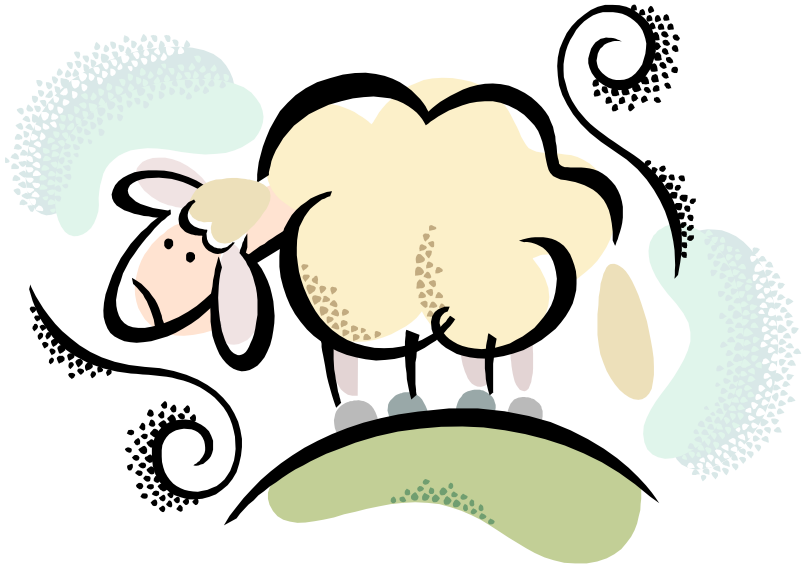
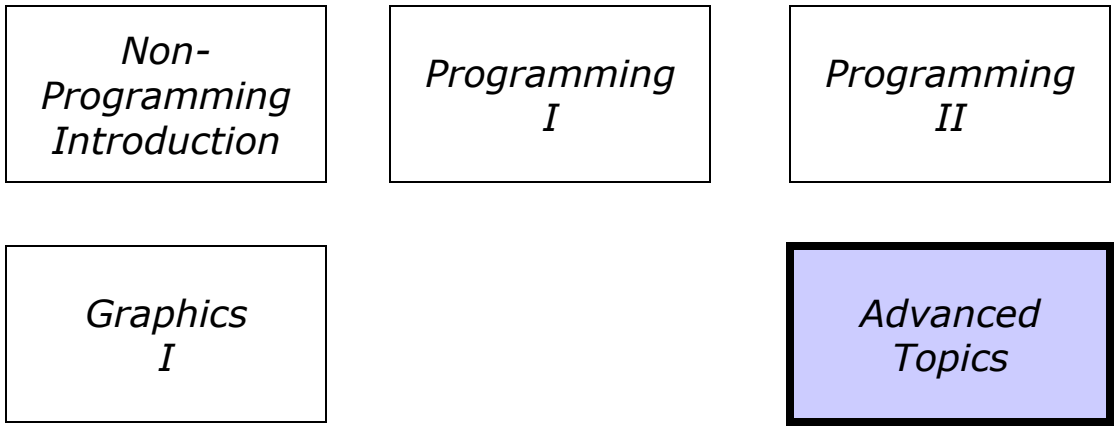


Advanced Topics

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



Charting Our Course

- Goal of this Presentation
- Creating Your Own R Package
- Using Regular Expressions
- S3/S4 Objects and Object Oriented Programming
- R Startup Parameters and Memory-related Functions
- Appendix

Goal of this Presentation

This presentation deals with more advanced topics than those that have been covered in the previous lectures.

In this presentation, we will look at how to create an R package so that you can share your code and data with other R users.

We will look into how to use Regular Expressions in R, a topic that was touched upon briefly in the Programming II lecture.

We will talk about object-oriented programming and the meaning of S3 and S4 objects in R.

Finally, we will look at the R startup parameters and how and why to change them in order to get R to perform better.

Creating Your Own R Package

Creating Your Own R Package

Throughout all of these R lectures, we have made use of R packages such as the `base` package all the way through the `lattice` package to create graphics. If a package was not available on your computer, you downloaded and installed it using the simple `install.packages()` function. These packages were made by other R users and were made available on CRAN, Source/R Forge or other web sites.

But, what if you've created a series of functions, or a few data frames, and you want to share them with other users within your department or organization? One way is to email your R functions to all of the potential users. This is probably not that way to go, especially if you modify your functions often: several users may not take the time to update your functions in their R environment leaving some users working with your older functions instead of the newer functions.

To solve this, R allows you to create your own R package complete with your functions, along with the associated documentation, data frames, and more.

This section explains how to create your own R package containing one function and one data frame. First, the data frame we will use is the Fat Kids data frame. Second, let's create a useful function called `MyMatch(string1,string2)` which computes a (terrible) match score between the two strings. Here is the R code (error checking has been removed to make the code smaller):

Creating Your Own R Package


```
MyMatch <- function(sString1,sString2) {

 ## Lowercase the incoming strings
 sString1 <- tolower(sString1)
 sString2 <- tolower(sString2)

 ## Replace any non-alphanumeric characters with a question mark
 sString1 <- gsub("[^[:alnum:]]","?",sString1)
 sString2 <- gsub("[^[:alnum:]]","?",sString2)

 ## Compute the match score by dividing the number of letters in sString1 by
sString2
 dScore=nchar(sString1)/nchar(sString2)

 return(dScore)
}
```

Creating Your Own R Package

For example, My`Match("FRED","FREDFRED")` yields a `0.5` which indicates that my matching algorithm stinks!

A package, when installed on your hard drive, is just a directory containing the following sub-directories (not all of which will be used in our example):

- A `man` sub-directory containing the documentation files
- An `R` sub-directory containing the R code
- A `data` sub-directory containing data frames
- A `src` sub-directory containing C, Fortran or C++ source code
- A `tests` sub-directory containing validation tests
- An `exec` sub-directory for other executables (such as Perl or Java)
- An `inst` sub-directory for miscellaneous content

In the root folder for your package, there are several files:

- A `DESCRIPTION` file with descriptions of the package, author, etc.
- A `configure` script used to check for required software, etc.

Creating Your Own R Package

Now, the creators of R realize that creating all of these sub-directories and files can be a chore, so they created a function called `package.skeleton()` that creates all of the files and folders necessary to create your package.

Let's create a package named `MYRTools` which will contain the `MyMatch()` function as well as the dfFatKids data frame:

```
> package.skeleton(name = "MYRTools", list=c("MyMatch","dfFatKids"),path = "C:\\TEMP")
Creating directories ...
Creating DESCRIPTION ...
Creating Read-and-delete-me ...
Saving functions and data ...
Making help files ...
Done.
Further steps are described in 'C:\TEMP/MYRTools/Read-and-delete-me'.
```

Take note that the list parameter is a character vector of all of the objects (i.e., functions, data frames, etc.) that are going to be placed in the `MYRTools` package. The name argument names the package (`MYRTools`, in our case) and the path parameter tells R where to put the skeleton.

