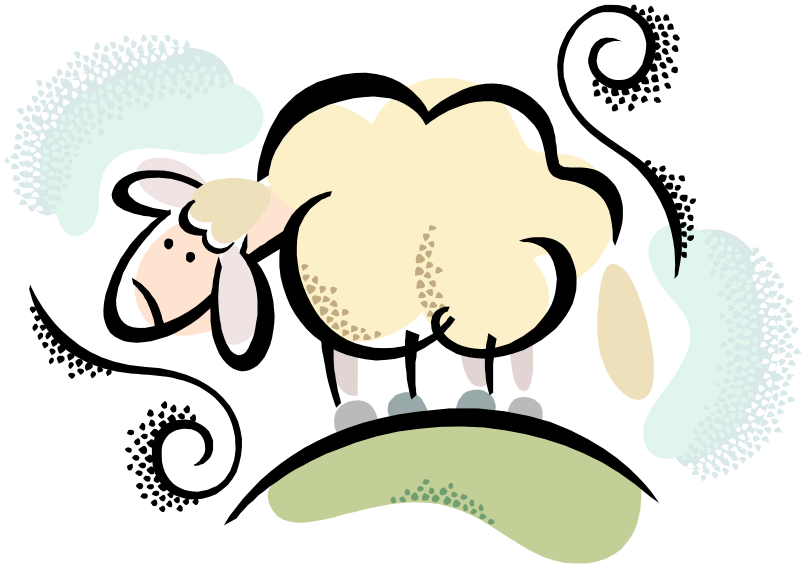


Programming II

Legal Stuff

This presentation as well as other presentations and documents found on the sheepsqueezers.com website may contain quoted material from outside sources such as books, articles and websites. It is our intention to diligently reference all outside sources. Occasionally, though, a reference may be missed. No copyright infringement whatsoever is intended, and all outside source materials are copyright of their respective author(s).

R Lecture Series

Non-Programming Introduction	*Programming I*	*Programming II*
Graphics I		*Advanced Topics*

Charting Our Course

- Goal of this Presentation
- What is R?
- Saying *Hello, World!* in R
- Installing and Running the Jaguar GUI
- Installing and Running the Eclipse IDE
- Installing Additional Packages
- R Data Types and Ranges
- R Data Structures
- Help! Using R's Built-in Help
- Understanding Packages
- Function Round-Up #1 – Mathematical Functions
- NA, NaN, +Inf, -Inf
- Reading in Data from Text Files
- Reading from and Writing to Excel 2007
- Factors
- More R Data Structures – The List
- R Options
- Function Round-Up #2 – Data Structure Initialization Functions
- Handling Dates in R
- Reading in Data from Text Files – Revisited
- Creating Your Own Functions in R

Charting Our Course

- Flow of Control
- Function Round-Up #3 – String Manipulation Functions
- Saving and Retrieving your R Workspace
- R and Database Interaction
- Function Round-Up #4 – Sorting and Ranking
- The `do.call()` Function
- Function Round-Up #5 – The Apply Series
- Matrix Operations
- Function Round-Up #6 – Random Numbers
- Using Formulas
- Attach and Detach and Why You Shouldn't Use Them
- Function Round-Up #7 – Recoding Variables
- Set Operations
- Appendix A: References
- Appendix B: R-Related Websites

Goal of this Presentation

The goal of this presentation is to teach you the more than just the basics of the R programming language. If you want to use R, but aren't interested in learning the language itself, please see the lecture *R Lecture Series – Non-Programming Introduction*. That particular lecture shows you how to interact with R by using only graphical user interfaces such as R Commander, Rattle, RExcel and RGGobi. If you are just starting to learn R, you might want to read the *Programming I* lecture first before you tackle this behemoth.

Learning a computer programming language can be boring. No, really. I know you don't believe that, but it is true! There are very few people who care about data types, data structures, etc., but this is what you need to learn in order to use R successfully.

Also, learning a new computer programming language requires practice, practice, practice. You cannot realistically expect to sit through a lecture – no matter how comprehensive – and then go back to your desk and write error-free code.

There are several ways to install R. Please see the lecture *R Lecture Series – Non-Programming Introduction* for installation instructions.

There are several ways to execute an R program and we show you several methods in this lecture. Pick the method you're most comfortable with to start. Later on, as you gain more proficiency in the language, you may want to change the method you use to execute your programs.

What is R?

R is a freely available programming language for statistical computing as well as the creation of publication-ready graphics. R is an implementation of the S programming language which itself was created by John Chambers, Rick Becker and Allan Wilks of Bell Laboratories. S is sold as a commercial product under the name *S-Plus*.

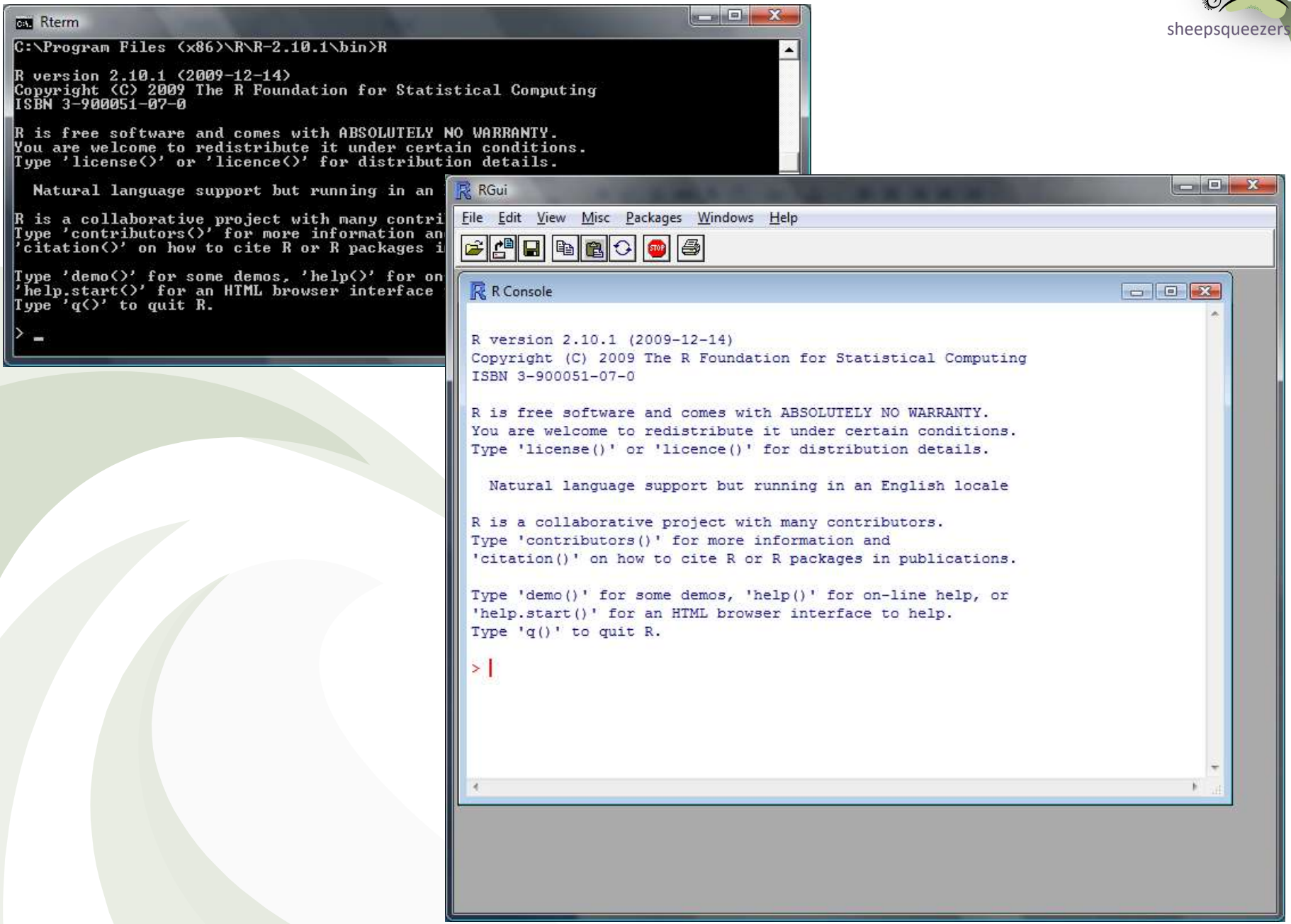
R is part of the GNU project which means that its source code is freely available under the GNU General Public License. The source code as well as pre-compiled versions of R are available at the R Project website (www.r-project.org) for a variety of platforms such as Windows and Linux.

R can be extended through the use of user-submitted packages (think SAS modules such as SAS/STAT and SAS/OR) and, at present, there are over two thousand freely available packages for download at the Comprehensive R Archive Network (CRAN) website (cran.r-project.org). A complete list of packages is located at cran.r-project.org/web/packages/index.html and topics cover basic and advanced statistics, genetic algorithms, ODBC/OLEDB database access, time series, financial computations, etc.

R comes with excellent documentation along with executable examples.

R comes with two interfaces which allow you to run R programs: Rterm, a command line interface, and RGui, a GUI interface. You can install several other interfaces to run your R programs such as Jaguar and the Eclipse IDE.

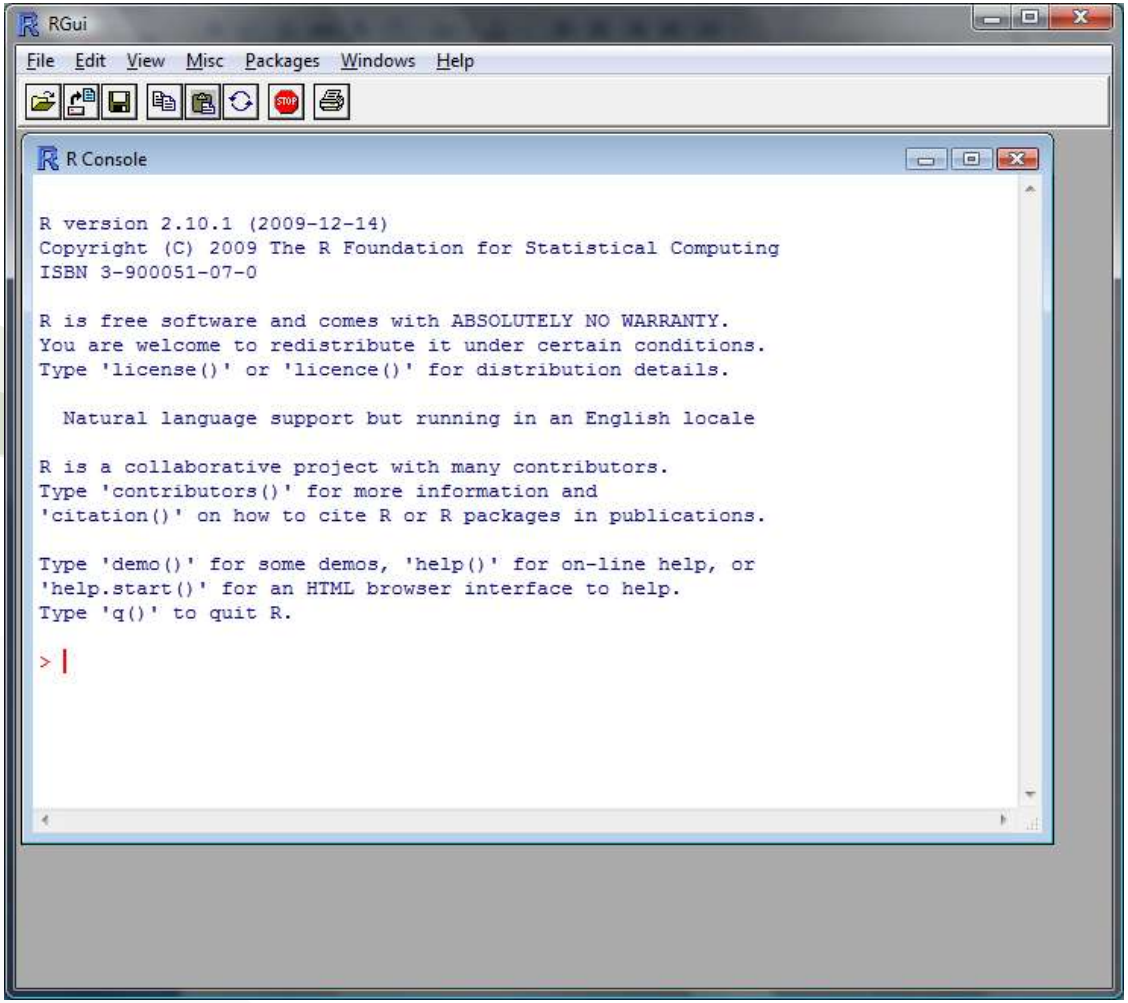
What is R?



Saying *Hello, World!* in R

Saying *Hello, World!* in R

Let's start off the lecture with a simple *Hello, World!* R program. Assuming we are running on Windows, start R by clicking on Start → All Programs → R → R 2.10.1. This will bring up the RGui as shown in the following screen:



Note that if you have run R Commander as specified in the previous lectures, you will see a Single Document Interface instead of the Multiple Document Interface shown above.

Saying *Hello, World!* in R

Take note of the red greater-than symbol (">"). This is called a *prompt* and indicates that R is ready to receive programming statements. Next, type in the following and hit the Enter key (you don't need to type in the prompt, just the statement):

```
> print("Hello, World!")
```

When you run this statement, you should see the following result:

```
[1] "Hello, World!"
>
```

Congratulations! You've just written your first R program! Okay, it's not very exciting, so let's pimp it out. Enter the following at the prompt:

```
> sText <- "Hello, World!"
> print(sText)
[1] "Hello, World!"
>
```

We just created our first variable called `sText` that contains the text *Hello, World!*. Note that we used the assignment operator `<-` as opposed to an equal sign (`=`) used in those other languages.

Saying *Hello, World!* in R

Next, let's create a variable and set it to a number. We will then concatenate that variable to our text variable sText to create a new variable called sAll:

```
> iChowder <- 86
> sText <- "Hello, World!"
> sAll <- paste(sText,iChowder,sep=" ")
> print(sAll)
[1] "Hello, World! 86"
>
```

Take note that assignment of numeric variables use the same `<-` assignment operator as for character variables. The `paste()` function takes two or more arguments (`sText` and `iChowder`, here) of variables to concatenate and ends with what type of character you'd like to use as a separator, a space in this example. Let's try a different separator:

```
> iChowder <- 86
> sText <- "Hello, World!"
> sAll <- paste(sText,iChowder,sep="+")
> print(sAll)
[1] "Hello, World!+86"
>
```

Saying *Hello, World!* in R

Now that we have this spiffy R program, let's save it in a file. Open up your favorite text editor, such as Notepad, TextPad, UltraEdit, etc., and place this program in the editor. Save the file as `MyFirstRProgram.R` on your hard-drive.

```
#*-------------------------------------------------------------------------*
#* Program:      MyFirstRProgram.R                                         *
#* R Version:    R 2.10.1                                                  *
#* Author(s):    sheepsqueezers.com                                        *
#* Date:         March 1, 2011                                             *
#*                                                                         *
#* Application:  None.                                                     *
#*                                                                         *
#* Abstract:     This program is my first R program...Woo-hoo!!            *
#*                                                                         *
#* Invocation:   None.                                                     *
#*                                                                         *
#* Assumptions:  None.                                                     *
#*                                                                         *
#* Parameters:   None.                                                     *
#*                                                                         *
#* Input(s):     None.                                                     *
#*                                                                         *
#* Output(s):    Printed output                                            *
#*                                                                         *
#* Example:      None.                                                     *
#*                                                                         *
#* Notes:        None.                                                     *
#*                                                                         *
#* Modification History:                                                   *
#* Date          Prog    Mod *   Reason                                    *
#* ---------     ----    -----   ------------------------------------------ *
#*-------------------------------------------------------------------------*
options(width=132)
iChowder <- 86   # Initialize iChowder
sText <- "Hello, World!"  # Initialize sText
sAll <- paste(sText,iChowder,sep="+")  # Create sAll
print(sAll)   # print out the results
```

Take note that comments begin with a number (pound) sign. The remaining part of the line is ignored.

Unlike SAS's /* … */, R has no way of commenting out an entire block of code.

Comments can be placed on statement lines as well. The remaining part of the line is ignored.

Saying *Hello, World!* in R

Now, in order to run your R program, you can copy-and-paste the code from the text editor into the RGui using CTRL-c and CTRL-v.

A better way is to pop in the code in the RGui using the `source()` function which is similar to SAS's `%include` statement. The first parameter (and the only one needed) is the location and filename of the R program you want to run:

```
> source("C:/TEMP/MyFirstRProgram.R")
```

Note that for both Windows and Linux, you can use forward slashes instead of backslashes normally used on Windows. If you want to use backslashes, you will need to double-up the slashes for `source()` to work properly:

```
> source("C:\\TEMP\\MyFirstRProgram.R")
```

In either case, the program executes and you will see the following output:

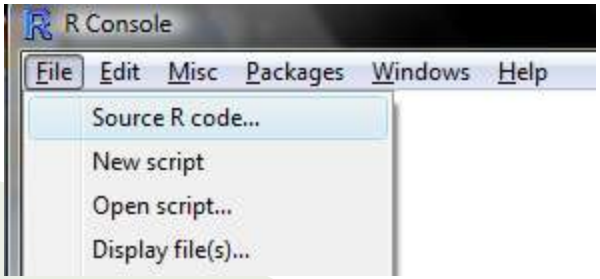
```
[1] "Hello, World!+86"
>
```

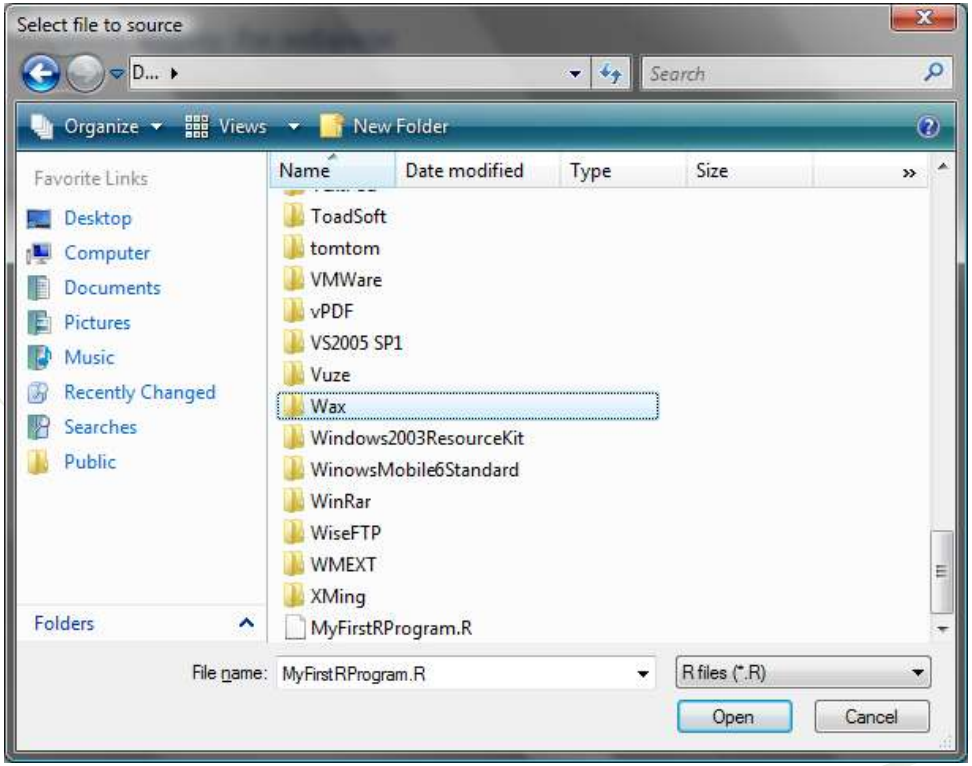
Note that the R code within MyFirstRProgram.R is not shown. In order to display the code, add additional parameters, as shown below:

Saying *Hello, World!* in R

```
> source("c:\\temp\\MyFirstRProgram.R",echo=TRUE,max.deparse.length=Inf)
```

Another way to run your R program is to use File…Source R code… from the RGui.





Note that this method is equivalent to `source("filename")`, but not as *programmy*.

Saying *Hello, World!* in R

And yet another way of running your R program is in batch mode at the command line *outside of* Rterm and RGui. This method works on both Windows and Linux.

At the Windows or Linux command prompt, type in the following and hit the enter key (can enter double-quotes around the path/filename):

```
$ R CMD BATCH C:/TEMP/MyFirstRProgram.R
```

This will run your R program and place results in `C:/TEMP/MyFirstRProgram.Rout` which looks like this:

```
[1] "Hello, World!+86"
>
> proc.time()
   user  system elapsed
   0.63    0.04    0.67
```

Note that your Rout file will contain some error messages at the end. You can eliminate these error messages by including the following statement at the end of your R program:

```
q("no") #Quit out of R without saving workspace image.
```

Saying *Hello, World!* in R

But, be careful! Adding `q("no")` to an R program you are copying and pasting into RGui will cause it to exit, closing your R session!

A *workspace image* is a unique beastie. Unlike SAS, when your SAS program ends, all work datasets are destroyed and all variables along with it. In R, if you have created several variables, you can save them all as a workspace image so that you can get them back when you restart R. We talk more about this later on in the lecture.

Finally, be aware that, unlike SAS, *R is case-sensitive*!

Installing and Running the Jaguar GUI

Installing and Running the Jaguar GUI

So far we've been running our R programs using the built-in RGui or from the operating system command prompt using the `R CMD BATCH` command. In order to make R programming a bit easier, we will now install and use the Jaguar GUI.

Jaguar (JGR) stands for Java GUI for R and is a cross-platform stand-alone R terminal that provides a friendly R-console complemented by a spreadsheet-like data editor and a script editor featuring syntax highlighting, autocompletion, and function argument hints. These last three features will be familiar to those programmers who have worked with VB or similar languages.

To install Jaguar on your computer, navigate your browser to http://rforge.net/JGR/files/ and download `jgr.exe`. Double-click on this program (you must run as Administrator on Vista) to install Jaguar. You will see the following icon on your desktop:

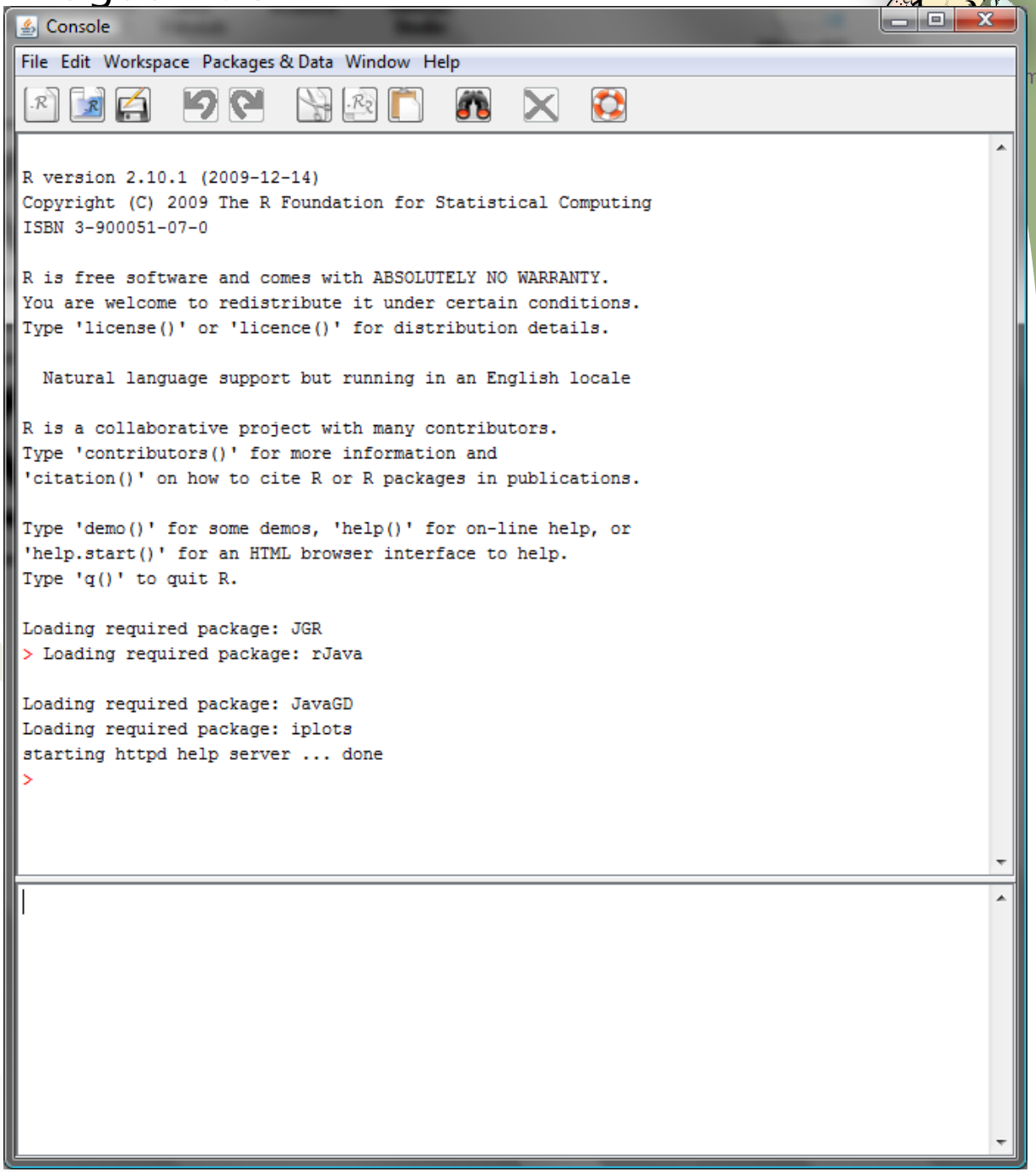
Double-click this icon to start Jaguar. Note that you do not need to be the administrator at this point. You will see the following screen when Jaguar starts:

Installing and Running the Jaguar GUI

The Jaguar GUI interface:

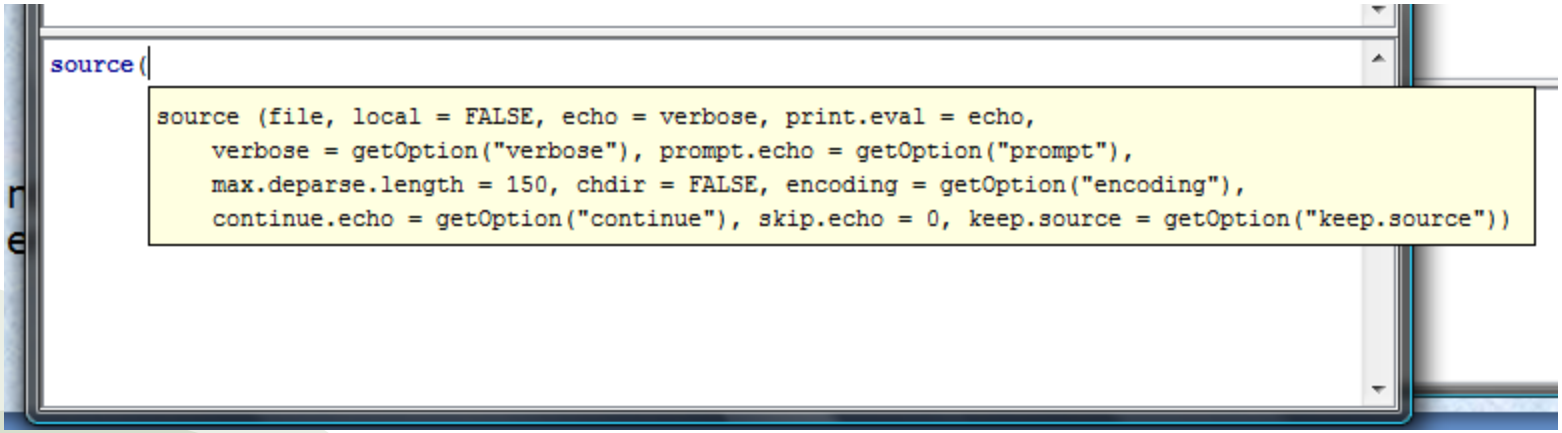
The results of running your R code appear here.

Enter your R statements here.



Installing and Running the Jaguar GUI

Notice that a hint comes up after you type the left parenthesis:



There's a lot more to this GUI interface than what has been explained here.

Installing and Running the Eclipse IDE

Installing and Running the Eclipse IDE

An Integrated Development Environment (IDE) is a software application that provides comprehensive facilities to computer programmers for software development and usually provides a source code editor and other tools to aid in development and debugging.

