is still there and will now be revealed if you type its name.
You can remove all objects from the workspace with the 'Misc\Remove all objects' menu command.

It's also worthwhile to know that you can save individual objects in the workspace with the 'save()' function, not just the entire workspace. If for example you type 'mph.to.fps <- 1.47' and then type 'save(mph.to.fps, file="speed_convert")', this object will be saved to the current directory with

the file name 'speed_convert'. Note that the object name is not in quotes but the file name is. You can reload that object with the command 'load("speed_convert")'. This will recreate the 'mph.to.fps' object in the workspace. Again, please note the quotes around the file name. More than one object can be saved with the 'save()' command. To do this, you should know something about lists, a topic for later this week.

That's it for this lesson. Here's a summary of the functions we covered this time:

'search()' shows the search path R follows to find an object name.

'get(pos=)' allows you specify where you want to retrieve an object from in the search path.

'ls(some_package_name)' allows you to look at the contents of a library that has been loaded.

'library()' lists all the libraries that are available on your computer.

'library(some_library_name)' loads a library into the workspace.

The menu command 'Packages\Install package(s) from CRAN ..' allows you to get other libraries.

'rm(some_object_name)' removes an object from the workspace.

'save(some_object_name, file="some_file_name")' will save a workspace object to disk.

'load("some_file_name")' will retrieve a saved object.

From: STABLER Benjamin
Sent: Wednesday, April 23, 2003 10:12 AM
Subject: R Week 2 Lesson 3, Vectors and Matrices

WEEK 2, LESSON #3: VECTORS AND MATRICES

DATA TYPES

R recognizes several types of data. Numbers and character strings are the most familiar of these, but it also recognizes logical (or Boolean), complex and NULL types. Logical data types are often used for program control and subsetting matrices (and other data structures) in R. TRUE and FALSE are how R represents logicals. Note that there are no quotes around the words since they are not character types, but logical types. If we assign to x the word "TRUE" ('x <- "TRUE"'), R will create x as a character type object. The 'typeof( )' function will return "character" for our x object. To assign the logical type value TRUE to x we just type x <- TRUE. Then 'typeof(x)' returns "logical". It is a subtle difference but can be important when your code expects a logical type value as opposed to a character type value.

NULL data types represent nothing. This is different than 0 or an empty character string (""). You can assign NULL to an object by typing 'x <- NULL'.

Complex data types represent complex numbers.

All of these data types can have a special value, NA, which represents a missing value. This is a useful value because R will keep track of the effect of missing values in computations. For example, adding a number to a NA results in a NA. NAs can also cause a lot of trouble in your analysis. If, for example, you take the mean of a set of numbers which includes one or more NAs, the result will be NA, unless you tell the 'mean()' function to ignore them. A very important function to remember for testing if NAs exist in your data is the 'is.na( )' function. For example if you type 'y <- c(2,4,NA)' and then type 'is.na(y)', you will get an answer of FALSE FALSE TRUE. Watch out for NAs!

VECTORS

Data types can be organized into several different data structures. The simplest data structure is the vector, which is collection of numbers. To create a vector we can use the 'c( )' function. For example: 'myvector <- c(2, 4, 67, 1, 1, 443)'. If we want to view what myvector looks like we just need to type myvector at the R prompt. This is the same as using the 'print( )' function with the R object as the input to the function – 'print(myvector).'

In addition to creating vectors with the 'c( )' function, we can create vectors with the : (colon) operator. The colon operator creates a sequence of integers from start:end. For example: 5:10 will return 5 6 7 8 9 10. The colon operator is a simplification of the 'seq()' function, which has the form 'seq(from, to, by)'. If you type 'seq(1, 10, 2)' for example, you will make a vector of odd numbers from 1 to 9.

The 'rep()' function creates vectors of repeating number sequences. For example, 'rep(1:3, 3)' will return 1 2 3 1 2 3 1 2 3.

All R operators and functions are vectorized. This means that R will operate on each element of a vector. For example, typing '5*c(3,4,5)' will return '15 20 25'. This is one of the strengths of R. There is no need to loop through each

element of the vector and multiply the element by 5. Try this with different operators. (You can find out about what operators are available by typing 'help(Arithmetic)'.) Finally, the vector data structure is the foundation of all R data structures, and is often used to build more complex data structures such as matrices.

MATRICES

A matrix is a two-dimensional data structure with each element being of the same type. You can think of a matrix as a table of cells, with each cell having the same type of data (such as numeric or character). For example:

```
2     4     6
5     7     9
1     2     3
```

is a 3 row by 3 column matrix. Matrices are easy to create in R with the 'matrix( )' function. The 'matrix( )' function usually takes three arguments:

1) the data to fill the matrix with
2) the number of rows
3) the number of columns

Therefore, we could create the matrix above with the following command: 'matrix(c(2,5,1,4,7,2,6,9,3),3,3)'. There are a couple of things to note about the previous command. First, you can use a vector of data to fill the matrix, and second, that the matrix was filled in column major order (i.e. it filled the first column from top to bottom then the second column from top to bottom and so on). We could simplify our matrix function call by assigning the vector to an object before using it to fill the matrix. 'mydata <- c(2,5,1,4,7,2,6,9,3)' then 'matrix(Mydata, 3, 3)' will create our matrix.

Often you will want to fill the matrix in row-major order instead of column-major order. There are two ways to do this. The first is to add the byrow=T argument to the 'matrix( )' function, so we would type 'matrix(mydata, 3, 3, byrow=T)'. The second is to transpose the matrix after the matrix has been created. The 't( )' function will transpose a matrix – so try 't(maxtrix(mydata, 3, 3))'. We should probably brake down our statement into the sub-statements 'mymatrix <- matrix(mydata, 3, 3)' and 'mytmatrix <- t(mymatrix)'. The final result should be:

```
2     5     1
4     7     2
6     9     3
```

As I mentioned earlier, all the elements of the matrix must be the same data type. As a result, 'matrix(c(2, 2, "hello", 3), 2, 2)' will not keep the numbers as numbers. Instead, R will demote the numbers to characters, which it signifies by putting quotes around them.

In addition to using the matrix function to create a matrix, you can also use the 'rbind( )' and 'cbind( )' functions to create matrices. The 'rbind( )' function will row-bind any number of vectors into a matrix, while the 'cbind( )' function will column-bind them. Therefore we could use rbind to create the matrix above with the following code: 'rbind(c(2,5,1), c(4,7,2), c(6,9,3)).'

DIMENSIONS

The R language includes a number of functions that return the dimensions of a data structure. The 'length( )' function returns the number of elements in a vector or the number of elements (cells) in a matrix. Because R is an object-oriented language, it knows how to find the length of a named object such as myvector, regardless if it is a vector, matrix, or a more complex data structure. Object-oriented programming can be a little intimidating, but don't worry, R handles all the object-oriented stuff behind the scenes, so all you have to do is type 'length(myobject)' and R returns the length of the object.

Unlike vectors, which don't really have dimensions (okay, maybe they have one dimension), matrices have two dimensions. The R function 'dim( )' will return the dimensions of a matrix. So in our example above, 'dim(mymatrix)' returns 3 3. If you try to 'dim(a vector)', R returns NULL, meaning that there is no dimension defined for vectors.

Well that should be enough R for today's lesson. Tomorrow Brian will introduce you to the more complex and very powerful list data structure. But before we close, here is a list of all the functions used today.

FUNCTIONS

c( ) - create a vector
print( ) - print an object
: (colon) operator - create a sequence vector
seq( ) - create a more complex sequence vector
rep( ) - replicate a number or vector as many times as specified
matrix( ) - create a matrix
t( ) - transpose a matrix
rbind( ) - row-bind vectors
cbind( ) - column-bind vectors
length( ) - return the length of a vector (or matrix)
dim( ) - return the dimensions of a matrix
typeof( ) - return the data type of the object
is.na( ) - test for NAs
mean( ) - return the mean of a numeric vector

Benjamin Stabler
Transportation Planning Analysis Unit
Oregon Department of Transportation
555 13th Street NE, Suite 2
Salem, OR 97301 Ph: 503-986-4104

From: GREGOR Brian J
Sent: Thursday, April 24, 2003 9:07 AM
Subject: R Week 2 Lesson 4, Lists

WEEK 2, LESSON #4: LISTS

Let's start with a note about object and value names. R is case sensitive. 'true' is not the same as 'TRUE'. We who are used of working in the DOS/Windows world get accustomed to the sloppy way that names are handled. R comes from the UNIX world which is more particular about naming.

Yesterday you learned about data types and the vector and matrix data structures. *** At least you should have learned it yesterday if you had been following the program, instead of doing other things and just waiting for some "convenient time" to read your R tutorial ;-) *** These data structures and the operators and functions that know how to operate on them offer some very powerful computational capabilities. One limitation of these structures, however, is that they require all their elements to have the same data type. If you try to combine types like this, 'c(3, 4, "a", 2)', R will coerce all the values into one type; characters in this case. Another limitation for matrices is that all the rows/columns have to be the same length. You can't for example 'rbind(c(1, 2, 3, 4), c(1, 2, 3))' and get what you want. R will recycle the second vector to make it as long as the first so you get this:

```
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    1    2    3    1
Warning message:
number of columns of result
        not a multiple of vector length (arg 2) in: rbind(c(1, 2, 3, 4), c(1, 2,
3))
```

LISTS TO THE RESCUE

Lists provide a data structure which allows you to combine different types and sizes of data. In fact, you can include any type of object in a list. For example, you can include a vector, a matrix, a function, and a list in a list. This often comes in very handy for keeping data about something together. The small urban area modeling code, for example, keeps the results of trip production calculations in a list. The first part of the list contains a set of matrices of trip productions by trip purpose and hour. The second part contains a set of matrices of trip productions by special generator and hour. The third part contains a vector of summary data.

You can create a list with the 'list()' function. You use the function in the same way that you use the 'c()' function. Entering 'list("Wednesday", c(1, 2, 3), matrix(1:9, 3, 3))' at the prompt generates the following:

```
[[1]]
[1] "Wednesday"

[[2]]
[1] 1 2 3

[[3]]
     [,1] [,2] [,3]
[1,]    1    4    7
```

```
[2,]    2    5    8
[3,]    3    6    9
```

Notice that each part (element) of the list is preceded with a number inside doubled brackets. This is how R references elements of a list. You can also name elements of the list when you make it. For example 'list(day="Wednesday", hours=c(1, 2, 3), trips=matrix(1:9, 3, 3))' returns:

```
$day
[1] "Wednesday"

$hours
[1] 1 2 3

$trips
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

You can then use these names to extract parts of the list. If you had assigned this list to the name "travel", for example, then entering 'travel$trips' at the prompt would give you:

```
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

This naming feature is not unique to the 'list()' function. The 'c()' function also allows you to name elements of vectors.

Our friend 'c()' also can be used for concatenating elements into lists. For example, we could add a vector of names to the "travel" list we created by typing 'travel <- c(travel, list(persons=c("Ted", "Ed", "Fred", "Ned")))'. This will make the vector of person names the fourth element of the list named persons. Note that this command uses the function 'list()' within it. This binds the whole persons vector together as an element of the list. If we had not used this function, then each element of the persons vector would have been added as a separate element of the list. Try entering 'c(travel, places=c("here", "there", "everywhere"))'

That's it for your short introduction to lists. Ben will continue tomorrow with presentation on a special type of list, the dataframe. Then next week we will cover a variety of functions about how to manipulate lists.

From: STABLER Benjamin
Sent: Friday, April 25, 2003 8:56 AM
Subject: R Week 2 Lesson 5, Data Frames

WEEK 2, LESSON #5: DATA FRAMES

As Brian introduced yesterday, lists are a data structure that supports different types and sizes of data. An example list might look something like:

```
[[1]]
[1] "Wednesday"

[[2]]
[1] 1 2 3

[[3]]
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

The list data structure is very useful, but does have its limitations. The major limitation of lists is that data elements across the list have no relationship. A couple of examples might better illustrate this point. In out list above, there is no relationship between the "2" in the second list element [[2]] and column two (numbers 4 5 6) [,2] in the third list element [[3]]. If for example we had a very simple list, say:

```
[[1]]
[1] "Oregon" "Washington" "California"

[[2]]
[1] "Salem" "Olympia" "Sacramento"

[[3]]
[1] 3500000 6000000 34000000
```

and we wanted to get the approximate population of each state (list element [[3]]) by entering the state's name, we would run into some trouble with lists. This is where data frames, a special type (or class) of lists, is extremely handy.

DATA FRAMES

A data frame is very similar to a spreadsheet. It consists of any number of rows and columns of data. It supports different types of data in each column (unlike matrices) but also recognizes that there are relationships across the rows (unlike lists, which don't really have rows). Let's get a simple data frame from the data that comes with R. The 'data( )' function will load a R dataset into the workspace. Type 'data(airquality)' to load the air quality dataset into R. Then type 'airquality' at the prompt to see the dataset. You will probably need to scroll up to see the beginning (or top rows) of the data frame.

If we type 'str(airquality)', R will return the structure of the airquality object, which looks like:

```
'data.frame':   153 obs. of  6 variables:
 $ Ozone  : int  41 36 12 18 NA 28 23 19 8 NA ...
 $ Solar.R: int  190 118 149 313 NA NA 299 99 19 194 ...
 $ Wind   : num  7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
 $ Temp   : int  67 72 74 62 56 66 65 59 61 69 ...
 $ Month  : int  5 5 5 5 5 5 5 5 5 5 ...
 $ Day    : int  1 2 3 4 5 6 7 8 9 10 ...
```

This tells us that the airquality object is a data frame with 153 observations or rows of 6 variables or columns. It also lists the name of each column (or field), the data type (int for integer, num for numeric) and a few sample values.

Data frames are great for storing data and for reading in existing data that is in spreadsheet form. There are R functions that will read Access, Excel, DBF, CSV, and other formats into data frames. But for all the good things about data frames there is one main thing to look out for when working with them. All the columns of data must have the same length of values. So you cannot have one column of length 10 and another of length 11. Accidentally changing the lengths of the columns is an easy way to corrupt your data frame.

CREATING DATA FRAMES

The easiest way to create a data frame is to use the 'data.frame( )' function. The data.frame function takes any number of vectors as its arguments. For example:

```
X <- c(4,5)
Y <- c(1,9)
Z <- data.frame(X,Y)
```

Z then looks like this:

```
  X Y
1 4 1
2 5 9
```

Notice that it took the X and Y vectors and made them into columns. R also numbered the rows 1 and 2 and named the columns with the name of the vector used to create it. If we had tried to create a data frame with a vector of length 2 and a vector of length 3 R would return the following error: "Error in data.frame(c(9, 9), c(3, 4, 5)) : arguments imply differing number of rows: 2, 3"

You can reference a column of a data frame by name using the '$' dollar sign. We could get the "Temp" column of the airquality data frame by typing 'airquality$Temp', which will return all the values of the temp column. You can perform operations on just the "Temp" column of the airquality data frame if you like. For example: 'mean(airquality$Temp)' will return the mean (77.88) of the temperature column.

Lastly, it is easy to create a new column in a data frame. If we wanted to get the sum of each record in data frame "airquality", and then save it as another column named "total", we would type: 'airquality$total <- rowSums(airquality)' The 'rowSums( )' function will sum each row of a data frame or matrix and return a vector. So rowSums(airquality) returns the sum of each row, which we then assigned to airquality$total (our new column named total).

That's all the R for this week. Enjoy the weekend.

FUNCTIONS

data( ) - Load a sample R dataset
data.frame( ) - Create a data frame object
$ - Reference a column of a data frame
mean( ) - Calculate the mean of a numeric vector
rowSums( ) - Sum each row of a data frame or matrix

Benjamin Stabler
Transportation Planning Analysis Unit
Oregon Department of Transportation
555 13th Street NE, Suite 2
Salem, OR 97301 Ph: 503-986-4104

From: GREGOR Brian J
Sent: Monday, April 28, 2003 12:22 PM
Subject: R Week 3 Lesson 1, Importing, Exporting and Editing Data

R Week 3 Lesson 1, Importing, Exporting and Editing Data

Now that we're past the preliminaries, we'll get into some more useful and fun stuff. Today I'll explain how to get data into R, how to export it from R so that I can be brought into another application, and how to use R's data editor. At the conclusion of today's lesson, you should be able to start using R to do some of your work.

SIMPLE DATA IMPORT

The most common way to important data into R is to use one of the functions that reads text files. Most applications like Excel can save files as text files. These can then be read into R. To show you how it is done, I'll step you through the process of saving an Excel file as a text file and opening the text file in R. I've attached an Excel file to this email for your use in this exercise.

Step 1: Save the Excel file as text. Place the "zoning.xls" in the directory that you'll be working from. Now go to the 'File\Save As' menu item and in the 'Save as type' drop down list box choose 'CSV (Comma delimited)(*.csv)'. You'll see that Excel kept the same file name but changed the extension to 'csv'. Now click 'Save'. (Don't worry about the warning that says Excel can save multiple sheets because you only have one.)

Step 2: Import the text file into R. Now start R in the folder where you saved your file. Then enter 'zoning <- read.csv("zoning.csv")' at the prompt. This will read the table into the object called 'zoning'. The 'read.csv()' function reads a text file in "comma-separated values" format. The table you read in is fairly large, so you probably will not want to look at the whole thing. You can look at the upper left corner of it by entering 'zoning[1:20,1:5]'. This shows the first 20 rows and first five columns of the table. (Well be explaining the syntax for subsetting portions of tables and matrices in a couple of days.) Note that the first line that is printed contains the column names. If you enter 'str(zoning)' at the prompt, you will see the structure of the data you imported. Note that it is a data.frame and that all but the first two columns are integers or other numbers.

MORE FLEXIBLE BUT STILL FAIRLY SIMPLE DATA IMPORT

The first two columns of the 'zoning' table are factors. The default behavior of R is to convert character strings into factors when importing tables with read.csv. Factors are special vectors used to hold a categorical value. This is useful for statistical analysis, but not always what you want.

If you don't want your character data to be changed into a factor, or if you want to control some other aspects of how your data is imported, you can use the 'read.table()' function. This function is a little harder to use, but it is much more flexible. If you enter 'args(read.table)' at the prompt, you can see the array of arguments (think of these as options) that the function can take. You should see the following:

```
function (file, header = FALSE, sep = "", quote = "\"'", dec = ".",
    row.names, col.names, as.is = FALSE, na.strings = "NA", colClasses = NA,
```

```
    nrows = -1, skip = 0, check.names = TRUE, fill = !blank.lines.skip,
    strip.white = FALSE, blank.lines.skip = TRUE, comment.char = "#")
```

Here are some of the arguments that you are most likely to use:

file	You have to have a file name of course (in quotations)
header	Set to TRUE if the first row contains the column names
sep	Identify the charater used to separate data. White space is the default
colClasses	Identify the type of data for each column.
nrows	Use to specify how many rows to read in
skip	You can specify how many rows to skip before you read data

Here's an example of how you would use 'read.table()'. This uses the "example_tables.txt" attachment. If you open up this file in your text editor, you will see that it has to tables plus some header lines and some separating blank lines. We can use R to read in these two tables and specify the column data classes as follows:

'table1 <- read.table("example_tables.txt", header=TRUE, sep="", colClasses=c("character", "character", "integer", "numeric", "numeric", "numeric"), nrows=19 ,skip=4)'

'table2 <- read.table("example_tables.txt", nrows=19, skip=27)'

You can now look at the tables by typing their names at the prompt. You can also look at their structures by typing 'str(table1)' for example. Notice that the second table did not have 'header', 'sep' or 'colClasses' arguments. When the arguments are not specified, the function uses the default values. These are shown by 'args(read.table)'. Also note that since there was no header, R labeled the columns 'V1, V2 ...'

EDITING DATA

R also has some rudimentary capabilities for editing data in tables. If you type 'edit(table1)' at the prompt, a window with a spreadsheet look will open up with your data in it. You can now click in cells and change the values. Try changing some values. When you're done, just click the close box. If you type 'table1' at the prompt, you will see that your changes were recorded.

EXPORTING DATA

Now, say you want to save this table so that you can open it in a spreadsheet or other application. You can write it to a file with the 'write.table()' function as follows:
'write.table(table1, "new_table.csv", sep=",", row.names=FALSE, col.names=TRUE)'
You can specify any type of data separator with the 'sep=' argument. In this case we specified the comma separator. You can then open this file in a text editor to see what was saved. You can also import it into Excel.

*** Note *** If you save row.names and col.names, and then import the saved table into Excel, the column names will be shifted one cell to the left. You can test this with the following commands. 'rownames(table1) <- letters[1:19]'
'write.table(table1, "new_table2.csv", sep=",", row.names=TRUE, col.names=TRUE)'

Additional arguments can be specified for the 'write.table()' function. The ones used in these examples are the ones that you will use most.

OTHER IMPORT AND EXPORT OPTIONS

With 'read.table()' and 'write.table()' you will be able to do most of the data importing and exporting you need to do, but that's only the tip of the iceberg. There are many additional ways to read and write data to and from R. Many are built into R and additional ones are available in add-on packages. Here are three that you may want to investigate:

'scan()' is a lower level data import function that is even more flexible than 'read.table()'.

The 'foreign()' library has functions for reading SPSS, SAS, Stata, Minitab and S data files.

The 'RODBC' library has functions for making database connections, doing SQL queries and reading and writing database tables. These functions allow you to read and write to Excel and Access files.

Check these out when you have the time. Ben and I would be happy to answer any data import/export questions you have.

Brian Gregor, P.E.
Transportation Planning Analysis Unit
Oregon Department of Transportation
Brian.J.GREGOR@odot.state.or.us
(503) 986-4120

From: STABLER Benjamin
Sent: Tuesday, April 29, 2003 8:58 AM
Subject: R Week 3 Lesson 2, Naming Data and Extracting Portions

WEEK 3, LESSON #2: NAMING DATA AND EXTRACTING PORTIONS

NAMING ELEMENTS OF VECTORS AND MATRICES

There are different ways to name elements of an object depending on the structure of the object. For example, type 'x<-1:3', followed by 'names(x) <- c("a","b","c")', and then just 'names(x)'. R will return "a" "b" "c". If we just type 'x', R will return the vector with the name of each element above the value of each element. If we type 'str(x)', R tells us that the vector is now an Named int (integer) instead of simply an int.

In order to name the rows and columns of a matrix, R uses the rownames() and colnames() functions. Type 'y<-matrix(1:25,5,5)', followed by 'rownames(y) <- c("a1","a2","a3","a4","a5")'. Before we assigned the our vector of "a"s concatenated to numbers as the rownames of object y, the matrix had no rownames. Try typing 'colnames(y)' to see that there are no column names set for the y object. The colnames of y can be set in the same manner as the rownames. In fact you could assign the rownames of y to the colnames of y by typing 'colnames(y) <- rownames(y)'.

A shortcut to naming matrices is the dimnames() function. The dimnames() function takes a list of two vectors (row and column names). Type 'dimnames(y) <- list(1:5,1:5)'. This will set the row names and column names of the y matrix to 1 2 3 4 5.

Note that the number of elements in the names vector must be equal to the number of elements in the vector that is being named. For vectors, if there are more names than numbers then R will return an error. If there are more numbers than names then R will pad the remaining names with NA. For matrices, the number of row names or column names must equal the number of elements in the dimension.

LIST AND DATA FRAME NAMES

First we will create a dummy list with the following command: 'z <- list(rep(11,5),seq(1:5))'. Then let's name the elements of the list with the names function by typing 'names(z) <- c("myrep","myseq")'.

Data frame names are a little more complicated because data frames have two dimensions like matrices. The rownames() and colnames() functions will assign names to data frames. Let's load the airquality data again by typing 'data(airquality)'. Now let's change the colnames of the airquality dataset by typing 'colnames(airquality) <- c("col.1","col.2","col.3","col.4","col.5","col.6")'.

This last command is rather longer, so it would be easier to use a function that would simply the process. We are essentially repeating "col." six times followed by the numbers 1:6 in our naming convention. The paste() function will paste strings and numbers together, recycling values if necessary. Type 'paste("col.",1:6,sep="") and see what R returns. R returns the same thing as the vector of names we typed earlier. The sep="" argument to paste tells R what kind of separator to use when pasting together the elements (in this case it is no space ""). Thus for the column names for the airquality data set we could have typed 'colnames(airquality) <- paste("col.",1:6,sep="")'.

If you want to remove the names of an object, just set the names() or rownames() or colnames() of that object to NULL.

EXTRACTING BY NAME

Names help better manage your data, but the real value in naming data in R is for extraction. If we wanted to get the "a3" row of our matrix named y, we would type into R 'y["a3",]'. The square braces [ ] are R notation for extraction. For vectors you simply put the name (or names) of the elements you want in the braces, such as 'x[c("b","c")]'. You can put the name more than once and R will return the appropriate value each time - 'x[c("b","b","b")]'.

For matrices and data frame, the syntax uses a comma to separate the rows from the columns. 'y["a3","a1"]' would extract the cell value for row "a3", column "a1", which is equal to 3. Note that the rows come before the comma and the columns after the comma. If you want all the rows or column then leave that part of the extraction blank, i.e. 'y["a3",] will return all the columns of row "a3". And like vectors, you can extract multiple row or columns at once such as 'airquality[,c("col.1","col.2")]'.

Lists are a little bit tricky when it comes to extraction because they use two square braces instead of one. In addition to the method discussed last week (the dollar sign $), you can use [["name""]]. To extract the "myseq" element of our z list, type 'z[["myseq"]]'. Like the other data structures, you can reference multiple names as well.

Extracting is key to programming in R. Tomorrow Brian will introduce you to extracting using indexes and numbers, which is even more powerful than names.

FUNCTIONS

names() - name a vector or list
rownames() - name the rows of a matrix or data frame
colnames() - name the columns of a matrix or data frame
dimnames() - name the rows and columns of a matrix or data frame
paste() - paste together strings or numbers, recycling values if necessary
[" "] - extract elements of a vector by name
[" ",] - extract rows of a matrix or data frame by name
[," "] - extract columns of a matrix or data frame by name
[[" "]] - extract an element of a list by name

Benjamin Stabler
Transportation Planning Analysis Unit
Oregon Department of Transportation
555 13th Street NE, Suite 2
Salem, OR 97301 Ph: 503-986-4104

From: GREGOR Brian J
Sent: Wednesday, April 30, 2003 9:54 AM
Subject: R Week 3 Lesson 3, More on Subsetting Data

R Week 3 Lesson 3, More on Subsetting Data
~~~~~~

Yesterday, Ben explained how names can be used to extract portions of vectors, matrices, lists and data frames. In today's lesson, I explain how numerical and logical indices can be used to subset data. First let's review the basic syntax for using brackets to subset data.
some.vector[index] extracts the indexed position(s) of some.vector
some.matrix[row.index, col.index] extracts the indexed rows and columns of some.matrix or data frame
some.matrix[, col.index] extracts all the rows for the indexed columns of some.matrix or data frame
some.matrix[row.index,] extracts all the columns for the indexed rows of some.matrix or data frame
some.list[[index]] extracts the indexed element(s) of some.list

Integer vectors and logical vectors or matrices may be used as indices. If an integer vector is used, the subsetting operation will extract data elements corresponding to the positions indicated by the indexing vector. The easiest way to understand this is to see some examples. You can extract the vowels from the letters of the alphabet by using their positions as indices like this: 'letters[c(1, 5, 9, 15, 21)]'. If we wanted to put the vowels in reverse order we would just reverse the order of the indices: 'letters[c(21, 15, 9, 5, 1)]'. To extract every other letter we could use a sequence of odd numbers as the indices: 'letters[seq(1, length(letters), 2)]'. To duplicate every other letter: 'letters[rep(1:26, rep(c(1,2),13))]' (Make sure you understand what's happening with these last examples. 'seq()' and 'rep()' are important functions to know.)

Subsetting matrices and data frames is done in the same way except that indices are needed for both the rows and columns. To see this let's start by loading the now familiar air quality data set: 'data(airquality)'. To get the first 10 rows of the data, enter 'airquality[1:10,]'. To extract the third and forth columns of the data enter, 'airquality[,c(3,4)]'. To get every other row and every other column of the data enter, 'airquality[seq(1, nrow(airquality), 2), seq(1, ncol(airquality), 2)]'. (Note that the 'nrow()' function counts the number of rows of a data frame or matrix and the 'ncol()' function counts the number of columns.)

You can also use logical vectors to subset data. Data will be extracted everywhere the vector value is TRUE. So if 'a <- 1:10', then 'a[c(T, T, F, T, T, F, T, F, F, F)]' results in '1 2 4 5 7'. Note that if the logical vector is not the same length as the vector it is indexing, R will recycle it as necessary to make it the same length. The following example illustrates how you can use this recycling to extract every third row of the air quality data: 'airquality[c(F,F,T),]'. Notice that the syntax is shorter and clearer than what was done using 'seq()' above.