As you see above, the output of the `package.skeleton()` function tells you what it's doing and tells you to read the file `Read-and-delete-me` for further instructions. The contents of `Read-and-delete-me` are on the next slide:

Creating Your Own R Package


```
* Edit the help file skeletons in 'man', possibly combining help files for multiple functions.
* Put any C/C++/Fortran code in 'src'.
* If you have compiled code, add a .First.lib() function in 'R' to load the shared library.
* Run R CMD build to build the package tarball.
* Run R CMD check to check the package tarball.

Read "Writing R Extensions" for more information.
```

As you see above, you are to edit the help files in the man sub-directory, and then issue the command `R CMD build MYRTools` at the command prompt as well as `R CMD check MYRTools` to check the build. (We will use a different command on Windows.) You should also edit the DESCRIPTION file and fill in the appropriate information for your package. Here is what the DESCRIPTION file looks like before it has been edited:

```
Package: MYRTools
Type: Package
Title: What the package does (short line)
Version: 1.0
Date: 2010-10-18
Author: Who wrote it
Maintainer: Who to complain to <yourfault@somewhere.net>
Description: More about what it does (maybe more than one line)
License: What license is it under?
LazyLoad: yes
```

Below is how I changed it:

Creating Your Own R Package


```
Package: MYRTools
Type: Package
Title: My specific Functions and Data Frames
Version: 1.0
Date: 2010-10-18
Author: sheepsqueezers.com
Maintainer: The Sheepster <comments@sheepsqueezers.com>
Description: This package contains my specific functions (such as MyMatch, etc.) and data (such as
dfFatKids, etc.) for use within the the organization.
License: Sheepsqueezers.com Only!
LazyLoad: yes
LazyData: yes
```

Next, edit the man page(s) in the man sub-directory. For this example, there are three man pages to edit:

- dfFatKids.Rd – the manual page for the dfFatKids data frame
- MyMatch.Rd – the manual page for the MyMatch function
- MYRTools-package.Rd – the manual page associated with the entire package itself

Unfortunately, the code within these three man pages is similar to the old LaTex (or Tex) language used to produce academic papers before modern word processors came along. We're not going to teach you how to read LaTex, but you only have to change where the double-tildes are located. Let's take a look at the man page for the dfFatKids data frame:

Creating Your Own R Package


```
\name{dfFatKids}
\alias{dfFatKids}
\docType{data}
\title{
%%   ~~ data name/kind ... ~~
}
\description{
%%  ~~ A concise (1-5 lines) description of the dataset. ~~
}
\usage{data(dfFatKids)}
\format{
  A data frame with 7 observations on the following 5 variables.
  \describe{
    \item{\code{FirstName}}{a factor with levels \code{ALBERT} \code{BUDDY} \code{FARQUAR} \code{LAUREN}
\code{ROSEMARY} \code{SIMON} \code{TOMMY}}
    \item{\code{Height}}{a numeric vector}
    \item{\code{Weight}}{a numeric vector}
    \item{\code{FattyIndex}}{a numeric vector}
    \item{\code{BMI}}{a numeric vector}
  }
}
\details{
%%  ~~ If necessary, more details than the __description__ above ~~
}
\source{
%%  ~~ reference to a publication or URL from which the data were obtained ~~
}
\references{
%%  ~~ possibly secondary sources and usages ~~
}
\examples{
data(dfFatKids)
## maybe str(dfFatKids) ; plot(dfFatKids) ...
}
\keyword{datasets}
```