Eclipse is a *general purpose IDE* with an extensible plug-in system. One of the plug-ins used to allow Eclipse to work with R is called *StatET* which can be found at www.walware.de/goto/statet. You can find a tutorial for *StatET* at www.splusbook.com/RIntro/R_Eclipse_StatET.pdf.

The installation process for Eclipse and StatET can be cumbersome since there are several moving parts, so here goes:

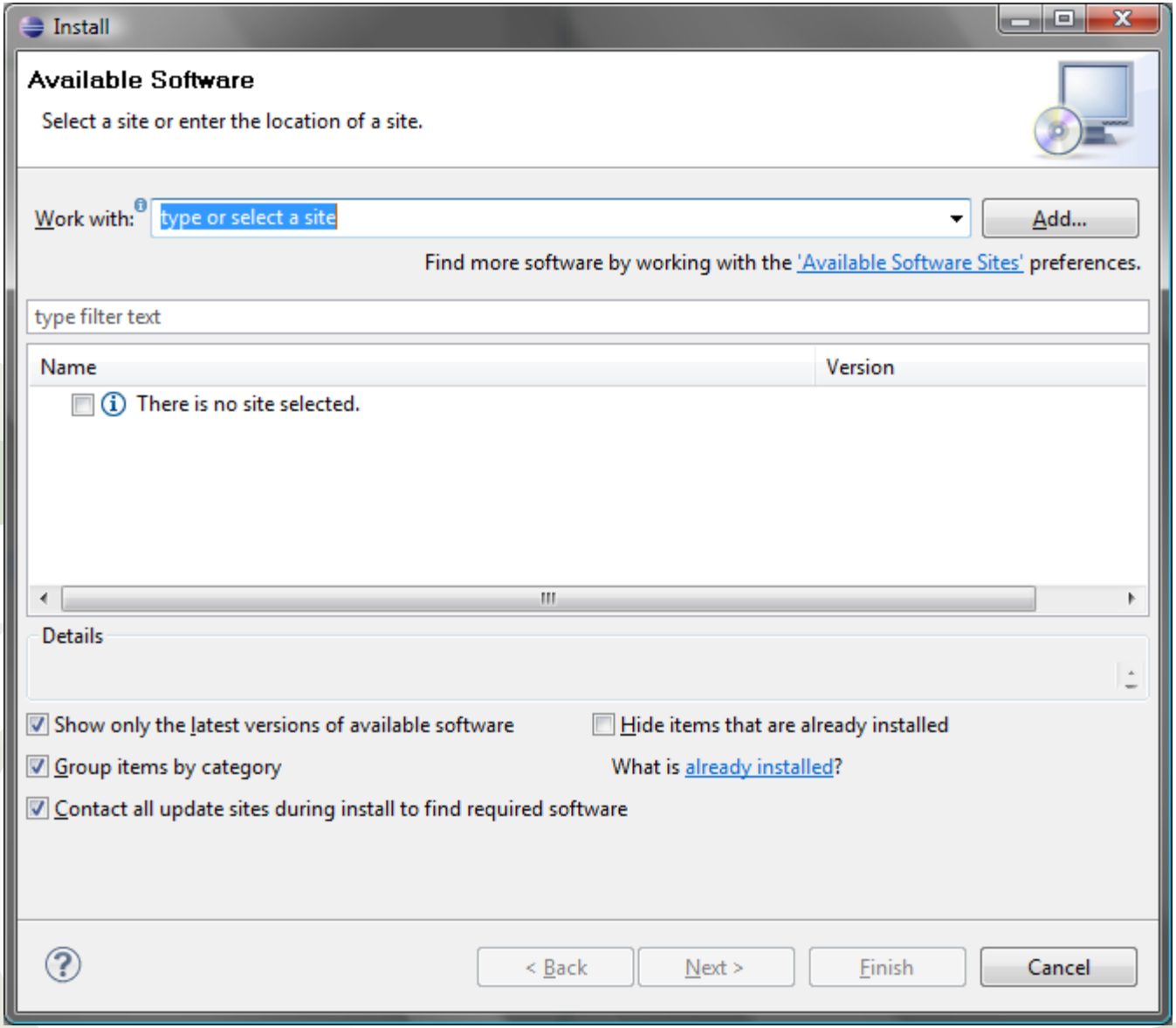
1. Download the Java Runtime Environment (JRE) `jre-6u21-windows-x64.exe` (or the appropriate executable for your operating system) from Sun's website at java.com/en/download/manual.jsp.
2. Download `eclipse-SDK-3.5.2-win32-x86_64.zip` (or the version appropriate for your operating system) from the Eclipse download page at download.eclipse.org/eclipse/downloads/drops/R-3.5.2-201002111343/winPlatform.php.
3. Since *StatET* is an Eclipse plug-in, there is nothing to download at this time. Once Eclipse is up-and-running, we will plug-in StatET from within Eclipse.

Installing and Running the Eclipse IDE

Next, let's install the software:

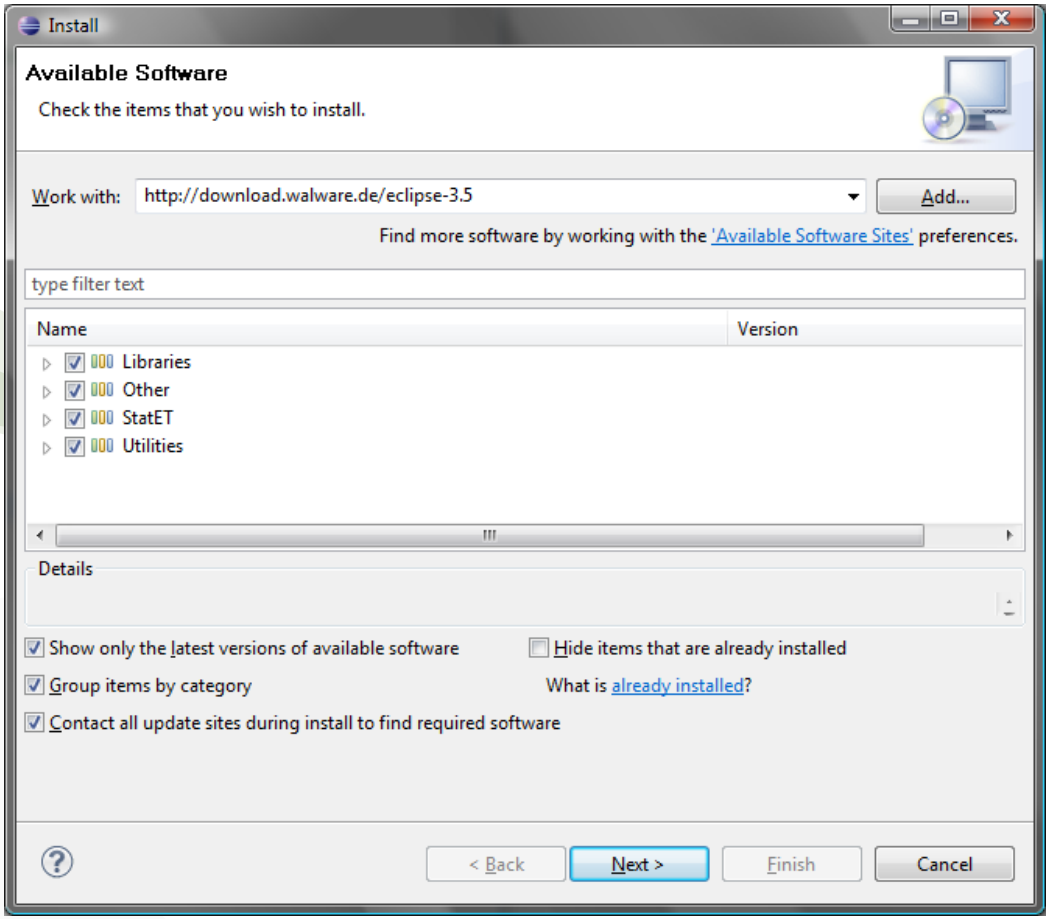
1. Double-click `jre-6u21-windows-x64.exe` (or similar) to start the JRE installation wizard. Follow the instructions until the installation is complete.
2. Unzip `eclipse-SDK-3.5.2-win32-x86_64.zip` (or similar) to a permanent folder on your computer. All you have to do is unpack this zip file. There is no installation wizard to run.
3. Create a shortcut to the Eclipse executable `eclipse.exe` and place it on your desktop.
4. Start Eclipse by double-clicking on the shortcut.
5. Once Eclipse has started, we can install the StatET plug-in. Click on the menu Help…Install New Software… The following dialog box will appear (see next slide):

Installing and Running the Eclipse IDE



Installing and Running the Eclipse IDE

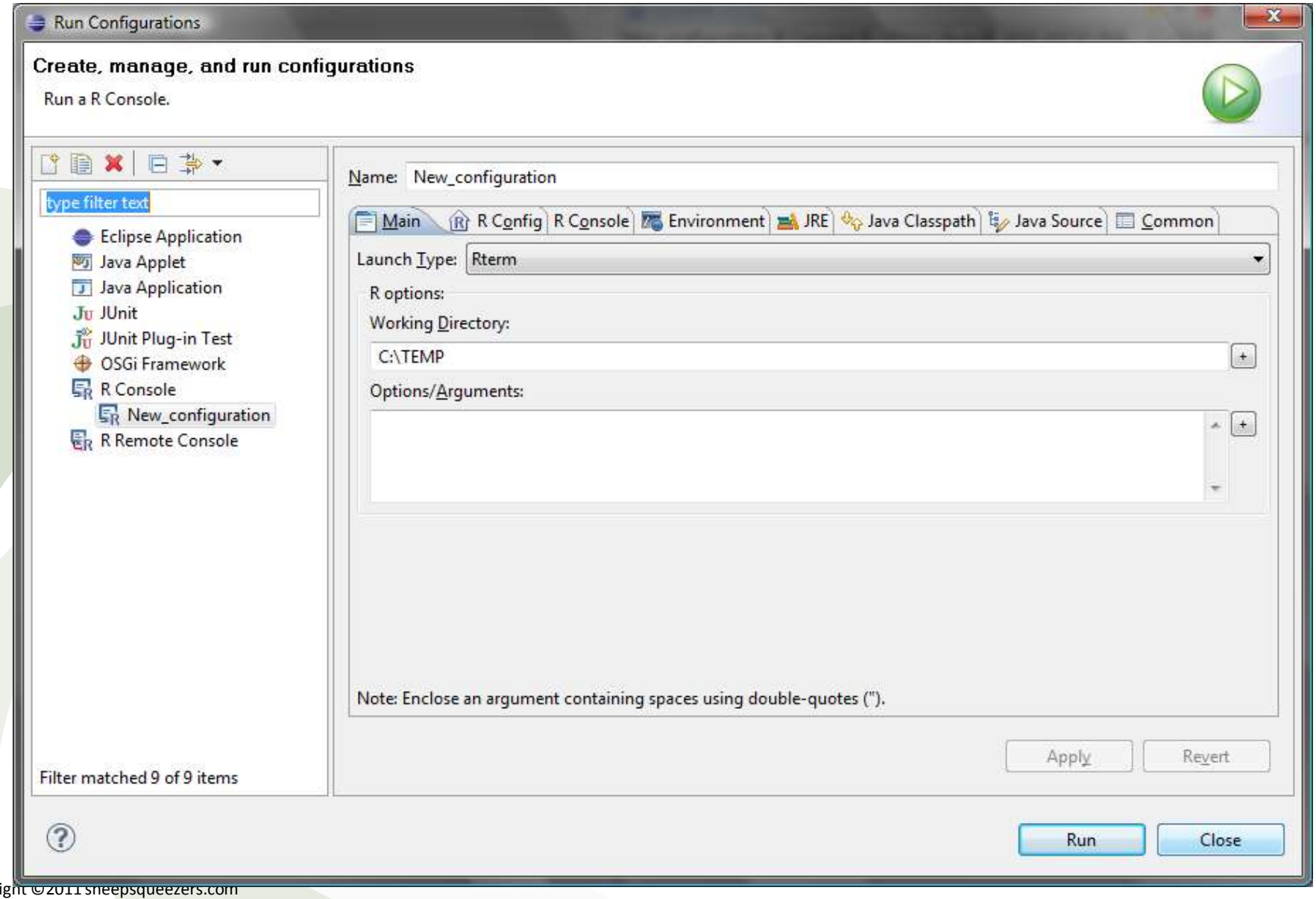
6. Place the following text in the input box to the right of the text `Work with: http://download.walware.de/eclipse-3.5` and hit the Enter key. The dialog box will now show four items in the main text box:



7. Check all four checkboxes and click Next. The software will download and you will be shown a license page. Click Finish and the software will install.
8. Click Finish to complete the plug-in installation process.

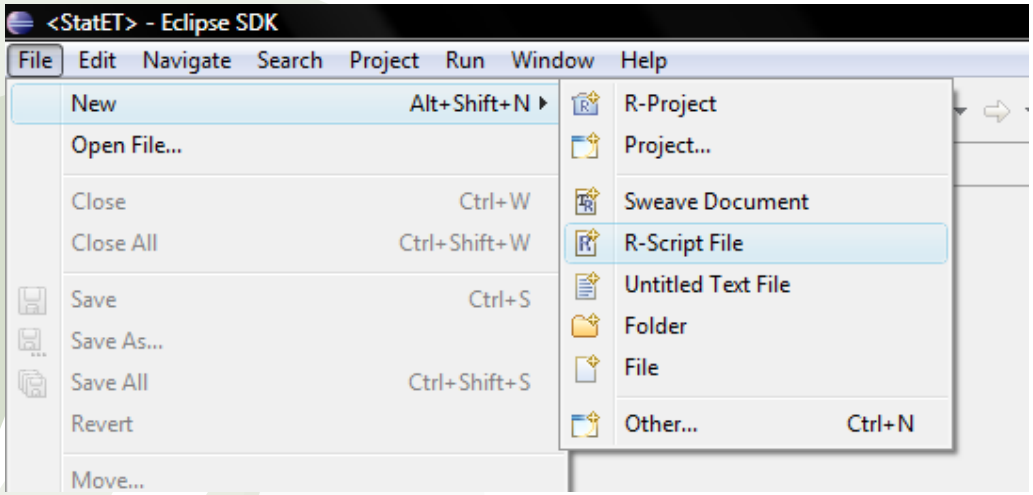
Installing and Running the Eclipse IDE

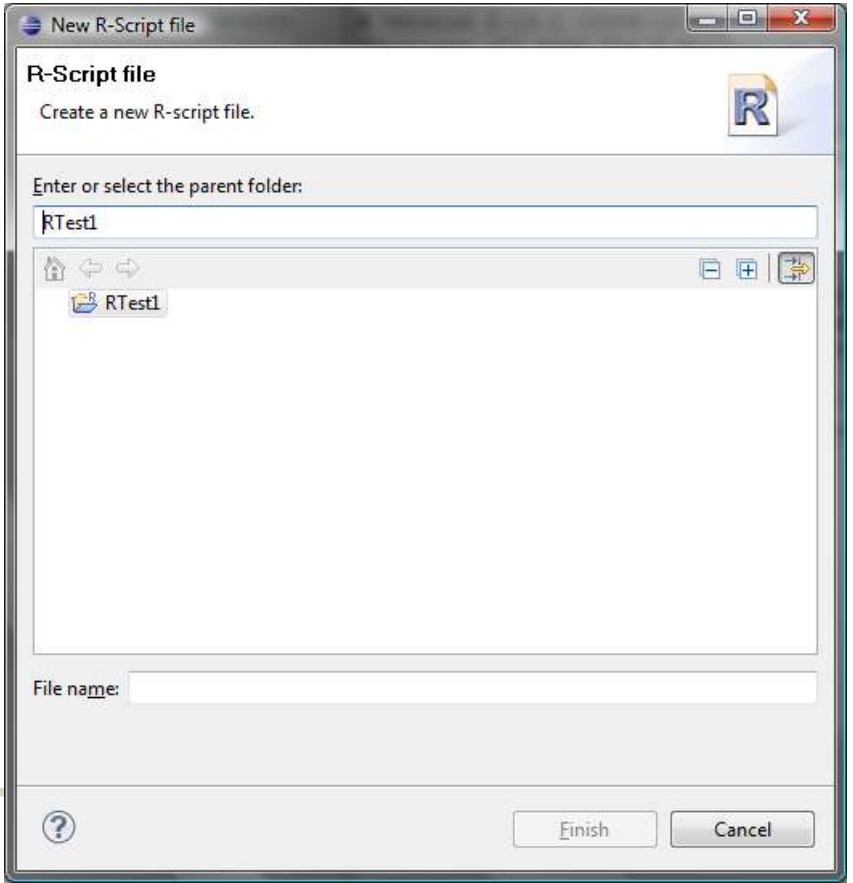
9. Next, we have to set up Eclipse to run R programs. Within Eclipse, click on Run…Run Configurations… You should see something similar to this:



Installing and Running the Eclipse IDE

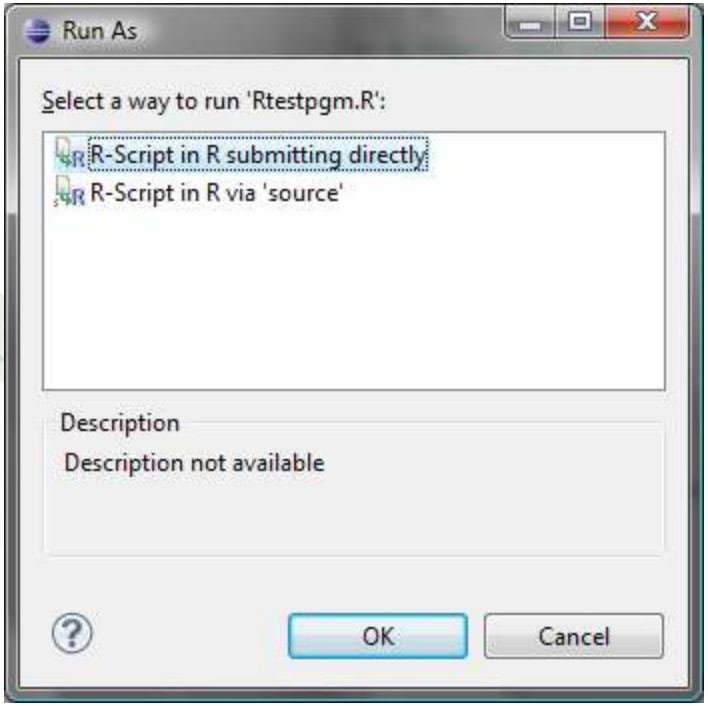
10. You should see New_configuration under the node R Console (on the left side). Ensure that launch type is Rterm and that you have chosen an appropriate working directory.
11. Click the Run button. This will start the R Console.
12. Next, let's create a new R program: Click File → New → R-Script File and the following dialog box will appear asking you to name your program. You do not need to end with *.R*. Click Finish. A new blank R program page will appear with a small comment section at the top.





Installing and Running the Eclipse IDE

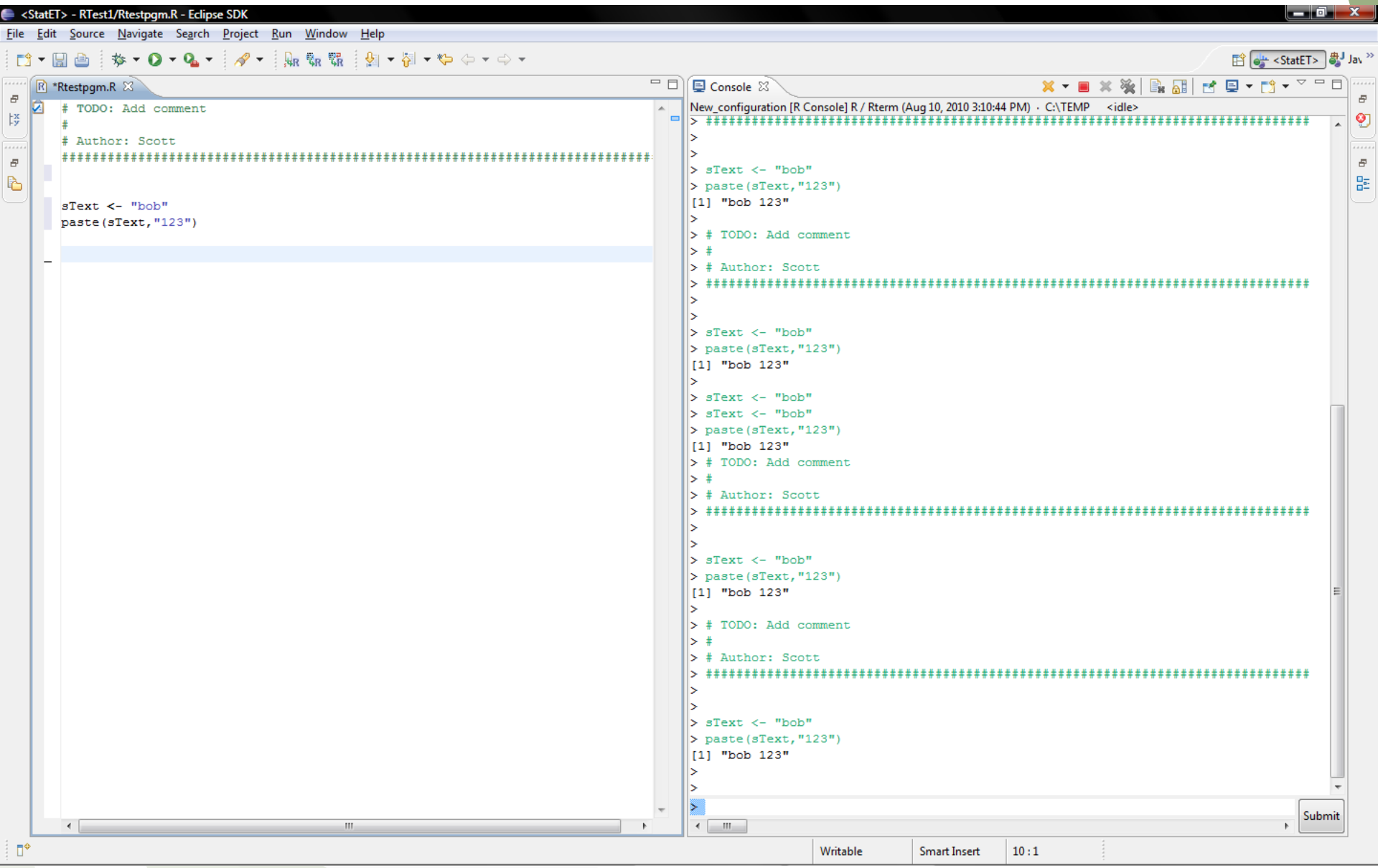
13. Enter the R commands from our simple *Hello, World!* example above and click the Run button at the top of the Eclipse IDE. The following dialog box will appear. Click on *R-Script in R submitting directly* then click OK. Your program will run in the console. Don't forget that you can arrange your windows however you prefer.



14. Since it's a pain to have the dialog box above appear all the time, we can set up a keyboard shortcut to run your code bypassing this box. Click on Windows → Preferences and the Preferences dialog box will appear. In the General section, click on the Keys link. Click in the Binding input box and click CTRL z. Search for the command *Run R-Script in R Submitting Directly* and Click OK. You should now be able to run your R program by clicking CTRL-z.

Installing and Running the Eclipse IDE

Here is how I arrange my windows (code on the left, output on the right): sheepsqueezers.com



Installing Additional Packages

Installing Additional Packages

In SAS, SAS/STAT, SAS/OR, SAS/IML, etc. are known as *modules*: additional functionality in the form of procedures (PROCs) that can be installed (for $$$) and made available to SAS programmers.

Similarly, R has additional functionality you can download and install (for free…woo-hoo!) and these are known as *packages*. There are well over two thousand packages available on the Comprehensive R Archive Network (CRAN) website: cran.r-project.org. Here is a list of some of them:

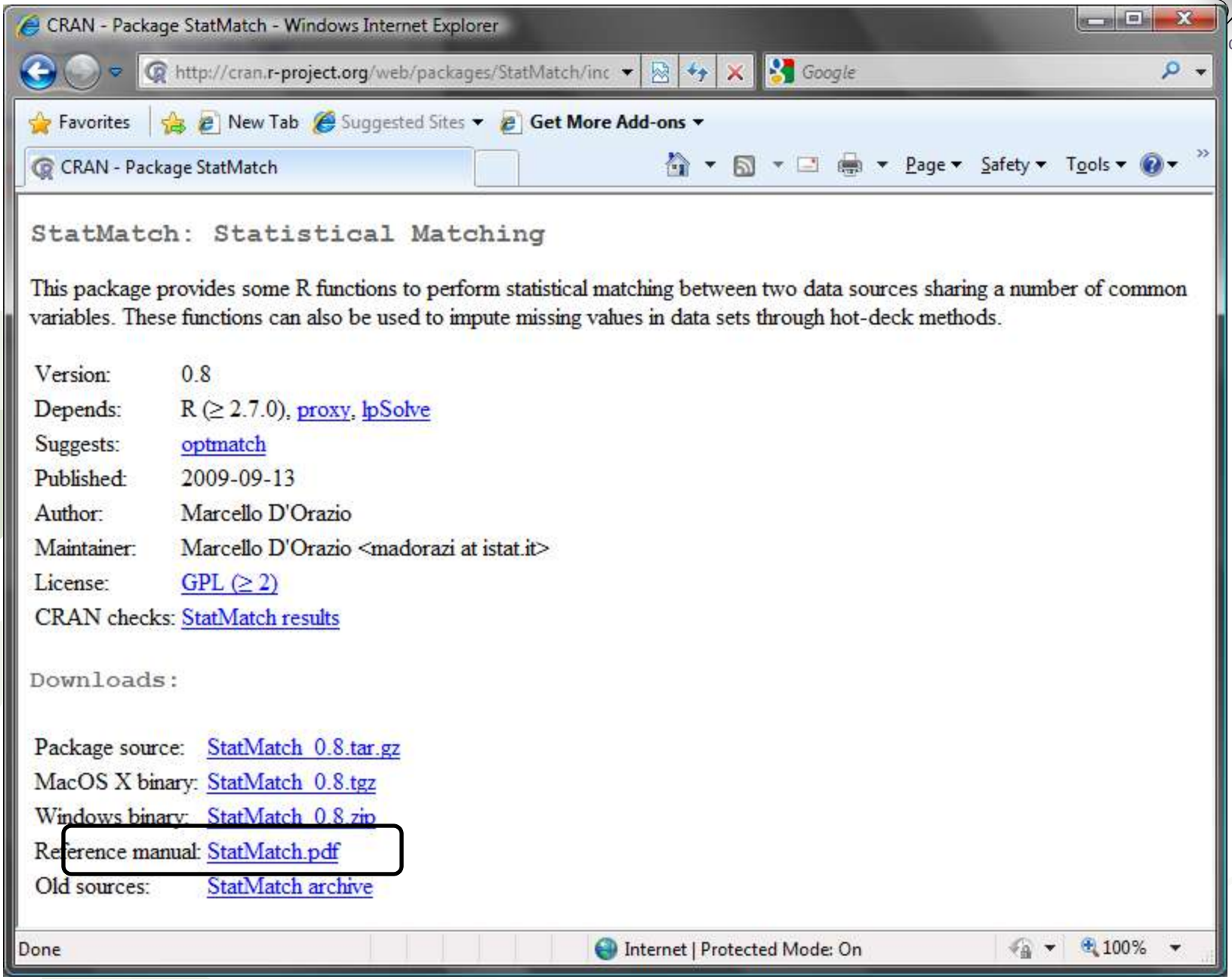
SampleSizeMeans	Sample size calculations for normal means
SampleSizeProportions	Calculating sample size requirements when estimating the difference between two binomial proportions
SciViews	SciViews GUI API - Main package
ScottKnott	The ScottKnott Clustering Algoritm
SemiPar	Semiparametic Regression
SenSrivastava	Datasets from Sen & Srivastava
StatMatch	Statistical Matching

If you want to install the StatMatch package, say, type in the following at the command line:

```
> install.packages("StatMatch",dependencies=TRUE)
```

To find out more about the StatMatch package, click on the StatMatch link (not shown above, but shown on the website) and you will see the following webpage:

Installing Additional Packages



Installing Additional Packages

If you would like a list of *your own* installed packages, type the following at the command line:

```
> .packages(all.available=TRUE)
  [1] "abind"                 "ada"                   "AER"
  [4] "akima"                 "amap"                  "anchors"
  [7] "ape"                   "aplpack"               "arules"
 [10] "biclust"               "biglm"                 "bitops"
 [13] "cairoDevice"           "car"                   "CarbonEL"
 [16] "caTools"               "cba"                   "chron"
 [19] "clue"                  "coda"                  "coin"
 [22] "colorspace"            "cubature"              "DAAG"
 [25] "DBI"                   "degreenet"             "deldir"
 [28] "descr"                 "digest"                "doBy"
 [31] "dynlm"                 "e1071"                 "effects"
 [34] "ellipse"               "ergm"                  "fBasics"
 [37] "fEcofin"               "fields"                "flexclust"
 [40] "flexmix"               "foreach"               "Formula"
 [43] "fpc"                   "gam"                   "gclus"
 [46] "gdata"                 "gee"                   "geoR"
 [49] "ggplot2"               "GPArotation"           "gpclib"
 [52] "gplots"                "graph"                 "gridBase"
 [55] "gtools"                "gWidgets"              "gWidgetsRGtk2"
 [58] "gWidgetstcltk"         "hexbin"                "HH"
 [61] "Hmisc"                 "HSAUR"                 "igraph"
 [64] "ineq"                  "iplots"                "ipred"
 [67] "isa2"                  "iterators"             "its"  …
```

R Data Types and Ranges

R Data Types and Ranges

As you know, SAS only has two data types: numeric and character. Dates are represented as numbers, unlike Oracle and SQL Server which have a separate data type for dates (DATE in Oracle, DATETIME/SMALLDATETIME in SQL Server).