You can use this capability to subset data based on logical conditions. For example if you enter 'a[a < 5]', R will evaluate 'a < 5' and create a logical vector which indicates which elements of 'a' are less than 5. Then R will use this logical vector to select elements of 'a'. As another example, you can select all records of the air quality data where temperature is greater than 80 degrees like this: 'airquality[airquality$Temp > 80,]'. As a final example,
~~~~~~

here's how you could eliminate all records having NA in any column: 'airquality[!is.na(rowSums(airquality)),]' In this example, since all the data is numeric, 'rowSums(airquality)' yields a number except where any elements of a row are NA. 'is.na' creates a logical vector identifying which of the results are NA (TRUE) and which are not (FALSE). The '!' symbol means not. It turns TRUE into FALSE and vise verse.

Note R does not care where the indexing vector comes from (although you should). Don't get stuck in the mode of thinking that your index to a table has to come from within the table. If you do, then you'll unnecessarily expand tables (remember 'cbind()') in order to associate your indexes with your data to be indexed. Indexes can come from anywhere. It does not matter to R. This flexibility allows you to simplify your calculations, but it also puts more responsibility on you to be aware of what you are doing. For example, if you use a logical vector that is not the same length as the vector you are subsetting, it will recycle and you may not get what you want.

Numerical and logical indexing is one of the most important things to understand about R. I suggest that you spend some time trying these ideas and be sure to ask questions if you don't understand. Here is a problem for you to try. Take the air quality data, split it in half by rows, and then interleave the two halves. Your output data frame would then have the 1st row of the 1st half followed by the 1st row of the 2nd half etc. There are several solutions to this problem. Mail yours back and we'll post them later this week.

From: GREGOR Brian J
Sent: Friday, May 09, 2003 1:27 PM
Subject: R Answer to Interleaving Problem

I haven't gotten any answers to this problem and I've heard that it is too hard. I thought I'd better send you answers before you get too frustrated. Remember, though, the drop out period for this course has passed so you're stuck with this course whether you like the problems or not. You better start studying for the final exam ;-)

You may have gone astray if you followed my instructions literally when I said "take the air quality data, split it in half by rows, and then interleave the two halves." You don't have to literally split the data frame. The solution to this problem involves indexing. Really understanding what you can do with indexing is a very important part of unlocking the power of R.

The first thing to realize is that you can reorder any data frame by using a vector with the right order as the row index for the data frame. Try this if you are having a hard time understanding what I just said: 'reversed.air <- airquality[153:1,]'. "153:1" generates a vector of the sequence from 153 down to one. When this is used as a row index, the first row of 'reversed.air' is the 153rd row of 'airquality'. The second row of 'reversed.air' is the 152nd row of 'airquality' and so on. So by using an index that was reversed, we reversed the order of the data frame.

What we need to do then is create the appropriate index vector to get the data frame in the desired row order. We need to choose the 1st row, followed by the 78th row, followed by the 2nd row and so forth. If you could somehow define "index <- c(1,78,2,79,3,80,4,81 ...)", then "airquality[index,]" would rearrange the rows of the air quality data frame in the desired order. The way I did this is to add two sequences: 1, 1, 2, 2, 3, 3, ... and 0, 77, 0, 77, 0, 77 ....

The first sequence can be generated with 'rep(1:77, each=2)'. The second sequence can be generated with 'rep(c(0,77), 77)'. If you add these two sequences together you have an index with the proper order: 'index <- rep(1:77, each=2) + rep(c(0,77), 77)'. You need to chop off the last value, however, to make the index the proper length: either 'index <- index[1:153]' or 'index <- index[-154]' will do the trick. Then with that index in hand you can get the airquality data in the desired order with 'airquality.result <- airquality[index,]'

You could generate the index in another way using a function from this Wednesday's lesson. First enter 'index.matrix <- rbind(1:77,78:154)'. This creates a matrix where the indices are split into two rows of a matrix. Then if you enter 'index <- as.vector(index.matrix)' you get a vector with the indices in the proper order. This occurs because 'as.vector()' converts a matrix to a vector in column major order. Finally, chop off the index vector to the proper length with 'index <- index[-154]'.

Later on, we'll show you a brute force method of solving the problem with a 'for' loop, but just to illustrate how to do it. It's usually better to use the indexing power of R.

Brian Gregor, P.E.
Transportation Planning Analysis Unit
Oregon Department of Transportation
Brian.J.GREGOR@odot.state.or.us
(503) 986-4120

From: STABLER Benjamin
Sent: Thursday, May 01, 2003 9:21 AM
Subject: R Week 3 Lesson 4, Functions to Describe Data

R Week 3 Lesson 4, Functions to Describe Data

Let's begin by loading the now famous airquality data set 'data(airquality)'. R has functions for all the standard descriptive statistics. To begin, let's look at:

min() - calculates the minimum value of a vector(s)
max() - calculates the maximum value of a vector(s)

Both functions can take multiple vectors as arguments and both functions return the min or max of all the vectors input. For example: 'min(airquality$Wind,airquality$Temp)'. Remember that most functions will return NA if there are any NAs in the data set. In addition to the min() and max() function, there are the following functions:

range() - returns the min and max of a vector(s) as a vector of length two
mean() - returns the mean of a vectors(s)
median() - returns the median of a vector(s)
sum() - sum a vector(s)
var() - returns the variance of a vector(s)
sd() - returns the standard deviation of a vector(s)

All of the functions listed so far have the "na.rm" optional argument. The default is set to FALSE, so R will NOT ignore NAs when calculating the function. But simply adding, for example: 'mean(airquality$Ozone, na.rm=T)', will ignore NAs in the calculation and thus return a number.

There are many other functions for describing data, but one of the best is the summary() function. The summary function takes any number of vectors, and will return the summary information for each column of a data frame or matrix if given as the argument. For the Ozone vector of the airquality data frame, the summary function returns:

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
   1.00   18.00   31.50   42.13   63.25  168.00   37.00
```

For the first five columns of the airquality data set, summary returns: 'summary(airquality[,1:5])'

```
     Ozone           Solar.R           Wind             Temp           Month
 Min.   :  1.00   Min.   :  7.0   Min.   : 1.700   Min.   :56.00   Min.:5.000
 1st Qu.: 18.00   1st Qu.:115.8   1st Qu.: 7.400   1st Qu.:72.00   1stQu.:6.000
 Median : 31.50   Median :205.0   Median : 9.700   Median :79.00   Median:7.000
 Mean   : 42.13   Mean   :185.9   Mean   : 9.958   Mean   :77.88   Mean:6.993
 3rd Qu.: 63.25   3rd Qu.:258.8   3rd Qu.:11.500   3rd Qu.:85.00   3rdQu.:8.000
 Max.   :168.00   Max.   :334.0   Max.   :20.700   Max.   :97.00   Max.:9.000
 NA's   : 37.00   NA's   :  7.0
```

The NA's is a count of the number of NAs in the vector.

The table() function will cross classify two or more vectors. For example: 'table(airquality$Temp,airquality$Month)' returns a matrix of the number of records for each value of airquality$Temp and airquality$Month. Table is a great way to build contingency tables.

Our final two functions are rowSums() and colSums(), which are pretty self explanatory. rowSums() will sum each row of a matrix or data frame, while colSums() will sum each column. For example, 'colSums(airquality)' returns:

```
Ozone Solar.R    Wind    Temp   Month     Day
     NA      NA  1523.5 11916.0  1070.0  2418.0
```

or colSums(airquality, na.rm=T):

```
  Ozone Solar.R    Wind    Temp   Month     Day
 4887.0 27146.0  1523.5 11916.0  1070.0  2418.0
```

Remember that the results of all of these functions can be assigned to new objects and then indexed to extract values. So 'column.totals <- colSums(airquality, na.rm=T)' creates a new vector named column.totals that is the totals of airquality columns. We could then 'rbind()' the column.totals to the airquality data set in order to add the totals to the bottom of the data frame. Try adding the column totals on your own.

Functions

min() - calculates the minimum value of a vector(s)
max() - calculates the maximum value of a vector(s)
range() - returns the min and max of the vector(s) as a vector of length two
mean() - returns the mean of the vectors(s)
median() - returns the median of the vector(s)
sum() - sums the vector(s)
var() - returns the variance of the vector(s)
sd() - returns the standard deviation of the vector(s)
summary() - returns summary statistics for a vector(s)
table() - create a cross-classification of multiple variables
rowSums() - sum the rows of a data frame or matrix
colSums() - sum the columns of a data frame or matrix

Benjamin Stabler
Transportation Planning Analysis Unit
Oregon Department of Transportation
555 13th Street NE, Suite 2
Salem, OR 97301 Ph: 503-986-4104

From: GREGOR Brian J
Sent: Monday, May 05, 2003 4:13 PM
Subject: R Week 4 Lesson 1, Using the apply() and sweep() functions

R Week 4 Lesson 1, Using the apply() and sweep() functions
~~~~~~
Well, most of you who responded suggested that we reduce the number of lessons per week. Three was suggested most often but some people suggested four. We will send you three or four a week, depending on the subject matter. This week we will cover a number of related functions that are useful for analyzing data. This will take four lessons.

Much of the ease of use and power of the R language comes from its ability to operate on vectors and matrices element-wise. For example, if you have one matrix 'a <- matrix(1:100, 10, 10)' and another matrix 'b <- matrix(1:3, 10, 10)' Now, you can add these matrices simply by entering 'a + b'. Many of R's functions are set up to work this way (for example, 'paste("count #", 1:10, sep="")'). They are "vectorized". Over the course of this week, we will introduce you to several very powerful functions that allow you to vectorize most calculations. They permit you to operate on matrices, data frames, vectors and lists quickly and simply.

The first of these functions is 'apply()'. This function allows you to sequentially apply a function to each row or column of a matrix or a data frame. (You may also use it on higher order arrays, but the usage is harder to understand so we will put that off for now.) I'll introduce 'apply()' with some examples. Start by loading a data set 'data(USPersonalExpenditure)'. This is a small data set so just enter 'USPersonalExpenditure' to take a look at it. You learned before that you can sum the column with the colSums function like this 'colSums(USPersonalExpenditure)'. Before the colSums function was available, you could do the same thing with the apply() function like this:
'apply(USPersonalExpenditure, 2, sum)'.

You read the arguments of apply() as follows:
1st is the name of the matrix or data frame you are operating on.
2nd is the dimension you are operating across. "1" is for rows and "2" is for columns.
3rd is the name of the function you are applying.
So apply() in this instance goes through the data extracting the first column vector and summing it, then moving to the second column vector and summing it, and so on.

Here's another example. If you wanted to get the range of values in each row, you would enter 'apply(USPersonalExpenditure, 1, range)'.

You can even apply custom functions that you define in this manner. Here is an example of how you do this. Suppose that you wanted to compare the values in each column to the column maximums. This could be done as follows:
'apply(USPersonalExpenditure, 2, function(x) x/max(x))'. With this, we are operating on the USPersonalExpenditure data by column as before. We are then telling apply() that we are defining the function to be applied with the 'function(x)' statement. Think of the "x" here as referring to the column that is being operated on. After 'function(x)' is the formula for the calculation being done. So in this case the formula is taking the column vector and dividing by the maximum value in the vector. Since these calculations work on vectors, the result is a vector. The resulting vectors are bound together in a new table as you see from the results.
~~~~~~

As you can see, the apply function offers a lot of flexibility and power for operating on matrices. You may have notices, however, that the examples above operate just on the data that is in each column. What if we want to bring other data into the calculation. There are a couple of options for this. The first is to define a function in the 'apply()' which references the other data. The second is to use the 'sweep()' function.

Let's start off with an explanation of 'sweep()' using an example. Suppose you wanted to compare the expenditures in the USPersonalExpenditure data frame in uninflated dollars. You would do the conversion by multiplying the values in each column by the corresponding deflator for that year. Start by making a vector of deflator values corresponding to the columns in USPersonalExpenditure: 'deflators <- c("1940"=0.079, "1945"=0.102 , "1950"=0.136, "1955"=0.151, "1960"=0.167)'. To calculate the adjusted expenditures enter 'sweep(USPersonalExpenditure, 2, deflators, "*")'. As with apply(), the first argument of sweep() is the matrix or data frame you are operating on and the second argument is the dimension you are operating on. The third argument is the vector of values that will be applied in the sweep. The last argument is the function or operator that will be used. Note that if you are specifying a single operator, as was done in this example, it has to be surrounded by quotes.

The other way to do this is by defining a function in the apply statement that uses the deflators vector: 'apply(USPersonalExpenditure, 1, function(x) x*deflators)'. What this does is multiply each row of the matrix by the deflators vector. Notice, however, that the resulting matrix is transposed from the original (rows and columns are switched). To get the table back into the same form we have to transpose it: 't(apply(USPersonalExpenditure, 1, function(x) x*deflators))'.

Here's another example of apply and sweep(). Load the air quality data set again: 'data(airquality)'. Now what we will do is standardize all the first four columns (Ozone, Solar.R, Wind, Temp) so that their mean value is zero and their values represent the number of standard deviations from the mean. So, first let's calculate the means: 'air.mean <- apply(airquality[,1:4], 2, function(x) mean(x,na.rm=T))'. Then calculate the standard deviations: 'air.sd <- apply(airquality[,1:4], 2, function(x) sd(x, na.rm=T))'. Then we standardize with respect to the mean: 'air.mean.standard <- sweep(airquality[,1:4], 2, air.mean, "-")'. Finally, we can standardize with respect to the standard deviation: 'air.standardized <- sweep(air.mean.standard, 2, air.sd, "/")'. (Note that there is a function that will standardize a data frame for you called scale.)

Enter the following to see that the standardization has occurred. Don't worry about what the commands mean, we'll cover that next week.
'par(mfrow=c(2,2)); hist(air.standardized[,1], main="Ozone"); hist(air.standardized[,2], main="Solar.R"); hist(air.standardized[,3], main="Wind"); hist(air.standardized[,4], main="Temp"); par(mfrow=c(1,1))'

Well that covers today's lesson. You will find that apply() and sweep() are very useful functions but take a while to understand well. If you can, take some time to work through some examples and try to use these commands to analyze your data. You will see this effort paying off in the future. To review:
apply() allows you to apply a function by row or column of a matrix or data frame.
sweep() allows you to operate on a matrix or data frame using a vector of values corresponding to the rows or columns.

Brian Gregor, P.E.
Transportation Planning Analysis Unit
Oregon Department of Transportation
Brian.J.GREGOR@odot.state.or.us
(503) 986-4120

From: STABLER Benjamin
Sent: Tuesday, May 06, 2003 11:48 AM
Subject: R Week 4 Lesson 2, Using the lapply() and sapply() functions

R Week 4 Lesson 2, Using the lapply() and sapply() functions

LAPPLY

Last week we learned that a list data structure can hold multiple data types, and is a great way to store data sets. Let's create a list from the state data that comes with R. Start by loading the state data with 'data(state)'. A 'ls()' returns:

```
[1] "state.abb"      "state.area"     "state.center"   "state.division"
[5] "state.name"     "state.region"   "state.x77"
```

First we will create a list from the different vectors. Then we will use the lapply() function to perform some calculations on the list. We create a list with the list() command, so 'state <-list(state.abb, state.area, state.center, state.division, state.name, state.region, state.x77)' will create a list of state data. To see the structure of the state list type 'str(state)', which returns:

```
List of 7
 $ : chr [1:50] "AL" "AK" "AZ" "AR" ...
 $ : num [1:50]  51609 589757 113909  53104 158693 ...
 $ :List of 2
  ..$ x: num [1:50]  -86.8 -127.3 -111.6  -92.3 -119.8 ...
  ..$ y: num [1:50] 32.6 49.3 34.2 34.7 36.5 ...
 $ : Factor w/ 9 levels "New England",..: 4 9 8 5 9 8 1 3 3 3 ...
 $ : chr [1:50] "Alabama" "Alaska" "Arizona" "Arkansas" ...
 $ : Factor w/ 4 levels "Northeast","South",..: 2 4 4 2 4 4 1 2 2 2 ...
 $ : num [1:50, 1:8]  3615   365  2212  2110 21198 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:50] "Alabama" "Alaska" "Arizona" "Arkansas" ...
  .. ..$ : chr [1:8] "Population" "Income" "Illiteracy" "Life Exp" ...
```

Remember that the elements of the list are not named unless you specifically name them when creating the list or afterwards with the names() function. It is also important to remember that you can reference element of a list with the [[element number]] syntax.