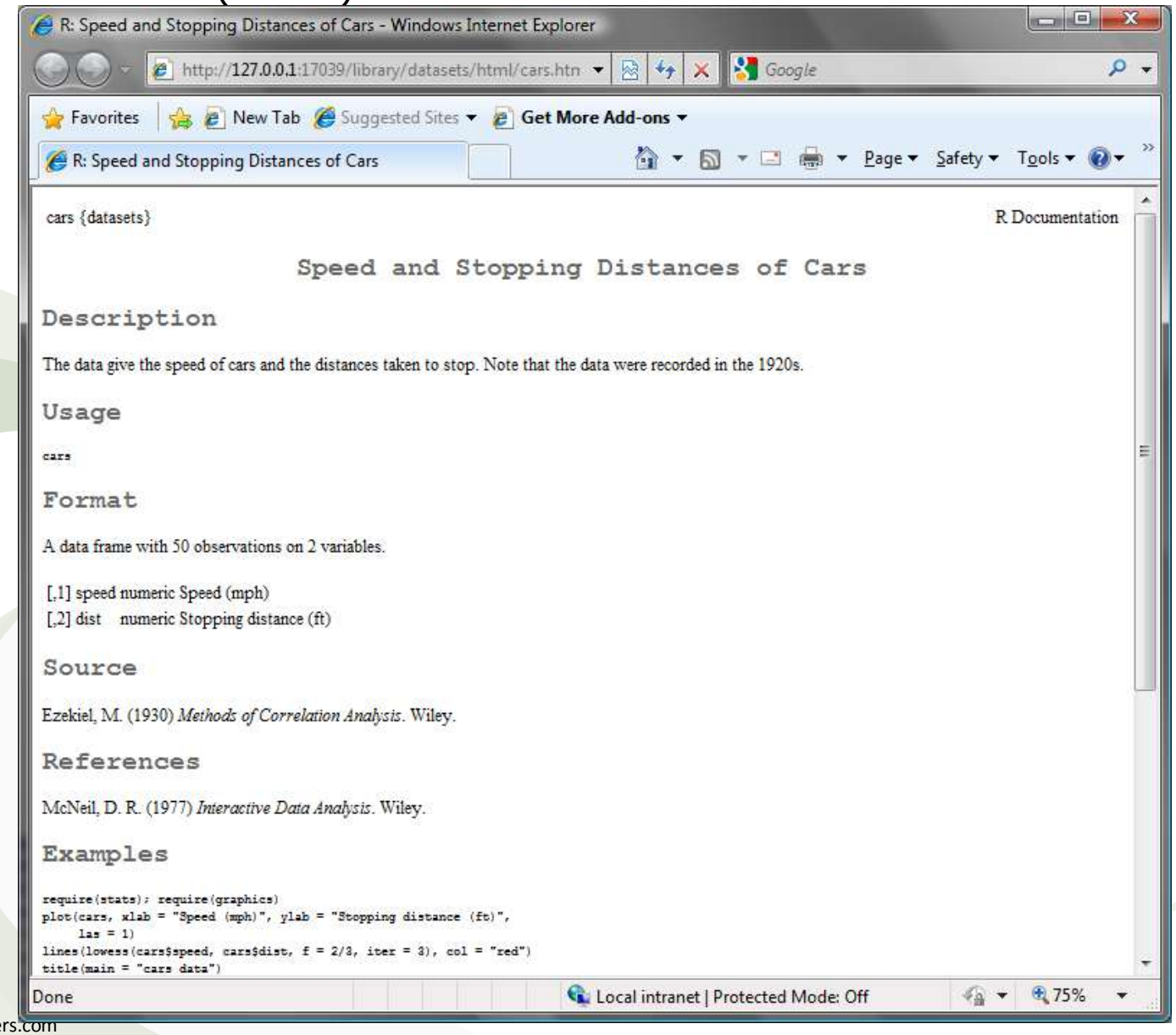
Creating Your Own R Package

Wherever you see the bold red code is where you have to change. For example, here is the modified version of this man page:

```
\name{dfFatKids}
\alias{dfFatKids}
\docType{data}
\title{
Height and Weight Information for Overweight Children
}
\description{
This data frame give the name, height, weight, fatty index (FI) and the body mass index (BMI) of seven overweight children.
}
\usage{data(dfFatKids)}
\format{
  A data frame with 7 observations on the following 5 variables.
  \describe{
    \item{\code{FirstName}}{a factor with levels \code{ALBERT} \code{BUDDY} \code{FARQUAR} \code{LAUREN} \code{ROSEMARY}
\code{SIMON} \code{TOMMY}}
    \item{\code{Height}}{a numeric vector}
    \item{\code{Weight}}{a numeric vector}
    \item{\code{FattyIndex}}{a numeric vector}
    \item{\code{BMI}}{a numeric vector}
  }
}
\details{
No additional details are provided.
}
\source{
Adipose,F.(2010), Journal of Farm Family Statisitics, v.1, no. 7, pp331-334.
}
\references{
No additional references given.
}
\examples{
dfFatKids
}
\keyword{datasets}
```

Creating Your Own R Package

Before I modify my own Rd files, I like taking a look at some of the pre-existing man pages. For example, the cars data frame in the `datasets` package looks like this (`?cars`):



Creating Your Own R Package

Before we can build the package for delivery, we must ensure that the R binaries are in the path by updating the PATH environment variable. Details for your operating system may vary, so contact the help desk. Also, on Windows you will need to install ActivePerl! The Perl provided with Cygwin does not work with the R package creation. You won't need to do this on a Unix/Linux machine, by the way. Also on Windows, you will need to install the R Tools located at http://www.murdoch-sutherland.com/Rtools. Run the Rtools.exe installer. Ensure that the PATH has been updated to include R, ActivePerl and the Rtools. Finally, we can create the package `MYRTools` at the command line. Open up the command line window, change directory to C:\TEMP (or wherever your package skeleton is located), and then issue the following command:

```
C:\TEMP>Rcmd build --binary MYRTools
* checking for file 'MYRTools/DESCRIPTION' ... OK
* preparing 'MYRTools':
* checking DESCRIPTION meta-information ... OK
* removing junk files
* checking for LF line-endings in source and make files
* checking for empty or unneeded directories
* building binary distribution
WARNING: some HTML links may not be found
* installing *source* package 'MYRTools' ...
** R
** data
** preparing package for lazy loading
** help
*** installing help indices
** building package indices ...
** MD5 sums
packaged installation of 'MYRTools' as MYRTools_1.0.zip
* DONE (MYRTools)
```

Creating Your Own R Package

As you see, the file MYRTools_1.0.zip has been created in the C:\TEMP folder (and NOT in the MYRTools folder as you might expect). Note that this does NOT install this package on your machine automatically

To test this out, I used the `rm()` command from within the RGui to remove both the `MyMatch()` function as well as the `dfFatKids` data frame. I then quit out of R selecting Yes when asked if I wanted to save my workspace. This will ensure that the next time I go into R, `MyMatch()` and `dfFatKids` will not exist any more.

Let's install our package. At the R command line, issue the following:

```
> install.packages("C:/TEMP/MYRTools_1.0.zip",
                   repos=NULL,
                   lib="C:/PROGRA~2/R/R-210~1.1/library")
package 'MYRTools' successfully unpacked and MD5 sums checked
```

Note that the first argument is the location of the zip file, the second argument is set to NULL to indicate that our package is local (as opposed to CRAN), and the third parameter is the location of the `library` sub-directory.

Note that you may need to modify the library sub-directory's security to allow yourself write privileges.