SAS, Perl, JavaScript, VBScript, VB, etc. are known as a *weakly typed* language since you do not have to define what data type a variable is up-front. Languages such as C, C++, C#, Java, PL/SQL, T-SQL, etc. are *strongly typed* languages because you are forced to define each variable along with its data type before using the variable.

Similar to SAS, R is a weakly typed language. You do not have to define what each variable's data type will be…trust me, R will know. In many cases, R will convert from one data type to another data type automatically without you knowing it. There are several conversion functions allowing you more control over data type conversions, but we'll talk about that later on.

R recognizes the following data types:

```
NULL < raw < logical < integer < real < complex < character < list < expression
```

The reason for the less-than symbols is to indicate the highest type R will convert your data to if necessary. Recall that the numeric variable iChowder was converted to a character string in the `paste()` function. Character is further to the right of the list above than integer or real.

R Data Types and Ranges

Note that R further categorizes integers and reals as *numeric*. Note that the single and double data types are the same as the numeric data type.

Recall in our simple example above we created two variables:

```
> iChowder <- 86
> sText <- "Hello, World!"
```

In order to determine what data type R is using for a variable, you can use the `mode()` function:

```
> mode(iChowder)
[1] "numeric"
> mode(sText)
[1] "character"
>
```

The range of numeric values such as is `+/-2.2E-308` to `+/-1.7E+308` and is similar to SAS, Oracle, SQL Server, etc. SAS is limited to a character string size of `32767`, but R is effectively unlimited. The maximum integer is 2,147,483,647.

R Data Types and Ranges

R also has another data type called `logical`, which holds the values `TRUE` (`T`) or `FALSE` (`F`) based on the result of a logical expression. Take note of the capitalization! `True` and `true` are incorrect!

For example, let's test whether iChowder, which is set to 86, is equal to 777. Let's see what R has to say about this:

```
> bRC <- iChowder==777
> print(bRC)
[1] FALSE
```

Take note that to test for equality you use two equal signs (==) unlike in SAS where you use just one equal sign (=).

We'll look more into the logical data type later on in the lecture.

There are several other data types, such as raw, complex, etc., but we will not talk about these in this lecture series.

R Data Structures

R Data Structures

A *data structure* is a way of organizing and storing data in a computer so that it can be used more efficiently. Excluding the SAS Hash and Hash Iterator Objects, Base SAS has no data structures, just numeric and character variables. These are referred to as *scalars* or *scalar variables* in SAS. That is, for example, a SAS variable can hold either the number 7 or the characters "U.S.A".

R has several data structures and we will introduce them one at a time.

A *vector* is a data structure consisting of a collection of elements identified by a single index. All of the elements must be the same data type. Recall from your Linear/Matrix Algebra class in college that a vector looks like this: $\mathbf{v} = [v_1, v_2, v_3, \ldots, v_n]$. The numbers 1, 2, …, n are the indices of the vector and v_1, v_2, v_3, … , v_n are the elements of the vector, **v**.

The easiest way to create a vector is R is by using the concatenation function, `c()`. This function takes a comma-delimited list of values of a specific data type (mode) and creates a vector. For example, below I create a vector v with a series of values:

```
> v <- c(11.7,13.6,12.5,-79.6,101.2)
> print(v)
[1]  11.7  13.6  12.5 -79.6 101.2
>
```

R Data Structures

Some of you may want to know whether this vector is *vertical* or *horizontal*, as shown below:

$$v = \begin{bmatrix} 11.7 \\ 13.6 \\ 12.5 \\ -79.6 \\ 101.2 \end{bmatrix} \qquad v = [11.7 \;\; 13.6 \;\; 12.5 \;\; -79.6 \;\; 101.2]$$

At this point, don't worry about the orientation of the vector, just think of it as a series of values stuffed into a variable name, v.

Recall that in our simple example, we set sText to "Hello, World!":

```
> sText <- "Hello, World!"
> print(sText)
[1] "Hello, World!"
>
```

What would happen if we used the `c()` function to create sText?

```
> sText <- c("Hello, World!")
> print(sText)
[1] "Hello, World!"
>
```

R Data Structures

Isn't that interesting! Whether you create sText by setting the variable to a value using double-quotes or by using the `c()` function, you get the same exact results! Why is this?

As colon-tighteningly scary as it may sound: *Unlike SAS, R has no scalar variables!* At a minimum, everything is a vector (or other data structure, as we shall see). So, why would R avoid scalars in its programming language? The reason is that, for the most part, statisticians operate on a lot of data and not just one singular value. For example, let's add one to iChowder:

```
> print(iChowder+1)
[1] 87
>
```

Now, let's add one to the vector v:

```
> print(v)
[1]  11.7  13.6  12.5 -79.6 101.2
> print(v+1)
[1]  12.7  14.6  13.5 -78.6 102.2
>
```

R Data Structures

What is R doing? In the first example, R successfully added one to iChowder to come up with 667, but it seems to have *added one to each element* of the vector v!! That's exactly what R is doing. This is called a *vectorizing computation* in R and is very useful!

Note that, at this point, the way you think about programming in R may be diverging from the way you think about programming in SAS. In SAS, you use a series of DATA Steps and procedures to perform calculations within the rows of a dataset as well as across the rows of a dataset, but in R you use vectorizing computations to arrive at the same thing.

For example, let's compute the average z-score given the vector v (which should be zero, by the way):

```
> MeanV <- mean(v)
> print(MeanV)
[1] 11.88
> StdDevV <- sd(v)
> print(StdDevV)
[1] 63.9336140070308
> ZScoresV <- (v - MeanV)/StdDevV
> print(ZScoresV)
[1] -0.00281542038246433  0.02690290587688115  0.00969755909515483
[4] -1.43085920326574945  1.39707415867617790
> print(mean(ZScoresV))
[1] 0
> print(sd(ZScoresV))
[1] 1
>
```

R Data Structures

Let's take a closer look at what R is doing when computing the variable ZScoresV.

Let's start with the subtraction of the mean:

```
> print(v)
[1]  11.7  13.6  12.5 -79.6 101.2
> print(MeanV)
[1] 11.88
> print(StdDevV)
[1] 63.93361
> print(v-MeanV)
[1]  -0.18   1.72   0.62 -91.48  89.32
>
```

Clearly, R is subtracting the value of MeanV from v one element at a time. Let's continue with the division of the standard deviation:

```
> print( (v-MeanV)/StdDevV )
[1] -0.002815420  0.026902906  0.009697559 -1.430859203  1.397074159
>
```

As you can see, we have divided by 63.93, the standard deviation of v, into each element of the vector v after the MeanV was subtracted.

But, how can R divided just **one** number, say the MeanV value 11.88, into each of the **five** elements of the vector v? Whenever one vector has a smaller number of elements than another vector, the *values in the smaller vector are repeated to match the number of elements in the larger vector*.

R Data Structures

Let's take a closer look at this. Let's create another vector, w, that contains six elements:

```
> w <- c(1,2,3,4,5,6)
> print(w)
[1] 1 2 3 4 5 6
>
```

Let's create two vectors, one with only one value and another with two values:

```
> t1 <- c(0.5)
> print(t1)
[1] 0.5
> t2 <- c(1,2)
> print(t2)
[1] 1 2
>
```

Next, let's compute w-t1 and w-t2:

```
> print(w-t1)
[1] 0.5 1.5 2.5 3.5 4.5 5.5
> print(w-t2)
[1] 0 0 2 2 4 4
>
```

R Data Structures

In the first example, w-t1, R computes the results as:

```
1-0.5   2-0.5   3-0.5   4-0.5   5-0.5   6-0.5
```

where 0.5 is repeated 6 times once for each element of the larger vector, w.

In the second example, w-t2, R computes the results as:

```
1-1   2-2   3-1   4-2   5-1   6-2
```

where t2 is repeated three times: c(1,2,1,2,1,2) to come up with a total of 6 elements to match the number of elements in w.

What would happen if the smaller vector were not a multiple of the first? Let's try that out:

```
> print(w)
[1] 1 2 3 4 5 6
> t3 <- c(1,2,3,4)
> print(w-t3)
[1] 0 0 0 0 4 4
Warning message:
In w - t3 : longer object length is not a multiple of shorter object length
>
```

R Data Structures

As you can see, R issues a warning message, but still performs the computations, in this case like this:

```
1-1   2-2   3-3   4-4   5-1   6-2
```

where t3 is repeated as much as is necessary!

Now, in order to access the individual elements of a vector, you use the bracket, or *indexing*, notation specifying which element you are interested in. For example, let's pull the first element of vector v:

```
> print(v)
[1]  11.7  13.6  12.5 -79.6 101.2
> print(v[1])
[1] 11.7
>
```

Note that in R, the indexing into a vector starts with the number one and not zero as in many other languages. This indexing lark is not just for fun, but can be enormously useful when you want to subset a vector based on a certain criteria (think SQL WHERE-Clause!). For example, let's take a look at vector v:

```
> print(v)
[1]  11.7  13.6  12.5 -79.6 101.2
>
```

R Data Structures

So, here's the goal: Let's create a new vector from v, but without the negative values:

```
> z <- v[v>=0]
> print(z)
[1]  11.7  13.6  12.5 101.2
> print(v)
[1]  11.7  13.6  12.5 -79.6 101.2
>
```

So, what's going on? First off, that zero you see in the indexing brackets is not a scalar, but a vector, `c(0)`. Second, since v contains five elements, that zero is repeated five times, `c(0,0,0,0,0)`. Third, R evaluates the expression v>=0 element by element and returns a TRUE or FALSE for each element returned:

```
> print(v>=0)
[1]  TRUE  TRUE  TRUE FALSE  TRUE
>
```

Take note that there is one FALSE returned and that is in the same position as the negative value of v, -79.6. Fourth, since the expression `v>=0` is within the indexing brackets, each logical value *where TRUE occurs* will see those elements returned.

R Data Structures

That is:

Hence, that's why v[v>=0] returns 11.7, 13.6, 12.5 and 101.2, but does not return -79.6.

Note that we can add additional subsetting criteria:

```
> print(v[v>=0 & v<100])
[1] 11.7 13.6 12.5
>
```

In this case the criteria is that each element of the vector v be non-negative as well as less than 100.

The symbol for `AND` is `&`, `OR` is `|`, and `NOT` is `!`.

Note: If you would like to perform bitwise AND, OR, XOR, see the `bitops` package.

R Data Structures

Similar to a SET Statement in SAS, you can append two vectors together by using the `append()` function:

```
> vMyVec1 <- c(1,2,3)
> vMyVec2 <- c(4,5,6)
> vMyVec3 <- append(vMyVec1,vMyVec2)
> print(vMyVec3)
[1] 1 2 3 4 5 6
>
```

R Data Structures

The next data structure we will learn about is called an *array*.

Here is a ray:

Here, too, is a ray:

R Data Structures

An *array* is a generalization of a vector; that is, whereas a vector only has one dimension, an array can have multiple dimensions. A special case of an array is the two-dimensional case, which we call a *matrix*.

We won't talk about the general case of arrays, but we will talk about the matrix case. Remember: a *vector* is a one-dimensional array, and a *matrix* is a two-dimensional array. Also, just like a vector, all of the elements of a matrix must have the same data type (that is, mode): either numeric, character, logical, etc. The vectors do not have to have the same number of elements, but it would make your life easier if they did. But, if they do not, as explained above, the smaller vector repeats to fill in the gaps.

Recall that to create a vector, we used the `c()` function. Believe it or not, there are several ways to create a matrix. One way is to slap together several vectors either horizontally or vertically. For example, let's create two vectors and then create a matrix from them:

```
> v1 <- c(1,2,3,4,5,6)
> v2 <- c(11,12,13,14,15,16)
> print(v1)
[1] 1 2 3 4 5 6
> print(v2)
[1] 11 12 13 14 15 16
```

R Data Structures

Next, let's combine the vectors v1 and v2 together column-wise using the column bind function, `cbind()` :

```
> m1 <- cbind(v1,v2)
> print(m1)
     v1 v2
[1,]  1 11
[2,]  2 12
[3,]  3 13
[4,]  4 14
[5,]  5 15
[6,]  6 16
>
```

As you see above, `cbind()` slapped v1 and v2 togther as columns of the matrix we are now calling m1. There is an equivalent row bind function, `rbind()`, to combine vectors as rows:

```
> m2 <- rbind(v1,v2)
> print(m2)
   [,1] [,2] [,3] [,4] [,5] [,6]
v1    1    2    3    4    5    6
v2   11   12   13   14   15   16
>
```

R Data Structures

Another method to create a matrix is to use the `matrix()` function:

```
> m3 <- matrix( c(1,2,3,4,5,6),nrow=3,ncol=2,byrow=TRUE )
> print(m3)
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
>
> m4 <- matrix( c(1,2,3,4,5,6),nrow=3,ncol=2,byrow=FALSE )
> print(m4)
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
>
```

Take note of the parameters of the `matrix()` function. The first parameter is a vector of values. The second parameter is the number of rows in the matrix. The third parameter is the number of columns in the matrix. Finally, the fourth parameter indicates how the data in the vector is to be layed out in the matrix. When `byrow=TRUE`, the values proceed across then down. If `byrow=FALSE`, the values proceed down the first column, then the second column, etc.

R Data Structures

Just like vectors, you can use indexing brackets to gather subsets of data from the matrix. In this case, you have two comma-delimited criteria instead of just one: `mat[rows,columns]`. If you want to return the matrix m3 with just the second column, do this:

```
> print(m3)
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
> print( m3[,c(2)] )
[1] 2 4 6
>
```

If you want to return the matrix m3 with the second and third rows and all of the columns, do this:

```
> print( m3[c(2,3),] )
     [,1] [,2]
[1,]    3    4
[2,]    5    6
>
```

Take note that in the first example, there is nothing to the left of the comma; in the second example, there is nothing to the right of the comma. This indicates **all** data will be returned: rows, in the first case; columns, in the second.

R Data Structures

If you want the second column and the first and second rows:

```
> print( m3[c(1,2),c(2)] )
[1] 2 4
>
```

If you have two matrices of the same size, you can add them together *pointwise* by using the usual plus-sign:

```
> print(m3)
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
> print(m4)
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> print(m3+m4)
     [,1] [,2]
[1,]    2    6
[2,]    5    9
[3,]    8   12
>
```

R Data Structures

You can use both `cbind()` and `rbind()` with matrices:

```
> print( cbind(m3,m4) )
     [,1] [,2] [,3] [,4]
[1,]    1    2    1    4
[2,]    3    4    2    5
[3,]    5    6    3    6
>
```

You can also use subsetting criteria such as:

```
> print(m3)
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
> print( m3[ m3[,2]==4 , ] )
[1] 3 4
>
```

We will talk about matrix manipulation such as, matrix multiplication, later on in the lecture.

The next data structure is called the *data frame* and is analogous to the data set in SAS. A *data frame* is a two dimensional array that allows for differing data types and also allows for **named** columns.

R Data Structures

One way to create a data frame is to use the `data.frame()` function. We talk more about this later on in the presentation, but here is a simple example:

```
> dfM1 <- data.frame(V1=c(1,3,5),V2=c(2,4,6))
```

Similar to PROC CONTENTS in SAS, you can see the structure of the data frame by using the `str()` function with the name of the data frame as the argument:

```
> str(dfM1)
'data.frame':   3 obs. of  2 variables:
 $ V1: num  1 3 5
 $ V2: num  2 4 6
>
```

If you become confused as to what your variable is (a matrix, data frame, etc.), then use the `class()` function to find out:

```
> class(m1)
[1] "matrix"
> class(dfM1)
[1] "data.frame"
>
```

To refer to a specific variable within a data frame, type the name of the data frame followed by a dollar-sign followed by the name of the column: *data_frame*$*variable_name*. For example, `dfM1$V1`. Note that you can use indexing notation on data frames just like with vectors and matrices.

R Data Structures

You can very easily create a new column using the $ syntax mentioned above. For example, let's create column V3 as the sum of V1 and V2:

```
> dfM1$V3 <- dfM1$V1 + dfM1$V2
> dfM1
  V1 V2 V3
1  1  2  3
2  3  4  7
3  5  6 11
```

Now, you can convert a data frame into a matrix by using `data.matrix()`:

```
> print(dfFatKids)
  FirstName Height Weight FattyIndex
1    ALBERT     45    150     3.3333
2  ROSEMARY     35    123     3.5143
3     TOMMY     78    167     2.1410
4     BUDDY     12    189    15.7500
5   FARQUAR     76    198     2.6053
6     SIMON     87    256     2.9425
7    LAUREN     54    876    16.2222
> mFatKids <- data.matrix(dfFatKids)
> print(mFatKids)
     FirstName Height Weight FattyIndex
[1,]         1     45    150     3.3333
[2,]         5     35    123     3.5143
[3,]         7     78    167     2.1410
[4,]         2     12    189    15.7500
[5,]         3     76    198     2.6053
[6,]         6     87    256     2.9425
[7,]         4     54    876    16.2222
>
```

R Data Structures

Now, given two data frames, you can merge them in several different ways. As we said above, we can create a new variable in the data frame by referring to the data frame, followed by a $-sign, followed by a variable name. But, instead of a variable name, you can use a vector or a column from *another* data frame. But, this assumes that the sort order for the data frame and the vector/column are the same! This may not be the case for two separate data frames, say, read in from two different text files or pulled from two different database tables.

In order to merge two data frames together, you can use the `merge()` function. This function is similar to the SAS Merge Statement and even allows for keeping just the rows that are in both data frames, or all rows from the data frame on the left, or all rows from the data frame on the right.

For example, here is the Fat Kids data frame:

```
> print(dfFatKids)
  FirstName Height Weight FattyIndex
1    ALBERT     45    150     3.3333
2  ROSEMARY     35    123     3.5143
3     TOMMY     78    167     2.1410
4     BUDDY     12    189    15.7500
5   FARQUAR     76    198     2.6053
6     SIMON     87    256     2.9425
7    LAUREN     54    876    16.2222
>
```

R Data Structures

and here is a data frame containing the names of the kids along with their telephone number:

```
> print(dfFatKidsPhone)
  FirstName  PhoneNumber
1    ALBERT 215-123-4567
2  ROSEMARY 215-123-5678
3     TOMMY 215-123-6789
4     BUDDY 215-123-7890
5   FARQUAR 215-123-8901
6     SIMON 215-123-9012
7    LAUREN 215-123-0123
8     BOBBY 610-123-9876
```

As you can see above, Bobby's phone number appears in this data frame but does not appear in the fat kids data frame. To merge the two data frames together, keeping only those rows that match *all of the common variables*, you do this:

```
> dfFatKidsALL <- merge(dfFatKids,dfFatKidsPhone)
> print(dfFatKidsALL)
  FirstName Height Weight FattyIndex  PhoneNumber
1    ALBERT     45    150     3.3333 215-123-4567
2     BUDDY     12    189    15.7500 215-123-7890
3   FARQUAR     76    198     2.6053 215-123-8901
4    LAUREN     54    876    16.2222 215-123-0123
5  ROSEMARY     35    123     3.5143 215-123-5678
6     SIMON     87    256     2.9425 215-123-9012
7     TOMMY     78    167     2.1410 215-123-6789
>
```

R Data Structures

Next, you can take control over what variables you want to use in the merge:

```
> dfFatKidsALL <- merge(x=dfFatKids,y=dfFatKidsPhone,by.x="FirstName",by.y="FirstName")
> dfFatKidsALL
  FirstName Height Weight FattyIndex  PhoneNumber
1    ALBERT     45    150     3.3333 215-123-4567
2     BUDDY     12    189    15.7500 215-123-7890
3   FARQUAR     76    198     2.6053 215-123-8901
4    LAUREN     54    876    16.2222 215-123-0123
5  ROSEMARY     35    123     3.5143 215-123-5678
6     SIMON     87    256     2.9425 215-123-9012
7     TOMMY     78    167     2.1410 215-123-6789
>
```

Finally, you can tell the `merge()` function to keep all of the matched data plus the data in the *y* data frame:

```
> dfFatKidsALL <-
            merge(x=dfFatKids,y=dfFatKidsPhone,by.x="FirstName",by.y="FirstName",all.y=TRUE)
> dfFatKidsALL
  FirstName Height Weight FattyIndex  PhoneNumber
1    ALBERT     45    150     3.3333 215-123-4567
2     BUDDY     12    189    15.7500 215-123-7890
3   FARQUAR     76    198     2.6053 215-123-8901
4    LAUREN     54    876    16.2222 215-123-0123
5  ROSEMARY     35    123     3.5143 215-123-5678
6     SIMON     87    256     2.9425 215-123-9012
7     TOMMY     78    167     2.1410 215-123-6789
8     BOBBY     NA     NA         NA 610-123-9876
>
```

R Data Structures

Recall that for vectors and matrices, we used a very strange notation with the indexes to get a subset of the data based on certain criteria. With data frames, you can use the `subset()` function to achieve the same results and avoid the crazy indexes.

For example, let's subset the dfFatKidsALL data frame to only those Heights that are non-missing. We use the `is.na()` function below to determine those Heights that are missing (or NA, in R terminology):

```
> subset(dfFatKidsALL,!is.na(Height))
  FirstName Height Weight FattyIndex  PhoneNumber
1    ALBERT     45    150     3.3333 215-123-4567
2     BUDDY     12    189    15.7500 215-123-7890
3   FARQUAR     76    198     2.6053 215-123-8901
4    LAUREN     54    876    16.2222 215-123-0123
5  ROSEMARY     35    123     3.5143 215-123-5678
6     SIMON     87    256     2.9425 215-123-9012
7     TOMMY     78    167     2.1410 215-123-6789
>
```

Next, let's pull back only the males by name using the `%in%` operator:

```
> subset(dfFatKids,FirstName %in% c("ALBERT","BUDDY","FARQUAR","SIMON","TOMMY","BOBBY"))
  FirstName Height Weight FattyIndex
1    ALBERT     45    150     3.3333
3     TOMMY     78    167     2.1410
4     BUDDY     12    189    15.7500
5   FARQUAR     76    198     2.6053
6     SIMON     87    256     2.9425
>
```

R Data Structures

Finally, you can select a subset of the variables when using the `subset()` function by using the select= parameter:

```
> subset(dfFatKids,Height>=30 & Weight<=150,select=c(FirstName,FattyIndex))
  FirstName FattyIndex
1    ALBERT     3.3333
2  ROSEMARY     3.5143
>
```

Help! Using R's Built-in Help!

Help! Using R's Built-in Help

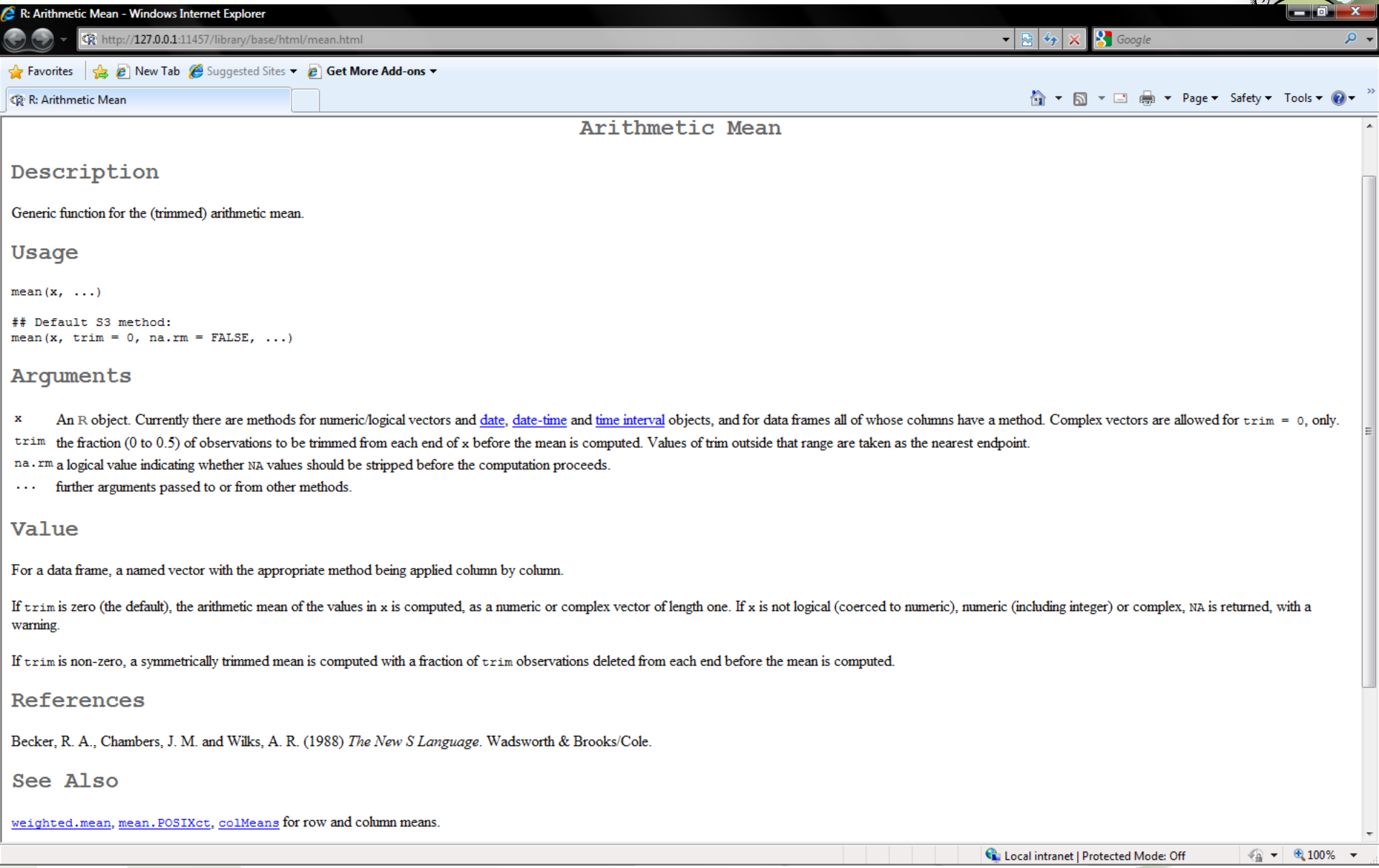
R has a great built-in help system that you should know about.

If you know the name of the function you want to look up, at the R command prompt, type in *?function-name*. For example, let's look up more about the `mean()` function:

```
> ?mean
```

On the next slide, you will see the R Help page for the `mean()` function:

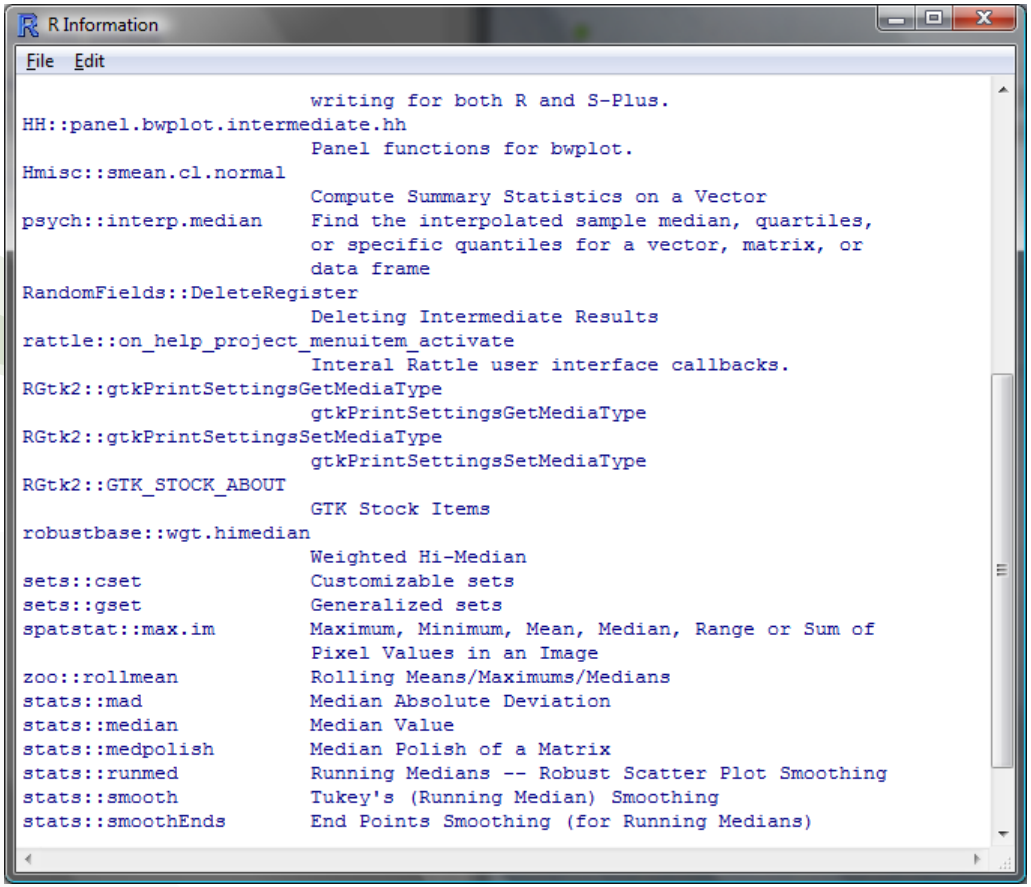
Help! Using R's Built-in Help



Help! Using R's Built-in Help

If you don't know the name of the function, type in two question marks followed by an appropriate keyword: ??keyword. This will bring up a list of possible candidates. For example,

```
> ??median
```



As you see above, there is a list of candidates for median that are double-colon separated. The text before the colon is the package name; the text after is the function name. So, `stats::median` is the median function in the `stats` package.