Now that we have a list, let's lapply() a function to each element of the list. A simple summary() of each element of the list might be useful. The lapply() function takes two arguments, the list to apply the function to and the function to apply to each element of the list. Try 'lapply(state, summary)'. This will return a summary of each element of the list in list form. Notice that the summary is different for each element since each element is a different data type and/or structure.

As with apply(), you can write your own functions to apply to a list by typing "function(x)" followed by the function definition for the function argument to the list. For example: 'lapply(state, function(x) x[1])'. This will return the 1st element of each element of the list. Notice that the 3rd element returned is a vector – that's because the 3rd element of the state list is a list, so the 1st element of the 3rd element of the state list is actually a vector. The

ability to store multiple complex data types and structures in a list is its main advantage, but it can also make the list difficult to understand.

Now that we have the basics about lapply(), let's use it to plot the center of each state along with the state abbreviation. Let's plot each state with 'plot(state.center, pch=20)'. As you can see, we get a plot that looks like the U.S. We will learn about the plot() function next week.

In order to label each point, we need to use the text() function. The text() function adds text to a plot and requires the X and Y coordinate and the label. We already have the X and Y coordinate, since we plotted them, but we probably want to shift the label up and over a little so it doesn't cover up the point. So we need to add 0.5 to both the X and Y of each point. Since state.center is a list of X and Y, we can use the lapply function to apply a simple addition of 0.5 to each element of each element of the list. Type: 'label.xy <- lapply(state.center, function(x) x+0.5)' to create a new list of X and Y points. Then all that is left is to add the text to the plot with 'text(label.xy, state.abb, col="blue")'.

SAPPLY

The sapply() function is very similar to the lapply() function except that it can apply the function to each element of a vector and it is also returns a vector instead of a list. If we type 'lapply(state.center, mean)', R will return the mean of the X and the mean of Y coordinates as a list of length two. But since the result is so simple, it would be better to return a vector of length two. Type 'sapply(state.center, mean)', and R will return a vector named X and Y. This is preferred in this case since it saves us the additional line of code when using the lapply() to convert the list of X and Y to a vector.

In addition to operating on lists, sapply() also operates on vectors. For example type 'sapply(1:10, function(x) x^2)', which will return the square of each element of the list. This is essentially the same as simply typing (1:10)^2. But sapply() allows more flexibility in that you can apply any custom function you write to each element of a vector.

Both lapply() and sapply() are useful functions for iterating through a list without having to write a for or while loop. These functions, in addition to the apply() and sweep() functions, can save you a lot of time in R.

Functions:
lapply() - apply a function to each element of a list and return a list
plot() - plot x and y data
text() - add text to a plot
sapply() - apply a function to each element of a list or vector and return a vector

Benjamin Stabler
Transportation Planning Analysis Unit
Oregon Department of Transportation
555 13th Street NE, Suite 2
Salem, OR 97301 Ph: 503-986-4104

From: GREGOR Brian J
Sent: Wednesday, May 07, 2003 10:48 AM
Subject: R Week 4 Lesson 3, tapply() and related functions

R Week 4 Lesson 3, tapply() and related functions
~~~~~~

Today I'll cover the versatile tapply() function. You will find that the more you learn about this function, the more uses you will have for it.

If you look at the help page for tapply(), you'll see the purpose described as applying a function over a ragged array. What does this mean? Well we've seen arrays in the form of matrices, which are two dimensional arrays. With matrices, all rows are the same length and all columns are the same length. They're not ragged. As you saw a couple of days ago, you can use the apply() function with matrices. (We did not cover how to use them with higher dimensional arrays.) You can also use apply() with data frames because they also are not ragged. Now suppose that you wanted to apply a function to parts of a data set that are not equal in length. Say you have Census data on households and you want to find the mean household income by the sex of the household head. An array of household income by sex would be a ragged array because the number of cases for each sex would not be equal. If you wanted to find the mean income by sex, you would use the tapply() function.

Here's an example of how this works. Start by loading the MASS library: 'library(MASS)'. Then load the "UScereal" data set: 'data(UScereal)'. This data set contains nutritional information on a list of cereals. It also identifies the manufacturer of each cereal. We can compare the average sugar content of cereals by manufacturer using tapply() like this: 'tapply(UScereal$sugars, UScereal$mfr, mean)' In this example, the first argument to tapply() is a vector of data we want to operate on. The second is the factor we're using to group the data. The third is the function we are applying.

If you wanted to see the means for several of the variables by manufacturer, you could do this using the combination of the apply() and tapply() functions like this: 'apply(UScereal[,c("fat", "sodium", "sugars")], 2, function(x) tapply(x, UScereal$mfr, mean))'. What's done here is to take three columns of the data and then apply the tapply() function to each of them in turn, finding the mean by manufacturer.

You are not limited to using a single grouping factor. If you want to use multiple factors, you enter them as a list in the second argument. We can see this with another example using the UScereal data. The data set also includes a code indicating whether the cereal is on the top, middle or bottom grocery shelf in the variable named "shelf". To see the mean sugar content by manufacturer and shelf enter 'tapply(UScereal$sugar, list(UScereal$mfr, UScereal$shelf), mean)'.

You can do similar grouping analysis with the aggregate() function. One advantage of aggregate is that it allows you to specify several data vectors in the first argument. This avoids having to combine apply() and tapply(). To get the same table of average fat, sodium and sugars by manufacturer you enter this: 'aggregate(UScereal[,c("fat", "sodium", "sugars")], list(UScereal$mfr), mean)'. Notice that you have to specify the grouping factor as a list even if it is just a single vector. You don't need to do this with tapply().

Finally, here's an example of how the power of tapply() can be applied to modeling. Often you need to collapse a zone-to-zone matrix to a district-to-
~~~~~~

district matrix where each district is comprised of one or more zones. You may, for example, want to convert an origin-destination matrix for zones to an equivalent one at the district level. You can do this easily with the tapply(). For this example, define your zones as follows: 'zones <- 100:109'. Then define the corresponding districts like this: 'districts <- c(1,1,1,2,2,3,3,4,4,4)'. Finally, make a matrix of trips using the sample() function: 'od <- matrix(sample(1:10,100,replace=T),10,10)'. The sample function takes a random sample from the vector that is the first argument. The size of the sample is the second argument. The "replace=T" argument says that the sampling is with replacement. To collapse this zonal od matrix into a district od matrix enter the following:
'tapply(as.vector(od), as.list(expand.grid(districts,districts)),sum)'.
Voila. There's your district od matrix like magic. If this was a matrix of zone-to-zone travel times, you could convert it into a district to district travel times using the "mean" function rather than "sum".

Here's how the this statement works. The as.vector(od) converts the od matrix into a vector in column-major order (first column followed by second column and so on). The expand.grid(districts,districts) makes a data frame where each row is a combination of the factors which are the arguments. The first factor varies fastest followed by the second and so on. (If you find this description confusing, try the function out and see how it operates:
'expand.grid(1:3,1:3,1:3)') Notice that 'expand.grid(districts,districts)' corresponds to the order of the od matrix when it is converted into a matrix. You can verify this by entering 'cbind(expand.grid(zones,zones),as.vector(od))'. You can see that the first column is the row zone. The second is the column zone and third is the value for that row and column. The data frame created by expand.grid() is converted into a list with as.list(). With the matrix converted to grid and the districts corresponding to zones put into a list of factors in the proper order, the tapply() function can then do it's magic and create the district-to-district matrix.

Here are the functions we covered today.
tapply() takes a vector and a list of corresponding factors and applies a function to the vector indexed by each combination of factors.
aggregate() does the same thing as tapply(). However, it can operate on a data frame as well as a vector.
sample() creates a random sample given a sample size and a vector from which the sample is drawn
as.vector() converts a matrix into a vector in column-major order.
expand.grid() creates a data frame of every combination of the factors that are passed to it.
as.list() converts a data frame into a list. Each column becomes an element of the list.

That is it for this week. Please take some time to experiment with the various apply family functions you learned. Better yet, start using them in one of your data analysis projects.

Brian Gregor, P.E.
Transportation Planning Analysis Unit
Oregon Department of Transportation
Brian.J.GREGOR@odot.state.or.us
(503) 986-4120

From: STABLER Benjamin
Sent: Monday, May 12, 2003 2:09 PM
Subject: R Week 5 Lesson 1: SORT, ORDER, SPLIT, UNIQUE and DUPLICATED

R Week 5 Lesson 1: SORT, ORDER, SPLIT, UNIQUE and DUPLICATED

Let's start by loading the UScereal data set into our workspace. "library(MASS)" then "data(UScereal)". Our first function is the sort() function.

SORT

The sort() function sorts a numeric vector in ascending or descending order. Type "sort(UScereal[,4])" to sort the UScereal$fat column. Notice that the vector is returned in ascending order. A good function to use with sort is rev(), which will reverse the order of a vector. Try "rev(sort(c(2,4,5,6,3,2)))" to sort the vector in ascending order and then reverse the result.

ORDER

The order() function allows you to sort data frames as well as vectors. It works in a different way, however, that may take a bit of time to fully understand. While sort(x) will return x sorted in ascending order, order(x) will return a vector that gives the indexes of x in the order of the values of x. For example, if "x <- c(3,7,8,1,4,9,5,10,2,6)", here's how x, and order(x) compare:

x	3	7	8	1	4	9	5	10	2	6
order(x)	4	9	1	5	7	10	2	3	6	8

What order(x) is showing you is that the fourth element of x is the smallest, followed by the ninth element, the 1st element, and so on. Now if you enter "x[order(x)]" at the prompt, you will get x in sorted order because you are indexing "x" with the indices that place x in ascending order.

Another example might help clarify the order() function. For example: "sort(UScereal[,4])" returns the actual values in ascending order, while "order(UScereal[,4])" returns the position of element in order. The first few elements of the UScereal[,4] vector are below for comparison.

```
UScereal[,4]
 [1] 3.0303030 3.0303030 0.0000000 2.6666667 0.0000000 2.6666667

sort(UScereal[,4])
 [1] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000

order(UScereal[,4])
 [1]  3  5  8 14 15 16
```

Notice that the third element of the USCereal[,4] vector is 0. When we sort() the vector, this becomes the first element returned. When we order() the vector, the element's position (3) becomes the first element returned. It is a subtle difference but is important because you can sort a data frame based on one of its columns using the order() function. Try "UScereal[order(UScereal$fat),]". This will return the UScereal data frame with the rows in order by the 'fat' column.

SPLIT

The split() function can be used to split a vector, matrix or data frame by a corresponding factor. Try: "split(c(1,2,3,4,5),c("a","a","b","b","b"))", which will return:

```
$a
[1] 1 2

$b
[1] 3 4 5
```

Thus we split the c(1,2,3,4,5) by the vector of factors c("a","a","b","b","b").

In our cereal data set, try "x <- split(UScereal, UScereal$mfr)". The x object we assigned our split result to is the UScereal data frame split into many data frames by the factors in the mfr column. We can then apply functions to the resulting list of data frames with the lapply() function. For example: "lapply(x, function(x) mean(x[,2]))" will return the mean calories of all the cereals produced by manufacturer.

UNIQUE & DUPLICATED

Sometimes it is necessary to find the unique values of a data set. The unique() function returns all the unique values of a vector. Thus, "unique(UScereal[,4])" will return all the unique values of fat content. The unique() function returns a vector, so you can then apply other functions to it if necessary.

Related to unique() is the duplicated() function, which will return TRUE or FALSE for each element in a vector or matrix if that number is duplicated in the data set. For example" "duplicated(c(1:5,5))" returns:

```
[1] FALSE FALSE FALSE FALSE FALSE TRUE
```

since the last two numbers of the vector supplied to duplicated() are 5. Often duplicated() is used for indexing in conjunction with ! (not) to select those records of a vector containing duplicates that are not duplicated.

That's it for today. Tomorrow I will cover merge(), match() and the %in% operator.

FUNCTIONS

sort() - sort a vector
rev() - reverse a vector
order() - order a vector (i.e. return the indexes of a vector in sorted order)
split() - split a data set based on a factor
unique() - find the unique values of a data set
duplicated() - find the duplicated values of a data set

Benjamin Stabler
Transportation Planning Analysis Unit
Oregon Department of Transportation
555 13th Street NE, Suite 2
Salem, OR 97301 Ph: 503-986-4104

From: STABLER Benjamin
Sent: Tuesday, May 13, 2003 8:14 AM
Subject: R Week 5 Lesson 2, MERGE, MATCH and %in%

R Week 5 Lesson 2, MERGE, MATCH and %in%

MERGE

The merge() function is similar to a database join command. It will merge all fields with duplicate names of the two data frames specified. For an example, let's try merging the Animals and mammals data frames from the MASS library. First, load the data sets with "library(MASS)" followed by "data(Animals)" and then "data(mammals)". Now that we have the data frames in our workspace, type: "merge(Animals, mammals, by="row.names")". R will return a merged data frame of all the matching animals in the Animals and mammals data sets.

```
merge(Animals, mammals, by="row.names")
          Row.names   body.x brain.x   body.y brain.y
1  African elephant 6654.000  5712.0 6654.000  5712.0
2    Asian elephant 2547.000  4603.0 2547.000  4603.0
3               Cat    3.300    25.6    3.300    25.6
4        Chimpanzee   52.160   440.0   52.160   440.0
5               Cow  465.000   423.0  465.000   423.0
6            Donkey  187.100   419.0  187.100   419.0
7           Giraffe  529.000   680.0  529.000   680.0
8              Goat   27.660   115.0   27.660   115.0
9    Golden hamster    0.120     1.0    0.120     1.0
10          Gorilla  207.000   406.0  207.000   406.0
11        Grey wolf   36.330   119.5   36.330   119.5
12       Guinea pig    1.040     5.5    1.040     5.5
13            Horse  521.000   655.0  521.000   655.0
14            Human   62.000  1320.0   62.000  1320.0
15           Jaguar  100.000   157.0  100.000   157.0
16         Kangaroo   35.000    56.0   35.000    56.0
17            Mouse    0.023     0.4    0.023     0.4
18              Pig  192.000   180.0  192.000   180.0
19           Rabbit    2.500    12.1    2.500    12.1
20              Rat    0.280     1.9    0.280     1.9
21    Rhesus monkey    6.800   179.0    6.800   179.0
22            Sheep   55.500   175.0   55.500   175.0
```

Note that I specified by="row.names" to tell R to merge the data frames based on the row names. You can also specify a column by referring to its name as well. If the column is named something different in both data frames, then use by.x= and by.y=, with x referring to the first data frame to merge and y referring to the second. It is important to note that the column name to be merged on is specified in quotes. If the "by" argument is omitted then R will merge on all common column names. For example, type "merge(Animals, Animals)" to merge one copy of the Animals data frame with another. It looks like nothing happened, but really R merged all the fields with duplicate names (all the fields). When column names are specified with by or by.x and by.y, R will not merge duplicated columns and returns both data frame versions of the duplicated column.