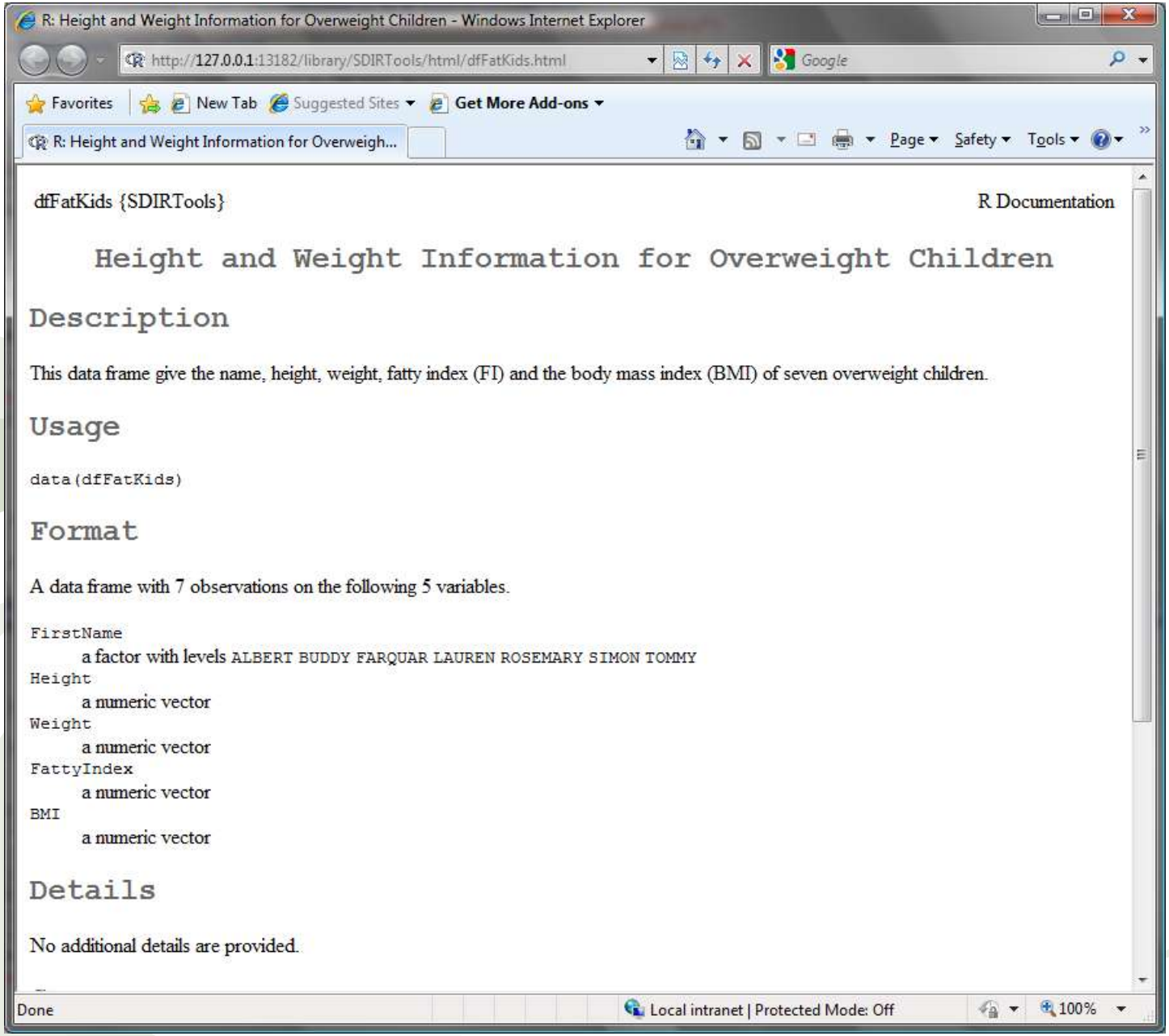
Next, let's test out our package:

Creating Your Own R Package


```
> library(MYRTools)
> search()
 [1] ".GlobalEnv"        "package:MYRTools" "package:stats"     "package:graphics"
"package:grDevices" "package:datasets"  "package:digest"    "package:reshape"
 [9] "package:plyr"      "package:grid"      "package:proto"     "package:rcom"
"package:rscproxy"  "package:utils"     "package:methods"   "Autoloads"
[17] "package:base"
> dfFatKids
  FirstName Height Weight FattyIndex       BMI
1    ALBERT     45    150     3.3333  52.07407
2  ROSEMARY     35    123     3.5143  70.58694
3     TOMMY     78    167     2.1410  19.29668
4     BUDDY     12    189    15.7500 922.68750
5   FARQUAR     76    198     2.6053  24.09868
6     SIMON     87    256     2.9425  23.77699
7    LAUREN     54    876    16.2222 211.18930
> MyMatch("Aetna","Aetna, Inc.")
[1] -9.09091
> ?dfFatKids
```

Take note that when you issue `dfFatKids` at the command line, the data is shown. If you did not use `LazyData: yes`, you would have to issue `data(dfFatKids)` first in order to make the data available. Regardless, the `MyMatch()` function is available for use and does not have to be "loaded" first.

Creating Your Own R Package



Using Regular Expressions

Using Regular Expressions

Recall that in the Programming II lecture, we briefly discussed regular expressions, but said that we would defer a more detailed lecture for a later time. Now's the time, baby!

If you don't know anything about Regular Expression (aka, RegEx's), prepare to have your world rocked! There may be a steep learning curve here, but once you *get it*, you'll be happy you spent the time to learn RegEx's.

First, let's talk about *wildcards*. Most of you will be familiar with at least two wildcards. The first is used most often when perform a directory listing using the `dir` or `l` commands at the Windows or Linux command prompt. For example, to list all of those files that begin with the letters `prog` followed by anything and ending in `.c`, you would type this at the command line:

```
dir prog*.c
l prog*.c
```

Basically, the asterisk means *any amount of letters or numbers*.

The second is used in the Structured Query Language (SQL) in WHERE Clauses. For example, to search the ADDRESS field for text containing the words "MAIN ST.", you could issue this SQL query:

Using Regular Expressions


```
SELECT *
 FROM MYTABLE
 WHERE ADDRESS LIKE '%MAIN ST.%';
```

As you see above, the percent signs mean "any amount of characters" and is analogous to the asterisk used at the command line. SQL has an additional wildcard, the underscore (_) meaning *exactly one letter or number*. So, to search a SSNum field for an appropriate social security number, we could issue this SQL query:

```
SELECT *
 FROM MYTABLE
 WHERE SSNUM LIKE '___-__-____';  /* 3 underscores, a dash, 2 underscores, a dash, 4 underscores */
```

Now, if you ponder the phrases *exactly one letter or number* and *any amount of letters or numbers* you'll find that there are two concepts being melded together in these phrases: **quantity** (such as *exactly one* and *any amount*) and **characters** (such as *letters or numbers*).

In Regular Expressions, you are not limited to *any amount* and *exactly one*, but are free to specify the quantity you need by using one or more *quantifiers*. In Regular Expressions, the combination of quantifiers and characters is called a *pattern* and it's this pattern that's used to search through a text string in order to determine if a match has been made, very similar to the WHERE Clause above which searches for the correct pattern of a social security number.

Using Regular Expressions

To whet your appetite, let's create a Regular Expression to match the pattern of a social security number:

```
[[:digit:]]{3}-[[:digit:]]{2}-[[:digit:]]{4}
```

The text `[[:digit:]]`, called a *Character Class*, indicates that we are searching for the numbers 0 through 9. The numbers in the curly braces are the quantifiers. The combination of everything you see above specifies the pattern we are looking for. In this case, exactly three digits followed by a single dash followed exactly two digits followed by a dash followed by exactly four digits.

So, why is this better than using the underscore, say, in our SQL example above? Because the underscore also allows for letters and non-alphanumeric characters (such as question marks and periods). So, if your SSNUM field contained the text `"A@C-DE-F?H$"`, your WHERE Clause would happily let this text be displayed. So, how would you remedy this in SQL? It looks like you'd have to code this SQL:

```
SELECT *
 FROM MYTABLE
 WHERE SUBSTR(SSNUM,1,1) IN ('0','1','2','3','4','5','6','7','8','9')
       AND SUBSTR(SSNUM,2,1) IN ('0','1','2','3','4','5','6','7','8','9')
       AND SUBSTR(SSNUM,3,1) IN ('0','1','2','3','4','5','6','7','8','9')
       AND SUBSTR(SSNUM,4,1)='-'
       AND ...and so on...
```

Using Regular Expressions

So, which would you rather code, the RegEx or the SUBSTR?