Help! Using R's Built-in Help

To see more information about `stats::median`, type in:

```
> ?stats::median
```

R: Median Value - Windows Internet Explorer

http://127.0.0.1:11457/library/stats/html/median.html

Favorites | New Tab | Suggested Sites | Get More Add-ons

R: Median Value

Page | Safety | Tools

Median Value

Description

Compute the sample median.

Usage

```
median(x, na.rm = FALSE)
```

Arguments

`x`	an object for which a method has been defined, or a numeric vector containing the values whose median is to be computed.
`na.rm`	a logical value indicating whether `NA` values should be stripped before the computation proceeds.

Details

This is a generic function for which methods can be written. However, the default method makes use of `sort` and `mean` from package **base** both of which are generic, and so the default method will work for most classes (e.g. "`Date`") for which a median is a reasonable concept.

Value

The default method returns a length-one object of the same type as `x`, except when `x` is integer of even length, when the result will be double.

If there are no values or if `na.rm = FALSE` and there are `NA` values the result is `NA` of the same type as `x` (or more generally the result of `x[FALSE][NA]`).

Done | Local intranet | Protected Mode: Off | 100%

Help! Using R's Built-in Help

We will soon be talking about the many functions available in R. To get a complete list of functions available in the base package, enter the following at the R Command line and hit Enter:

```
> library(help=base)
```

Or, in general, you can get a list of available functions from within any package by using the follow code:

```
> library(help=package_name)
```

Note that sometimes you will have to place double-quotes around the text you want to search for when using the question mark so as to not confuse R. `?if` will confuse R whereas `?"if"` will search through the help system for if-related information.

Understanding Packages

Understanding Packages

As mentioned earlier, there are well over two thousand packages freely available from the Comprehensive R Archive Network (CRAN) website at cran.r-project.org.

Now, when you install R, several packages are automatically available for you to use. Other packages will have to be installed as discussed in the section *Installing Additional Packages* towards the beginning of the lecture.

In order to see the automatically available packages, type in the `search()` function at the R command line:

```
> search()
 [1] ".GlobalEnv"        "package:stats"     "package:graphics"
 [4] "package:grDevices" "package:datasets"  "package:rcom"
 [7] "package:rscproxy"  "package:utils"     "package:methods"
[10] "Autoloads"         "package:base"
>
```

In order to see more about the package itself and the functions contained within, type in the following:

```
> help(package=package-name)
> library(help=package-name)
```

Note that the two packages rcom and rscproxy are part of the RExcel install and do not appear if RExcel has not been installed. So, the following packages are available by default:

Understanding Packages

- `stats`: This package contains functions related to statistics.
- `graphics`: This package contains functions related to creating graphics.
- `grDevices`: This package contains functions used to output graphics and other output to PDF, JPEG, etc. files.
- `datasets`: This package contains many data frames for you to play with.
- `utils`: This package contains utility functions such as `str()` which we discussed earlier.
- `methods`: This package contains functions related to object-oriented programming
- `base`: This package contains basic functions such as `max()` and `min()`.

Now, in order to load a package (assuming you've installed it already), you must use the `library()` function to load it in. For example, to load in the `car` package, type the following at the command line:

```
> library(car)
Loading required package: MASS
Loading required package: nnet
Loading required package: survival
Loading required package: splines
>
```

As you see above, the `car` package will load in additional packages as it sees fit. The additional packages contain functions that are used by the `car` package. Let's take a look at the search path now that we added the `car` package:

Understanding Packages


```
> search()
 [1] ".GlobalEnv"        "package:car"       "package:survival"
 [4] "package:splines"   "package:nnet"      "package:MASS"
 [7] "package:stats"     "package:graphics"  "package:grDevices"
[10] "package:datasets"  "package:rcom"      "package:rscproxy"
[13] "package:utils"     "package:methods"   "Autoloads"
[16] "package:base"
>
```

As you see above, the `car` package now appears in the first position, followed by the survival, splines, nnet and MASS packages which the `car` package needs.

The most important thing to remember about the search path is that when you use a function, R will search this path in the order shown above. If there are two conflicting function names – yes, it does happen, but not very often – R will use the function that it finds first in the search path.

Another nice function you should be aware of is the `ls()` function. This function lists all of the variables, functions, etc. that you have made within your R session:

```
> ls()
[1] "m1" "m2" "m3" "m4" "v" "dfM1"
>
```

Now, you may want to list just certain types of variables, such as the character variables, etc. For this, you use a variant of the `ls()` function called `ls.str(mode="`*`mode_name`*`")`.

Understanding Packages

Let's find all of the character, logical and numeric variables we have created:

```
> ls.str(mode="character")
     cVec :  chr [1:4] "A" "B" "C" "D"
     sDates :  chr [1:3] "2010-01-01" "2010-01-02" "2010-01-03"
     sDates2 :  chr [1:3] "01Jan2010" "02Jan2010" "03Jan2010"
>
> ls.str(mode="logical")
     bitty :  logi TRUE
>
> ls.str(mode="numeric")
     adate : Class 'Date'  num 5950
     bdate : Class 'Date'  num 6315
     cdate : Class 'Date'  num 14831
     dDates : Class 'Date'  num [1:3] 14610 14611 14612
     dEnd : Class 'Date'  num 14640
     diff :  num 365
     dStart : Class 'Date'  num 14624
     w :  num [1:5] 1 3 5 7 9
     x :  num [1:10] 1 1.44 1.89 2.33 2.78 ...
     z : Class 'Date'  num 14841
>
```

Finally, if you want to remove a variable, you can use the `rm()` function:

```
> rm(v)
```

Function Round-Up #1

Mathematical Functions

Function Round-Up #1 – Mathematical Functions

In this section we present some math functions. Since there are so many functions in R, we will present them piecemeal so as to not drive you insane. We will not spend a lot of time on the more familiar functions.

The standard arithmetic operators are available in R:

```
x + y: addition
x - y: subtraction
x * y: multiplication
x / y: (floating-point) division
x ^ y: x^y
x %% y: x mod y
x %/% y: (integer) division
abs(x): absolute value
colSums(): sums down the columns of an array(vector, matrix, data frame)
rowSums(): sums across the rows of an array (")
colMeans(): computes means down the columns of an array (")
rowMeans(): computes means across the rows of an array (")
```

Example:

```
> m3 <- matrix( c(1,2,3,4,5,6),nrow=3,ncol=2,byrow=TRUE )
> print(m3)
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
```

Function Round-Up #1 – Mathematical Functions

Example (continued):

```
> print(colSums(m3))
[1]  9 12
> print(rowSums(m3))
[1]  3  7 11
> print(colMeans(m3))
[1] 3 4
> print(rowMeans(m3))
[1] 1.5 3.5 5.5
>
```

```
cos(x): cosine of x
sin(x): sine of x
tan(x): tangent of x
acos(x): arc cosine of x
asin(x): arc sine of x
atan(x): arc tangent of x
atan2(y,x): equivalent to atan(y/x)
cosh(x): hyperbolic cosine of x
sinh(x): hyperbolic sine of x
tanh(x): hyperbolic tangent of x
acosh(x): arc hyperbolic cosine of x
asinh(x): arc hyperbolic sine of x
atanh(x): arc hyperbolic tangent of x
cumsum(x): cumulative sum of elements of an array
cumprod(x): cumulative product of elements of an array
cummax(x): maximum element from the first element to the current element of an array
cummin(x): minimum element from the first element to the current element of an array
```

Function Round-Up #1 – Mathematical Functions

Example:

```
> v <- c(1,2,3,4,5,6)
> cumsum(v)
[1]  1  3  6 10 15 21
> cummax(v)
[1] 1 2 3 4 5 6
> cummin(v)
[1] 1 1 1 1 1 1
> cumprod(v)
[1]   1   2   6  24 120 720
>
```

log(x, base = exp(1)): log of x with base=e (can change base to desired base)
logb(x, base = exp(1)): log of x with base=e (preferred, can change base to desired base)
log10(x): log of x with base=10
log2(x): log of x with base=2
log1p(x): equivalent to log(1+x)
exp(x): e^x
expm1(x): e^x-1
max(x): maximum value of array x
min(x): minimum value of array x
sign(x): returns +1 if x>0, -1 if x<0, 0 if x=0
seq(from=,to=,by=,length.out=NULL,along.with=NULL): Sequence from *from* to *to* by *by*.
seq_along(along.with): Sequence from 1 to the number of items in along.with.
seq_len(length.out): Sequence from 1 to the value of length.out

Function Round-Up #1 – Mathematical Functions

Example (seq_* series):

```
> seq_len(5)
[1] 1 2 3 4 5
> seq_len(10)
 [1]  1  2  3  4  5  6  7  8  9 10
>
> cVec <- c("A","B","C","D")
> seq_along(cVec)
[1] 1 2 3 4
>
> w <- seq(1,10,2)
> w
[1] 1 3 5 7 9
> print(w)
[1] 1 3 5 7 9
> x <- seq(1,5,length.out=10)
> x
 [1] 1.000000 1.444444 1.888889 2.333333 2.777778 3.222222 3.666667 4.111111
 [9] 4.555556 5.000000
>
```

Note that a cheap way of getting a integral sequence is by using the *colon-operator*. For example, let's create a vector of the numbers 1 to 10:

```
> vMyVec <- 1:10
> vMyVec
 [1]  1  2  3  4  5  6  7  8  9 10
>
```

Function Round-Up #1 – Mathematical Functions

Note that the colon-operator always adds a one from the starting value until it gets to the ending value. If the addition of one exceeds the ending value, the ending value is not placed in the vector:

```
> 1.1:10.7
 [1]  1.1  2.1  3.1  4.1  5.1  6.1  7.1  8.1  9.1 10.1
>
```

Note that you can use the colon-operator anywhere you'd place a vector of numbers. Also, can do 10:1 and you will get a vector of numbers from 10 to 1.

Additional mathematical functions are as follows:

```
beta(a,b) - Beta function
lbeta(a,b) - natural logarithm of the Beta function
gamma(x) - Gamma function
lgamma(x) - natural logarithm of the Gamma function
psigamma(x,deriv=0) - Polygamma function
digamma(x) - digamma function
trigamma(x) - trigamma function
choose(n,k) - C(n,k) combinations function
lchoose(n,k) - natural logarithm of C(n,k)
factorial(x) - x!
lfactorial(x) - natural logarithm of x!
```

These functions can be found in the documentation under `?Special`.

Function Round-Up #1 – Mathematical Functions

Additional useful mathematical functions include the following:

`ceiling(x)` - ceiling takes a single numeric argument x and returns a numeric vector containing the smallest integers not less than the corresponding elements of x.

`floor(x)` - floor takes a single numeric argument x and returns a numeric vector containing the largest integers not greater than the corresponding elements of x.

`trunc(x, ...)` - trunc takes a single numeric argument x and returns a numeric vector containing the integers formed by truncating the values in x toward 0.

`round(x,digits=0)` - round rounds the values in its first argument to the specified number of decimal places (default 0).

`signif(x,digits=6)` - signif rounds the values in its first argument to the specified number of significant digits.

A nice function to produce all combinations of vectors or factors (we'll talk about factors later) is the `expand.grid()` function. For example, given the following three vectors, `expand.grid()` produces all combinations of them and produces a data frame:

```
> A <- c(1,2)
> B <- c("red","blue")
> C <- c("M","F")
```

Function Round-Up #1 – Mathematical Functions


```
> expand.grid(IDNum=A,Color=B,Gender=C)
  IDNum Color Gender
1     1   red      M
2     2   red      M
3     1  blue      M
4     2  blue      M
5     1   red      F
6     2   red      F
7     1  blue      F
8     2  blue      F
>
```

Notice that the first variable moves the fastest, the second the next fastest and the third the slowest.

NA, NaN, +Inf, -Inf

NA, NaN, +Inf, -Inf

Just like SAS, R has the concept of a missing value. Although SAS has several missing values (., ._, .A, … , .Z), R only has one: NA. You can place NA anywhere you would place a period (.) or blank (" ") in SAS:

```
> w <- c(1,2,NA,4,5,6)
> w
[1]  1  2 NA  4  5  6
>
```

Take note that you do NOT place quotes around NA even if it's a character variable! Recall that we subsetted elements of a vector v above. You can return the missing values, or return all of the non-missing values as follows:

```
> print( w[is.na(w)] )  # is.na(w) returns TRUEs and FALSEs!!
[1] NA
> print( w[!is.na(w)] )
[1] 1 2 4 5 6
>
```

where the function `is.na()` returns TRUE or FALSE for each element of the vector supplied to it. The `!is.na()` returns the opposite.

NA, NaN, +Inf, -Inf

Note that some functions allow you to remove any missing values before the computation is performed. For example, the `mean()` function's first argument is the vector, while the second argument is `na.rm=TRUE|FALSE`. For example,

```
> print(w)
[1]  1  2 NA  4  5  6
> print(mean(w))
[1] NA
> print( mean(w,na.rm=TRUE) )
[1] 3.6
>
```

Note that for most functions, `na.rm=FALSE` and no missing values will be removed which is different from how SAS handles missing values.

If you divide a positive number by zero, you will get `Inf`. If you divide a negative number by zero, you will get `-Inf`. If you provide a function with an argument it cannot perform its action on, then you get back `NaN`. For example,

```
> z <- 1/0
> print(z)
[1] Inf
> z <- -1/0
> print(z)
[1] -Inf
> z <- log(-1)
Warning message:
In log(-1) : NaNs produced
> print(z)
[1] NaN
```

NA, NaN, +Inf, -Inf

Note that `TRUE` can be abbreviated as `T`, and `FALSE` can be abbreviated as `F`. They, too, must be capitalized! In R, t is the transpose function, `t()`, and is not equivalent to T.

Note that `NA`, `NaN`, `+Inf` and `-Inf` are referred to as numeric constants in R. See the documentation under `?NumericConstants`. Other constants available in R are as follows:

`LETTERS` – a vector of 26 elements from A to Z.

`letters` – a vector of 26 elements from a to z.

`month.abb` – a vector of 12 elements from Jan to Dec.

`month.name` – a vector of 12 elements from January to December.

`pi` – a vector of 1 element containing pi: 3.141593

Note that there are additional functions used to test whether a value is NaN, is finite or is infinite:

```
is.finite(x) - returns a vector of the same length as x the jth element of which is TRUE if
      x[j] is finite (i.e., it is not one of the values NA, NaN, Inf or -Inf).
is.infinite(x) - returns a vector of the same length as x the jth element of which is TRUE if
     x[j] is infinite (i.e., equal to one of Inf or -Inf).
is.nan(x) - tests if a numeric value is NaN.
```

Reading In Data From Text Files

Reading In Data From Text Files

Unlike the graphical user interfaces (GUIs) we talked about in the first lecture – such as Rattle, R Commander and RExcel – when programming R you have to use functions to read in data. There are several functions you can use to read in data from text files and these functions have several parameters used to control how data is read in:

```
read.csv(file, header = TRUE, sep = ",", quote="\"", dec=".", fill = TRUE, comment.char="", ...)
          - Read in comma-delimited data (sep=","), character fields are quoted (quote="\""),
                    period represent decimals (dec="."), the first row is a header (header=TRUE),
                    any data after the comment character is ignored (comment.char="")

read.csv2(file, header = TRUE, sep = ";", quote="\"", dec=",", fill = TRUE, comment.char="", ...)
          - Read in semi-colon-delimited data (sep=";"), character fields are quoted (quote="\""),
                    commas represent decimals (dec=","), the first row is a header (header=TRUE),
                    any data after the comment character is ignored (comment.char="")

read.delim(file, header = TRUE, sep = "\t", quote="\"", dec=".", fill = TRUE, comment.char="", ...)
          - Read in tab-delimited data (sep="\t"), character fields are quoted (quote="\""),
                    period represent decimals (dec="."), the first row is a header (header=TRUE),
                    any data after the comment character is ignored (comment.char="")

read.delim2(file, header = TRUE, sep = "\t", quote="\"", dec=",", fill = TRUE, comment.char="", ...)
          - Read in tab-delimited data (sep="\t"), character fields are quoted (quote="\""),
                    commas represent decimals (dec=","), the first row is a header (header=TRUE),
                    any data after the comment character is ignored (comment.char="")
```

In all four cases above, the parameter `fill=TRUE` indicates to add in blank columns to a row of data if it has less columns than other rows of data (in SAS, MISSOVER).

Reading In Data From Text Files

In general, you can use the `read.table()` function to read in textual data. The four functions shown above are based on `read.table()` and are provided for convenience. Here is the syntax for `read.table()`:

```
read.table(file,
      header = FALSE,
      sep = "",
      quote = "\"'",
      dec = ".",
      row.names,
      col.names,
      as.is = !stringsAsFactors,
      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 = "#",
      allowEscapes = FALSE,
      flush = FALSE,
      stringsAsFactors = default.stringsAsFactors(),
      fileEncoding = "",
      encoding = "unknown")
```

In all five cases, a data frame is produced. Note that if you are dealing with a fixed-format file, then use the function `read.fwf()` instead.

Reading In Data From Text Files

Next, let's read in the fat kids data. Recall that the FatKids.csv file is comma-delimited. Let's use `read.csv()` to read in the data:

```
> dfFatKids <- read.csv("C:\\Users\\Scott\\Desktop\\R Series\\FatKids.csv")
> print(dfFatKids)
  FirstName Height Weight FattyIndex
1    ALBERT     45    150     3.3333
2  ROSEMARY     35    123     3.5143
3     TOMMY     78    167     2.1410
4     BUDDY     12    189    15.7500
5   FARQUAR     76    198     2.6053
6     SIMON     87    256     2.9425
7    LAUREN     54    876    16.2222
> str(dfFatKids)
'data.frame':   7 obs. of  4 variables:
 $ FirstName : Factor w/ 7 levels "ALBERT","BUDDY",..: 1 5 7 2 3 6 4
 $ Height    : int  45 35 78 12 76 87 54
 $ Weight    : int  150 123 167 189 198 256 876
 $ FattyIndex: num  3.33 3.51 2.14 15.75 2.61 ...
>
```

Note that there was no need to provide additional parameters to the `read.csv()` function since all of the default arguments were fine. Also, take note that the names of the columns were derived from the header (due to header=TRUE).

If you take a look at the structure of dfFatKids, above, you will see that Height and Weight are integers, FattyIndex is numeric, but FirstName is "factor" instead of "character". We talk more about factors later in the presentation.

Reading In Data From Text Files

Rather than printing out all of the data from a data frame (or vector, or matrix, etc.), you can use the `head()` function to show the first 6 rows of data. Similarly, you can use the `tail()` function to show the last 6 lines of the data.

```
> head(dfFatKids)
  FirstName Height Weight FattyIndex
1    ALBERT     45    150     3.3333
2  ROSEMARY     35    123     3.5143
3     TOMMY     78    167     2.1410
4     BUDDY     12    189    15.7500
5   FARQUAR     76    198     2.6053
6     SIMON     87    256     2.9425
> tail(dfFatKids)
  FirstName Height Weight FattyIndex
2  ROSEMARY     35    123     3.5143
3     TOMMY     78    167     2.1410
4     BUDDY     12    189    15.7500
5   FARQUAR     76    198     2.6053
6     SIMON     87    256     2.9425
7    LAUREN     54    876    16.2222
```

You can also add a number as the second parameter indicating the number of rows you'd like to see (besides the default of 6):

```
> head(dfFatKids,2)
  FirstName Height Weight FattyIndex
1    ALBERT     45    150     3.3333
2  ROSEMARY     35    123     3.5143
>
```

Reading In Data From Text Files

Finally, you can write out a data frame to a file by using one of the `write()` functions. For example, let's write out dfFatKidsALL to a comma-delimited file:

```
> write.csv(dfFatKidsALL,"c:\\temp\\dfFatKidsALL.csv")
```

Here is what that file looks like:

```
"","Gender","FirstName","Height","Weight","FattyIndex","BMI","Freq"
"1","F","ROSEMARY",35,123,3.5143,70.5869387755102,2
"2","F","LAUREN",54,876,16.2222,211.189300411523,2
"3","M","ALBERT",45,150,3.3333,52.0740740740741,5
"4","M","TOMMY",78,167,2.141,19.2966798159106,5
"5","M","BUDDY",12,189,15.75,922.6875,5
"6","M","FARQUAR",76,198,2.6053,24.0986842105263,5
"7","M","SIMON",87,256,2.9425,23.7769850706831,5
```

You can prevent the row names column above (on the left) from being written out by using the `row.names=FALSE` option:

```
> write.csv(dfFatKidsALL,"c:\\temp\\dfFatKidsALL.csv",row.names=FALSE)
```

Check the R documentation for more on the `write()` series of functions.

Reading From and Writing to Excel 2007

Reading From and Writing To Excel 2007

The package `xlsx` allows you to read from and write to Excel 2007 workbooks. First, to install this package, type the following at the R command line:

```
> install.packages("xlsx",dependencies=TRUE)
```

You will be asked to choose your CRAN Mirror. Do so and click OK. The package will be installed. Next, don't forget to use the `library()` function to make the package available:

```
> library(xlsx)
Loading required package: xlsxjars
Loading required package: rJava
> dfFatKids_xlsx <-
        read.xlsx("C:\\TEMP\\FatKids.xlsx",sheetName="FatKids",as.data.frame=TRUE,header=TRUE)
> print(dfFatKids_xlsx)
  FirstName Height Weight FattyIndex
1    ALBERT     45    150     3.3333
2  ROSEMARY     35    123     3.5143
3     TOMMY     78    167     2.1410
4     BUDDY     12    189    15.7500
5   FARQUAR     76    198     2.6053
6     SIMON     87    256     2.9425
7    LAUREN     54    876    16.2222
```

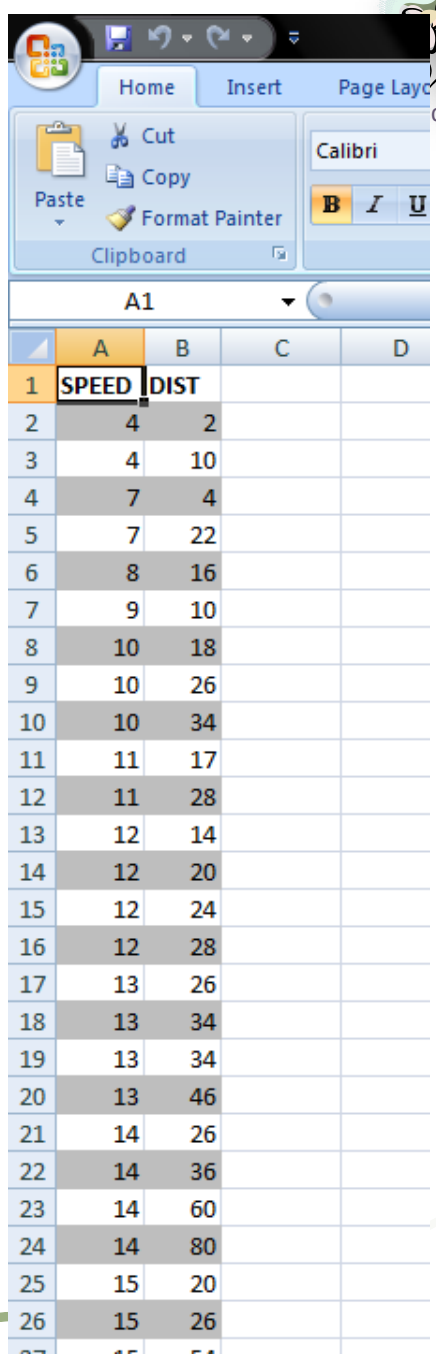
Reading From and Writing To Excel 2007

Here is an example of writing to an Excel 2007 spreadsheet. This example takes the cars data frame and loads it into an Excel 2007 spreadsheet.

```
# Bring in the xlsx library
library(xlsx)

# Bring in the gtools library
library(gtools)

# Bring in the cars dataframe
data(cars)

# Create Workbook
wb <- createWorkbook()

# Create Sheet in the Workbook
sh <- createSheet(wb,sheetName="Client Data")

# Create a bold font object
fo_bold <- createFont(wb,isBold=TRUE)

# Create a cell style object using the bold font
cs_bold <- createCellStyle(wb,font=fo_bold)

# Create a cell style object that has a grey background
cs_gray_back <-
createCellStyle(wb,fillBackgroundColor="grey",fillPattern="SOLID_FOREGROUND",fillForegroundColor="grey")

# Based on the number of rows and columns in the cars dataframe
# create that many rows and columns in the sheet
rows_in_cars <- dim(cars)[1]
cols_in_cars <- dim(cars)[2]
rows <- createRow(sh,1:(rows_in_cars+1)) # Add one for the column headers
cells <- createCell(rows,1:cols_in_cars)

# Place the column name on the first row
setCellValue(cells[[1,1]],toupper(names(cars)[1]))
setCellValue(cells[[1,2]],toupper(names(cars)[2]))
```

Reading From and Writing To Excel 2007

```
# Make the headers bold
setCellStyle(cells[[1,1]],cs_bold)
setCellStyle(cells[[1,2]],cs_bold)

# Place the data into the cells
for(i in 1:rows_in_cars) {

 # Place SPEED in the first column
 setCellValue(cells[[i+1,1]],cars$speed[i])
 if (odd(i)) { setCellStyle(cells[[i+1,1]],cs_gray_back) }

 # Place DIST in the second column
 setCellValue(cells[[i+1,2]],cars$dist[i])
 if (odd(i)) { setCellStyle(cells[[i+1,2]],cs_gray_back) }

}

# Autosize the two columns
autoSizeColumn(sh,1)
autoSizeColumn(sh,2)

# Save the workbook
saveWorkbook(wb,"C:\\temp\\test1.xlsx")
```



	A	B	C	D
1	SPEED	DIST		
2	4	2		
3	4	10		
4	7	4		
5	7	22		
6	8	16		
7	9	10		
8	10	18		
9	10	26		
10	10	34		
11	11	17		
12	11	28		
13	12	14		
14	12	20		
15	12	24		
16	12	28		
17	13	26		
18	13	34		
19	13	34		
20	13	46		
21	14	26		
22	14	36		
23	14	60		
24	14	80		
25	15	20		
26	15	26		

There's more to this than what's shown above. Type in `library(help="xlsx")` at the R command line and read on!

The `gtools` package was used for the `odd()` function.

Factors

Factors

The vast majority of the time, when you read in character data from a text file or database table, those text columns will be used on the BY or CLASS statements in SAS. For example, age groups, regions, state codes, etc., all would be used on the BY or CLASS statements. Also, in most cases, this type of data will repeat several times within the dataset. Since character data can be quite large and take up a lot of space within a dataset, some of us take the time to replace large character strings with either numbers (1,2,3,…) or smaller character strings ('1','2','3',…). This reduces down the size of the dataset and, in the end, replacing '40 to 49' with 4 (say) results in the same computation using less space. But, once the dataset has been summarized, you have to replace the 4 with '40 to 49'…not a big deal (see PROC FORMAT).

When R reads in character data either using `read.csv()`, etc. or pulling data from the database, it maps this character data to numbers. These numbers are called *factors*. You never see any difference in the computations because R replaces the factors with the original character strings for display. This happens behind the scenes.