MATCH

The match() function returns a vector of the positions or indexes of the (first) match of the first argument in the second argument. Type

"match(rownames(Animals), rownames(mammals))" to try out the match function. R returns:

```
[1] NA  4  5  6  8 NA 19 21 22 NA 25 28 29 32 33 NA 35 36 38 39 43 44 45 46 52
NA NA 56
```

which tells us that the first row name of the Animals data frame is not in the row names of the mammals data frame and that the second row name in the Animals data frame is in the fourth position of the row names of the mammals data frame. We can verify this by looking at the rownames of each data frame: "rownames(Animals)" and "rownames(mammals)" will do this.

```
> rownames(Animals)
 [1] "Mountain beaver"  "Cow"              "Grey wolf"        "Goat" . . .

> rownames(mammals)
 [1] "Artic fox"                "Owl monkey"               "Mountian beaver"
 [4] "Cow"                      "Grey wolf"                "Goat"
```

It looks like the first row name of Animals is in the row names of mammals but actually it is not since there is a misspelling in the mammals version. But the "Cow" row name matched correctly, since the second value returned by our match() is 4, which is the position of the row name in the mammals data set.

Match is similar to merge, except that match returns the index of the matched value where as merge merges the data frames. This is the same relationship as we had with sort() and order() - one returns the values in order while the other returns the indexes in order. The match() function is great for appending new columns to data frames based on indexes of common fields.

%in%

A related operator to the match() function is %in%. Before I describe the %in% operator let me write a few things about operators. Operators such as + or - are really just functions written another way. So sum(2,3) is the same as 2+3. The main difference is in how we write the function. Try: ""+"(2,3)" and R will return 5 - note that we put the + inside quotes. Other than the syntax difference, operators and functions work the same.

The %in% binary operator (or function) will return TRUE or FALSE indicating if there is a match between the elements of two vectors. The %in% operator is great for selecting data. For example: "1:5%in%1:3" returns T T T F F since 4 and 5 are not in the second data set. We can use this to build an index (basically a query) of a data frame by creating a vector of indexes of the records we want. For example, say you want to extract data from the Animals data set for the selected list of animals: "dinosaurs <- c("Dipliodocus", "Triceratops", "Brachiosaurus")". You could do this with "Animals[rownames(Animals) %in% dinosaurs,]". This tells R to return all the rows of the Animals data set where the rownames are found in the dinosaurs vector.

```
              body brain
Dipliodocus   11700  50.0
Triceratops    9400  70.0
Brachiosaurus 87000 154.5
```

The main difference between match() and %in% is that match() returns the position number of the match, where as %in% just returns TRUE or FALSE if there is a match. The %in% operator is also faster than match().

HOMEWORK

Since the last homework assignment went so well, let's try another. Type: "problem <- data.frame(1:100, 100:1, rep(letters[1:25], 4))" to create a problem data frame. The problem is to find the sum of the numbers of the first two columns based on the letters. So, for example, what is the total of the first two fields with the letter "a" in the third field? There are many different ways to accomplish this. One way that I can think of is "lapply(split(problem, problem[,3]), function(x) sum(x[,1:2]))".

FUNCTIONS

merge() - merge two data frames
match() - return the index of each element of a vector in another vector
%in% - is an element value of a vector an element value in another vector?

Benjamin Stabler
Transportation Planning Analysis Unit
Oregon Department of Transportation
555 13th Street NE, Suite 2
Salem, OR 97301 Ph: 503-986-4104

From: GREGOR Brian J
Sent: Wednesday, May 14, 2003 2:07 PM
Subject: R Week 5 Lesson 3, Beginning Statistical Analysis

R Week 5 Lesson 3, Beginning Statistical Analysis

Last week you were introduced to a number of functions for describing and summarizing data. These included:
min() and max() to calculate the minimum and maximum values of a vector;
range() to calculate the lowest and highest values of a vector;
mean() and median() to calculate the mean and median of a vector;
var() and sd() to calculate the variance and standard deviation of a vector;
summary() to show summary statistics; and
table() to tabulate cases.

This lesson starts off with additional uses of the table() function, shows how tapply() can be use to create tables of weighted data, and shows how continuous variables can be made into factors with cut(). Tomorrow's lesson will introduce functions for carrying out significance tests and linear regression.

As you may recall, table() produces multi-way classification tables. To see this, first load the MASS library: 'library(MASS)'. Then load the 'quine' data set: 'data(quine)'. This is data from a study of school absences among Australian school children of different sex, ethnicity, age and learner group. To get a crosstab of sex and age you enter 'table(quine$Sex, quine$Age)', resulting in:

```
    F0 F1 F2 F3
  F 10 32 19 19
  M 17 14 21 14
```

The rows are the factors corresponding to the first table() argument and the columns corresponding to the second argument.

Likewise a three-way table would be produced by adding another data column argument: 'table(quine$Sex, quine$Age, quine$Lrn)'. Notice that this produces an array. An array extends the matrix concept to higher dimensions. In this case, it is a three dimensional array because we asked for a table of three variables. The rows and columns are sex and age respectively. Each table in the array corresponds to the learner group.

Multi-way tables in this format can be hard to understand. The ftable() function helps by producing a "flat" table. Try this: 'ftable(quine$Sex, quine$Age, quine$Lrn)'.

If you have assigned the results of table() to an object, you can use the margin.table() functions to compute the table margins. For example: 'quine.table <- table(quine$Sex, quine$Age)'; 'margin.table(quine.table,1)' to get row margin; 'margin.table(quine.table,2)' to get column margin. Note that you can also get these results by using the rowSums(), colSums() and apply() functions.

Sometimes you will want to tabulate a value, rather than the number of cases (as table() does); for example, when tabulating population totals from a survey using expansion factors. Table will not work here, but tapply() will. You can test this with the quine data. Lets tabulate the number of days absent by sex and age: 'tapply(quine$Days, list(quine$Sex, quine$Age), sum)'. You could calculate the average days absent by each combination of sex and age by combining the previous calculations like this: 'tapply(quine$Days, list(quine$Sex, quine$Age), sum) / table(quine$Sex, quine$Age)'.

To tabulate data, it's often necessary to convert continuous data, such as income, into categorical data (factors). The 'cut()' function is used for this. Let's try this out on the 'UScereal' data set: 'data(UScereal)'. Now let's create categorical variables for fat content. First take a look at the distribution of fat content by cereal by entering 'hist(UScereal$fat)'. Then cut the data into categories using the breakpoints of 0-1, 1-2, 2-4, 4-10: 'fat <- cut(UScereal$fat, c(0,1,2,4,10), labels=c("low", "med", "high", "oh my"), include.lowest=TRUE)'. The first argument is the 'UScereal$fat' vector that is to be converted. The second is a vector of all the break points including the minimum and maximum. The third is a vector of labels to be given to each category. If this is omitted, then the default labels will be the ranges in the following form: [0,1], (1,2], (2,4], (4,10] where parentheses indicate an open interval (e.g. greater than) and brackets indicate a closed interval (e.g. less than or equal to). The other option for labels is 'labels=FALSE'. If this is entered, the factor levels will be named with simple integer codes. The last argument, 'include.lowest=TRUE', is necessary to have the function include the lowest values; zero in this case.

The new 'fat' variable is a factor (check this with 'str(fat)'). As such, it can be tabulated. For example, you can see a table of cereals by manufacturer and fat content by entering 'table(UScereal$mfr, fat)'. You can also now use this variable to see how average sugar content varies by fat content: 'tapply(UScereal$sugar, fat, mean)'.

Tomorrow we'll continue our examination of cereals and other data using statistical measures. Here's a summary of functions covered today.
table() creates cross tabulations of categorical values (factors).
ftable() also creates cross tabulations, but displays them in a flat vs. array format that makes multi-way tabulations easier to understand.
margin.table() calculates the margins of a table.
tapply() creates tables of continuous variables.
cut() creates a factor variable from a continuous variable.

Brian Gregor, P.E.
Transportation Planning Analysis Unit
Oregon Department of Transportation
Brian.J.GREGOR@odot.state.or.us
(503) 986-4120

From: GREGOR Brian J
Sent: Thursday, May 15, 2003 11:20 AM
Subject: R Week 5 Lesson 4, Elementary Statistical Analysis

R Week 5 Lesson 4, Elementary Statistical Analysis

Today we will pick up from where we left off and cover some basic statistical tests. So many statistical analyses are available with R that it would take books and years to cover them all. All I can hope to do here is get you started with a few functions and explain some syntax that is common to many statistical analyses.

First let's start with using the 't test' to compare means of some of the data we've been working on. We do this with the 't.test()' function. To try this out, load the 'MASS' library and the 'quine' data. We'll compare the mean number of days absent by various characteristics, starting with sex. The first example is of a one-sample test. Let's assume for the purposes of this example that the population mean of days absent for girls is the mean in the quine data: 'girl.mean <- mean(quine$Days[quine$Sex=="F"])'. Then to test whether the mean for boys is significantly different from this enter 't.test(quine$Days[quine$Sex=="M"], mu=girl.mean)'. The first argument is a vector of the data for boys and the second argument is the mean value that is the reference. The results of the t test are printed out at the prompt. It should look like this:

```
        One Sample t-test

data:  quine$Days[quine$Sex == "M"]
t = 1.3331, df = 65, p-value = 0.1871
alternative hypothesis: true mean is not equal to 15.225
95 percent confidence interval:
 13.86540 22.04369
sample estimates:
mean of x
 17.95455
```

You can see from the t statistic in the third line and the corresponding p-value that boy mean is not significantly different than the girl mean ($t < 2$, $p > 0.05$). You can also see on following lines the 95% confidence interval and the boy mean.

To show the one-sample test, I made the assumption that the girl population mean was equal to the sample mean. Because it's not, we should use the two-sample t test. Here's one way to do so: 't.test(quine$Days[quine$Sex=="M"], quine$Days[quine$Sex=="F"])'. The syntax is similar, but instead of specifying a vector of data and a reference mean, you specify two data vectors. You can see that the t statistic is lower in this case, because the two-sample test is accounting for the sample variation of both means.

The preceding example is not the simplest way of specifying a t test. R provides an alternative for specifying model formulas which looks like this: 't.test(Days ~ Sex, data=quine)'. You can read this as 'Days by Sex' or 'Days as a function of Sex' for the quine data. Model formulae simplify the specification of statistical models, tests and graphs and are commonly used in R. If you attach your data frame, the specification becomes even easier: 'attach(quine)'; 't.test(Days ~ Eth)'.

Model formulae are also used in the specification of regression models. We'll look at this with a simple linear regression. We'll use the 'whiteside' data set for this. Load the data set 'data(whiteside)'. This data set compares gas consumption in a house for different temperatures before and after the walls were insulated. Let's first plot the data. (Don't worry about the meaning of the plot commands just copy and paste them at the command prompt. We'll explain what they mean next week.)

```
opar <- par(mfrow = c(1,2)) # a graphic parameter call to plot two graphs in one
window
before <- subset(whiteside, Insul=="Before", select=c(Temp,Gas))
after <- subset(whiteside, Insul=="After", select=c(Temp,Gas))
plot(Gas~Temp, data=before, xlim=range(whiteside$Temp),
ylim=range(whiteside$Gas), main="Before Insulation")
abline(lm(Gas~Temp, data=before))
plot(Gas~Temp, data=after, xlim=range(whiteside$Temp),
ylim=range(whiteside$Gas), main="After Insulation")
abline(lm(Gas~Temp, data=after))
par <- opar # a graphic parameter call to set things back to previous conditions
```

You can see from these plots that insulation appears to have reduced gas consumption. Now let's estimate regression models for these two cases and compare them. The function to estimate a linear regression model is 'lm()'. As you see in the code above, that's what we used to plot the trend lines in the before and after plots. If you enter 'lm(Gas~Temp, data=before)' at the prompt, R reports the coefficients for the linear model:

```
Call:
lm(formula = Gas ~ Temp, data = before)

Coefficients:
(Intercept)         Temp
     6.8538      -0.3932
```

The first coefficient is the y intercept of the line and the second is the slope.

You can also assign the results of the linear model to an object: 'before.model <- lm(Gas~Temp, data=before); after.model <- lm(Gas~Temp, data=after)' This object can then be an argument for other functions. For example, the results of the linear models for before and after were arguments, individually, to the abline() function which took the intercept and slope information and drew a line on the plots. The object-oriented nature of R makes this sort of functionality possible. (Is R cool or not.) Summary is another function that can use a lm object. Try 'summary(before.model)'. You will see a number of summary statistics including the distribution of residual errors, the model coefficients, their standard errors, t statistic and corresponding probabilities. This is followed by residual standard errors, the squared correlation coefficient and the F statistic of overall significance.

Correlations between variables can be calculated with the cor() function. For example, 'cor(before$Gas, before$Temp)' gives a correlation coefficient of -0.9714978, the square of which is the R-Squared value in the regression statistics. You can do multiple correlations on a data frame easily by specifying the data frame as the argument to correlation. For example, we could analyze multiple correlations between attribute of the UScereal data as follows.

First load the data set: 'data(UScereal)'. Then enter 'cor(UScereal[,3:9])' and you get a table of correlations between all the selected variables.

```
          protein       fat    sodium     fibre       carbo      sugars
shelf
protein 1.0000000 0.4112661 0.5727222 0.8096397  0.54709029  0.18484845
0.3963311
fat     0.4112661 1.0000000 0.2595606 0.2260715  0.18285220  0.41567397
0.3256975
sodium  0.5727222 0.2595606 1.0000000 0.4954831  0.42356172  0.21124365
0.2341275
fibre   0.8096397 0.2260715 0.4954831 1.0000000  0.20307489  0.14891577
0.3578429
carbo   0.5470903 0.1828522 0.4235617 0.2030749  1.00000000 -0.04082599
0.2604599
sugars  0.1848484 0.4156740 0.2112437 0.1489158 -0.04082599  1.00000000
0.2900511
shelf   0.3963311 0.3256975 0.2341275 0.3578429  0.26045989  0.29005112
1.0000000
```

Here are a few references for more information on statistics using R:
Some free sources on the web:
"Simple R", John Verzani
"Data Analysis and Graphics Using R - An Introduction", J.H. Maindonald
Some books:
"Introductory Statistics with R", Peter Dalgaard
"Modern Applied Statistics with S-PLUS", W.N. Venables & B.D. Ripley

Here are the functions we covered:
t.test() does one-sample, two-sample and paired t tests. Tests can be specified in several ways including model formula.
lm() does linear regression. If the results of lm() are assigned to an object, then other functions can be used to use or analyze the results.
abline() plots the regression line.
summary() gives a summary of the regression results.
cor() computes the correlation coefficient for two vectors or, if given a data frame as an argument, the correlations between all pairs of variables.