Let's talk more about the quantifiers in Regular Expressions. As we've seen, you can use the curly braces containing a number to indicate the quantity of the characters that should appear. Quantifiers always follow *on the right* of the character set in the RegEx pattern. Here are the quantifiers available:

```
? - The preceding item is optional and will be matched at most once (i.e., 0 or 1 times).
* - The preceding item will be matched zero or more times.
+ - The preceding item will be matched one or more times.
{n} - The preceding item is matched exactly n times.
{n,} - The preceding item is matched n or more times.
{n,m} - The preceding item is matched at least n times, but not more than m times.
```

As you see above, our curly braces are represented, but can take on two additional forms. The form we learned above for the social security example indicates the exact number of times the character set (on the left) is to appear:

`[[:digit:]]{3}` ➔ a single digit must appear *exactly 3 times*

There are two other forms:

`[[:digit:]]{3,}` ➔ a single digit must appear 3 or more times
`[[:digit:]]{3,5}` ➔ a single digit must appear exactly 3 times but not more than 5 times.

Using Regular Expressions

As you can see above, the asterisk (*) is represented with its usual meaning of match zero or more times. Along with this quantifier, we have the question mark (?) meaning to match exactly zero or one times, and the plus sign (+) meaning to match one or more times.

Let's create a pattern to match a nine digit zip code regardless of the dash between the first five digits and the second four digits:

```
[[:digit:]]{5}-?[[:digit:]]{4}
```

We've seen the curly braces before as well as the digits Character Class, so let's concentrate on the `-?`. The question mark (?) indicates that the character before it should appear either zero times (that is, there is no dash appearing), or exactly one time (that is, there is a dash appearing). This pattern would match, say, `19115-2151` as well as `191152151`. Note that if we used an asterisk instead of a question mark, the pattern would allow for more than one dash to be allowed. For example, this pattern

```
[[:digit:]]{5}-*[[:digit:]]{4}
```

would match the zip code `19115---2151` as well as `191152151`.

Using Regular Expressions

We've seen that digits Character Class, `[:digits:]`, represents the single digits from 0 to 9. There are several other Character Classes available to you:

```
[:alnum:] - Alphanumeric characters: [:alpha:] and [:digit:].

[:alpha:] - Alphabetic characters: [:lower:] and [:upper:].

[:blank:] - Blank characters: space and tab. (This is an extension to the POSIX standard.)

[:cntrl:] - Control characters. In ASCII, these characters have octal codes 000 through 037, and 177
(DEL). In another character set, these are the equivalent characters, if any.

[:digit:] - Digits: 0 1 2 3 4 5 6 7 8 9.

[:graph:] - Graphical characters: [:alnum:] and [:punct:].

[:lower:] - Lower-case letters in the current locale.

[:print:] - Printable characters: [:alnum:], [:punct:] and space.

[:punct:] - Punctuation characters: ! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [ \ ] ^ _ ` { | } ~.

[:space:] - Space characters: tab, newline, vertical tab, form feed, carriage return, and space.

[:upper:] - Upper-case letters in the current locale.

[:xdigit:] - Hexadecimal digits: 0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f.
```

As you see, the alphabetic characters (both upper and lower case letters, a-z and A-Z) are represented by [:alpha:], and alphanumeric characters (both numbers from 0-9 as well as a-z and A-Z) are represented by [:alnum:].

Now, you'll notice that it seems that I've left off the additional left and right brackets around these Character Classes. Actually, the additional brackets

Using Regular Expressions

around a Character Class name have another meaning. Instead of using the digits Character Class in the patterns above, `[[:digits:]]`, I could have instead coded digits like this: `[0123456789]`. For example, here's our social security number pattern re-written:

```
[0123456789]{3}-[0123456789]{2}-[0123456789]{4}
```

Alternatively, you can use 0-9 to mean 0123456789. By the way, you can use a-z to mean all of the lowercase letters from a to z, and A-Z to mean all of the uppercase letters from A to Z. Here is our pattern again:

```
[0-9]{3}-[0-9]{2}-[0-9]{4}
```

So, in Regular Expressions, the brackets indicates a list of characters you want to be included in your pattern. For example, let's assume we have a product code number that must appear in the following form:

two characters, either P or Z or 7, exactly one must appear
four numbers, 0 to 9, all four digits must appear
four or five letters, must appear from the set W, X, A, Y

For example, P1234WXAY is correct as well as Z4792XXAA is correct. The regular expression pattern to match this is:

Using Regular Expressions


```
[PZ7]{1}[0-9]{4}[WXAY]{4,5}
```

Alternatively, you can use the following equivalent pattern:

```
[PZ7]{1}[[:digit:]]{4}[WXAY]{4,5}
```

Or, equivalently,

```
[PZ7]{1}[0123456789]{4}[WXAY]{4,5}
```

Note that the dash in `[0-9]` indicates a range, in this case a range of numbers from 0 to 9. If you want to include an actual dash in the match, it should appear *first*: `[-0-9]`.

Now, before we move on, let's see how you can use Regular Expressions in R. The function `grepl(`*`pattern`*`,`*`variable`*`)` takes a Regular Expression pattern, within quotes, and a variable (vector, etc.) and returns a logical vector indicating whether the pattern matched (TRUE) or did not match (FALSE) the pattern. So, given the following vector variable,

```
> vSSN <- c("111-22-3333","444-55-6666","ABC-DE-FFFF","11-2222-333","BOB 111-22-3333 WILMA")
> vSSN
[1] "111-22-3333" "444-55-6666" "ABC-DE-FFFF" "11-2222-333" "BOB 111-22-3333 WILMA"
```

Using Regular Expressions

let's use `grepl()` to determine what is a valid or invalid social security number:

```
> vSSN <- c("111-22-3333","444-55-6666","ABC-DE-FFFF","11-2222-333","BOB 111-22-3333 WILMA")
> vSSN
[1] "111-22-3333" "444-55-6666" "ABC-DE-FFFF" "11-2222-333" "BOB 111-22-3333 WILMA"
> grepl('[[:digit:]]{3}-[[:digit:]]{2}-[[:digit:]]{4}',vSSN)
[1]  TRUE  TRUE FALSE FALSE  TRUE
>
```

Note that both the first and second elements are TRUE, which makes perfect sense. But, note that grepl() is telling us that fifth entry is also true. Sure, there is a valid social security number in that string, but it's surrounded by the text BOB and WILMA. What gives?