To determine if your character data is actually a factor, you can use the `str()` function to display the structure of a data frame. If you see "Factor w/ # levels" then your character string was changed to a factor. Again, you will not see a difference when performing computations since R quickly replaces the factors with the original character string for display.

Factors

Now, since we read in the fat kids data into the data frame dfFatKids, using the `str()` function produces the following:

```
> str(dfFatKids)
'data.frame':   7 obs. of  4 variables:
 $ FirstName : Factor w/ 7 levels "ALBERT","BUDDY",..: 1 5 7 2 3 6 4
 $ Height    : int  45 35 78 12 76 87 54
 $ Weight    : int  150 123 167 189 198 256 876
 $ FattyIndex: num  3.33 3.51 2.14 15.75 2.61 ...
>
```

As you see, the FirstName has been replaced with a factor. To see the individual levels, use the function `levels()`:

```
> levels(dfFatKids$FirstName)
[1] "ALBERT"   "BUDDY"    "FARQUAR"  "LAUREN"   "ROSEMARY" "SIMON"    "TOMMY"
> nlevels(dfFatKids$FirstName)
[1] 7
>
```

The `nlevels()` function returns the *number* of levels.

Take note that the `class()` of a factor is "factor" whereas the `mode()` is "numeric":

```
> class(dfFatKids$FirstName)
[1] "factor"
> mode(dfFatKids$FirstName)
[1] "numeric"
>
```

This is because factors are stored as integers internally (see the bold red above).

Factors

Besides the reduction in storage, several R functions expect a factor to do their *thang*. For example, the R `by()` function takes a function (such as the mean) and performs it "by" a factor. Think CLASS or BY statements in SAS. Here is the syntax:

```
by(dataframe-variable, factor-variable, function-name)
```

For example, let's compute the average weight for the males vs. females in the dfFatKids data frame. The first thing we need to do is create a gender column. We will use the R data editor to add this column. At the R Command line type in the following:

```
> dfFatKids <- edit(dfFatKids)
```

When the editor pops up, click on the heading labeled var5 and replace the text var5 with Gender. Next, add in M's and F's as required. Finally, click File…Close.

	FirstName	Height	Weight	FattyIndex	Gender
1	ALBERT	45	150	3.3333	M
2	ROSEMARY	35	123	3.5143	F
3	TOMMY	78	167	2.1410	M
4	BUDDY	12	189	15.7500	M
5	FARQUAR	76	198	2.6053	M
6	SIMON	87	256	2.9425	M
7	LAUREN	54	876	16.2222	F

Factors

Next, let's compute the average weight by Gender. Type in the following:

```
> by(dfFatKidsGender[,"Weight"],dfFatKids$Gender,mean)
```

OR

```
> with(dfFatKidsGender,by(Weight,Gender,mean))
```

The results are as follows:

```
dfFatKids$Gender: F
[1] 499.5
------------------------------------------------------------
dfFatKids$Gender: M
[1] 192
```

As you can see, the output is a little sparse. If you want to save this data into a variable, enter in the following:

```
> Weight_Mean <- by(dfFatKids[,"Weight"],dfFatKids$Gender,mean)
> print(Weight_Mean)
dfFatKids$Gender: F
[1] 499.5
------------------------------------------------------------
dfFatKids$Gender: M
[1] 192
>
```

We kind of cheated here since Gender is actually a character string and not a factor. Why did this work? The `by()` function actually converts to factors itself!!

Factors

If you really needed to convert a character string into a factor, you can use the `as.factor()` function. For example, let's re-create the fat kids data frame changing the Gender character string into a factor:

```
> dfFatKids2$Gender <- as.factor(dfFatKids2$Gender)
> str(dfFatKids2)
'data.frame':   7 obs. of  5 variables:
 $ FirstName : Factor w/ 7 levels "ALBERT","BUDDY",..: 1 5 7 2 3 6 4
 $ Height    : num  45 35 78 12 76 87 54
 $ Weight    : num  150 123 167 189 198 256 876
 $ FattyIndex: num  3.33 3.51 2.14 15.75 2.61 ...
 $ Gender    : Factor w/ 2 levels "F","M": 2 1 2 2 2 2 1
>
```

If you would prefer the Gender factors to be labeled Male instead of M and Female instead of F, then you can rename the factors. First, determine the order of the factors:

```
> levels(dfFatKids2$Gender)
[1] "F" "M"
```

Next, using the `levels()` function, rename the levels in the same order as printed above:

```
> levels(dfFatKids2$Gender) <- c("Female","Male")
> levels(dfFatKids2$Gender)
[1] "Female" "Male"
```

Factors

Next, let's re-run the `by()` example:

```
> by(dfFatKids2[,"Weight"],dfFatKids2$Gender,mean)
dfFatKids2$Gender: Female
[1] 499.5
------------------------------------------------------------
dfFatKids2$Gender: Male
[1] 192
>
```

As you see, *Female* and *Male* are now being printed out.

Now, if you want to see the factor's internal numeric value, you can use the `as.numeric()` function:

```
> as.numeric(dfFatKids$FirstName)
[1] 1 5 7 2 3 6 4
```

Recall that the structure of the fat kids dataset is:

```
> str(dfFatKids)
'data.frame':   7 obs. of  5 variables:
 $ FirstName : Factor w/ 7 levels "ALBERT","BUDDY",..: 1 5 7 2 3 6 4
 $ Height    : num  45 35 78 12 76 87 54
 $ Weight    : num  150 123 167 189 198 256 876
 $ FattyIndex: num  3.33 3.51 2.14 15.75 2.61 ...
 $ Gender    : chr  "M" "F" "M" "M" ...
>
```

Factors

So far we've seen factors produced for text strings like FirstName and Gender, but we can also create factors from numeric variables such as Height and Weight using the `cut()` function. In the next example, we break up the Weight variable into three evenly divided pieces:

```
> Weight_Groups <- cut(dfFatKids$Weight,3)
> print(Weight_Groups)
[1] (122,374] (122,374] (122,374] (122,374] (122,374] (122,374] (625,877]
Levels: (122,374] (374,625] (625,877]
> class(Weight_Groups)
[1] "factor"
```

As you can see, we created a factor called Weight_Groups that is broken up into three groups: (122,374] (374,625] (625,877]. We can also label these groups:

```
> Weight_Groups <- cut(dfFatKids$Weight,3,labels=c("Low","Medium","High"))
> Weight_Groups
[1] Low  Low  Low  Low  Low  Low  High
Levels: Low Medium High
>
```

To apply this factor back to the data as a new variable, instead of creating Weight_Groups, we can create dfFatKids$Weight_Groups:

```
> dfFatKids$Weight_Groups <- cut(dfFatKids$Weight,3,labels=c("Low","Medium","High"))
```

Factors

Now, the fat kids data frame looks like this:

```
   FirstName Height Weight FattyIndex Gender Weight_Groups
1     ALBERT     45    150     3.3333      M          Low
2   ROSEMARY     35    123     3.5143      F          Low
3      TOMMY     78    167     2.1410      M          Low
4      BUDDY     12    189    15.7500      M          Low
5    FARQUAR     76    198     2.6053      M          Low
6      SIMON     87    256     2.9425      M          Low
7     LAUREN     54    876    16.2222      F         High
>
```

Notice that Medium does not appear in the Weight_Groups. This is due to the fact that the `cut()` function just divides up the data into *evenly distributed* pieces and does not automatically compute quantiles or other statistics. To compute quantiles, you can use the `quantile()` function and use it in the `cut()` function. Here is what the `quantile()` function produces by itself:

```
> qnt <- quantile(dfFatKids$Weight)
> print(qnt)
   0%   25%   50%   75%  100%
123.0 158.5 189.0 227.0 876.0
>
```

Next, let's use qnt in the `cut()` function:

Factors


```
> dfFatKids$Weight_Groups2 <-
          cut(dfFatKids$Weight,qnt,labels=c("Grp1","Grp2","Grp3","Grp4"),include.lowest=TRUE)
> print(dfFatKids)
```

	FirstName	Height	Weight	FattyIndex	Gender	Weight_Groups	Weight_Groups2
1	ALBERT	45	150	3.3333	M	Low	Grp1
2	ROSEMARY	35	123	3.5143	F	Low	Grp1
3	TOMMY	78	167	2.1410	M	Low	Grp2
4	BUDDY	12	189	15.7500	M	Low	Grp2
5	FARQUAR	76	198	2.6053	M	Low	Grp3
6	SIMON	87	256	2.9425	M	Low	Grp4
7	LAUREN	54	876	16.2222	F	High	Grp4

Notice that my labels are four groups and not five. This is because the ranges are from [123,158] (158,189] (189,227] (227,876] which is four groups. Also, notice that I am using the parameter include.lowest=TRUE. This parameter, when set to TRUE, allows the left end-point of the lowest cut to be matched with the data. If I did not include this parameter, ROSEMARY would have a Weight_Groups2 set to NA since she is exactly 123 pounds.

To remove the column Weight_Groups (the column that's bogus), you can do this:

```
> dfFatKids3 <- dfFatKids[,-6]  # Weight_Groups is the 6th column. -6 indicates removal of 6th column.
> dfFatKids3
```

	FirstName	Height	Weight	FattyIndex	Gender	Weight_Groups2
1	ALBERT	45	150	3.3333	M	Grp1
2	ROSEMARY	35	123	3.5143	F	Grp1
3	TOMMY	78	167	2.1410	M	Grp2
4	BUDDY	12	189	15.7500	M	Grp2 …

Factors

R has several additional functions associated with factors. In particular, the `gl()` function generates factors based on the arguments to the function. Here is the syntax for the `gl()` function:

```
gl(n,k,length=n*k,labels=1:n,ordered=FALSE)
```

where

n - an integer giving the number of levels.

k - an integer giving the number of replications.

length - an integer giving the length of the result.

labels - an optional vector of labels for the resulting factor levels.

ordered - a logical indicating whether the result should be ordered or not.

For example, let's produce a factor with only two levels but that repeats 3 times (for a total of 2*3=6 values):

```
> gl(2,3)
[1] 1 1 1 2 2 2
Levels: 1 2
```

Next, let's do the same, but use labels "F" and "M":

Factors


```
> gl(2,3,labels=c("F","M"))
[1] F F F M M M
Levels: F M
>
```

The `split()` function takes a data frame as its first parameter and a factor as its second and `split()` will return a **list** of data frames broken up by the factor you specify (we talk more about lists in the next section):

```
> lSplits <- split(dfFatKids2,dfFatKids2$Gender)
> lSplits
$F
  FirstName Height Weight FattyIndex Gender
2  ROSEMARY     35    123     3.5143      F
7    LAUREN     54    876    16.2222      F

$M
  FirstName Height Weight FattyIndex Gender
1    ALBERT     45    150     3.3333      M
3     TOMMY     78    167     2.1410      M
4     BUDDY     12    189    15.7500      M
5   FARQUAR     76    198     2.6053      M
6     SIMON     87    256     2.9425      M
>
```

Note that the class of lSplit$F and lSplit$M is data.frame. See `unsplit()` on how to merge the list data back into a single data frame.

More R Data Structures – The List

More R Data Structures – The List

In the first section on data structures, we learned about the vector, the array, the matrix and the data frame. There is one more data structure you need to know about and it is called the **List**.

Recall that a vector, array and matrix must have the same mode (that is, data type) for all of the elements; that is, all elements must be numeric, or all elements must be character, etc.

Recall that the data frame, on the other hand, contains one or more columns of data each potentially of a different mode, and each column must contain the same number of rows. This is similar to a SAS dataset!

A list is similar to a vector and a data frame combined. Each element in a list can contain a different mode and each element does not necessarily have to have the same number of rows. A list can be thought of as a generalization of the data frame.

For example, here is how we form a vector:

```
> vMyVec <- c(1,2,3,4,5)
> print(vMyVec)
[1] 1 2 3 4 5
>
```

More R Data Structures – The List

Here is how we form a list using the `list()` function:

```
> lMyList <- list(x=c(1,2),y=c("A","B","C"))
> print(lMyList)
$x
[1] 1 2

$y
[1] "A" "B" "C"
```

Notice that unlike a vector and similar to a data frame, we can give names to the elements of a list: x and y. As you see, the first *element* of our list, x, is a vector containing the values 1 and 2. The second "element" of our list, y, is a vector containing the values A, B and C. Note that the term *element*, unlike a vector, does not refer to an individual number, like 2, or letter, like C, but refers to the entire vector c(1,2) or vector c("A","B","C").

Now, to refer to an individual element of our list, we can use the names we've given each element of the list. In general, it's *list_name*$*element_name*, as shown below:

```
> print(lMyList$x)
[1] 1 2
> print(lMyList$y)
[1] "A" "B" "C"
>
```

More R Data Structures – The List

If you want to see the structure of a list, use the `str()` function:

```
> str(lMyList)
List of 2
 $ x: num [1:2] 1 2
 $ y: chr [1:3] "A" "B" "C"
>
```

You can save the elements of a list to a variable in the normal way:

```
> xFromList <- lMyList$x
> mode(xFromList)
[1] "numeric"
> class(xFromList)
[1] "numeric"
> xFromList
[1] 1 2
> yFromList <- lMyList$y
> mode(yFromList)
[1] "character"
> class(yFromList)
[1] "character"
> yFromList
[1] "A" "B" "C"
>
```

More R Data Structures – The List

So, why use a list? Well, a list is an organized way to refer to information such as parameters needed later on in your program that is stored in one variable as opposed to being stored in several variables. For example, let's create a list containing the information needed to pull script data, say:

```
> lRXInfo <- list(ndcs=c("'11111111111'","'22222222222'"),
                  start_date="'2010-01-01'",
                  end_date="'2010-03-31'")
> print(lRXInfo)
$ndcs
[1] "'11111111111'" "'22222222222'"
$start_date
[1] "'2010-01-01'"
$end_date
[1] "'2010-03-31'"
```

Now, you can refer to each of these elements separately in say a SQL Query:

```
> sSQLQuery <- paste("SELECT * FROM MYRXS WHERE NDC_KEY IN (",
                     paste(lRXInfo$ndcs,collapse=","),
                     ") AND DATE_OF_SERVICE BETWEEN DATE ",
                     lRXInfo$start_date,
                     "AND DATE ",
                     lRXInfo$end_date
                    )
>
> sSQLQuery
[1] "SELECT * FROM MYRXS WHERE NDC_KEY IN ( '11111111111','22222222222' ) AND DATE_OF_SERVICE
BETWEEN DATE  '2010-01-01' AND DATE  '2010-03-31'"
>
```

More R Data Structures – The List

Recall that we referred to elements of a vector using an index into the vector itself:

```
> vMyVec <- c(1,2,3,4,5)
> print(vMyVec[2])
[1] 2
> print(vMyVec[3])
[1] 3
>
```

Due to the complexity of lists, you have to refer to each element in a list using double-left and double-right brackets:

```
> lRXInfo[[1]]
[1] "'11111111111'" "'22222222222'"
>
```

Since lRXInfo[[1]] refers to the vector of NDC codes, how do we refer to each element of the vector at this point? To refer to the second NDC code, do this:

```
> lRXInfo[[1]][2]
[1] "'22222222222'"
>
```

R Options

R Options

Similar to SAS's System Options, you can modify R's system options. To obtain a full list of system options and their currently set values, type in the function name `options()` at the command line. Note that `options()` returns a List:

```
> options()
$add.smooth
[1] TRUE

$check.bounds
[1] FALSE

$continue
[1] "+ "

$contrasts
        unordered           ordered
"contr.treatment"      "contr.poly"

$defaultPackages
[1] "datasets"  "utils"     "grDevices" "graphics"  "stats"     "methods"

…skipped several…

$verbose
[1] FALSE

$width
[1] 80
```

R Options

To change the linesize of the page, in SAS you would specify LS=#; in R use the list element `width`. To retrieve the current value of a specific option, use the following code:

```
> options()$width
[1] 80
>
```

or

```
> options("width")
$width
[1] 80
```

To set the width to a new value, use the following syntax:

```
> options(width=132)
> options("width")
$width
[1] 132
```

Another nice option is the digits option. This is the number of digits to display for numeric data. Currently, it is set to 7, but can be changed as we show on the next slide:

R Options


```
> options("digits")
$digits
[1] 7
> print(pi)
[1] 3.141593
> options(digits=12)
> print(pi)
[1] 3.14159265359
```

Note that if you want a detailed explanation of most of the options in R, see the documentation at `?options`.

You can save the current set of options in a variable by using the following:

```
> op <- options()
```

You can then restore those options like this:

```
> options(op)
```

Now, besides options, R has several objects (functions, lists, etc.) to tell you about your R setup. The `.Machine` list tells you about the numerical characteristics of your machine. For example, `$double.xmax` tells you what the largest number can be used in R. See the documentation for an explanation of the list elements produced.

R Options

The `.Platform` list tell you about the platform on which R is installed. For example, `$OS.type` is "windows" on my computer. See the documentation for an explanation of the list elements produced.

The `capabilities()` function lists certain capabilities available with the installed version of R.:

```
> as.data.frame(capabilities())
         capabilities()
jpeg               TRUE
png                TRUE
tiff               TRUE
tcltk              TRUE
X11               FALSE
aqua              FALSE
http/ftp           TRUE
sockets            TRUE
libxml             TRUE
fifo              FALSE
cledit             TRUE
iconv              TRUE
NLS                TRUE
profmem            TRUE
cairo             FALSE
>
```

As you see above, you can produce jpeg's, png's, etc. with this version of R, but the X11 (Unix) and aqua (Mac) display functions are not available (which makes sense since I'm running on Windows).

R Options

In addition, the functions `R.Version()` list and `R.home()` vector display the version of R and the home directory where R is installed:

```
> R.Version()
$platform
[1] "i386-pc-mingw32"

$arch
[1] "i386"

$os
[1] "mingw32"

$system
[1] "i386, mingw32"

$major
[1] "2"

$minor
[1] "10.1"

$version.string
[1] "R version 2.10.1 (2009-12-14)"

> R.home()
[1] "C:\\PROGRA~2\\R\\R-210~1.1"
>
```

R Options

Finally, if you would like to have certain actions take place during the startup of R, you can modify the `Rprofile.site` file located in the /etc folder on Linux. On Windows, it is located in C:\Program Files (x86)\R\R-2.10.1\etc (for my version 2.10.1). This file, if found, will be sourced into your R session.

You can also place a `.Rprofile` file in your current working directory or in the user's home directory. This file, if it exists, will be sourced after `Rprofile.site`.

Here is what my Rprofile.site file looks like:

```
# Things you might want to change
# options(papersize="a4")
# options(editor="notepad")
# options(pager="internal")

# set the default help type
# options(help_type="text")
  options(help_type="html")

# set a site library
# .Library.site <- file.path(chartr("\\", "/", R.home()), "site-library")

# set a CRAN mirror
# local({r <- getOption("repos")
#       r["CRAN"] <- "http://my.local.cran"
#       options(repos=r)})

library(utils)
if (.Platform$GUI %in% c("Rgui","Rterm")){
  if(!("rcom" %in% installed.packages()[, "Package"])){
    install.packages("rcom", dep = T);
    require(rcom);
    dummy<-comRegisterRegistry()} else {
    require(rcom)}}
```

R Options

There are two special functions which you can place in these files: `.First()` and `.Last()`. `.First()` will be executed when R starts, and `.Last()` will be executed when R is closed. You can place `library()` functions within the `.First()` function to load in your favorite packages…this prevents having to type them in each time you start R!! For example,

```
.First <- function() {
 cat("\nLoading chron package!\n")
 require(chron)
}

.Last <- function(){
 cat("\nGoodbye at ", date(), "\n")
}
```

Note that I used the function `require()` instead of `library()`. Both do the same thing, but `library()` will report an error if the package is not found, but `require()` will give just a warning.

Function Round-Up #2

Data Structure Initialization

Function Round-Up #2 – Data Structure Initialization

Besides having several built-in data frames for your use, like the cars data frame (speed and stopping distance of cars) and faithful (Old Faithful Geyser data), R has several built-in vectors, factors, matrices, etc. that you might find useful:

state.abb: character vector of 2-letter abbreviations for the state names.

state.area: numeric vector of state areas (in square miles).

state.center: list with components named x and y giving the approximate geographic center of each state in negative longitude and latitude. Alaska and Hawaii are placed just off the West Coast.

state.division: factor giving state divisions (New England, Middle Atlantic, South Atlantic, East South Central, West South Central, East North Central, West North Central, Mountain, and Pacific).

state.name: character vector giving the full state names.

state.region: factor giving the region (Northeast, South, North Central, West) that each state belongs to.

state.x77: matrix with 50 rows and 8 columns giving the following statistics in the respective columns:

Population: population estimate as of July 1, 1975
Income: per capita income (1974)
Illiteracy: illiteracy (1970, percent of population)
Life Exp: life expectancy in years (1969–71)
Murder: murder and non-negligent manslaughter rate per 100,000 population (1976)
HS Grad: percent high-school graduates (1970)
Frost: mean number of days with minimum temperature below freezing (1931–1960) in capital or large city
Area: land area in square miles

Function Round-Up #2 – Data Structure Initialization

For example, let's produce a data frame that contains the two-letter state codes as well as which one of the nine divisions the state belongs to:

```
> dfStateInfo <- data.frame(STABB=state.abb,STDIV=state.division)
> dfStateInfo
      STABB              STDIV
1         AL East South Central
2         AK            Pacific
3         AZ           Mountain
4         AR West South Central
5         CA            Pacific
6         CO           Mountain
7         CT        New England
8         DE     South Atlantic
9         FL     South Atlantic
10        GA     South Atlantic
11        HI            Pacific
12        ID           Mountain
13        IL East North Central
…
```

Note that the `data.frame()` function is one of several function you can use to create data structures in R besides the methods we've showed you thus far.

```
data.frame(colname1=var1, colname2=var2, … , colnamen=varn)
```

Where colname# is the name you want that column to be call in the data frame, and var# is the vector, etc. you want to be included in the data frame.

Function Round-Up #2 – Data Structure Initialization

Here are the data structure initialization functions:

`vector(mode = "`*`mode`*`", length = 0)` – produces a vector of mode *mode* and length *length*.
`numeric(length=0)` – produces a vector of zeroes of length *length*.
`character(length=0)` – produces a vector of null strings ("") of length *length*.
`logical(length=0)` – produces a vector of FALSEs of length *length*.
`integer(length=0)` – produces a vector of integral zeroes of length *length*.
`list(…)` – produces a list from the arguments. See the section on Lists in this lecture.

```
> z <- numeric(5)
> z
[1] 0 0 0 0 0
> y <- vector(mode="logical",length=5)
> y
[1] FALSE FALSE FALSE FALSE FALSE
> y <- vector(mode="numeric",length=5)
> y
[1] 0 0 0 0 0
> y <- vector(mode="character",length=5)
> y
[1] "" "" "" "" ""
> x <- character(length=5)
> x
[1] "" "" "" "" ""
> w <- logical(length=5);
> w
[1] FALSE FALSE FALSE FALSE FALSE
```

Note that the functions `real()`, `double()` and `single()` are equivalent to `numeric()`.

Function Round-Up #2 – Data Structure Initialization

Along with the functions listed on the previous slide, each one comes with an associated conversion function (`as.`*`function_name`*`()`) and a test function (`is.`*`function_name`*`()`). If you want to convert from a character string to a number, you can use `as.numeric()` function:

```
> z <- c("123","1.245")
> z
[1] "123"   "1.245"
> y <- as.numeric(z)
> y
[1] 123.000   1.245
>
```

If the `as.numeric()` function cannot convert the character value to a number, it produces a warning, but replaces the problem child with NA:

```
> z <- c("123","456","ABC")
> y <- as.numeric(z)
Warning message:
NAs introduced by coercion
> y
[1] 123 456  NA
>
```

Handling Dates in R

Handling Dates in R

Recall that in SAS, a SAS date is the number of **days** since January 1st, 1960, and a SAS datetime is the number of seconds since January 1st, 1960. Well, it's probably no surprise that R has a way to store dates and times as well.

One of the easiest ways to represent dates is by using the `as.Date()` conversion function to represent dates as the number of days since January 1st, 1970. This function takes a character string in either the "%Y-%m-%d" or "%Y/%m/%d" formats (similar to Oracle's "YYYY-MM-DD" or "YYYY/MM/DD" formats) and converts it to a date. If your character string is not in one of those two default formats, you can provide an additional `format="`*`format_string`*`"` parameter to the `as.Date()` function using the formats shown below:

```
%d – day of the month
%m – month (number)
%b – month (abbreviated name)
%B – month (full name)
%y – year (two-digit)
%Y – year (four-digit)
```

For example, to represent the date April 17, 1986, enter the following:

```
> adate <- as.Date("April 17, 1986","%B %d,%Y")
> print(adate)
[1] "1986-04-17"
>
> class(adate)
[1] "Date"
>
```

Handling Dates in R

If you have a second date, you can perform subtraction to get back the number of days between those two days:

```
> bdate <- as.Date("April 17, 1987","%B %d,%Y")
> print(bdate-adate)
Time difference of 365 days
> class(diff)
[1] "difftime"
>
```

If you would prefer to work with a number rather than the difftime class as shown above, you can use the `as.numeric()` conversion function:

```
> diff <- as.numeric(bdate-adate)
> print(diff)
[1] 365
```

As mentioned above, if your character string is in one of the two default formats, you don't need the format parameter:

```
> cdate <- as.Date("2010-08-10")
> print(cdate)
[1] "2010-08-10"
>
```

You can perform other arithmetic on these dates such as subtraction (shown above) and addition:

Handling Dates in R


```
> print(bdate)
[1] "1987-04-17"
> print(bdate+365)
[1] "1988-04-16"
>
```

If you would like today's date, use `Sys.Date()` function:

```
> z <- Sys.Date()
> print(z)
[1] "2010-08-20"
```

Note that dates using the `as.Date()` function have a class of Date and a mode of numeric:

```
> class(z)
[1] "Date"
> mode(z)
[1] "numeric"
>
```

Once you have a variable with one or more dates in it, you can use the `format()` function to format those dates back to character strings:

Handling Dates in R


```
> sDates <- c("2010-01-01","2010-01-02","2010-01-03")
> dDates <- as.Date(sDates)
> print(dDates)
[1] "2010-01-01" "2010-01-02" "2010-01-03"
> print(dDates+31)
[1] "2010-02-01" "2010-02-02" "2010-02-03"
> sDates2 <- format(dDates,"%d%b%Y")
> print(sDates2)
[1] "01Jan2010" "02Jan2010" "03Jan2010"
> class(sDates2)
[1] "character"
> mode(sDates2)
[1] "character"
>
```

As you see above, sDates2 is an actual character vector. Now, dates produced by the `as.Date()` function can use the following functions to produce the month name, weekday name and quarter:

```
> months(dDates)
[1] "January" "January" "January"
> months(dDates,abbreviate=T)
[1] "Jan" "Jan" "Jan"
> weekdays(dDates,abbreviate=T)
[1] "Fri" "Sat" "Sun"
> weekdays(dDates,abbreviate=F)
[1] "Friday"   "Saturday" "Sunday"
> quarters(dDates)
[1] "Q1" "Q1" "Q1"
```

Handling Dates in R

Another nice feature with dates produced from the `as.Date()` function is the ability to produce sequences of dates using the `seq()` function we discussed in the first function round-up. The parameter names are the same, but you can put in dates as the arguments:

```
> dStart <- as.Date("2010-01-15")
> dEnd <- as.Date("2010-01-31")
> print(dStart)
[1] "2010-01-15"
> print(dEnd)
[1] "2010-01-31"
> seq(dStart,dEnd,1)
 [1] "2010-01-15" "2010-01-16" "2010-01-17" "2010-01-18" "2010-01-19"
 [6] "2010-01-20" "2010-01-21" "2010-01-22" "2010-01-23" "2010-01-24"
[11] "2010-01-25" "2010-01-26" "2010-01-27" "2010-01-28" "2010-01-29"
[16] "2010-01-30" "2010-01-31"
>
```

The next way to produce dates is to use the `chron` package. This package is not installed by default, so you will have to install it (`install.packages("chron", dep=T)`). Unfortunately, this package expects times to be separated from dates and its precision is not the best, so we will skip discussing this package.

Handling Dates in R

Two additional ways to represent dates are the POSIXct and POSIXlt datetime classes.

The POSIXct class represents datetimes in the "usual" manner as the number of **seconds** since 1/1/1970.

The POSIXlt class represents dates in a list data structure containing the list elements: `sec` (0-61, number of seconds), `min` (0-59), `hour` (0-23), `mday` (1-31, day of the month), `mon` (0-11, 0=JAN, …, 11=DEC), `year` (year number since 1900), `wday` (0-6 day of week, 0=Sunday), and `yday` (0-365, day of year).

If you do not need the list concept, then the usual way to store dates is using POSIXct classes. Just as a reminder, the `as.Date()` function returns the number of **days** since 1/1/1970 and the `as.POSIXct()` function returns the number of **seconds** since 1/1/1970. This is analogous to SAS date and datetime values, resp.