Brian Gregor, P.E.
Transportation Planning Analysis Unit
Oregon Department of Transportation
Brian.J.GREGOR@odot.state.or.us
(503) 986-4120

From: STABLER Benjamin
Sent: Tuesday, May 20, 2003 12:02 PM
Subject: R Week 6 Lesson 1, The plot() function

R Week 6 Lesson 1, The plot() function

It is now time to start visualizing some of that data you have been manipulating and analyzing. The key function for visualizing data is plot(). The plot() function is a high-level function, which means that it can be used to start a plot (or graph). Low-level functions such as lines(), which will be covered later, affect already existing plots. Low-level plotting functions are a great way to add additional information to a plot (such as point labels and lines). But before we get into the more advanced parts of plotting in R, let's start with the plot() function.

The most significant two arguments to plot() are x and y, which represent vectors of the x points and y points that you want to plot. For example, "plot(x=c(1,2,3,4,5), y=c(2,2,3,3,4))" will plot the following points:

```
 x,y
(1,2)
(2,2)
(3,3)
(4,3)
(5,4)
```

Since this plot is pretty boring, let's try plotting some more exciting data. Load the UScereal data set with "library(MASS)" followed by "data(UScereal)". Next type "plot(UScereal$sugars, UScereal$calories)" to see if there is a linear relationship between sugar content and calories.

As you can see, R will figure out default x and y ranges for the plot area. But you can specify these with the xlim and ylim arguments to plot(). Try "plot(UScereal$sugars, UScereal$calories, xlim=c(0,20), ylim=c(0,300))". The xlim and ylim arguments expect a vector of length two which represents the lower and upper bounds.

Let's customize a few more of the plot settings with the main, xlab, ylab, and pch arguments. The main argument is what you set to specify the plot title. The xlab and ylab arguments are the x and y labels, while pch is the plotting character to use. Combining all of these into our previous plot function call we get: "plot(UScereal$sugars, UScereal$calories, xlim=c(0,20), ylim=c(0,300), main="Sugar v. Calories", xlab="Sugar Content", ylab="Calories", pch=20)". The plot now has a nice title, axis labels, and the point symbol used is smaller and has a solid fill. The range of values for pch is from 1 to 25 - I'll let you explore what each one is on your own.

At this point I think it is important to note a few things about plotting in R. Plotting commands can get rather long, and you don't want to have to type things over and over again. The best way to deal with this is write your R commands in a text editor and then copy and paste them into the R console. It is also nice to refer back to previous plot code that you have written to remember what arguments to use.

A final argument to plot() is the type argument. The type argument specifies what type of plot to draw. Choices for type include "p" for points, "l" for lines, "b" for both and "n" for no plotting. Let's use the "b" type plot with a

new data set. Load the airmiles data set with "data(airmiles)". The airmiles data set is of type time series, which means that R knows the interval between points and the beginning and ending dates of the data. We can plot the airmiles data with the following command: "plot(1937:1960, airmiles, type="b")". There is an easier way to plot this data however. Simply type "plot(airmiles)" and R will figure out the x-axis based on information stored in the time series data structure. R's object-oriented programming make things simplier for us again.

As I mentioned earlier, low-level functions such as lines() can be used to add data to a plot. Let's plot the airmiles data as points with "plot(1937:1960, airmiles, type="p", pch=8)". We can then use low-level function title() to add a title to the plot (this is the same as the main argument to plot). For example: "title("Growth in Airmiles")". We could connect the points that we drew using the lines() function. Try: "lines(1937:1960, airmiles, col="red")". It draws lines between each point in the same order as the order of the x and y vectors. The lines() function is one way to draw a model network in R.

In addition to continuous data, R can plot factor or categorical data. Try "plot(UScereal$shelf, UScereal$sugars, pch=16)" to see that R plotted the sugar content on the y-axis of the UScereal data set based on the cereal's shelf in the grocery store. Besides simple plots, R has a number of other plotting functions for specific purposes, such as plotting each column of a data frame on the same plot (see matplot()).

Because R is a programming language in addition to a program, R provides tremendous flexibility in creating plots and even more complex graphics. For a demonstration of the plotting capabilities of R, type: "demo(graphics)".

Finally, the hist() function produces a histogram. Try: "hist(UScereal$sugars)". Many of the same arguments that can be passed to plot(), such as xlab, ylab, and col can also be passed to hist().

FUNCTIONS

plot() - plot x and y data
hist() - create a histogram of a vector
title() - add a title to an existing plot
lines() - add lines to an existing plot

Benjamin Stabler
Transportation Planning Analysis Unit
Oregon Department of Transportation
555 13th Street NE, Suite 2
Salem, OR 97301 Ph: 503-986-4104

From: GREGOR Brian J
Sent: Wednesday, May 21, 2003 2:20 PM
Subject: R Week 6 Lesson 2, Lower Level Plotting Functions

R Week 6 Lesson 2, Lower Level Plotting Functions

Yesterday, Ben described how you use the higher level plotting commands plot() and hist(). He also introduced a few lower level plotting commands: title() and lines(). As you gain more understanding about R graphics, you will find that you can use these lower level commands to plot almost any type of graph you wish to. You are not confined to the canned graph formats that Excel or other packages provide. I'll get you started on building graphs from their elements. I'll also introduce you to a few commands that allow you to interact with your graphs.

Let's start by loading some suitable data: 'library(MASS); data(hills)'. The hills data frame includes data on the distance(dist), elevation gain(climb) and time(time) for a number of mountain climbs in Great Britain. Next add another variable indicating the overall steepness of each climb:
'hills$steep <- as.numeric(resid(lm(climb~dist, data=hills))>0)'. This statement first does a linear regression on climb vs. distance. The regression line shows the average slope. Climbs are steeper than average where the residuals of the regression are positive. This logical test is converted to a numeric vector (FALSE becomes 0 and TRUE becomes 1). Finally attach the data so that we can refer to the columns directly: 'attach(hills)'.

Now we'll build a graph of time vs. distance. Start by looking at the data to identify how to set up the graph axes. You can do this with 'summary(dist)' and 'summary(time)'. We'll have the x axis run from 0 to 30 and the y axis run from 0 to 200. Now plot an empty graph: 'plot(dist,time,type="n", xlab="", ylab="", xlim=c(0,30), ylim=c(0,200), axes=F)'. The type="n" argument tells R that nothing will be plotted. Assigning an empty string to xlab and ylab causes no x or y axis labels to be printed. Finally, the 'axes=F' argument turns off printing of axes. So what you now have is a graphics window with nothing in it. However, the the coordinate system has been set up so that you can add elements to the graph and they will be put in their proper places.

The x axis can be added with 'axis(1, at=seq(0,30,by=5))' and the y axis with 'axis(2, at=seq(0,200,by=20))'. The first argument to the axis() function identified which access to add. Axis numbering is as follows: 1 = bottom, 2 = left side, 3 = top, 4 = right side. The second argument to axis is a vector which identifies the values for the tick marks for the axis. In this case, the x axis has tick marks at 0, 5, 10, 15, 20, 35, 30. axis() can use an additional 'labels' argument which can be a vector of labels to place at each of the tick marks. If this argument is not used, R will print the 'at' values.

You can add a box around the whole plot area with 'box()'. You can specify the color and linetype of the box, but we'll just use the default values of solid black.

Now lets add data points and color them according to whether the climbs are more or less steep:
'points(dist[steep==0], time[steep==0], pch=16, col="blue")'
'points(dist[steep==1], time[steep==1], pch=16, col="red")'
Note that the syntax of points is similar to that of plot. The x values come first and the y values second. Then other arguments can be used to specify

various plotting characteristics. In this case, we plotted less steep climbs in blue and more steep in red.

We can place titles with 'title(main="Climbing Time vs. Distance", xlab="miles", ylab="minutes")'. The arguments have the same meaning as for the plot() function.

Next we'll add a legend, but before we do this, I'll show you how to identify the x and y coordinates of places in the graph. Enter 'locator(3)' at the console and then click on three different points in the graph with your mouse. Notice that the console returns the x and y values of the places you clicked. This can be used for any number of purposes, but we'll use it to place the legend. Enter 'legend.loc <- locator(1)' and then click a place in the graph where you would like the upper left hand corner of the legend to be. The coordinates will be stored in 'legend.loc'. Now you can place the legend using these coordinates as follows: 'legend(legend.loc$x,legend.loc$y,legend=c("Less Steep", "Steep"), col=c("blue", "red"), pch=16)'. Notice that the first two arguments are the x and y coordinates of the legend box. The third argument specifies the legend labels. The fourth argument specifies the corresponding colors and the last argument specifies the character type. Look at the help page for 'legend' to see the options that are available.

You can identify data points on the graph with 'identify()'. Enter 'identify(dist, time, rownames(hills), n=3)', and then click next to three of the data points. If you click slightly off to any side of a dot, the label will be printed on that side. The first argument to 'identify' is the vector of x values and the second argument is the vector of y values. The third argument is the corresponding vector of labels to be used. In this case the row names. The last argument specifies the number of points to be identified. If no value is given for this argument, you can keep on identifying points until you click on the 'stop locator' menu item of the graphics window.

We'll finish up by adding an annotation to the graph. This will be a short bit of text and an arrow pointing from the text to one of the dots. Start by locating the dot the arrow will point to. Enter 'arrow.head <- locator(1)' and click on the blue dot located at about 3.5 miles and 80 minutes. Then locate the text location by entering 'text.loc <- locator(1)' and clicking in the empty area above the dot. Then enter 'text(text.loc$x, text.loc$y, "Not feeling well")'. Finally, enter 'arrows(text.loc$x, text.loc$y, arrow.head$x, arrow.head$y)'. The arguments to 'text' in this example should be fairly self explanatory. Check the help page to see the options that are available. The first two arguments to the arrows function are the x and y coordinates of the arrow tail. The second two arguments are the x and y coordinates of the arrow head. This function also has a number of options such as the size and angle of the arrow head, the direction of the arrow, and whether it is to be single or double-headed.

This may have seemed a lot of work to do a graph. If you have a simple graph, it's best to use the higher level plot functions. The lower level functions are there to allow you to build custom graphs. In most cases, you would be defining a function which includes the command you want. You would then use this as your own higher-level plotting function.

Here are the plotting commands we used today:
plot() we used to set up a blank plot
axis() draws an axis
box() puts a box around the plot area

points() adds points to a graph
title() adds titles
locator() returns the x and y values of mouse clicks in the graph area
legend() adds a legend
identify() identifies data points
text() adds text to a graph
arrow() allows you to draw arrows

From: STABLER Benjamin
Sent: Thursday, May 22, 2003 3:30 PM
Subject: R Week 6 Lesson 3, par() and other graphical functions

R Week 6 Lesson 3, par() and other graphical functions

Start with "data(attitude)" to load the Chatterjee-Price Attitude data set. Today, we are going to experiment with the par() function, which sets global graphical parameters. Simply type "par()", which returns a list of 68 items. These are the defaults for graphical parameters for plotting in R. For example, par()$type is the setting for the default type of plot - "p" for points. Let's change one of the settings and see what happens. Type "par(lty=3)" followed by "plot(attitude$complaints,attitude$ratings, type="l")". R will plot a dotted line of the data, as opposed to the default solid line that we are use to.

The par parameters are the same as those used in the plotting functions describe earlier this week. So when you set "lwd=2" (line width equal to 2), you set the graphical parameter "lwd" to 2. As an argument to a plot() function or a low-level function such as lines(), the setting was only temporary. When you set these with par(parameter = new value)" you change them until you restart R or set the graphical parameters back to the defaults (more on this later).

We can set parameters with the following syntax: "par(parameter = new value)". The best way to learn the par commands is to refer to the help page for par(). Type "?par" to load the par help page in your Internet browser. Different parameters include plot type, background color, margin sizes, colors, font sizes, line widths, axis settings, symbol sizes, and plotting window settings such as mfrow and mfcol.

The mfrow and mfcol graphical parameters are very useful. To show you what they do type "par(mfrow=c(2,3))". Then copy and paste the following lines to R.

```
plot(attitude$rating,attitude$complaints)
plot(attitude$rating,attitude$privileges)
plot(attitude$rating,attitude$learning)
plot(attitude$rating,attitude$raises)
plot(attitude$rating,attitude$critical)
plot(attitude$rating,attitude$advance)
```

As you can tell, mfrow split the plotting window into a matrix of 2x3 plotting windows and plotted each plot in its own frame. The only difference between mfrow and mfcol is the order that R moves from one frame to the next - mfrow is in row major order while mfcol is in column major order. When we get to programming in R, we will see a better way to plot these six plots. This plot of rating versus the other six variables is rather useful, but it might be even better to plot every variable against every other variable. In R this is easy. Type "plot(attitude)" to produce a 6x6 matrix to visualize the relationships. Our six plots from earlier as now just the first column of this more complicated plot. Again, it is that object-oriented nature of R making things easier.

There are alternatives to mfcol and mfrow. The layout() function allows more flexibility in splitting up the plotting region as does split.screen(). A totally different, and much more complex, environment for plotting in R is represented by Trellis graphics. The lattice package is where R stores the Trellis graphics functions. It is built on a different graphical model called grid and is often incompatible with traditional R plot() and par() graphics. If you are looking for a more consistent graphical "style" and/or advanced

conditioning plots then Trellis is for you. Otherwise, R's traditional plotting environment should work.

Sometimes while experimenting with different par settings you realize that you want to revert back to the defaults. This is easy to in R since everything is stored as an object. par() is just a list so it can be assigned to an object by just typing "opar <- par()". Then use "par(opar)" to set the values of par to the values of the opar list.

All graphical data must be sent to a graphical device. When you plot() some data, the default device opened by R is a window. A window is simply the plot window that we have been using in this lesson. A new window with no data can be created with "windows()". But other graphical devices can receive data. Type "pdf("test.pdf")" to open a pdf file in the working directory that will receive all R graphical output until told otherwise. Type "plot(attitude$rating,attitude$advance)" and notice that R does not open a plotting window. Instead it sent the plot to the PDF file. R will save each new plot to a new page in the pdf file. The dev.list() and dev.off() functions can be used to list all currently active devices and to close the device number returned by dev.list. Graphical windows can be saved to various format by using the File drop down menu in the graphical window or by using the savePlot() function. The savePlot() function takes three arguments: 1) the filename to save to, 2) the file type (pdf, jpg, wmf, bmp, ps) and 3) the device to save (the default being the current device).

That should be enough for today. Tomorrow Brian is going to present some more specialized plotting functions such as barplot(), matplot(), and dotchart().