A pattern will attempt to match anywhere within the text string and not just the beginning and/or end. Hence, the fifth element is TRUE. But, how can we override this to ensure that our string is just a social security number and nothing else? Regular Expressions provides additional character you can use to modify how the pattern is evaluated:

```
^ - indicates that the pattern must match the beginning of the string
$ - indicates that the pattern must match the ending of the string
```

These two additional characters are known as *metacharacters* and they modify how the pattern is evaluated. Both of these metacharacters can be used singly or at the same time. For example, let's change our pattern to match the social

Using Regular Expressions

security number ensuring that the string contains ONLY a social security number:

```
> vSSN <- c("111-22-3333","444-55-6666","ABC-DE-FFFF","11-2222-333","BOB 111-22-3333 WILMA")
> vSSN
[1] "111-22-3333" "444-55-6666" "ABC-DE-FFFF" "11-2222-333" "BOB 111-22-3333 WILMA"
> grepl('^[[:digit:]]{3}-[[:digit:]]{2}-[[:digit:]]{4}$',vSSN)
[1]  TRUE  TRUE FALSE FALSE FALSE
>
```

The caret is used at the beginning of the pattern to indicate that what follows MUST start at the beginning of the text string; and the dollar sign is used at the end of the pattern to indicate that what preceded MUST end at the end of the text string. Hence, your pattern is pegged between the start and the end of your text string. Any matches that *might have occurred* within the text string, but are not pegged between the beginning and the ending, do not match.

Now, so far we've known exactly how our patterns should look because we've been dealing with known text strings such as the social security number (three numbers, dash, two numbers, dash, four numbers), etc. But, what if you don't exactly know what might appear, but you know the form of the string? For example, here are some street addresses:

```
123 MAIN ST
A1B7 FIRST RD
A15-24 SECOND AVE
```

Using Regular Expressions

Notice that the house numbers can be any series of letters, numbers, and punctuation. The name of the street (MAIN, FIRST, SECOND) is a single name, though. And the street type (ST, RD, AVE) follow. So, let's try to create a pattern that matches these three examples; that is, `grepl()` should return TRUE for all three of these examples.

But, before we create the pattern, we need two additional pieces of information.

First, in order to match a single character (a-z, A-Z, 0-9, punctuation characters), we can use the period (.) metacharacter. If we follow the period metacharacter with an asterisk quantifier, giving us `.*`, we have a pattern that means *match zero or more characters (including blanks and nulls)*. If we create the pattern .+, we have a pattern that means *match one or more characters (including blanks, but not nulls)*. So, for example, let's start to create our pattern:

```
> vADDR <- c("123 MAIN ST","A1B7 FIRST RD","A15-24 SECOND AVE")
> vADDR
[1] "123 MAIN ST"       "A1B7 FIRST RD"     "A15-24 SECOND AVE"
> grepl('^.+ .+ ST$',vADDR)
[1]  TRUE FALSE FALSE
>
```

Notice that our pattern says to match one or more characters, followed by a blank followed by one or more characters followed by a blank followed by ST. As you see, `grepl()` is TRUE for the first element of vADDR, but not the rest.

Using Regular Expressions

Now, in order to match RD and AVE as well as ST, we need to know about the left parenthesis (( ), right parenthesis ( )) and vertical bar (|) metacharacters which are used together to give you the ability to provide alternative choices to your pattern. The left and right parentheses provide a place to list your alternatives and the vertical bar is used as a separator between the alternatives indicating the logical OR condition. This is called *alternation*.

For example, to indicate in our pattern that RD, AVE and ST are alternative possibilities, we would create our pattern like this:

```
> vADDR <- c("123 MAIN ST","A1B7 FIRST RD","A15-24 SECOND AVE")
> vADDR
[1] "123 MAIN ST"        "A1B7 FIRST RD"     "A15-24 SECOND AVE"
> grepl('^.+ .+ (ST|RD|AVE)$',vADDR)
[1] TRUE TRUE TRUE
>
```

It's the (ST|RD|AVE) which tells the pattern to recognize ST or RD or AVE which is why `grepl()` returns all TRUE above. Note that there is no need for a quantifier after the parenthesized alternation since match exactly one is implied. If this bothers you, you can add a `{1}` after the right parenthesis.

Now, you might be asking yourself: What happens if I want to search for a left parenthesis, or a period, or other metacharacter in my pattern? In order to search for these, you should put two backslashes (\\) before the character. This

Using Regular Expressions

indicates that the single metacharacter that follows loses its metacharacter meaning and becomes just one of the boys, so to speak. This is called *escaping*. For example, let's create a pattern that matches dollar values. Here is an example:

```
> vBUCKS <- c("$1234.76","$476","974.21")
> vBUCKS
[1] "$1234.76" "$476" "974.21"
> grepl('^\\${1}[[:digit:]]{1,}\\.{1}[[:digit:]]{2}$',vBUCKS)
[1]  TRUE FALSE FALSE
>
```

Note that the \\$ indicate a $ character, the \\. indicates a period character, both lose their metacharacter meaning in those instances. But, the $ at the end of the pattern keeps is metacharacter meaning since it has not been escaped.

Now, there is a problem with the example above. If you look at the second element of vBUCKS, you'll see that it did not match the pattern. Let's modify the pattern so that is matched as well:

```
> grepl('^\\${1}[[:digit:]]{1,}(\\.[[:digit:]]{2})?$',vBUCKS)
[1]  TRUE  TRUE FALSE
>
```

Note the we placed the expression starting from the period to the end within

parentheses and followed the right parenthesis with a question mark (indicating the pattern can occur zero or one times).

So far we've used the `grepl()` function to return a logical vector indicating whether your regular expression pattern matches your text string. This function is very useful for testing your regular expression patterns. Another nice function that is related to `grepl()` is `grep()`. This function returns a vector of integers indicating which elements are true. That is, instead of TRUEs and FALSEs, you only get the indices where the pattern matches. For example:

```
> grepl('^\\${1}[[:digit:]]{1,}(\\.[[:digit:]]{2})?$',vBUCKS)
[1]  TRUE  TRUE FALSE
> grep('^\\${1}[[:digit:]]{1,}(\\.[[:digit:]]{2})?$',vBUCKS)
[1] 1 2
>
```

Two additional functions, `sub()` and `gsub()` allow you to replace the matched pattern with substitute text. For example, let's replace ST, STR and STRE in the street address below with STREET:

```
> vADDR1 <- c("123 MAIN ST","A1B7 FIRST STR","A15-24 SECOND STRE")
> gsub('^.+ .+ (ST|STR|STRE)$','STREET',vADDR1)
[1] "STREET" "STREET" "STREET"
>
```

Using `gsub()` replaces the matched pattern with the word STREET. But, what

happened to the rest of the address? Both `sub()` – which replaces only the *first* occurrence of the matched pattern – and `gsub()` – which replaces *all* occurrences of the matched pattern – are doing exactly what they were designed to do: replace the matched pattern with the substitution text. In order to keep the rest of the matched information, we have to learn about *back referencing*.