Just to be clear, both the dates produced with `as.Date()` and the datetimes produced with `as.POSIXct()` play nicely. That is, you can convert a date produced by `as.Date()` into a datetime using `as.POSIXct()`.

Handling Dates in R

For example,

```
> dDate1 <- as.Date("2010-04-25")
> dDate1
[1] "2010-04-25"
> dDate2 <- as.POSIXct(dDate1)
> dDate2
[1] "2010-04-24 20:00:00 EDT"
> as.numeric(dDate2)
[1] 1272153600
> as.numeric(dDate1)
[1] 14724
>
```

Note that dates without times are treated as midnight UTC (GMT)! Add 4 or 5 hours (as seconds!) to correct this:

```
> dDate2 <- as.POSIXct(dDate1) + 4*60*60
> print(dDate2)
[1] "2010-04-25 EDT"
```

Recall that `Sys.Date()` returns the number of days since 1/1/1970. You can get both the date and time using `Sys.time()` (note the capitalization!) which returns the number of seconds since 1/1/1970:

Handling Dates in R


```
> Sys.time()
[1] "2010-08-21 13:04:37 EDT"
> Sys.Date()
[1] "2010-08-21"
> as.numeric(Sys.Date())
[1] 14842
> as.numeric(Sys.time())
[1] 1282410311
>
```

Note that the `date()` function is equivalent to the `Sys.time()` function.

The other datetime option is to use POSIXlt which is a list structured version of datetimes:

```
> lDate2 <- as.POSIXlt(dDate2)
> print(lDate2)
[1] "2010-04-25 EDT"
> str(lDate2)
 POSIXlt[1:9], format: "2010-04-25"
> unlist(lDate2)
  sec   min  hour  mday   mon  year  wday  yday isdst
    0     0     0    25     3   110     0   114     1
> lDate2$mday
[1] 25
```

Recall that we can use the sequence `seq()` function to produce sequences with dates produced with `as.Date()`. We can do the same for datetimes produced by `as.POSIXct()`:

Handling Dates in R


```
> vDateSeq <- seq(from=dtBegDate,to=dtEndDate,by="1 days")
> head(vDateSeq)
[1] "2010-01-01 EST" "2010-01-02 EST" "2010-01-03 EST" "2010-01-04 EST" "2010-01-05 EST" "2010-01-06 EST"
> length(vDateSeq)
[1] 89
> vDateSeq
 [1] "2010-01-01 00:00:00 EST" "2010-01-02 00:00:00 EST" "2010-01-03 00:00:00 EST" "2010-01-04 00:00:00 EST"
 [5] "2010-01-05 00:00:00 EST" "2010-01-06 00:00:00 EST" "2010-01-07 00:00:00 EST" "2010-01-08 00:00:00 EST"
 [9] "2010-01-09 00:00:00 EST" "2010-01-10 00:00:00 EST" "2010-01-11 00:00:00 EST" "2010-01-12 00:00:00 EST"
[13] "2010-01-13 00:00:00 EST" "2010-01-14 00:00:00 EST" "2010-01-15 00:00:00 EST" "2010-01-16 00:00:00 EST"
[17] "2010-01-17 00:00:00 EST" "2010-01-18 00:00:00 EST" "2010-01-19 00:00:00 EST" "2010-01-20 00:00:00 EST"
[21] "2010-01-21 00:00:00 EST" "2010-01-22 00:00:00 EST" "2010-01-23 00:00:00 EST" "2010-01-24 00:00:00 EST"
[25] "2010-01-25 00:00:00 EST" "2010-01-26 00:00:00 EST" "2010-01-27 00:00:00 EST" "2010-01-28 00:00:00 EST"
[29] "2010-01-29 00:00:00 EST" "2010-01-30 00:00:00 EST" "2010-01-31 00:00:00 EST" "2010-02-01 00:00:00 EST"
[33] "2010-02-02 00:00:00 EST" "2010-02-03 00:00:00 EST" "2010-02-04 00:00:00 EST" "2010-02-05 00:00:00 EST"
[37] "2010-02-06 00:00:00 EST" "2010-02-07 00:00:00 EST" "2010-02-08 00:00:00 EST" "2010-02-09 00:00:00 EST"
[41] "2010-02-10 00:00:00 EST" "2010-02-11 00:00:00 EST" "2010-02-12 00:00:00 EST" "2010-02-13 00:00:00 EST"
[45] "2010-02-14 00:00:00 EST" "2010-02-15 00:00:00 EST" "2010-02-16 00:00:00 EST" "2010-02-17 00:00:00 EST"
[49] "2010-02-18 00:00:00 EST" "2010-02-19 00:00:00 EST" "2010-02-20 00:00:00 EST" "2010-02-21 00:00:00 EST"
[53] "2010-02-22 00:00:00 EST" "2010-02-23 00:00:00 EST" "2010-02-24 00:00:00 EST" "2010-02-25 00:00:00 EST"
[57] "2010-02-26 00:00:00 EST" "2010-02-27 00:00:00 EST" "2010-02-28 00:00:00 EST" "2010-03-01 00:00:00 EST"
[61] "2010-03-02 00:00:00 EST" "2010-03-03 00:00:00 EST" "2010-03-04 00:00:00 EST" "2010-03-05 00:00:00 EST"
[65] "2010-03-06 00:00:00 EST" "2010-03-07 00:00:00 EST" "2010-03-08 00:00:00 EST" "2010-03-09 00:00:00 EST"
[69] "2010-03-10 00:00:00 EST" "2010-03-11 00:00:00 EST" "2010-03-12 00:00:00 EST" "2010-03-13 00:00:00 EST"
[73] "2010-03-14 00:00:00 EST" "2010-03-15 01:00:00 EDT" "2010-03-16 01:00:00 EDT" "2010-03-17 01:00:00 EDT"
[77] "2010-03-18 01:00:00 EDT" "2010-03-19 01:00:00 EDT" "2010-03-20 01:00:00 EDT" "2010-03-21 01:00:00 EDT"
[81] "2010-03-22 01:00:00 EDT" "2010-03-23 01:00:00 EDT" "2010-03-24 01:00:00 EDT" "2010-03-25 01:00:00 EDT"
[85] "2010-03-26 01:00:00 EDT" "2010-03-27 01:00:00 EDT" "2010-03-28 01:00:00 EDT" "2010-03-29 01:00:00 EDT"
[89] "2010-03-30 01:00:00 EDT"
```

Besides *days*, you can provide the by= parameter with *secs*, *hours*, *mins*, *days*, and *weeks*.

Note above that we used the `length()` function. This function returns the number of elements in a vector.

Handling Dates in R

Two additional helper functions `ISOdate()` and `ISOdatetime()`, both of which return POSIXct *datetimes*, are similar to the `dhms()` function in SAS. These two functions have the following parameters:

```
ISOdatetime(year, month, day, hour, min, sec, tz = "")
ISOdate(year, month, day, hour = 12, min = 0, sec = 0, tz = "GMT")
```

Note that for `ISOdatetime()`, the first six parameters must be specified, whereas only the first three need to be specified when using `ISOdate()`:

```
> dDate1 <- ISOdate(2010,1,1)
> print(dDate1)
[1] "2010-01-01 12:00:00 GMT"
> dDate2 <- ISOdatetime(2010,1,1,0,0,0)
> print(dDate2)
[1] "2010-01-01 EST"
> dDate1 <- ISOdate(2010,1,1,hour=0,tz="EST")
> print(dDate1)
[1] "2010-01-01 EST"
>
```

Recall in SAS, in order to find the number of weeks, say, between two dates, we use the `INTCK()` function. In R, we can use the `difftime()` function to obtain a similar value:

Handling Dates in R

For example, let's determine the number of weeks between January 1, 2010 and March 31, 2010:

```
> dDate1 <- ISOdate(2010,1,1,hour=0,tz="EST")
> dDate2 <- ISOdate(2010,3,31,hour=0,tz="EST")
> print(dDate2-dDate1)
Time difference of 89 days
> difftime(dDate2,dDate1)
Time difference of 89 days
> difftime(dDate2,dDate1,units="weeks")
Time difference of 12.71429 weeks
> 89/7
[1] 12.71429
>
```

Note that you can still use the `format()` function with POSIX datetimes (as we showed above for dates produced with `as.Date()`):

```
> format(dDate1,"%B %d, %Y")
[1] "January 01, 2010"
>
```

Two additional functions for use with POSIX dates are `strptime()`, used to read in character strings and convert them into dates; and `strftime()` is similar to `format()` for datetimes. We won't discuss these here, so please see the R Manual.

Reading in Data from Text Files - Revisited

Reading in Data from Text Files – Revisited

Recall that we introduced the functions `read.csv()`, `read.delim()`, `read.table()`, etc. Recall that when we read in a character string using one of these functions, R converted that string to a *factor*. Now that we've talked about factors, we can look into this strange behavior more fully.

The reason R converts text strings to factors is to save space in the data frame. If you repeat the same text string "How Now Brown Cow?" over and over again, you will be taking up 18 bytes on each row of data. If you have a million rows of data, you have sucked up about 18MBs of space. R will replace the text with a factor such as an integer such as 1, 2, 3, … which takes up much less space in the data frame. You can think of factors as a space-saving device.

Now, there are several additional pieces of information related to factors that you need to be aware of when using the read-family of functions. First, there is the function `default.stringsAsFactors()` used by R to determine if you want strings to be automatically converted to factors or not. This function returns TRUE (indicating that you **do** want strings to be converted to factors) or FALSE (no thanks, factors stink PU!). The default installation of R has `default.stringsAsFactors()` return TRUE. When you use the read-family of functions, it determines whether to convert strings to factors based on the return value of `default.stringsAsFactors()`.

Reading in Data from Text Files – Revisited

While you can change the result of this function by changing the global option using `options(stringsAsFactors=FALSE)`, you can use one of several parameters provided in the read-family of functions.

The `stringsAsFactors=` parameter is set to `default.stringsAsFactors()`. You can override this by provided this parameter as well as the logical value you want:

```
> dfFatKids <- read.csv("C:\\Users\\Scott\\Desktop\\R Series\\FatKids.csv")
> str(dfFatKids)
'data.frame':   7 obs. of  4 variables:
 $ FirstName : Factor w/ 7 levels "ALBERT","BUDDY",..: 1 5 7 2 3 6 4
 $ Height    : int  45 35 78 12 76 87 54
 $ Weight    : int  150 123 167 189 198 256 876
 $ FattyIndex: num  3.33 3.51 2.14 15.75 2.61 ...

> dfFatKids <- read.csv("C:\\Users\\Scott\\Desktop\\R Series\\FatKids.csv",
                        stringsAsFactors=FALSE)
> str(dfFatKids)
'data.frame':   7 obs. of  4 variables:
 $ FirstName : chr  "ALBERT" "ROSEMARY" "TOMMY" "BUDDY" ...
 $ Height    : int  45 35 78 12 76 87 54
 $ Weight    : int  150 123 167 189 198 256 876
 $ FattyIndex: num  3.33 3.51 2.14 15.75 2.61 ...
>
```

Another useful parameter is the `as.is=` parameter. You can set it to TRUE or FALSE as well. TRUE indicates no conversion to factors whereas FALSE indicates conversion to factors.

Reading in Data from Text Files – Revisited

For example:

```
> dfFatKids <- read.csv("C:\\Users\\Scott\\Desktop\\R Series\\FatKids.csv",as.is=TRUE)
> str(dfFatKids)
'data.frame':   7 obs. of  4 variables:
 $ FirstName : chr  "ALBERT" "ROSEMARY" "TOMMY" "BUDDY" ...
 $ Height    : int  45 35 78 12 76 87 54
 $ Weight    : int  150 123 167 189 198 256 876
 $ FattyIndex: num  3.33 3.51 2.14 15.75 2.61 ...

> dfFatKids <- read.csv("C:\\Users\\Scott\\Desktop\\R Series\\FatKids.csv",as.is=FALSE)
> str(dfFatKids)
'data.frame':   7 obs. of  4 variables:
 $ FirstName : Factor w/ 7 levels "ALBERT","BUDDY",..: 1 5 7 2 3 6 4
 $ Height    : int  45 35 78 12 76 87 54
 $ Weight    : int  150 123 167 189 198 256 876
 $ FattyIndex: num  3.33 3.51 2.14 15.75 2.61 ...
>
```

Recall that SAS allows you to use the `LENGTH` and `INFORMAT` statements to indicate to SAS whether a variable is numeric or character and how to read in the data from the text file – that is, what *informat* to use. The read-family of functions provides an additional parameter named `colClasses=` which indicates what mode (i.e., data type) and class each column should be as well as whether the column should be read into the data frame at all!

Reading in Data from Text Files – Revisited

The syntax for the `colClasses=` parameter are as follows:

```
colClasses=c("mode","mode", …, "mode")
```

where *mode* is one of the following:

```
"NULL" – used if you do not want that column to be inputted into the data frame
"logical" – boolean column
"integer" – integer column
"numeric" – numeric column
"character" – character column (no conversion to factors is done!)
"factor" – character-to-factor conversion performed
"Date" – date produced in a similar manner to as.Date()
"POSIXct" – datetime produced in a similar manner to as.POSIXct()
```

Note that the comma-delimited list of modes must be in the same order as the columns you are reading in!

Now, let's pimp out our code. Here is the original:

```
> dfFatKids <- read.csv("C:\\Users\\Scott\\Desktop\\R Series\\FatKids.csv")
> str(dfFatKids)
'data.frame':   7 obs. of  4 variables:
 $ FirstName : Factor w/ 7 levels "ALBERT","BUDDY",..: 1 5 7 2 3 6 4
 $ Height    : int  45 35 78 12 76 87 54
 $ Weight    : int  150 123 167 189 198 256 876
 $ FattyIndex: num  3.33 3.51 2.14 15.75 2.61 ...
>
```

Reading in Data from Text Files – Revisited

Let's use the `colClasses=` parameter to read in the Fat Kids data:

```
> dfFatKids2 <- read.csv("C:\\Users\\Scott\\Desktop\\R Series\\FatKids.csv",
                         colClasses=c("character","integer","integer","numeric"))
> str(dfFatKids2)
'data.frame':   7 obs. of  4 variables:
 $ FirstName : chr  "ALBERT" "ROSEMARY" "TOMMY" "BUDDY" ...
 $ Height    : int  45 35 78 12 76 87 54
 $ Weight    : int  150 123 167 189 198 256 876
 $ FattyIndex: num  3.33 3.51 2.14 15.75 2.61 ...
>
```

Next, let's tell R *not* to read in the Weight column:

```
> dfFatKids3 <- read.csv("C:\\Users\\Scott\\Desktop\\R Series\\FatKids.csv",
                          colClasses=c("character","integer","NULL","numeric"))
> str(dfFatKids3)
'data.frame':   7 obs. of  3 variables:
 $ FirstName : chr  "ALBERT" "ROSEMARY" "TOMMY" "BUDDY" ...
 $ Height    : int  45 35 78 12 76 87 54
 $ FattyIndex: num  3.33 3.51 2.14 15.75 2.61 ...
>
```

Finally, let's read in dates from a modified Fat Kids text file containing their date of birth as well as the other columns already in that text file. Note that these dates must be in one of the default formats in order to be read in correctly!

Reading in Data from Text Files – Revisited

Here is what our new text file looks like:

```
FirstName,Height,Weight,FattyIndex,BirthDate
ALBERT,45,150,3.3333,2001/01/15
ROSEMARY,35,123,3.5143,2002/02/16
TOMMY,78,167,2.1410,2003/03/17
BUDDY,12,189,15.7500,2004/04/18
FARQUAR,76,198,2.6053,2005/05/19
SIMON,87,256,2.9425,2006/06/20
LAUREN,54,876,16.2222,2007/07/21
```

Here is our code to read in that data:

```
> dfFatKidsWithBirthDates <-
                        read.csv("C:\\Users\\Scott\\Desktop\\RSeries\\FatKidsWithBirthDates.csv",
                                 colClasses=c("character","integer","NULL","numeric","Date"))
    > str(dfFatKidsWithBirthDates)
    'data.frame':   7 obs. of  4 variables:
     $ FirstName : chr  "ALBERT" "ROSEMARY" "TOMMY" "BUDDY" ...
     $ Height    : int  45 35 78 12 76 87 54
     $ FattyIndex: num  3.33 3.51 2.14 15.75 2.61 ...
     $ BirthDate :Class 'Date'  num [1:7] 11337 11734 12128 12526 12922 ...
    > dfFatKidsWithBirthDates
      FirstName Height FattyIndex  BirthDate
    1    ALBERT     45     3.3333 2001-01-15
    2  ROSEMARY     35     3.5143 2002-02-16
    3     TOMMY     78     2.1410 2003-03-17
    4     BUDDY     12    15.7500 2004-04-18
    5   FARQUAR     76     2.6053 2005-05-19
    6     SIMON     87     2.9425 2006-06-20
    7    LAUREN     54    16.2222 2007-07-21
    >
```

Creating Your Own Functions in R

Creating Your Own Functions in R

So far we've learned a lot about the functions provided by R's base package. You'll be happy and thrilled to know that you – lowly R programmer wannabe – can create your own functions in R. Here is the general syntax for creating a function in R:

```
function_name <- function(arg1=default_value1, arg2=default_value2, … , argn=default_valuen)
{
 body_of_function
 return(value)
}
```

where

function_name is the name of your function,
arg# are the formal arguments to the function,
default_value# are the default values provided (if desired),
body_of_function are the R commands to produce the desired results,
value is the desired return value from the function.

Note that similar to SAS Macros, you do not need to provide *named* formal arguments to the function, but can provide *positional* arguments. I recommend avoiding this option if at all possible since positional arguments can be inadvertently mixed up.

Creating Your Own Functions in R

Note that arguments to functions can be one of the data structures we've talked about in this lecture series, such as vectors, data frames, lists, etc.

Let's create a simple function called `fAddOne` that adds the number one to each element of a vector passed into our function:

```
> fAddOne <- function(pVec=NA)
 {
  tmpVec <- pVec
  tmpVec <- tmpVec + 1
  return(tmpVec)
 }
> vNums <- c(1,2,3)
> vNums
[1] 1 2 3
> fAddOne(vNums)
[1] 2 3 4
>
```

Notice that you are passing a vector into the function, but you are also returning a vector from the function using the `return()` function.

Note that if you do not use a `return()` function, the value of the last variable or function will be returned.

Creating Your Own Functions in R

A nice feature of R's user-defined functions is that you can define a symbol (such as "+") as a function name instead of an actual name (such as "sum"). For example, let's define the tilde (~) symbol to be the square root of the sum of squares of two given numbers (the "norm" of two numbers):

```
> `~` <- function(x,y) {

  return(sqrt(x*x + y*y))

 }
> 3~4
[1] 5
>
```

Note that the two arguments can be any name you want. Note that this function call is equivalent to `` `~`(x,y) `` which makes it seem like a real function, such as `mean(5,2)`.

You can list the arguments and default values to any function, by using the `args(`*`function_name`*`)` function, where *`function_name`* is the name of the function (such as mean) or the full symbol (such as `` `~` ``):

```
> args(`~`)
function (x, y)
NULL
> args(mean)
function (x, ...)
NULL
```

Creating Your Own Functions in R

You can show the body (i.e., the code) of a function by using the `body(function_name)` function:

```
> body(`~`)
{
    return(sqrt(x * x + y * y))
}
>
```

If you've run out of symbols to use to define your functions, R let's you use `%text%` instead:

```
> `%fNorm%` <- function(x,y) {

  return(sqrt(x*x + y*y))

 }
> 3 %fNorm% 4
[1] 5
>
```

Now, if your own function needs to clean up after itself before it exits, you can use the `on.exit()` function to tell R what to do just before the function exits. The syntax is:

```
on.exit(expression, add=TRUE|FALSE)
```

Creating Your Own Functions in R

where

expression is an expression or function to run

add= indicates whether to replace the original expression (when FALSE) or to add an additional expression to the list of things to run (when TRUE)

For example, let's create function that changes the number of digits used to display a number, and then at the end, reset the option back to its original value (default is 7):

```
fMyFunc <- function(x) {

 on.exit(options(digits=7),add=FALSE)

 options(digits=22)
 print(x/3)

}
> fMyFunc(10)
[1] 3.333333333333333
> 10/3
[1] 3.333333
>
```

Creating Your Own Functions in R

Recall that we learned about the `by()` function earlier in the presentation. This function will compute the mean (or other function you specify) on a group variable. This is similar to PROC MEANS with a CLASS Statement. The nice thing about the `by()` function (and functions like it) is that you can specify your *own function* rather than just using pre-defined R functions.

For example, let's compute the average body mass index (BMI) by gender on the Fat Kids data frame by creating our own function:

```
fBMI <- function(pDFHW) {

 tDF <- pDFHW
 tDF$BMI <- 703*tDF$Weight/tDF$Height^2
 return(mean(tDF$BMI))

}
> by(dfFatKids2[,c("Height","Weight")],dfFatKids2$Gender,fBMI)
dfFatKids2$Gender: F
[1] 140.8881
------------------------------------------------------------
dfFatKids2$Gender: M
[1] 208.3868
>
```

Take note that a data frame is passed to our function by the `by()` function! Note that you didn't have to specify the columns and could have passed the entire data frame to the fBMI function!

Creating Your Own Functions in R

You can use the `missing()` function to determine if the value of a specific parameter is missing and then take appropriate action:

```
fBMI <- function(pDFHW) {

 if ( missing(pDFHW) ) {
  cat("Missing parameter!\n")
  return(NA)
 }

 tDF <- pDFHW
 tDF$BMI <- 703*tDF$Weight/tDF$Height^2
 return(mean(tDF$BMI))

}
> fBMI()
Missing parameter!
[1] NA
>
```

As you can see in the example above, I used the `cat()` function instead of the `print()` function to print output to the screen. The problem with `print()` is that, if you are assigning the results of a function to a variable, the output from `print()` has this tendency to look like results output:

Creating Your Own Functions in R


```
> fMyFunc <- function(x) {

  print(x)
  return(x+1)

 }
> fMyFunc(5)
[1] 5
[1] 6
>
```

As you can see above, the 5 is the output from the `print()` function! Be aware, though, that the actual output from the `return` is what is placed in any variable assignment:

```
> fMyFunc <- function(x) {

  print(x)
  return(x+1)

 }
> z <- fMyFunc(5)
[1] 5
> z
[1] 6
>
```

The `[1] 5` above is due to the `print()` function. Note that z is only assigned 6!

Creating Your Own Functions in R

Now, you can prevent this print issue from occurring by using the `cat()` function instead within your function. Note that `cat()` has no assignable output (NULL) and only prints to the screen:

```
> fMyFunc <- function(x) {

   cat("The input parameter, x, is set to",x,"\n")
   return(x+1)

 }
> fMyFunc(5)
The input parameter, x, is set to 5
[1] 6
>
```

The `\n` tells R to output a carriage return and line feed. Without the `\n`, anything printed next will appear on the same line as your message:

```
> fMyFunc <- function(x) {

   cat("The input parameter, x, is set to",x)
   return(x+1)

 }
> fMyFunc(5)
The input parameter, x, is set to 5[1] 6
>
```

Creating Your Own Functions in R

Note that `\n` is called a *character constant*. There are several more character constants, as shown below:

```
\n newline
\r carriage return
\t tab
\b backspace
\a alert (bell)
\f form feed
\v vertical tab
\\ backslash \
\nnn character with given octal code (1, 2 or 3 digits)
\xnn character with given hex code (1 or 2 hex digits)
```

You can prevent a function of yours from printing anything out if the user does not assign the results to a variable. The function you use is `invisible(var)`:

```
>  fMyFunc <- function(x) {

    y <- x+1
    invisible(y)

  }
> fMyFunc(5)              ← Notice that nothing printed out!!
> z<-fMyFunc(5)           ← But z is assigned the correct value!!
> z
[1] 6
>
```

Creating Your Own Functions in R

Along with the `return()` function, you can halt the execution a function by using the `stop()` function. The first parameter is zero or more objects which can be coerced to character and are pasted together and printed out:

```
> fMyFunc <- function(x) {

  if (x==5) {
   stop("ERROR: You provided the evil number=",x)
  }

  return(x)
 }
> fMyFunc(4)
[1] 4
> fMyFunc(5)
Error in fMyFunc(5) : ERROR: You provided the evil number=5
>
```

Another nice function is `stopifnot()` which does the same thing as `stop()` only if one or more of the expressions passed to it is **not** TRUE. That is, FALSEs cause a stop:

```
> v <- c(1,2,3)
> w <- c(1,2,3)
> stopifnot(length(v)==length(w))
>
> v <- c(1,2,3)
> w <- c(1,2,3,4)
> stopifnot(length(v)==length(w))
Error: length(v) == length(w) is not TRUE
>
```

Creating Your Own Functions in R

Finally, you can see the code behind many of R's functions by typing the name of the function at the command line without the parentheses or arguments:

```
> rank
function (x, na.last = TRUE, ties.method = c("average", "first", "random", "max", "min"))
{
    nas <- is.na(x)
    nm <- names(x)
    ties.method <- match.arg(ties.method)
    if (is.factor(x))
        x <- as.integer(x)
    y <- switch(ties.method, average = , min = , max = .Internal(rank(x[!nas],
        ties.method)), first = sort.list(sort.list(x[!nas])),
        random = sort.list(order(x[!nas], stats::runif(sum(!nas)))))
    if (!is.na(na.last) && any(nas)) {
        yy <- integer(length(x))
        storage.mode(yy) <- storage.mode(y)
        yy <- NA
        NAkeep <- (na.last == "keep")
        if (NAkeep || na.last) {
            yy[!nas] <- y
            if (!NAkeep)
                yy[nas] <- (length(y) + 1L):length(yy)
        }
        else {
            len <- sum(nas)
            yy[!nas] <- y + len
            yy[nas] <- 1L:len
        }
        y <- yy
        names(y) <- nm
    }
    else names(y) <- nm[!nas]
    y
}
```

Flow of Control

Flow of Control

Just like SAS, R has flow of control statements such as *if*, *for*/*while*/*repeat* loops, as well as loop control statements such as *break* and *next*. These will come in handy in creating our own functions, but…

…just because you see a for-loop listed above, this does NOT mean that you should avoid *vectorizing computations* by performing the loops yourself. For example, if you want to add one to a vector, we can just execute `vecNew <- vecOld +1` instead of setting up a for-loop and using indexes.

In R, vectorizing computations are much faster than performing loops…I'm just sayin'.

Nevertheless, sometimes you will have to perform loops and the flow of control statements can be used in those situations…I'm just sayin', again.

The IF statement has the following syntax(take note of the curly braces!):

```
if (condition) {
  if_body
}


if (condition) {
 if_body
} else {
 else_body
}
```

Flow of Control

For example, let's print out a message if two numbers are the same:

```
if (1 == 1) {
 print("Hello")
} else {
 print("Goodbye")
}
[1] "Hello"
>

if (1 == 2) {
 print("Hello")
} else {
 print("Goodbye")
}
[1] "Goodbye"
>
```

Now, there's a problem with the code above despite the fact that it's working. Note that we are attempting fall back on our old way of thinking and trying to compare two scalars when R has no scalars. Let's try this example again, but using vectors with only one element:

```
if (c(1) == c(2)) {
 print("Hello")
} else {
 print("Goodbye")
}
[1] "Goodbye"
>
```

Flow of Control

As you see, the code above works fine. Let's try it again with more than two elements:

```
if (c(1,2) == c(1,2)) {
 print("Hello")
} else {
 print("Goodbye")
}
[1] "Hello"
Warning message:
In if (c(1, 2) == c(1, 2)) { :
  the condition has length > 1 and only the first element will be used
>
```

As you see, only the first element is being used for the ==. If you want to test that two vectors (or other data structures) are equal you should use the `identical()` function:

```
if ( identical(c(1,2),c(2,1)) ) {
 print("Hello")
} else {
 print("Goodbye")
}
[1] "Goodbye"

>
```

Flow of Control

Besides testing equality, you can test for inequality by using a ! (meaning NOT) in front of the `identical()` function:

```
if ( !identical(c(1,2),c(2,1)) ) {
 print("Hello")
} else {
 print("Goodbye")
}
[1] "Hello"
>
```

An alternative to the if statement is the `ifelse()` function whose syntax is:

```
ifelse(test_condition,true_code,false_code)
```

For example, let's test if the total number of elements in a vector – returned by using the `length()` function – is exactly 5 or not:

```
> vMyVec <- c(1,2,3,4,5)
> vMyVec
[1] 1 2 3 4 5
> ifelse(length(vMyVec)==5,TRUE,FALSE)
[1] TRUE
> vMyVec <- c(1,2,3,4,5,6)
> vMyVec
[1] 1 2 3 4 5 6
> ifelse(length(vMyVec)==5,TRUE,FALSE)
[1] FALSE
>
```

Flow of Control

The for Loop in R is similar to the DO Loop in SAS. The syntax is as follows:

```
for (var in seq) {
 body
}
```

where

var is a temporary variable

seq is a sequence (created using `seq()` function, the colon operator, a vector, or more)

For example,

```
> vMyVec <- seq(1,10,2)
> for(i in vMyVec) {
  print(i)
 }
[1] 1
[1] 3
[1] 5
[1] 7
[1] 9
>
```

Flow of Control

The while Loop is similar to the WHILE Loop in SAS. Let's create a function that creates the factorial of a given integer argument:

```
> fMyFact <- function(factOf=5) {

  if(factOf<=0) {
   cat("Please provide a number greater than 0\n")
   return(-1) # or invisible(-1)
  }

  n<-factOf
  factorial <- 1
  while(!identical(n,1)) {
   factorial <- factorial*n
   n <- n - 1
  }

  return(factorial)

 }
> fMyFact(5)
[1] 120
> fMyFact(4)
[1] 24
> fMyFact(3)
[1] 6
> fMyFact(-10)
Please provide a number greater than 0
[1] -1
>
```

Flow of Control

The `repeat` statement will repeat a block of code until it encounters the `break` statement; otherwise, it will repeat infinitely. Let's repeat the factorial function above again, but let's use a `repeat` instead of a `while`:

```
> fMyFact <- function(factOf=5) {

  if(factOf<=0) {
   cat("Please provide a number greater than 0\n")
   return(-1)
  }

  n<-factOf
  factorial <- 1
  repeat {
   factorial <- factorial*n
   n <- n - 1
   if(identical(n,1)) break
  }

  return(factorial)

 }
> fMyFact(10)
[1] 3628800
>
```

Flow of Control

The `next` statement forces a loop back to the top of the loop avoiding code below it. For example,

```
> for(i in 1:10) {

  if(i==5) next

  print(i)

 }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
>
```

[Note that, for some reason, `identical()` does not work in this case! I'll update this slide when I figure out what's going on!!!]