Functions

par() - to review or set graphical parameters
pdf() - to create a pdf file for graphical output
savePlot() - to save a plot window to file
windows() - to create a new graphical window
dev.list() - to list the active graphical devices
dev.off() - to close a graphical device

Benjamin Stabler
Transportation Planning Analysis Unit
Oregon Department of Transportation
555 13th Street NE, Suite 2
Salem, OR 97301 Ph: 503-986-4104

From: GREGOR Brian J
Sent: Thursday, May 29, 2003 3:00 PM
Subject: R Week 7 Lesson 1, More High Level Graphing Functions

R Week 7 Lesson 1, More High Level Graphing Functions

In this last plotting lesson, I will cover a variety of handy plotting functions. These include curve(), matplot(), barplot(), dotchart(), stripplot(), and boxplot()

curve() allows you to plot expressions or other functions that return numerical results. Here's an example: 'curve(-0.02*x^3 + 0.4*x^2 - 3*x, from=0, to=10, n=100)'. The first argument is the expression that is to be graphed. The second and third arguments are the lower and upper x values. These could be specified with the xlim argument instead. 'curve(-0.02*x^3 + 0.4*x^2 - 3*x, xlim=c(0,10), n=100)' gives the same result. The fourth argument gives the number of points to plot. If you want to add a curve to an existing plot, you use the 'add' argument like this: 'curve(-0.02*x^3 + 0.5*x^2 - 3*x, from=0, to=10, n=30, add=T, type="b", pch=1, col='red')'. Notice that this last example also used 'pch' and 'col' arguments to specify the plotting character and color.

matplot() allows you to plot a whole matrix of values. For this example, lets plot the same polynomial expression as above, but vary the coefficient on the squared term from 0.1 to 0.5. We can make a matrix of values using the 'outer()' function:
'poly.mat <- outer(seq(1,10,by=0.5), seq(0.1,0.5,by=0.1), function(x,y) - 0.02*x^3 + y*x^2 - 3*x)'
Then we can plot all the respective curves with matplot() like so: 'matplot(seq(1,10,by=0.5), poly.mat)'. The first argument is the vector of x values. The second argument is a matrix of y values. matplot() plots each of the columns in turn. As you see from this plot, the default behavior of matplot is to plot points using numbers; each in a different color. By using the type="l" argument, the plot is changed to a line plot: 'matplot(seq(1,10,by=0.5), poly.mat, type="l")'.

Now load some data to display in the next set of charts:'data(airquality) ; data(VADeaths)'.

barplot() will plot either a vector of values as a single set of bars, or a matrix of values with stacked bars or side-by-side bars. Let's start with plotting a single set of bars using the air quality data. We'll plot mean temperature by month: 'barplot(tapply(airquality$Temp, airquality$Month, mean))'. Notice how the tapply() function is used to calculate the vector of means. Barplot then makes the plot from this vector. We can add the axis labels and title as follows: 'barplot(tapply(airquality$Temp, airquality$Month, mean), names.arg=c("May", "June", "July", "August", "September"), ylab="Degrees Fahrenheit", main="Mean NYC Temperatures")'. You can easily change the orientation of the bars by using the 'horiz=T' argument.

Now we'll plot the VADeaths data to show how barplot() handles multiple data series. VADeaths is a matrix of death rates in the state of Virginia by age (rows) and sex and location (columns). 'barplot(VADeaths)' produces stacked barplots with each column being one grouping. Side-by-side bars can be shown by using the 'beside=T' argument: 'barplot(VADeaths, beside=T)'. A legend will automatically be added with the 'legend.text' argument: 'barplot(VADeaths, beside=T, legend.text=c("50-54", "55-59", "60-64", "65-69", "70-74"), col=rainbow(5))'.

Notice also that the plotting colors can be changed. In this case, five colors were chosen from the rainbow palette. You could instead specify a vector of colors such as 'c("salmon", "springgreen", "tomato", "skyblue", "violetred")' from R's large selection of colors. You can get a list of all the available color names by entering 'colors()'.

If you want to have some fun, install the RColorBrewer package from CRAN and load the library. This package contains a number of nice color palettes for graphing and mapping. Then enter: 'barplot(VADeaths, beside=T, legend.text=c("50-54", "55-59", "60-64", "65-69", "70-74"), col=brewer.pal(5, "Pastel1"))'

dotchart() provides an alternative to barplot() that may be a preferred way of showing data in some instances. You can test this out with 'dotchart(VADeaths)'. For me, this presentation of the data is less cluttered than the barplot of the same data.

Sometimes it's easier to see patterns in data by ordering it as well. For example, we could order the dotchart categories in the order of their mean values. This is easy to do in R:
'dotchart(VADeaths[,order(apply(VADeaths,2,mean))])'. The apply statement calculates the mean values for each column. By feeding that to the order() function, you get the indexes to use to put the columns of the VADeath data in the order you want for graphing.

stripchart() is a one-dimensional scatterplot for categorical data. For example, we can show the distribution of the New York City temperature data by month as follows: 'stripchart(airquality$Temp~airquality$Month, pch=20)'. Notice that stripchart() allows you to use model formula notation. Here we're plotting temperature as a function of the month. If you have a lot of data, some of the points may be hidden by others. The default behavior of stripchart() is to "overplot" coincident data points. You can see all the points by specifying their display method as "jitter" or "stack". Try the following examples to see what these do.
'stripchart(airquality$Temp~airquality$Month, method="jitter", pch=20)'
'stripchart(airquality$Temp~airquality$Month, method="stack", pch=20)'

Data distributions can also be compared using boxplot(). This function also allows you to use model formulae as well:
'boxplot(airquality$Temp~airquality$Month)' This shows boxplots of the temperature distributions for each month. The boxes show the interquartile range with the line across the middle showing the median values. The "whiskers" on either side of the box encompass all data points within some specified distance of the interquartile range. The default value is 1.5 times the interquartile range. Points beyond the whiskers are plotted individually. (If you're not familiar with boxplots, it may help you to enter 'tapply(airquality$Temp, airquality$Month, summary)' and compare the results with the plot.) You can add a notches to the boxplots to facilitate comparison of medians with the notch=T argument.

That should get you going with plotting data. Here are the functions we covered:
curve() plots mathematical expressions
matplot() plots matrices. Each column is a data series.
barplot() plots bar plots
dotchart() is an alternative to barplot()
stripchart() plots one-dimensional scatterplots of data organized by category

boxplot() provides another way to plot data distributions organized by category

Brian Gregor, P.E.
Transportation Planning Analysis Unit
Oregon Department of Transportation
Brian.J.GREGOR@odot.state.or.us
(503) 986-4120

From: STABLER Benjamin
Sent: Friday, May 30, 2003 11:44 AM
Subject: R Week 7 Lesson 2, Writing Your Own Functions

R Week 7 Lesson 2, Writing Your Own Functions

So far we have pretty much been using R's built-in functions. These include such functions as rowSums() and mean(). Today we will write our own functions, which is probably the most productive aspect of R. Many of the built-in R functions such as mean.default() (which is the function that R usually uses when you call the mean() function) are also written in R, so once you understand how to write your own functions, you will be able to deconstruct many of R's built-in functions. Let's start with a simple function by typing:

```
myexp <- function(x, y) {
      result <- x^y
      result
}
```

The purpose of our "myexp" function is to exponentiate x to the y power. So myexp(2,3) will return 8. But first let's discuss function definition.

1) Function definitions are almost always more than one line. This is really the first time in the course that we have introduced multiple line expressions. In order to extend an expression to multiple lines, you need to use "{" and "}" (curly braces). These tell R that the expression does not end on this line, but rather it extends until the next curly brace. R assumes that an expression ends when it comes across a new line, but this can be overridden by curly braces.

2) Functions are defined with the "function" function. The "function" function takes the arguments of the function in parentheses, separated by commas. This is the same as when we call R functions such as plot(x,y), except that this time we get to decide what to call the arguments and what order to put them in. Thus "myexp" could have been defined as follows:

```
myexp <- function(base, exponent) {
      result <- base^exponent
      result
}
```

3) The function definition is assigned to an R object. This object is created with the name of our user defined function. So we can then call our function by name: "myexp(2,3)".

4) Functions return a single object. As a result, the last line of a function definition should be the object that you want the function to return. In our function above, we calculated x^y and assigned the result to "result". As you know, when you assign the results of a calculation to an object, R does not show you the results. Functions are similar in this way in that the assignment takes place within the function environment and in order to show you the results of the function you must explicitly ask R to return the result.

5) Actually, functions return the last unevaluated expression - which is often just an object as I described in point four. What this means is that our exponent function could be simplified to:

```
myexp <- function(base, exponent) {
```

```
      base^exponent
}
```

Since "base^exponent" is the last expression of the function, R will return the resulting value of that expression, which is the same as the value for "result" in our previous function definition. This is similar to inputting expressions at the R prompt since 2^3 will automatically return 8. If we had written our function like:

```
myexp <- function(base, exponent) {
      result <- base^exponent
}
```

Then R would not return result since the expression was evaluated and assigned to the result object, which only exists within the function environment.

6) As item four and five hinted at, calculations within functions work in their own environment. When you type x <- 5 at the prompt in R, R evaluates that expression in the global environment. But when you put x <- 5 inside a function, R evaluates the expression within a local function environment. Each time a function is called, it creates its own environment to work in. So try typing the following:

```
x <- 2

myx <- function(y) {
      x <- y
      x
}

myx(5)

x
```

For the last line R returns 2 for x since that is the value of x within the global environment, which is where x is defined. We sent the myx function the value of 5 for y and then R evaluates the insides of the function within a separate "local" environment for the function. So 5 is then assigned to a new x inside the myx environment. But when we finish the function, we terminate the local function environment and all calculations within it are lost. Function environments are temporary.

7) Default values can be specified in function definitions. So we could rewrite our function as:

```
myexp <- function(x, y=2) {
      result <- x^y
      result
}
```

Notice the =2 after the y in the function definition. This tells R that if no value is supplied for the y in the function call, then use the value of 2. So "myexp(5)" will return 25. This is a nice way to set defaults in functions. For example, I wrote a function to plot a model network in R that has a centroid connectors argument with the default set to FALSE. That way R will not plot centroid connectors, but if the user wants them then they can easily be plotted by changing the value to TRUE.

8) As I mentioned earlier, functions usually return a single object. So if you want to return multiple objects, then just put all the objects in a list object and then return the list.

9) Finally, just as with R's built-in functions, you can assign the results of a function to an object. So "newvalue <- myexp(5)" will set newvalue to 25.

The only function covered today is the function() function.

Benjamin Stabler
Transportation Planning Analysis Unit
Oregon Department of Transportation
555 13th Street NE, Suite 2
Salem, OR 97301 Ph: 503-986-4104

From: GREGOR Brian J
Sent: Tuesday, June 03, 2003 10:18 AM
Subject: R Week 8 Lesson 1, Building Programs

R Week 8 Lesson 1, Building Programs

Last week, Ben introduced you to defining functions. That is a very important element of building programs in R. Today I will introduce you another important part of building R programs: writing scripts.

A R script is simply a text file of R commands that is brought into R with the source() function. R scripts are typically named with the ".R" extension. I've attached a script file "analyze_commute.R" to this email for use in this lesson. Save it to the directory you're using for these lessons and then start your Rgui. To "source in" the script you just saved, you can enter 'source("analyze_commute.R")' at the command prompt. You can also use the menu command 'File/Source R Code'. You can then choose the file you want to source in using the file selection window that pops up.

Now open the "analyze_commute.R" file in your text editor. If you haven't installed a text editor which supports syntax highlighting for R, or if you haven't activated syntax highlighting for R, I strongly suggest that you do so. It makes reading and writing R programs much easier. UltraEdit, the editor which most people in TPAU use, has an R syntax highlighting definition. You can also get syntax highlighting with jEdit and several other text editors. You can see what's available at http://www.sciviews.org/.

The first thing to notice in this script is the use of the "#" symbol to designate comments. These can be place anywhere in a line. All text that follows a "#" is ignored by the R interpreter. If you comment your script liberally, you can document your analysis in the code itself. This is one of the big advantages of using R for analysis. People can see exactly what you have done and can replicate your analysis exactly. Also notice that you can use blank lines in scripts to separate sections and make them easier to read.

What this script does is open a connection to the Census 2000 county to county worker flow text file for Oregon on the internet. (Note: If the internet connection fails, check the "Target" property for the Rgui shortcut. It should include the "--internet2" option. This was covered in the first lesson on installing R.) This file contains records of the number of commutes from each Oregon County to every other county in the US (and to foreign countries). The script then reads that file into an R object and does some processing of the fields to get them into shape for the analysis. Then it extracts the records for instate commuting and does some analysis of them. Finally it saves the results as a tab-delimited text file.

I won't explain each command in the script because I've put the documentation in the file. Most of the commands have been covered in previous lessons. You can use the R help to look up the new commands. Email me if you have any questions.

Brian Gregor, P.E.
Transportation Planning Analysis Unit
Oregon Department of Transportation
Brian.J.GREGOR@odot.state.or.us
(503) 986-4120

analyze_commute.R attachment

```
# analyze_commute.R
# This script downloads Census 2000 commuting data and performs a variety of
analyses

# Connect to the Census 2000 county to county commute file for Oregon by
residence county
# For file data documentation see
https://www.census.gov/population/cen2000/commuting/coxcolayout.txt
census.con <-
url("https://www.census.gov/population/cen2000/commuting/2KRESCO_OR.txt")

# Read downloaded file into a data.frame
    # specify field widths (varies from Census documentation because a space is
    located between each field
    census.width <- c(2,4,5,5,42,3,4,5,5,42,7)
    # set field names
    census.name <- c("res.state", "res.county", "res.msa", "res.pmsa",
    "res.name", "wrk.state", "wrk.county", "wrk.msa", "wrk.pmsa", "wrk.name",
    "count")
    # set field types
    census.type <- c("character", "character", "character", "character",
    "character", "character", "character", "character", "character",
    "character", "numeric")
    # read in file
    commute <- read.fwf(census.con, widths=census.width, col.names=census.name,
    colClasses=census.type)

# Close the connection to the Census file
close(census.con)

# Format commute data frame
commute$res.name <- gsub("^ +","", commute$res.name) #remove leading spaces from
res.name
commute$res.name <- gsub(" +$","", commute$res.name) #remove trailing spaces
from res.name
commute$wrk.name <- gsub("^ +","", commute$wrk.name) #remove leading spaces from
wrk.name
commute$wrk.name <- gsub(" +$","", commute$wrk.name) #remove trailing spaces
from wrk.name
commute$wrk.state <- substring(commute$wrk.state, 2, 3) #remove leading zero
from wrk.state fips

# Set up an index of county fips codes and names
county.index <- commute[!duplicated(commute$res.name),c("res.county",
"res.name")] # take the res.county and res.names fields only where res.name is
not duplicated

# Select the records of commutes within the state
instate.commute <- commute[commute$wrk.state == unique(commute$res.state),]

# Do some analysis of in state commuting

    attach(instate.commute)
```

```
summary(count) # summary of intercounty commutes within Oregon
hist(count) # summary of intercounty commutes within Oregon

# Make a county to county origin and destination matrix
commute.od <- tapply(count, list(res.county, wrk.county), sum) # sum
commutes by origin and destination combination
commute.od[is.na(commute.od)] <- 0  # NAs need to be converted to zeros
rownames(commute.od) <- colnames(commute.od) <- county.index$res.name  #
name rows and columns with the county names

# Summarize OD matrix in several ways
commute.origins <- rowSums(commute.od) # compute computes by origin county
commute.destinations <- colSums(commute.od) # compute commutes by
destination county
internal <- commute.od[row(commute.od)==col(commute.od)] # internal commutes
are on the diagonal of of the od matrix
internal.pct <- 100 * internal / commute.origins # compute the internal
percentage
internal.pct <- round(internal.pct, 1)  # round the internal percentage to
the first decimal place
outflow <- commute.origins - internal # compute the number of commuters
leaving the county
inflow <- commute.destinations - internal # compute the number of commuters
entering the county

# Build a data frame of the results
commutes <- data.frame(cbind(origins=commute.origins,
destinations=commute.destinations, internal, internal.pct, outflow, inflow))
# put the results together

# Save the results table
write.table(commutes, "OR_Commutes.txt", row.names=T, col.names=T, sep="\t")

detach(instate.commute)
```

From: STABLER Benjamin
Sent: Wednesday, June 04, 2003 8:53 AM
Subject: R Week 8 Lesson 2, Program Control

R Week 8 Lesson 2, Program Control

FOR

A common program control mechanism is the for loop. Essentially a for loop will execute an expression i number of times, with i usually being a counter object. Let's look at a for loop in R.

```
for (i in 1:10) {
      print(letters[i])
}
```

which returns the first ten letters of the alphabet. The basic structure of the for loop is a) for, b) (name for each element in a vector), c) { expression to execute in curly braces if it extends over one line }. So to put the code in words, it would be "for i in each value in one to ten, print letter at position i." So R starts by setting i equal to the first value in the vector after the "in" in the condition, which is 1, and then prints letters[1]. It cycles through all the values in the 1:10 vector and then ends. For loops are not functions though, so the i value at the end of the cycle (10 in this case) stays assigned in the global workspace. For loops are a great way to automate repeative tasks.