Back referencing allows you to indicate which parts of a pattern are to be held aside so that you can refer to them later on, especially when using `sub()` and `gsub()`. To indicate that a part of a pattern is to be a back reference, surround it with parentheses. To use the back reference, place two backslashes followed by the number of the back reference in your substitution text.

For example, let's back reference the street number as well as the street name in our pattern:

```
> gsub('^(.+) (.+) (ST|STR|STRE)$','\\1 \\2 STREET',vADDR1)
[1] "123 MAIN STREET" "A1B7 FIRST STREET" "A15-24 SECOND STREET"
>
```

As you see, the first back reference is the street number (123, A1B7, A15-24) and is associated with the back reference \\1. The second back reference is the street name (MAIN, FIRST, SECOND) and is associated with the back reference \\2. We've placed both of these back references in the substitution portion of `gsub()`.

Using Regular Expressions

Back references can be very useful to swap columns within a text string. For example,

```
> vSTATES <- paste(state.abb,state.name,sep="-")
> vSTATES
 [1] "AL-Alabama"        "AK-Alaska"         "AZ-Arizona"        "AR-Arkansas"       "CA-California"
"CO-Colorado"       "CT-Connecticut"    "DE-Delaware"
 [9] "FL-Florida"        "GA-Georgia"        "HI-Hawaii"         "ID-Idaho"          "IL-Illinois"
"IN-Indiana"        "IA-Iowa"           "KS-Kansas"
[17] "KY-Kentucky"       "LA-Louisiana"      "ME-Maine"          "MD-Maryland"       "MA-Massachusetts"
"MI-Michigan"       "MN-Minnesota"      "MS-Mississippi"
[25] "MO-Missouri"       "MT-Montana"        "NE-Nebraska"       "NV-Nevada"         "NH-New Hampshire"
"NJ-New Jersey"     "NM-New Mexico"     "NY-New York"
[33] "NC-North Carolina" "ND-North Dakota"   "OH-Ohio"           "OK-Oklahoma"       "OR-Oregon"
"PA-Pennsylvania"   "RI-Rhode Island"   "SC-South Carolina"
[41] "SD-South Dakota"   "TN-Tennessee"      "TX-Texas"          "UT-Utah"           "VT-Vermont"
"VA-Virginia"       "WA-Washington"     "WV-West Virginia"
[49] "WI-Wisconsin"      "WY-Wyoming"

> gsub('^(.*)-(.*)$','\\2-\\1',vSTATES)
 [1] "Alabama-AL"        "Alaska-AK"         "Arizona-AZ"        "Arkansas-AR"       "California-CA"
"Colorado-CO"       "Connecticut-CT"    "Delaware-DE"
 [9] "Florida-FL"        "Georgia-GA"        "Hawaii-HI"         "Idaho-ID"          "Illinois-IL"
"Indiana-IN"        "Iowa-IA"           "Kansas-KS"
[17] "Kentucky-KY"       "Louisiana-LA"      "Maine-ME"          "Maryland-MD"       "Massachusetts-MA"
"Michigan-MI"       "Minnesota-MN"      "Mississippi-MS"
[25] "Missouri-MO"       "Montana-MT"        "Nebraska-NE"       "Nevada-NV"         "New Hampshire-NH"
"New Jersey-NJ"     "New Mexico-NM"     "New York-NY"
[33] "North Carolina-NC" "North Dakota-ND"   "Ohio-OH"           "Oklahoma-OK"       "Oregon-OR"
"Pennsylvania-PA"   "Rhode Island-RI"   "South Carolina-SC"
[41] "South Dakota-SD"   "Tennessee-TN"      "Texas-TX"          "Utah-UT"           "Vermont-VT"
"Virginia-VA"       "Washington-WA"     "West Virginia-WV"
[49] "Wisconsin-WI"      "Wyoming-WY"
>
```

S3/S4 Objects and Object Oriented Programming

S3/S4 Objects and Object Oriented Programming

Have you been continually amaze by the `print()` function throughout this lecture series? You may have wondered why I continually used the `print()` function to print out vectors or data frames instead of just using typing in the variable's name on the R command line. Both do the same thing:

```
> print(dfFatKids)
  FirstName Height Weight FattyIndex       BMI
1    ALBERT     45    150     3.3333  52.07407
2  ROSEMARY     35    123     3.5143  70.58694
3     TOMMY     78    167     2.1410  19.29668
4     BUDDY     12    189    15.7500 922.68750
5   FARQUAR     76    198     2.6053  24.09868
6     SIMON     87    256     2.9425  23.77699
7    LAUREN     54    876    16.2222 211.18930
> dfFatKids
  FirstName Height Weight FattyIndex       BMI
1    ALBERT     45    150     3.3333  52.07407
2  ROSEMARY     35    123     3.5143  70.58694
3     TOMMY     78    167     2.1410  19.29668
4     BUDDY     12    189    15.7500 922.68750
5   FARQUAR     76    198     2.6053  24.09868
6     SIMON     87    256     2.9425  23.77699
7    LAUREN     54    876    16.2222 211.18930
>
```

Here, dfFatKids is a data frame. But, haven't we used the `print()` function to print out results from a linear regression? For example,

S3/S4 Objects and Object Oriented Programming


```
> lrFATKIDS <- lm(Height ~ Weight,data=dfFatKids)
> print(lrFATKIDS)

Call:
lm(formula = Height ~ Weight, data = dfFatKids)

Coefficients:
(Intercept)       Weight
  53.529156     0.006277

>
```

Isn't it interesting that when using the `print()` function with data frames, the printed results are in a completely different format than when using the `print()` function with a variable containing the results from a linear regression? Why is this?

The answer comes from the `class()` function:

```
> class(dfFatKids)
[1] "data.frame"
> class(lrFATKIDS)
[1] "lm"
>
```

As you see, the `class()` for a data frame returns `data.frame`, whereas the `class()` function for a variable containing the results from a linear regression is `lm`. So, the printed results vary depending on the class you are working with.

S3/S4 Objects and Object Oriented Programming

Each object – a data frame, a vector, a linear regression, a time series, etc. – in R is associated with a specific class – `data.frame`,`numeric`/`character`/`logical`, `lm`, `ts`, etc. Unlike a function in R, which takes certain inputs, performs a task, and outputs the result, a *class* is a template for an object that contains functions (called *methods* in the OOP world), attributes (called *properties* in the OOP world) and additional variables. Each time you create a variable containing the results from the linear regression `lm()` function, say, you are creating an *instance* of the `lm` class. Once you have an instance of a class, you can then call any of the methods associated with that class as well as access information via properties.