Function Round-Up #3

String Manipulation

Function Round-Up #3 – String Manipulation

R has many functions used to manipulate text strings. In this section, we outline these functions.

To change a vector to a comma-delimited string, use the `toString()` function:

```
> toString(dfFatKids$FirstName)
[1] "ALBERT, ROSEMARY, TOMMY, BUDDY, FARQUAR, SIMON, LAUREN"
>
```

To find a substring of a string, use the `substr(text,first,last)` function:

```
> substr("ABCDEF",2,5)
[1] "BCDE"
>
```

Just like with SAS's `substr()` function, you can replace the text:

```
> z <- c("ABCDEF")
> substr(z,2,5) <- "CATS"
> z
[1] "ACATSF"
>
```

Function Round-Up #3 – String Manipulation

To trim a string to a certain length of characters, you can use the `strtrim()` function. Note that unlike SAS's TRIM() function, the `strtrim()` function does **not** remove leading and trailing blanks. This function is similar to the `substr()` function, but always starts at 1. Below, we get back the first two characters from the string:

```
> strtrim("ABCDEF",2)
[1] "AB"
>
```

Given a vector of character strings, you can use the `strsplit()` function to parse these character strings into individual substrings. Note that the return type of this function is a **list**. The first list element is associated with the first character string, etc. Within the first list element, you have an array of substrings:

```
> mydata <- c("123:ABC:456:DEF","XX:YY:ZZ:AA")
> s <- strsplit(mydata,":")
> s
[[1]]
[1] "123" "ABC" "456" "DEF"

[[2]]
[1] "XX" "YY" "ZZ" "AA"

> length(s)
[1] 2
```

Function Round-Up #3 – String Manipulation

You can place single or double quotes around your strings by using the `sQuote()` or `dQuote()` functions, resp. For example,

```
> v <- c("ABC","DEF")
> w1 <- sQuote(v)
> w1
[1] "‘ABC’" "‘DEF’"
> w2 <- dQuote(v)
> w2
[1] "“ABC”" "“DEF”"
>
```

Don't be confused with the double quotes surrounding the quoted text strings! R places double-quotes around text for printing purposes. Now, you will note that the single and double quotes shown above are the fancy upper-class quotes: ‘ABC’ and “ABC”. In order to get the low-class quotes, change the `useFancyQuotes` option from TRUE to FALSE: `options(useFancyQuotes=FALSE)`:

```
> v <- c("ABC","DEF")
> w1 <- sQuote(v)
> w1
[1] "'ABC'" "'DEF'"
> w2 <- dQuote(v)
> w2
[1] "\"ABC\"" "\"DEF\""
>
```

Function Round-Up #3 – String Manipulation

Note the backslash character before the double-quotes above. This backslash character is called an escape character and it protects the character to its right from being mis-interpreted by R.

If you want to append the elements from one vector together with another vector, you can use the `paste()` function. The first element of the first vector is concatenated with the first element of the second vector, and so on, as in these examples:

```
> v <- c("ABC","DEF")
> w <- c(123,456,789)
> paste(v,w)
[1] "ABC 123" "DEF 456" "ABC 789"
>
> v <- c("ABC","DEF")
> w <- c(123,456,789)
> paste(v,w,sep="-")
[1] "ABC-123" "DEF-456" "ABC-789"
>
>
```

Take note that the smaller vector is being recycled above! Also, the optional parameter `sep=" "` allows you to specify a separator character, blank by default, as shown in the second example above. An optional third argument is `collapse=` which allows you to further concatenate everything together in `toString()` fashion. By default, collapse is NULL, indicating that the strings remain separate.

Function Round-Up #3 – String Manipulation

For example,

```
> v <- c("ABC","DEF")
> w <- c(123,456,789)
> paste(v,w,sep="-",collapse=":")
[1] "ABC-123:DEF-456:ABC-789"
>
```

To concatenate everything together, like `toString()`, only with a dash, say, do this:

```
> paste(v,w,sep="-",collapse="-")
[1] "ABC-123-DEF-456-ABC-789"
>
```

If you want to know how many characters are in each string in a vector, then use the `nchar()` function:

```
> A <- c("A","AB","ABC")
> nchar(A)
[1] 1 2 3
>
```

If your goal is to determine if a string in a vector is empty or not, use the `nzchar()` function instead. The result is a vector of logical elements of TRUEs (if non-empty string), or FALSE (if empty string).

Function Round-Up #3 – String Manipulation

For example,

```
> Z <- c("A","","ABC","B")
> nzchar(Z)
[1]  TRUE FALSE  TRUE  TRUE
>
```

Similar to SAS's `index()` function, R has the `match()` function. Given a vector of values in the first parameter and a lookup vector in the second parameter, `match()` will return either the number of the element in the first vector's parameter if found in the second vector, or a NA:

```
> match(c("A","B","C"),c("C","A"))
[1]  2 NA  1
>
```

Note that "A" in the first vector is the second element (2) in the lookup vector. "B" does not exist in the lookup vector. And "C" is in the first position (1).

Now, whereas the `match()` function attempts to find an exact match within the lookup vector, the `pmatch()` function attempts to find a partial match. This is the difference between comparing "APPLES" against the lookup vector "APPLES","ORANGES" and getting back 1 since "APPLES" clearly matches "APPLES" versus comparing "APP" against the lookup vector and still getting a 1 since "APP" *partially* matches "APPLES". If more than one match, then NA returned.

Function Round-Up #3 – String Manipulation

For example,

```
> vMyData <- c("APP","OR","YUM","P")
> vMyLookup <- c("APPLES","ORANGES","PEARS","ORANGUTAN")
> match(vMyData,vMyLookup)
[1] NA NA NA NA
> pmatch(vMyData,vMyLookup)
[1]  1 NA NA  3
>
```

In a similar way that `pmatch()` matches partially, the function `char.expand()` matches partially, but then goes one step further by returning the match found in the lookup table:

```
> vMyLookup <- c("APPLES","ORANGES","PEARS","ORANGUTAN")
> char.expand("OR",vMyLookup)
character(0)
> char.expand("AP",vMyLookup)
[1] "APPLES"
>
```

Note that the first parameter to `char.expand()` must be a single character string.

And in a similar way that `match()` returns an integer indicating the element found, the function `charmatch()` does partial matching similar to `pmatch()` but returns the matched value within the lookup vector:

Function Round-Up #3 – String Manipulation

For example,

```
> vMyLookup <- c("APPLES","ORANGES","PEARS","ORANGUTAN")
> charmatch("OR",vMyLookup)
[1] 0
> charmatch("AP",vMyLookup)
[1] 1
>
```

You can upper case your characters by using the `toupper()` function and you can lowercase by using `tolower()`.

Similar to SAS's `translate()` function, R provides the `chartr()` function. Here is the syntax:

```
chartr(old, new, x)
```

where

old is a string of characters to look up

new is a string of characters to change to

x is the string to modify

For example,

Function Round-Up #3 – String Manipulation


```
> vMyVec <- c("Open the door!")
> chartr("Othd","Udiz",vMyVec)
[1] "Upen die zoor!"
>
```

Note that there is a one-to-one match between the old and the new vectors.

You are all familiar with the asterisk (*) as a wildcard, as in dir *.txt. You are also familiar with the % and _ wildcard in SQL's LIKE statement. In R, there is a methodology to search with more specificity than the asterisk, etc., allow. This methodology is called *regular expressions*. You can find out more information on regular expressions in R by entering in `?regex` at the RGui command line. While we don't have time to go over regular expressions in this lecture, be aware that they exist. You can find a lot of information on the Interweb on this topic. To whet your appetite, let's say that you have a vector containing addresses:

```
> vAddr <- c("123 Main St",
             "1476 Charleston Rd Suite 702",
             "9845 Voledemort Road Apt 222",
             "1476 Simple St Suite 702 Apt 107")
```

Now, let's use regular expressions to see if we can match each one of these patterns:

Function Round-Up #3 – String Manipulation


```
> vAddr <- c("123 Main St","1476 Charleston Rd Suite 702","9845 Voledemort Road Apt 222","1476 Simple Road Suite 702 Apt 107")
> vAddr
[1] "123 Main St" "1476 Charleston Rd Suite 702" "9845 Voledemort Road Apt 222" "1476 Simple Road Suite 702 Apt 107"
>
> # Match 123 MAIN STREET SUITE 123 APT 456
> re0 <- "^([0-9]+) ([a-zA-Z]+) (St|Street|Rd|Road)+ (Suite|Suit)+ ([0-9]+) (Apartment|Apt){1} ([0-9]+)$"
> bRC0 <- grepl(re0,vAddr)
> bRC0
[1] FALSE FALSE FALSE  TRUE
>
> # Match 123 MAIN STREET SUITE 123
> re1 <- "^([0-9]+) ([a-zA-Z]+) (St|Str|Street|Rd|Road)+ (Suite|Ste)+ ([0-9]+)$"
> bRC1 <- grepl(re1,vAddr)
> bRC1
[1] FALSE  TRUE FALSE FALSE
>
> # Match 123 MAIN STREET APT 123
> re2 <- "^([0-9]+) ([a-zA-Z]+) (St|Str|Street|Rd|Road)+ (Apartment|Apt)+ ([0-9]+)$"
> bRC2 <- grepl(re2,vAddr)
> bRC2
[1] FALSE FALSE  TRUE FALSE
>
> # Match 123 MAIN STREET
> re3 <- "^([0-9]+) ([a-zA-Z]+) (St|Str|Street|Rd|Road)+$"
> bRC3 <- grepl(re3,vAddr)
> bRC3
[1]  TRUE FALSE FALSE FALSE
>
```

Although the additional parentheses around each address element are not necessary when using the `grepl()` function – which returns TRUE for the array element that matches the regular expression, FALSE otherwise – the next function we will look at does need it.

So, let's now break out each of the addresses into its constituent parts:

Function Round-Up #3 – String Manipulation


```
> # Match 123 MAIN STREET SUITE 123 APT 456
> re0 <- "^([0-9]+) ([a-zA-Z]+) (St|Street|Rd|Road)+ (Suite|Suit)+ ([0-9]+) (Apartment|Apt){1} ([0-9]+)$"
> bRC0 <- grepl(re0,vAddr)
> bRC0
[1] FALSE FALSE FALSE  TRUE
> sub(re0,"\\1",vAddr)
[1] "123 Main St"                 "1476 Charleston Rd Suite 702" "9845 Voledemort Road Apt 222" "1476"
> sub(re0,"\\2",vAddr)
[1] "123 Main St"                 "1476 Charleston Rd Suite 702" "9845 Voledemort Road Apt 222" "Simple"
> sub(re0,"\\3",vAddr)
[1] "123 Main St"                 "1476 Charleston Rd Suite 702" "9845 Voledemort Road Apt 222" "Road"
> sub(re0,"\\4",vAddr)
[1] "123 Main St"                 "1476 Charleston Rd Suite 702" "9845 Voledemort Road Apt 222" "Suite"
> sub(re0,"\\5",vAddr)
[1] "123 Main St"                 "1476 Charleston Rd Suite 702" "9845 Voledemort Road Apt 222" "702"
> sub(re0,"\\6",vAddr)
[1] "123 Main St"                 "1476 Charleston Rd Suite 702" "9845 Voledemort Road Apt 222" "Apt"
> sub(re0,"\\7",vAddr)
[1] "123 Main St"                 "1476 Charleston Rd Suite 702" "9845 Voledemort Road Apt 222" "107"
>
```

Notice above that the `sub()` function is taking the regular expression we are passing to it and returning the first (\\1), the second (\\2), etc. component of the address. Notice that the fourth element of vAddr matched the regular expression since TRUE was returned in that case. Obviously, more work needs to be done to generate an address cleaning function, but I think you see the power of regular expressions. We talk more about regular expressions in the Advanced R lecture. The following is a list of the regular expressions functions and you can find more at `?grep`:

Function Round-Up #3 – String Manipulation

 # Searching Functions

grep(pattern, x, ignore.case = FALSE, extended = TRUE, perl = FALSE, value = FALSE, fixed = FALSE, useBytes = FALSE, invert = FALSE) –

grep(value = FALSE) returns an integer vector of the indices of the elements of x that yielded a match (or not, for invert = TRUE).

grep(value = TRUE) returns a character vector containing the selected elements of x

grepl(pattern, x, ignore.case = FALSE, extended = TRUE, perl = FALSE, fixed = FALSE, useBytes = FALSE) - grepl returns a logical vector (match or not for each element of x).

 # Substitution Functions (The two *sub functions differ only in that sub replaces only the first occurrence of a pattern whereas gsub replaces all occurrences.)

sub(pattern, replacement, x, ignore.case = FALSE, extended = TRUE, perl = FALSE, fixed = FALSE, useBytes = FALSE) - For sub and gsub return a character vector of the same length and with the same attributes as x (after possible coercion).

gsub(pattern, replacement, x, ignore.case = FALSE, extended = TRUE, perl = FALSE, fixed = FALSE, useBytes = FALSE) - For sub and gsub return a character vector of the same length and with the same attributes as x (after possible coercion).

 # Starting Position Match

regexpr(pattern, text, ignore.case = FALSE, extended = TRUE, perl = FALSE, fixed = FALSE, useBytes = FALSE) - returns an integer vector of the same length as text giving the starting position of the first match or *-1* if there is none, with attribute "match.length", an integer vector giving the length of the matched text (or *-1* for no match). The match positions and lengths are in characters unless useBytes = TRUE is used, when they are in bytes.

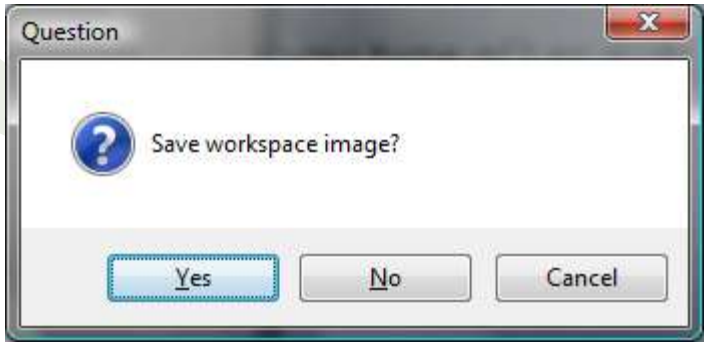
gregexpr(pattern, text, ignore.case = FALSE, extended = TRUE, perl = FALSE, fixed = FALSE, useBytes = FALSE) - returns a list of the same length as text each element of which is of the same form as the return value for regexpr, except that the starting positions of every (disjoint) match are given.

Saving and Retrieving Your R Workspace

Saving and Retrieving Your R Workspace

When you close a SAS session, all of your work – temporary datasets, macro variables, user-defined formats, etc. – is deleted. Whenever you close your R session, you are asked whether you want to save your workspace. In R, you can save all of your data frames, vectors, matrices, etc. in one file called a *workspace image*. Later on, you can load in that workspace image and be right back where you left off!

To save a workspace image, when you type `q()` or `quit()` at the command line to end the R Session, you will be presented with a dialog box, or query text, asking if you would like to save your workspace image:



If you click Yes, then R will save the workspace image, but not give you a choice as where to save it and what to name the file. By default, R saves the workspace image in a file called .RData in a folder dependent on the operating system.

To override the default location of this file, you use the `save.image()` function.

Saving and Retrieving Your R Workspace

The syntax for the `save.image()` function is:

```
save.image(file=".RData")
```

where

file – is the directory and name of the file to save (defaults to .RData)

For example,

```
> save.image("C:\\TEMP\\MyRImage.RData")
```

As mentioned towards the beginning of the lecture, you can use the `q()` or `quit()` function to end an R session. What I did not mention was that both `q()` and `quit()` take a character string as a parameter. If you want to end an R session, type `q("no")` and no workspace image will be saved. Type `q("yes")` if you want a workspace image to be saved. Note again that this method does not allow you to specify a directory and filename like the `save()` function.

Now, if you did use `q("yes")`, the next time you start R, your .RData workspace image will be loaded by default (assuming one exists). You will receive the following message on RGui:

```
[Previously saved workspace restored]
```

To load another workspace image, you use the `load()` function.

Saving and Retrieving Your R Workspace

The syntax for the `load()` function is:

```
load(filename_to_load)
```

For example,

```
> load("C:\\TEMP\\MyRImage.RData")
```

Note that if you capture the output of `load()` or place parentheses around it, you will be shown all of the variables that were loaded in:

```
> (load("c:\\temp\\MyRImage.RData"))
 [1] ".Random.seed" "df" "dfFatKids" "dfFatKids2" "dfFatKids3" "f" "fBMI" "fGender"
     "fMyFunc" "subdf" "v"
[12] "w" "y" "Z"
>
```

As you see above, there are 14 previously created items loaded into my workspace.

Now, as mentioned before, you can use the `rm()` function to remove an object you created in your current R session. But, you can also use `rm()` to remove **all** of your objects (please be careful with this!!):

```
> rm(list=ls())
```

R and Database Interaction

R and Database Interaction

There are several ways to interact with an Oracle, SQL Server, etc. database from within R: RODBC or ROracle.

RODBC uses an ODBC connection setup using the Data Sources (ODBC) application in Windows Control Panel. On a Unix/Linux machine, you have to use unixODBC (or similar) piece of software to set up your ODBC data source.

ROracle uses the Oracle connection software to talk directly with the Oracle database. This does not require you to set up a data source as you do when using RODBC.

Note that my attempts to compile RODBC on a 64-bit machine failed due to a 64-bit issue. This may change in the future, so don't give up on it. But, note that, in general, ROracle should pull data back from the database faster than RODBC.

Here is an example of how to use RODBC:

```
library(RODBC)
odbcDataSources()
ch <- odbcConnect("oracle_path","userid","password")
dfResults <- sqlQuery(ch,"SQL_query")
odbcClose(ch)
```

R and Database Interaction

Here is an example of how to use ROracle:

```
library(ROracle)
m <- dbDriver("Oracle")
con <- dbConnect(m,username="userid",password="password",dbname="oracle_path")
rs <- dbSendQuery(con,"SQL_query")
dfResults <- fetch(rs,n = -1)
oraCloseConnection(con)
oraCloseDriver(m)
```

Please review the documentation for these two packages.

Function Round-Up #4

Sorting and Ranking

Function Round-Up #4 – Sorting and Ranking

Recall that SAS has the PROC SORT procedure to sort a dataset. R is no different, but there are several methods you can use.

First, to reverse the elements of a vector, you can use the `rev()` function:

```
> vAnimals <- c("CAT","ANT","ZEBRA","DOG","DOG")
> vAnimals
[1] "CAT"   "ANT"   "ZEBRA" "DOG"   "DOG"
> rev(vAnimals)
[1] "DOG"   "DOG"   "ZEBRA" "ANT"   "CAT"
>
```

The `order()` function, rather than actually sorting the data, returns a permutation that indicates the order of the data. For example, given the vAnimals vector above, here is what `order()` returns:

```
> vAnimals <- c("CAT","ANT","ZEBRA","DOG","DOG")
> order(vAnimals)
[1] 2 1 4 5 3
>
```

Note that the 2 indicates that ANT comes first. The 1 indicates that CAT follows. The 4 and 5 indicate that DOG and DOG come next. Then the 3 indicates that ZEBRA ends the lineup. To use `order()`, place it in the vector's index:

Function Round-Up #4 – Sorting and Ranking


```
> vAnimals <- c("CAT","ANT","ZEBRA","DOG","DOG")
> vAnimalsORDER <- order(vAnimals)
> vAnimals[vAnimalsORDER]
[1] "ANT"   "CAT"   "DOG"   "DOG"   "ZEBRA"
>
```

You can provide more than one vector to the `order()` function, but each vector must be the same length and the sort order is taken over the same element for all vectors to determine the sort. For example, given these two vectors:

```
> v1 <- c(3,1,1,2)
> v2 <- c(9,7,6,8)
> order(v1,v2)
[1] 3 2 4 1
```

Note that this result doesn't make sense until you see that data slapped together using the `cbind()` function:

```
> cbind(v1,v2)
     v1 v2
[1,]  3  9
[2,]  1  7
[3,]  1  6
[4,]  2  8
```

Row (1,6) is first which is why `order()` produced a 3 first. Row (1,7) is next, which is why 2 follows. And so on. To sort descending, place a minus sign in front of the variable(s):

Function Round-Up #4 – Sorting and Ranking


```
> order(v1,-v2)
[1] 2 3 4 1
> cbind(v1,v2)
     v1 v2
[1,]  3  9
[2,]  1  7
[3,]  1  6
[4,]  2  8
>
```

As you can see, rows (1,6) and (1,7) have flipped positions. Now, you can also use the results of `order()` to sort a data frame. Below, I sort the Fat Kids data by Height and Weight:

```
> dfFatKidsORDER <- order(dfFatKids$Height,dfFatKids$Weight)
> dfFatKids[dfFatKidsORDER, ]
  FirstName Height Weight FattyIndex
4     BUDDY     12    189    15.7500
2  ROSEMARY     35    123     3.5143
1    ALBERT     45    150     3.3333
7    LAUREN     54    876    16.2222
5   FARQUAR     76    198     2.6053
3     TOMMY     78    167     2.1410
6     SIMON     87    256     2.9425
>
```

Now, `order()` returns the permutation of the sort order. The `sort()` function returns the data sorted…no permutations to deal with. But, `sort()` only accepts one variable! Note: `sort(x)` is the same as `x[order(x)]`.

Function Round-Up #4 – Sorting and Ranking

To rank a vector, you can use either the `rank()` function which lets you choose among a list of ties options, or the `xtfrm()` function which is the same as the `rank()` function using the "min" ties method.

Let's rank a vector using the `xtfrm()` function:

```
> v <- c(5,7,8,9,3,4,5,7,4,2)
> xtfrm(v)
 [1]  5  7  9 10  2  3  5  7  3  1
>
```

Notice that 3's and 5's repeat indicating tied ranks. Also, note that 4's and 6's do not appear since there were ties. Using the `rank()` function:

```
> v <- c(5,7,8,9,3,4,5,7,4,2)
> rank(v)
 [1]  5.5  7.5  9.0 10.0  2.0  3.5  5.5  7.5  3.5  1.0
>
```

Notice that the tied ranks are replaced, by default, with the average of the tied ranks. With the `rank()` function, you can choose from among several ties methods. Here is the syntax for the `rank()` function:

Function Round-Up #4 – Sorting and Ranking


```
rank(x, na.last = TRUE, ties.method = c("average", "first", "random", "max", "min"))
```

where

```
x - a numeric, complex, character or logical vector.
na.last - controls the treatment of  NAs.
          If TRUE, missing values in the data are put last;
          if FALSE, they are put first;
          if NA, they are removed;
          if "keep" they are kept with rank NA.
ties.method - a character string specifying how ties are treated.  With some values equal
           (called 'ties'), the argument ties.method determines the result at the
           corresponding indices:
                    The "first" method results in a permutation with increasing values
                     at each index set of ties.
                    The "random" method puts these in random order
                    The "average" (default) method replaces them by their mean,
                    The "max" and "min" replaces them by their maximum and minimum
                     respectively, the latter being the typical sports ranking.
```

The `xtfrm()` function is equivalent to

```
rank(x, ties.method="min", na.last="keep")
```

Similar to PROC SORT NODUPKEY, you can return a unique list of elements by using the `unique()` function:

Function Round-Up #4 – Sorting and Ranking


```
> w <- c(1,1,2,2,3,3,4,5,6,6)
> w
 [1] 1 1 2 2 3 3 4 5 6 6
> unique(w)
[1] 1 2 3 4 5 6
```

Note that this also works on data frames:

```
> unique(dfFatKids2[,c("Gender","Category")])
  Gender Category
1      M        A
2      F        D
4      M        B
6      M        C
>
```

The `do.call()` Function

The `do.call()` Function

The `do.call()` function allows you to call a function by providing it the name of a function as well as the parameters to that function and it will construct the function call for you and call it. That is, you don't have to call the function yourself.

For example, recall that we learned about the `by()` function. This performs a SAS PROC MEANS-like idea and returns several rows of data. Let's use the Tukey `fivenum()` function to produce the minimum, lower hinge, median, upper hinge and maximum values. This function returns a vector of just five numbers. For example,

```
> w <- c(10,15,25,34,56,99)
> fivenum(w)
[1] 10.0 15.0 29.5 56.0 99.0
>
```

Now, let's use the `by()` function using `fivenum()` to produce these statistics for each Gender on the Weight:

```
> (z<-by(dfFatKids2[,"Weight"],dfFatKids2$Gender,fivenum))
dfFatKids2$Gender: F
[1] 123.0 123.0 499.5 876.0 876.0
------------------------------------------------------------------------------------------
dfFatKids2$Gender: M
[1] 150 167 189 198 256
>
```

The `do.call()` Function

Now, let's say that we want a data frame with these values in it. Unfortunately, due to the nature of the `by()` function, we cannot use `data.frame()` or `as.data.frame()` to produce a data frame from this data. Recall that we learned about the `rbind()` function which takes a series of vectors (or matrices or data frames) and appends them together. Let's see what happens when we use the `rbind()` function:

```
> (z<-by(dfFatKids2[,"Weight"],dfFatKids2$Gender,fivenum))
dfFatKids2$Gender: F
[1] 123.0 123.0 499.5 876.0 876.0
------------------------------------------------------------
dfFatKids2$Gender: M
[1] 150 167 189 198 256
> rbind(z)
  F         M
z Numeric,5 Numeric,5
>
```

Clearly, this is not what we want. The `do.call()` function can help us out in this case:

```
> do.call(rbind,z)
  [,1] [,2]  [,3] [,4] [,5]
F  123  123 499.5  876  876
M  150  167 189.0  198  256
>
```

This returns a matrix.

The `do.call()` Function

As you see, `do.call()`'s first parameter is the function you want to run and the second parameter is the data you want to pass to the function named in the first argument. Now, to create a data frame, we can use the `as.data.frame()` function around it:

```
> dfZ <- as.data.frame(do.call(rbind,z))
> dfZ
   V1  V2    V3  V4  V5
F 123 123 499.5 876 876
M 150 167 189.0 198 256
>
```

Now, let's rename those column names to something more appropriate using the `names()` function:

```
> names(dfZ) <- c("MIN","LOWER_HINGE","MEDIAN","UPPER_HINGE","MAX")
> dfZ
  MIN LOWER_HINGE MEDIAN UPPER_HINGE MAX
F 123         123  499.5         876 876
M 150         167  189.0         198 256
>
```

But, what's the deal with the M's and F's in an unnamed column? That column is actually the *row name* and you've been staring at it for the entire presentation! We won't talk about row names, but see the documentation for the functions `rownames()` and `colnames()`.