Unlike some other programming languages, R allows you to use an element value of a vector instead of just a counter number. An example will make this clearer. The traditional way is:

```
states <- c("Washington","Oregon","California","Arizona","New Mexico")
for (i in 1:length(states)) {
      print(states[i])
}
```

while the more efficient way to write it in R is:

```
states <- c("Washington","Oregon","California","Arizona","New Mexico")
for (i in states) {
      print(i)
}
```

Thus i in the second for loop is equal to the vector element (the state name in this example). Sometimes you will use the new method and other times you will use the traditional method – it depends.

WHILE

Related to for() is the while() function. The while() function will execute an expression as long as the conditional statement is met. Let's illustrate this with an example.

```
x <- 1
while ( x<5 )  {
      print("hello there")
      x <- x+1
```

```
}
```

As you can see, this little bit of code will print "hello there" four times. At the start of each loop, the while condition is checked. Each time the while loop is evaluated, R checks the value of x versus 5. If x is less than 5 then it runs the code inside the curly braces. If x is greater than or equal to 5 then the loop is terminated. Note that greater than or equal to is >= while less than or equal to is <=. It is best to hand trace this loop if you are having any troblem figuring it out.

IF and ELSE

Conditional execution of statements can be very valuable. In R, the if() else() functions provide conditioning. Try:

```
x <- 5
if( x==5 ) { print("You got a match")
 } else { print("sorry no match") }
```

If x is equal to 5 then evaluate the first expression, else evaluate the second expression. R will return "You got a match".

There are a few things to note about the if() test. 1) To test if a value is equal to another value you use ==, not = (since = assigns). 2) It is best to put all your if() else() statements in curly braces so that R does not get confused. The reason for this is the way that R parses code. In other programming languages, you often have to end a line of code with a semicolon (to tell it that it has reached the end of a line). But R just assumes a new line signifies the end of the expression. This leads to trouble in if() else() situations because you will often want to write the TRUE and FALSE expressions on different lines. The solution - curly braces. Remember from our functions lesson that curly braces allows you to split expression across multiple lines. That means that it is best write your if() else() expression like the one above,

```
i.e. if (condition) { TRUE expression
 } else { FALSE expression }
```

The key here is the closing curly brace of the TRUE expression is on the next line. This lets R know that the entire if() statement is not yet done. Finally, else() statements are optional. Furthermore, you can have mutiple if()s if necessary.

IFELSE

The ifelse() function is a more compact version of if() else() above. The basic structure of ifelse() is ifelse(condition, true expression, false expression).

BREAK

The break() function will break out of a loop. So for example, try:

```
x <- 1
while ( x<5 )  {
      print("hello there")
      if (x==3) break()
      x <- x+1
}
```

Notice that R only printed three "hello there"s instead of four. That is because R broke out of the loop when x was equal to three.

NEXT

The next() function is related to break() except that next will skip to the next cycle in a loop. For example:

```
for (i in 1:10) {
      if(i==4) { next() }
      print(letters[i])
}
```

The result is that R does not print "d" since d is letters[4] and if i was equal to 4 then R skipped to the next cycle and did not finish evaluating the code that remained in cycle four.

&, |, and !

The "and" (&), "or" (|) and "not" (!) operators are often used in conditional tests. For example:

```
x<-1
y<-4
if(x==1 & y==4) { print("hello") }
if(x==1 & y==5) { print("goodbye") }
if(x==1 | y==1) { print("hello") }
if(x!=5) { print("goodbye") }
```

As you can see, these operators, often called boolean operators, can be very useful in evaluating TRUE/FALSE conditions.

FUNCTIONS

for() - repeat the expression that follows i number of times
while() - repeat the expression that follows as long as the condition is satisfied
if() - execute the expression that follows if the condition is satisfied
else() - execute the expression that follows - note this must come after an if()
ifelse() - a compact form of if() and else()
break() - terminate a loop / break out of a loop
next() - skip to the next cycle in a loop
& - and
| - or
! - not

Benjamin Stabler
Transportation Planning Analysis Unit
Oregon Department of Transportation
555 13th Street NE, Suite 2
Salem, OR 97301 Ph: 503-986-4104

From: GREGOR Brian J
Sent: Friday, June 06, 2003 8:46 AM
Subject: R Week 8 Lesson 3, Using Functions and Program Control to Automate Your Script

R Week 8 Lesson 3, Using Functions and Program Control to Automate Your Script

In this lesson you will see how writing functions and using some of the program control functions that Ben introduced can be used to generalize and add automation to a script. Specifically, I show you how the script I sent a couple of lessons ago can be modified so that it can retrieve and process the Census commuting files for any number of states. Moreover, the new script can retrieve either the table of commutes by residence or commutes by workplace. The results are contained in the script that is attached to this lesson.

The first step in the process of generalizing the script was to convert most of it into functions. Then the functions could be called repeatedly for as many states as the analysis is desired for. Most of the script could be wrapped into one big function, but I decided to create two functions to separate the file reading and formatting from the analysis of instate commutes. This way the file reading function could be used in another application without needing to be changed.

The first function is named "get.commute()". It takes two arguments, the state abbreviation and the type (RES = by residence, WRK = by workplace). Most of the function is like the previous script but there are some differences. Near the top of the function are a couple of "if" statements which check whether the argument values given to the function are valid. If they are not, program execution is stopped with the "stop()" function. The argument to this function is the text message to be displayed if the stop condition is triggered.

After the "if" statements, you see how the "state" and "type" arguments to the function are used in two "paste" functions to build up the url that identifies the file to be downloaded. With these two lines of code, the file downloading procedure was generalized so that it can process a request for any state’s data.

I’ll leave other changes in the script for you to identify and figure out what they do.

The second function, "summarize.instate.commutes", takes the data frame created by the "get.commute" function, removes the out-of-state commute records and does most of the analysis carried out by the previous script. (I left out the summarization of counts and the creation of a histogram.) This function returns a list whose elements are an origin-destination matrix of instate commutes and a summary of instate commutes by origin.

Following these function definitions are a few lines of script which control the process of getting the data for the desired states, analyzing it, and saving it. The first couple of lines define the vectors of states and types of commutes to be analyzed. The abbreviations of the states for which the data is to be pulled are contained in the "states" vector. The types of files to get are in the "types" vector. These are the only lines of script you need to change. Then the script has two nested for loops to cycle through each combination of state and type. The lines within the loops call the functions we defined using the state and type arguments coming from the for loop indexes and then save the results to files.

Try this script out using different combinations of states and see how it works.

That's it for this lesson and for the structured part of this course. At this point you should know enough about R to use it in your work. From now on, you will receive occasional emails with more advanced topics and scripts.

Brian Gregor, P.E.
Transportation Planning Analysis Unit
Oregon Department of Transportation
Brian.J.GREGOR@odot.state.or.us
(503) 986-4120

```
# analyze_instate_commute.R
# 6/5/03

# This script downloads Census 2000 county to county commuting data for a list
of states and then:
#     Creates and saves the data on instate commutes
#     Creates and saves an origin-destination matrix of the instate commutes
#     Summarizes the instate commutes and saves the results


# Define function for retrieving a commuting file from the Census
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
get.commute <- function(state="OR", type="RES"){

      # Check for valid state and type codes
      state.abb <- c("AK", "AL", "AR", "AZ", "CA", "CO", "CT", "DC", "DE", "FL",
"GA", "HI", "IA", "ID", "IL", "IN", "KS", "KY", "LA", "MA", "MD", "ME", "MI",
"MN", "MO", "MS", "MT", "NC", "ND", "NE", "NH", "NJ", "NM", "NV", "NY", "OH",
"OK", "OR", "PA", "RI", "SC", "SD", "TN", "TX", "US", "UT", "VA", "VT", "WA",
"WI", "WV", "WY")
      types <- c("RES", "WRK")
      if(!(state %in% state.abb)) stop("Must use valid state abbreviation") if(!
      (type %in% types)) stop("Type must be RES or WRK")

      # Make the file name and the url to get the data from
      commute.file <- paste("2K", type, "CO_", state, ".txt", sep="")
      url.file <- paste("https://www.census.gov/population/cen2000/commuting/",
commute.file, sep="")

      # Connect to the Census 2000 county to county commute file by residence or
work place
      # For file data documentation see
https://www.census.gov/population/cen2000/commuting/coxcolayout.txt
      census.con <- url(url.file)

      # Read downloaded file into a data.frame
      # specify field widths (varies from Census documentation because a space
is located between each field
      census.width <- c(2,4,5,5,42,3,4,5,5,42,7)
      # set field names
      census.name <- c("res.state", "res.county", "res.msa", "res.pmsa",
"res.name", "wrk.state", "wrk.county", "wrk.msa", "wrk.pmsa", "wrk.name",
"count")
```

```
      # set field types
      census.type <- c("character", "character", "character", "character",
"character", "character", "character", "character", "character", "character",
"numeric")
      # read in file
      commute <- read.fwf(census.con, widths=census.width,
col.names=census.name, colClasses=census.type)

      # Close the connection to the Census file
      close(census.con)

      # Format commute data frame
      commute$res.county <- gsub("^ +","", commute$res.county)
      #remove leading spaces from res.county
      commute$res.msa <- gsub("^ +","", commute$res.msa)
      #remove leading spaces from res.msa
      commute$res.pmsa <- gsub("^ +","", commute$res.pmsa)
      #remove leading spaces from res.pmsa
      commute$res.name <- gsub("^ +","", commute$res.name)
      #remove leading spaces from res.name
      commute$res.name <- gsub(" +$","", commute$res.name)
      #remove trailing spaces from res.name
      commute$wrk.county <- gsub("^ +","", commute$wrk.county)
      #remove leading spaces from wrk.county
      commute$wrk.msa <- gsub("^ +","", commute$wrk.msa)
      #remove leading spaces from wrk.msa
      commute$wrk.pmsa <- gsub("^ +","", commute$wrk.pmsa)
      #remove leading spaces from wrk.pmsa
      commute$wrk.name <- gsub("^ +","", commute$wrk.name)
      #remove leading spaces from wrk.name
      commute$wrk.name <- gsub(" +$","", commute$wrk.name)
      #remove trailing spaces from wrk.name
      commute$wrk.state <- substring(commute$wrk.state, 2, 3)
      #remove leading zero from wrk.state fips

    # Return the result
    commute

    }


# Define a function for creating a county to county origin destination matrix
and summarizing commutes
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

summarize.instate.commutes <- function(x=commute){

      # Set up an index of county fips codes and names
      county.index <- x[!duplicated(x$res.name),c("res.county", "res.name")] #
take the res.county and res.names fields only where res.name is not duplicated

      # Select the records of commutes within the state
      x <- x[(x$wrk.state == x$res.state) & (x$wrk.county != "000"),]

      # Make a county to county origin and destination matrix
      commute.od <- tapply(x$count, list(x$res.county, x$wrk.county), sum) # sum
commutes by origin and destination combination
```

```
      commute.od[is.na(commute.od)] <- 0  # NAs need to be converted to zeros
      county.index <- x[!duplicated(x$res.name),c("res.county", "res.name")] #
take the res.county and res.names fields only where res.name is not duplicated
      rownames(commute.od) <- colnames(commute.od) <- county.index$res.name  #
name rows and columns with the county names

      # Summarize OD matrix in several ways
      commute.origins <- rowSums(commute.od)
      # compute computes by origin county
      commute.destinations <- colSums(commute.od)
      # compute commutes by destination county
      internal <- commute.od[row(commute.od)==col(commute.od)]
      # internal commutes are on the diagonal of of the od matrix
      internal.pct <- 100 * internal / commute.origins
      # compute the internal percentage
      internal.pct <- round(internal.pct, 1)
      # round the internal percentage to the first decimal place
      outflow <- commute.origins - internal
      # compute the number of commuters leaving the county
      inflow <- commute.destinations - internal
      # compute the number of commuters entering the county

      # Build a data frame of the results
      commute.summary <- data.frame(cbind(origins=commute.origins,
destinations=commute.destinations, internal, internal.pct, outflow, inflow))  #
put the results together
      rownames(commute.summary) <- county.index$res.name

      # Return the OD matrix and results data frame as a list
      list(OD=commute.od, smry=commute.summary)
}


# Call the functions and save the results for a list of states and types
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

states <- c("OR", "WA")
types <- c("RES", "WRK")

for(st in states){
      for(ty in types){
            commute <- get.commute(state=st, type=ty)
            instate.summary <- summarize.instate.commutes(commute)
            commute.file <- paste(st, ty, "_commutes.txt", sep="_")
            write.table(commute, commute.file, row.names=F, col.names=T,
            sep="\t")
            od.file <- paste(st, ty, "instate_commute_od.txt", sep="_")
            write.table(instate.summary$OD, od.file, row.names=T, col.names=T,
            sep="\t")
            summary.file <- paste(st, ty, "instate_commute_summary.txt",
            sep="_")
            write.table(instate.summary$smry, summary.file, row.names=T,
            col.names=T, sep="\t")
            rm(commute, instate.summary, commute.file, od.file, summary.file)
      }
}
```