The `do.call()` Function

As another example of the `do.call()` function, let's use the `paste()` function to concatenate all of the variables in a data frame together:

```
> tmp <- expand.grid(letters[1:2], 1:3, c("+", "-"))
> tmp
   Var1 Var2 Var3
1     a    1    +
2     b    1    +
3     a    2    +
4     b    2    +
5     a    3    +
6     b    3    +
7     a    1    -
8     b    1    -
9     a    2    -
10    b    2    -
11    a    3    -
12    b    3    -
```

One method is to use the `paste()` function like this:

```
> paste(tmp$Var1,tmp$Var2,tmp$Var3,sep="")
 [1] "a1+" "b1+" "a2+" "b2+" "a3+" "b3+" "a1-" "b1-" "a2-" "b2-" "a3-" "b3-"
>
```

but this requires that we list all of the variables out. We can use the `do.call()` function to save us from typing all of those variables out:

The `do.call()` Function


```
> do.call("paste", c(tmp, sep=""))
 [1] "a1+" "b1+" "a2+" "b2+" "a3+" "b3+" "a1-" "b1-" "a2-" "b2-" "a3-" "b3-"
>
```

As you see, using the `do.call()` function, we only have to pass the name of the data frame and the separator we want to use in order to easily paste all of the text together on a row-by-row basis.

Function Round-Up #5

The Apply Series

Function Round-Up #5 – The Apply Series

The Apply series of functions – namely, `apply()`, `eapply()`, `lapply()`, `mapply()`, `rapply()`, `sapply()` and `tapply()` – apply a function over a specified set of data (vector, array, list, matrix or data frame).

Before detailing each function, let's show a simple example. Given the fat kids data, let's use `apply()` to sum down the Height and Weight columns of the data:

```
> apply(dfFatKids[,c("Height","Weight")],2,sum)
Height Weight
   387   1959
>
```

The first parameter is the data frame along with the appropriate columns we want to keep; the second parameter, 2, means "apply the function down the columns", and 1 means "apply the function across the rows"; the third parameter is the name of the function to apply to the data. If we used a 1 instead of 2, this is what we would get (the sum of the height and weight within row):

```
> apply(dfFatKids[,c("Height","Weight")],1,sum)
[1] 195 158 245 201 274 343 930
>
```

Let's try that again with the `mean()` function:

Function Round-Up #5 – The Apply Series


```
> apply(dfFatKids[,c("Height","Weight")],1,mean)
[1]  97.5  79.0 122.5 100.5 137.0 171.5 465.0
> apply(dfFatKids[,c("Height","Weight")],2,mean)
   Height    Weight
 55.28571 279.85714
>
```

Note that you can use the functions `colSums()`, `rowSums()`, `colMeans()` and `rowMeans()` instead of the methods shown. These four functions actually use `apply()` behind the scenes.

Note that just using the `mean()` or `sum()` functions instead of `apply()` will compute the sum or mean across all of the data and not just rows or columns!

As you could have guessed, the `apply()` function takes an array (or something that can be coerced into any array, like a data frame) and applies the given function across the rows or down the columns.

The remaining functions are more specific versions of the `apply()`. The `lapply()` function works on lists and returns a list:

```
> lMyList <- list(a=1:10,b=10:15,c=100:105)
> lMyList
$a
 [1]  1  2  3  4  5  6  7  8  9 10
$b
[1] 10 11 12 13 14 15
$c
[1] 100 101 102 103 104 105
```

Function Round-Up #5 – The Apply Series


```
> lapply(lMyList,sum)
$a
[1] 55
$b
[1] 75
$c
[1] 615
>
```

Note that `lapply()` summed the elements within each list element…not across. The `sapply()` function is similar to `lapply()` except that it attempts to create a vectors or matrix instead of a list:

```
> sapply(lMyList,sum)
  a   b   c
 55  75 615
>
```

Note that both the `lapply()` and `sapply()` functions do not specify a margin like the `apply()` function.

The `mapply()` function applies the function for the first elements of the listed data, then applied the function to the second elements of the listed data, etc.:

Function Round-Up #5 – The Apply Series


```
> vV1 <- 1:10
> vV1
 [1]  1  2  3  4  5  6  7  8  9 10
> vV2 <- 11:20
> vV2
 [1] 11 12 13 14 15 16 17 18 19 20
> vV3 <- 21:30
> vV3
 [1] 21 22 23 24 25 26 27 28 29 30
> mapply(sum,vV1,vV2,vV3)
 [1] 33 36 39 42 45 48 51 54 57 60
>
```

As you see above, 33 is the sum of 1+11+21, etc.

The remaining functions are esoteric, so I'll leave it up to you to read about those!

Matrix Operations

Matrix Operations

Along with all of the stuff I've just showed you, R comes with a built-in matrix language similar to SAS/IML. There are a lot more matrix functions built-in to the `Matrix` package as well as several other packages. We will focus just on the base package, so check out the other packages at CRAN.

Given the following two matrices,

```
mMyMat1 <- matrix(c(1,3,5,7, 2,4,6,8, 1,11,13,19, 15,13,12,13),nrow=4,ncol=4,byrow=TRUE)
> print(mMyMat1)
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
[3,]    1   11   13   19
[4,]   15   13   12   13
>

mMyMat2 <- matrix(c(1,2,3,4, 5,6,7,8, 9,10,11,12, 13,14,15,16),nrow=4,ncol=4,byrow=TRUE)
> print(mMyMat2)
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
[4,]   13   14   15   16
>
```

You can perform element-wise addition, subtraction, multiplication and division by using the standard arithmetic operators:

Matrix Operations


```
> mMyMat1 + mMyMat2
     [,1] [,2] [,3] [,4]
[1,]    2    5    8   11
[2,]    7   10   13   16
[3,]   10   21   24   31
[4,]   28   27   27   29
> mMyMat1 - mMyMat2
     [,1] [,2] [,3] [,4]
[1,]    0    1    2    3
[2,]   -3   -2   -1    0
[3,]   -8    1    2    7
[4,]    2   -1   -3   -3
> mMyMat1 * mMyMat2
     [,1] [,2] [,3] [,4]
[1,]    1    6   15   28
[2,]   10   24   42   64
[3,]    9  110  143  228
[4,]  195  182  180  208
> mMyMat1 / mMyMat2
          [,1]      [,2]      [,3]     [,4]
[1,] 1.0000000 1.5000000 1.6666667 1.750000
[2,] 0.4000000 0.6666667 0.8571429 1.000000
[3,] 0.1111111 1.1000000 1.1818182 1.583333
[4,] 1.1538462 0.9285714 0.8000000 0.812500
>
```

Note again that these operations are element-wise! To perform standard matrix multiplication, you use the %*% operator :

Matrix Operations


```
> mMyMat1 %*% mMyMat2
     [,1] [,2] [,3] [,4]
[1,]  152  168  184  200
[2,]  180  200  220  240
[3,]  420  464  508  552
[4,]  357  410  463  516
>
```

To find the inverse of a matrix, you can use the `qr.solve()` function:

```
> mMyMat1_INV = qr.solve(mMyMat1)
> mMyMat1_INV
      [,1] [,2]  [,3]          [,4]
[1,]  2.30 -2.1 -0.10  2.000000e-01
[2,] -1.25  0.5  0.25 -8.984729e-16
[3,] -8.40  8.8 -0.20 -6.000000e-01
[4,]  6.35 -6.2  0.05  4.000000e-01
>
```

To check that the inverse worked, you can multiply this inverse against the original matrix and you should be back the identity matrix:

```
> mMyMat1 %*% mMyMat1_INV
              [,1]          [,2]          [,3]         [,4]
[1,] 1.000000e+00 -2.131628e-14 -3.677614e-16 1.110223e-15
[2,] 2.131628e-14  1.000000e+00 -5.551115e-17 1.332268e-15
[3,] 2.042810e-14 -1.421085e-14  1.000000e+00 6.661338e-16
[4,] 7.993606e-15  0.000000e+00 -1.040834e-16 1.000000e+00
>
```

Matrix Operations

To find the determinant of a matrix, use the `det()` function:

```
> det(mMyMat1)
[1] 40
```

To pull out the diagonal elements of a matrix, use the `diag()` function:

```
> mMyMat1
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
[3,]    1   11   13   19
[4,]   15   13   12   13
> diag(mMyMat1)
[1]  1  4 13 13
```

The `diag()` function returns a vector, but if you want a matrix, you can use `as.matrix()`:

```
> as.matrix(diag(mMyMat1))
     [,1]
[1,]    1
[2,]    4
[3,]   13
[4,]   13
```

But, notice that you are not given a square matrix, one way to do this is to use

Matrix Operations

the `diag()` function again to create a 4x4 matrix:

```
> diag(diag(mMyMat1),4,4)
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    4    0    0
[3,]    0    0   13    0
[4,]    0    0    0   13
>
```

You can also assign to these results!

To compute the eigenvalues and eigenvectors, you can use the `eigen()` function which returns a *list* with elements `$values` (containing the eigenvalues), and `$vectors` (containing the eigenvectors):

```
> lmMyMat1_EIGEN <- eigen(mMyMat1)
> lmMyMat1_EIGEN
$values
[1] 36.8185300 -5.1505077 -0.9018984  0.2338761

$vectors
          [,1]       [,2]        [,3]       [,4]
[1,] 0.2431646  0.2164310 -0.04907062 -0.2342337
[2,] 0.2932454  0.1581128 -0.16859394  0.1285268
[3,] 0.6611905  0.6431354  0.83956402  0.7699817
[4,] 0.6463012 -0.7173108 -0.51410156 -0.5794339
>
```

Matrix Operations

To test if a matrix is symmetric or not, use the `isSymmetric()` function:

```
> isSymmetric(mMyMat1)
[1] FALSE
> isSymmetric(mMyMat2)
[1] FALSE
>
```

To pull out the lower or upper triangular parts of a matrix, you can use the `lower.tri()` and `upper.tri()` functions. Note that these functions return booleans (TRUEs and FALSEs), so you will have to include these in the indexes to get back the actual data. For example,

```
> mMyMat1 <- matrix(c(1,3,5,7, 2,4,6,8, 1,11,13,19, 15,13,12,13),nrow=4,ncol=4,byrow=TRUE)
> mMyMat1_UT <- mMyMat1
> mMyMat1_UT[lower.tri(mMyMat1)] <- NA
> mMyMat1_UT
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]   NA    4    6    8
[3,]   NA   NA   13   19
[4,]   NA   NA   NA   13
>
```

Note that you are setting the lower triangular part of the matrix to missing values. If you want to include the main diagonal, use `diag(mat,diag=TRUE)`.

Matrix Operations

To transpose a matrix, use the `t()` function:

```
> mMyMat1
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
[3,]    1   11   13   19
[4,]   15   13   12   13
> t(mMyMat1)
     [,1] [,2] [,3] [,4]
[1,]    1    2    1   15
[2,]    3    4   11   13
[3,]    5    6   13   12
[4,]    7    8   19   13
>
```

There are two cross-product functions available in the base package: `crossprod(mat1,mat2)` which is equivalent to `t(mat1) %*% mat2`, and `tcrossprod(mat1,mat2)` which is equivalent to `mat1 %*% t(mat2)`.

To solve a system of linear equations using matrices, you can use the `solve()` function. For example, given the two linear equations,

```
3x + 4y = 2
7x - 3y = 10
```

Let's solve the equation using the `solve()` function:

Matrix Operations


```
> A <- matrix( c(3,4, 7,-3),2,2,byrow=TRUE )
> B <- c(2,10)
> solve(A,B)
[1]  1.2432432 -0.4324324
>
```

To check,

```
> 3*(1.2432432) + 4*(-0.4324324)
[1] 2
> 7*(1.2432432) - 3*(-0.4324324)
[1] 10
>
```

The base package also has the following functions:

```
svd - computes the singular value decomposition
backsolve - solves an upper/lower triangular system
chol - the Choleski decomposition
chol2inv - inverse from Choleski (or QR) decomposition
kappa - computes the condition number of a matrix
kronecker - kronecker products on arrays
qr - The QR decomposition of a matrix
qr.X - reconstruct the Q, R or X matrixes from a QR object
```

Please check the CRAN website for other packages containing matrix functions.

Matrix Operations

Finally, to determine the dimensions of a matrix, use the `dim()` function:

```
> dim(mMyMat1)
[1] 4 4
> dim(mMyMat2)
[1] 4 4
>
```

Function Round-Up #6

Random Numbers

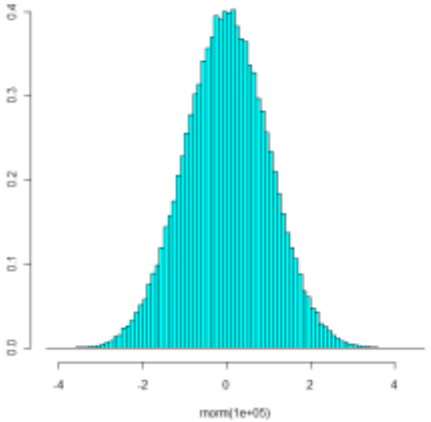
Function Round-Up #6

Just like SAS, R has several functions you can use to generate random numbers:

```
rexp(n, rate=1) - generate n random numbers from the exponential distribution
rgamma(n, shape, scale=1) - generate n random numbers from the gamma distribution
rnorm(n,mean=0,sd=1) – generate n random numbers from the Normal N(0,1) distribution
rpois(n, lambda) - generate n random numbers from the Poisson distribution
rweibull(n, shape, scale=1) - generate n random numbers from the Weibull distribution
rcauchy(n, location=0, scale=1) - generate n random numbers from the Cauchy distribution
rbeta(n, shape1, shape2) - generate n random numbers from the beta distribution
rt(n, df) - generate n random numbers from the Student (t) distribution
rf(n, df1, df2) - generate n random numbers from the Fisher–Snedecor (F) (c2) distribution
rchisq(n, df) - generate n random numbers from the Pearson distribution
rbinom(n, size, prob) - generate n random numbers from the binomial distribution
rgeom(n, prob) - generate n random numbers from the geometric distribution
rhyper(nn, m, n, k) - generate n random numbers from the hypergeometric distribution
rlogis(n, location=0, scale=1) - generate n random numbers from the logistic distribution
rlnorm(n, meanlog=0, sdlog=1) - generate n random numbers from the lognormal distribution
rnbinom(n, size, prob) - generate n random numbers from the negative binomial distribution
runif(n,min=0,max=1) – generate n random numbers from the Uniform distribution
rwilcox(nn, m, n), rsignrank(nn, n) - generate n random numbers from the
```

To produce the image below, I used this code:

```
> library(MASS)
> truehist(rnorm(100000))
```



Function Round-Up #6

You can set a random number seed so that you will be assured that you generate the same random numbers….as humorous as that may sound. Use the `set.seed()` function to set a random seed:

```
> rnorm(10)
 [1]  2.15599797  0.08803553  0.04413455  1.14435636 -0.05785080  0.12766980  1.04508568  0.62282303 -0.71812510  0.10443367
> rnorm(10)
 [1] -2.1948725 -1.1161109 -0.6612492  0.1559121  1.1837928  0.7067434 -0.4520887 -0.8152409 -2.4699628 -1.1530996
> set.seed(5)
> rnorm(10)
 [1] -0.84085548  1.38435934 -1.25549186  0.07014277  1.71144087 -0.60290798 -0.47216639 -0.63537131 -0.28577363  0.13810822
> rnorm(10)
 [1]  1.2276303 -0.8017795 -1.0803926 -0.1575344 -1.0717600 -0.1389861 -0.5973131 -2.1839668  0.2408173 -0.2593554
> set.seed(5)
> rnorm(10)
 [1] -0.84085548  1.38435934 -1.25549186  0.07014277  1.71144087 -0.60290798 -0.47216639 -0.63537131 -0.28577363  0.13810822
>
```

Using Formulas

Using Formulas

In some statistical and graphing functions available in R, instead of entering in a long list of variables that you want to use, you are required to enter in a *formula*.

In R's statistical functions, a formula indicates which variable(s) plays the role of dependent variable(s), which variable(s) play the role of independent variable(s), and which variable(s) play the role of interaction terms.

In R's graphical functions, a formula indicates which variables play the role of which axis (x/y or x/y/z), which variable(s) are grouping variables, etc. See the Programming I lecture for an example of this.

A formula is made up of variables plus several symbols. The tilde (~) symbol indicates which variable is the dependent variable (on the left of the tilde) and which variable is the independent variable (on the right). For example, let's compute a linear regression using the `lm()` function on the Fat Kids data with Weight being the dependent variable and Height being the independent variable:

```
> lmResults1 <- lm(Weight ~ Height,data=dfFatKids)
```

As you see above, Weight is to the left of the tilde indicating that it is to play the role of the dependent variable, and Height is to the right of the tilde indicating that it is to play the role of the independent variable. Note that there is no need to indicate the intercept since that is automatically placed in the formula for you. Below are the results using the `summary()` function:

Using Formulas


```
> summary(lmResults1)

Call:
lm(formula = Weight ~ Height, data = dfFatKids)

Residuals:
      1       2       3       4       5       6       7
-123.54 -144.40 -126.81  -64.27  -94.58  -43.34  596.93

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 245.8936   267.7686   0.918    0.401
Height        0.6143     4.4158   0.139    0.895

Residual standard error: 291 on 5 degrees of freedom
Multiple R-squared: 0.003856,   Adjusted R-squared: -0.1954
F-statistic: 0.01935 on 1 and 5 DF,  p-value: 0.8948

>
```

As you see, the model stinks. Now, if you wanted to add the variable FattyIndex as another independent variable, you use a plus-sign (+):

```
> lmResults2 <- lm(Weight ~ Height + FattyIndex,data=dfFatKids)
```

Using Formulas


```
> summary(lmResults2)

Call:
lm(formula = Weight ~ Height + FattyIndex, data = dfFatKids)

Residuals:
      1       2       3       4       5       6       7
  88.71  122.64  -68.80 -198.21  -44.70  -77.62  177.98

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -397.786    239.635  -1.660   0.1723
Height         6.899      3.076   2.243   0.0883 .
FattyIndex    44.584     12.932   3.448   0.0261 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 163.3 on 4 degrees of freedom
Multiple R-squared: 0.7492,     Adjusted R-squared: 0.6238
F-statistic: 5.974 on 2 and 4 DF,  p-value: 0.06291

>
```

As you see, the model is better. Note that the intercept is not significant, so let's remove it by introducing a "-1" in the model:

```
> lmResults3 <- lm(Weight ~ Height + FattyIndex - 1,data=dfFatKids)
```

Using Formulas

The results of this new model without an intercept is:

```
> summary(lmResults3)

Call:
lm(formula = Weight ~ Height + FattyIndex - 1, data = dfFatKids)

Residuals:
      1       2       3       4       5       6       7
 -41.46  -51.59  -63.55 -276.08  -41.11  -16.60  305.74

Coefficients:
           Estimate Std. Error t value Pr(>|t|)
Height        2.191      1.383   1.584   0.1741
FattyIndex   27.859      9.423   2.957   0.0316 *
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 189.8 on 5 degrees of freedom
Multiple R-squared: 0.815,      Adjusted R-squared: 0.741
F-statistic: 11.01 on 2 and 5 DF,  p-value: 0.01472

>
```

Again, there is no need to add a "+1" to indicate an intercept since this is automatic.

Using Formulas

If would like to introduce interaction terms, use a colon (:):

```
> lmResults4 <- lm(Weight ~ Height + FattyIndex + Height:FattyIndex,data=dfFatKids)
> summary(lmResults4)

Call:
lm(formula = Weight ~ Height + FattyIndex + Height:FattyIndex,
    data = dfFatKids)

Residuals:
         1          2          3          4          5          6          7
 1.305e-03 -5.080e-04  1.259e-03  5.566e-06 -3.542e-03  1.517e-03 -3.588e-05

Coefficients:
                   Estimate Std. Error    t value Pr(>|t|)
(Intercept)       -7.790e-04  4.723e-03     -0.165    0.879
Height             1.716e-05  7.010e-05      0.245    0.822
FattyIndex         2.716e-05  3.909e-04      0.069    0.949
Height:FattyIndex  1.000e+00  7.584e-06 131862.935 9.62e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.002476 on 3 degrees of freedom
Multiple R-squared:     1,      Adjusted R-squared:     1
F-statistic: 2.311e+10 on 3 and 3 DF,  p-value: 4.833e-16
>
```

You can achieve the same thing as above by using the asterisk (*):

Using Formulas


```
> lmResults5 <- lm(Weight ~ Height*FattyIndex,data=dfFatKids)
> summary(lmResults5)

Call:
lm(formula = Weight ~ Height * FattyIndex, data = dfFatKids)

Residuals:
         1          2          3          4          5          6          7
 1.305e-03 -5.080e-04  1.259e-03  5.566e-06 -3.542e-03  1.517e-03 -3.588e-05

Coefficients:
                     Estimate Std. Error    t value Pr(>|t|)
(Intercept)       -7.790e-04  4.723e-03     -0.165    0.879
Height             1.716e-05  7.010e-05      0.245    0.822
FattyIndex         2.716e-05  3.909e-04      0.069    0.949
Height:FattyIndex  1.000e+00  7.584e-06 131862.935 9.62e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.002476 on 3 degrees of freedom
Multiple R-squared:     1,      Adjusted R-squared:     1
F-statistic: 2.311e+10 on 3 and 3 DF,  p-value: 4.833e-16

>
```

There's more to formulas than this. See `?formula` in R.

There's more to the regression function `lm()`. See ?lm for more. Also, see the Programming I presentation for an example of plotting regression statistics.

The `attach()` and `detach()` Functions and Why you Shouldn't Use Them

Attach and Detach and Why You Shouldn't Use Them

If you decide to purchase books on R, you will no doubt come across the `attach()` and `detach()` functions. Recall that up to now, when referencing a variable within a data frame we had two choices:

1. *`data_frame`*`$`*`variable_name`* construct
2. `with(`*`data_frame`*`, …code…)` function

The `attach()` function gives you a third option: add your data frame to the search path which means that you can refer to any variable within your data frame *without* using 1 or 2 above. But, as John Chambers says in his book:

> *Using attach() has the advantage that you can type an arbitrary expression involving the variables without wrapping the expression in a call to with(). But the corresponding disadvantage is that the variable names may hide or be hidden by other objects. R will warn you in some cases, but not in all. For this reason, I recommend using a construction such as with() to avoid pitfalls that may seem unlikely, but could be disastrous.*

If one of the authors of the language says not to use `attach()`, I'll take it for granted that the guy knows what he's talking about and avoid that function. The `detach()` function undoes what `attach()` does.

'Nuf said.

Attach and Detach and Why You Shouldn't Use Them

Recall that we created the fat kids data frame, dfFatKids, by using the `read.csv()` function to read it into a data frame. If you issued the command `ls()`, you would see the list of data frames (as well as vectors, functions, etc.) available in your workspace.

One nice function related to this topic is the `data()` function. If you issue the command `data()` without any arguments, you will see a list of all the data frames available to you in all of the packages in the `search()` path. If you know of a data frame that you wish to have available in your workspace, then you can issue the following command to have it available in your workspace just as if you read it in with, say, `read.csv()`:

```
> ls()
 [1] "A" "adate" "aFatKids" "B" "bdate" "bladder" "bladder1" "bladder2
> data("baseball",package="plyr")
> ls()
 [1] "A" "adate" "aFatKids" "B" "baseball" "bdate" "bladder" "bladder1" "bladder2"
```

As you see above, before I issued the `data()` command to add the baseball data frame to my workspace, it does not appear (see first `ls()` command). But, after I issue the `data()` command, the second `ls()` shows that it appears in my workspace.

If you want to remove the data frame from your workspace, use the `rm()` command. This will remove the copy from your workspace and not the copy within the package itself!!

Function Round-Up #7

Recoding Variables

Function Round-Up #7: Recoding Variables

The package car comes with a function called `recode()` which allows you to recode data, either numeric or character, from one set of values to another. In order to use this function, though, you will have to issue a `library(car)` to make all of the car package's functions available to you.

For example, here is the Fat Kids data frame with a Category column:

```
> dfFatKids2
  FirstName Height Weight FattyIndex       BMI Gender Category
1    ALBERT     45    150     3.3333  52.07407      M        A
2  ROSEMARY     35    123     3.5143  70.58694      F        D
3     TOMMY     78    167     2.1410  19.29668      M        A
…and so on…
```

Let's recode the Category column so that A and D are called Grp1, B is called Grp2, and C is Grp3:

```
> library(car)
> dfFatKids2$CatGrp <-
                    recode(dfFatKids2$Category,"c('A','D')='Grp1';c('B')='Grp2';else='Grp3'")
> dfFatKids2
  FirstName Height Weight FattyIndex       BMI Gender Category CatGrp
1    ALBERT     45    150     3.3333  52.07407      M        A   Grp1
2  ROSEMARY     35    123     3.5143  70.58694      F        D   Grp1
3     TOMMY     78    167     2.1410  19.29668      M        A   Grp1
4     BUDDY     12    189    15.7500 922.68750      M        B   Grp2
5   FARQUAR     76    198     2.6053  24.09868      M        B   Grp2
6     SIMON     87    256     2.9425  23.77699      M        C   Grp3
7    LAUREN     54    876    16.2222 211.18930      F        D   Grp1
>
```

Set Operations

Set Operations

Similar to SQL, R has set operators such as `union()`, `intersect()`, `setdiff()` and `setequal()`. For example, given the following two vectors:

```
> vA <- c("bananas","oranges","kiwi","grapes")
> vB <- c("grapes","kaluha","vodka","gin")
> vA
[1] "bananas" "oranges" "kiwi"    "grapes"
> vB
[1] "grapes" "kaluha" "vodka"  "gin"
>
```

Let's compute each of those functions of these two vectors:

```
> union(vA,vB)
[1] "bananas" "oranges" "kiwi"    "grapes"  "kaluha"  "vodka"   "gin"
> intersect(vA,vB)
[1] "grapes"
> setdiff(vA,vB)
[1] "bananas" "oranges" "kiwi"
> setequal(vA,vB)
[1] FALSE
>
```

Appendix

Appendix A: References

Click the titles below to be taken to Amazon.com's website.

- SAS and R, 1st Edition, Ken Kleinman and Nicholas J. Horton (ISBN: 9781420070576)
- R for SAS and SPSS Users, 1st Edition, Robert A. Muenchen (ISBN: 9780387094175)
- Data Manipulation with R, 1st Edition, Phil Spector (ISBN: 9780387747309)
- R In A Nutshell, Joseph Adler (ISBN: 9780596801700)
- Software for Data Analysis: Programming with R, John M. Chambers (ISBN: 9780387759357)
- R Through Excel, Richard Heiberger and Erich Neuwirth, (ISBN: 9781441900517)
- The R Book, Michael J. Crawley (ISBN: 9780470510247)
- Interactive and Dynamic Graphics for Data Analysis with R and GGobi, Dianne Cook and Deborah F. Swayne (ISBN: 9780387717616)
- Lattice: Multivariate Data Visualization with R, Deepayan Darkar (ISBN: 9780387759685)
- ggplot2: Elegant Graphics for Data Analysis, Hadley Wickham (ISBN: 9780387981406)
- Introductory Statistics with R, 2nd Edition, Peter Dalgaard (ISBN: 9780387790541)
- Modern Applied Statistics with S, W.N. Venables and B.D. Ripley (ISBN: 9781441930088)
- Manuals Provided with R Software: An Introduction to R, R Data Import/Export, R Language Definition, R Installation and Administration, R Internals, Writing R Extensions
- GGobi Manual (http://www.ggobi.org/rggobi/introduction.pdf), Deborah F. Swayne, Hadley Wickham, et. al.
- cran.r-project.org/web/packages/RcmdrPlugin.HH/RcmdrPlugin.HH.pdf: Documentation for the RcmdrPlugin.HH plug-in to R Commander.

Appendix B: R-Related Websites

- www.r-project.org: This is the main R Software Site and contains the software itself for various platforms as well as documentation.
- cran.r-project.org: This is the website of the Comprehensive R Archive Network (CRAN) and it houses all of the R packages you could ever possibly want.
- journal.r-project.org: This is the website of the R Journal which is a refereed journal of the R Project and contains articles on a variety of topics related to R.
- rwiki.sciviews.org/doku.php: This is the R Wiki website and houses a variety of searchable content.
- sourceforge.net: This is the main SourceForge website and contains free software not necessarily related to R (but does contain a lot of R-related software).
- r-forge.r-project.org: This is the main Rforge website and contains R-related development software such as packages, graphical user interfaces, etc.
- rcom.univie.ac.at: R and Friends website with RCOM package and also R bundled with RCOM.
- www.ggobi.org/downloads: GGobi website with free download of GGobi.
- www.ggobi.org/demos: GGobi demo movies and screen shots.
- cran.r-project.org/web/packages/RcmdrPlugin.HH: Website for the RcmdrPlugin.HH plug-in to R Commander.

Support sheepsqueezers.com

If you found this information helpful, please consider supporting sheepsqueezers.com. There are several ways to support our site:

- ☐ Buy me a cup of coffee by clicking on the following link and donate to my PayPal account: Buy Me A Cup Of Coffee?.
- ☐ Visit my Amazon.com Wish list at the following link and purchase an item: http://amzn.com/w/3OBK1K4EIWIR6

Please let me know if this document was useful by e-mailing me at `comments@sheepsqueezers.com`.