The `print()` function is not just a lone function hanging out in R; it is a method associated with many (if not all) of the classes available in R. That is, there is a print-specific method for the `lm` class; there is a print-specific method for the `ts` class; there is a print-specific function for the `data.frame` class, etc. And, based on the class of the variable(s) placed in the `print()` function's arguments, that class's print-specific function is the one that is called.

To put it another way, there is NOT one huge `print()` function that is modified or updated every time an R programmer creates a new class. That would be silly!

To see what methods are associate with what class, you can use the `methods()` function:

S3/S4 Objects and Object Oriented Programming


```
> methods(class=data.frame)
 [1] $<-.data.frame           [.data.frame             [[.data.frame            [[<-.data.frame          [<-.data.frame
aggregate.data.frame
 [7] anyDuplicated.data.frame as.data.frame.data.frame as.list.data.frame       as.matrix.data.frame     by.data.frame
cbind.data.frame
[13] dim.data.frame           dimnames.data.frame      dimnames<-.data.frame    duplicated.data.frame    edit.data.frame*
format.data.frame
[19] formula.data.frame*      head.data.frame*         is.na.data.frame         Math.data.frame          mean.data.frame
melt.data.frame
[25] merge.data.frame         na.exclude.data.frame*   na.omit.data.frame*      Ops.data.frame           plot.data.frame*
print.data.frame
[31] prompt.data.frame*       rbind.data.frame         rescaler.data.frame      row.names.data.frame     row.names<-
.data.frame    rowsum.data.frame
[37] split.data.frame         split<-.data.frame       stack.data.frame*        str.data.frame*          subset.data.frame
summary.data.frame
[43] Summary.data.frame       t.data.frame             tail.data.frame*         transform.data.frame     unique.data.frame
unstack.data.frame*
[49] within.data.frame

   Non-visible functions are asterisked

> methods(class=lm)
 [1] add1.lm*           alias.lm*          anova.lm           case.names.lm*     confint.lm*        cooks.distance.lm*
deviance.lm*       dfbeta.lm*         dfbetas.lm*
[10] drop1.lm*          dummy.coef.lm*     effects.lm*        extractAIC.lm*     family.lm*         formula.lm*
hatvalues.lm       influence.lm*      kappa.lm
[19] labels.lm*         logLik.lm*         model.frame.lm     model.matrix.lm    plot.lm            predict.lm
print.lm           proj.lm*           residuals.lm
[28] rstandard.lm       rstudent.lm        simulate.lm*       summary.lm         variable.names.lm* vcov.lm*

   Non-visible functions are asterisked
```

As you see above, there is a print method associated with the `data.frame` class, called `print.data.frame`; and there is a print method associate with the `lm` class, called `print.lm`.

S3/S4 Objects and Object Oriented Programming

So, why use classes and object-oriented programming (OOP)? Object-oriented programming leads to code that is faster to write, easier to maintain and less likely to contain errors. OOP has been around for quite a while is an is not something that is a flash-in-the-pan concept. Many of the programs you use on a daily basis are written using the OOP paradigm.

But, if you are a casual user of R and just read in data, perform some data manipulations, compute some statistics and print out results and/or graphs, you may not necessarily need to use the OOP paradigm in R. But, if you are going to write a package, either within your organization or in the outside world, you will definitely want to look into the OOP paradigm in R.

There are two different flavors of OOP in R: S3 and S4. These do similar things, but S3 was added to R in 1990 when version 3 of S was released (S3). When version 4 of S was released (S4) around 1991, a stricter form of OOP was implemented in S. We will not discuss the S3 methods since they are older, but will talk about the new S4 implementation of OOP.

To create an S4 class, we use the `setClass()` function, and to create a method within our S4 class, we use a combination of `setGeneric()` and `setMethod()` functions. There are additional functions, but we will not discuss them in this lecture.

S3/S4 Objects and Object Oriented Programming

Let's create a class that holds the height and weight needed to compute the Body Mass Index (BMI). We will call this class *bmi*.

This means that we will need to have two variables, or *slots*, in our class: one to hold the weight and one to hold the height. The height variable will be called *ht*, and the weight variable will be called *wt*.

Next, let's create a method associated with our class, called *print*, that computes the BMI in the usual manner and spits it out.

To create the class, we use the `setClass()` function:

```
setClass("bmi",
         representation(ht="numeric",
                        wt="numeric")
        )
```

The first parameter to `setClass()` is the name of the class in quotes. The second parameter is the `representation()` function containing the names of the slots along with the mode for each slot, in this case, numeric.

S3/S4 Objects and Object Oriented Programming

We can now create a variable of class *bmi* by using the `new()` function. This function creates an instance of the class *bmi*:

```
> bmi1 <- new("bmi",ht=45,wt=150)
> class(bmi1)
[1] "bmi"
```

We can see some information about our bmi variable bmi1, by just entering bmi1 at the R command line:

```
> bmi1
An object of class "bmi"
Slot "ht":
[1] 45
Slot "wt":
[1] 150
```

Now, if you wanted to access just the ht slot, you can use the following notation:

```
> bmi1@wt
[1] 150
> bmi1@ht
[1] 45
>
```

S3/S4 Objects and Object Oriented Programming

Next, let's create our *print* function. First, we create a "normal" function, but call it *method_name.class_name*:

```
print.bmi <- function(object) {
 return( 703*object@wt/(object@ht^2) )
}
```

Note the use of the parameter `object` above. This is a *bmi* object coming through as an argument, so the slots are available to you.

Next, we associate this function with our class using the `setMethod()` function:

```
setMethod("print.bmi",
          signature="bmi",
          definition=function(object) {
                      return( 703*object@wt/(object@ht^2) )
                     }
          )
```

Note that the first argument is the name of our function, print.bmi, the signature is the name of our class, and the definition is the function body to print.bmi. Why this has to repeat is not obvious to me, but I'll update this section when I learn more about this.

In any case, let's re-create our bmi1 variable:

S3/S4 Objects and Object Oriented Programming


```
> bmi1 <- new("bmi",ht=45,wt=150)
```

Next, let's use the print() function to see what we get:

```
> print(bmi1)
[1] 52.07407
>
```

As you see, the print function associated with out bmi class computes the BMI and shows it to the screen.

Next, let's see if we can add two BMI class variables together. While this doesn't make sense in real life, let's say for this lecture that adding two *bmi*'s together is equivalent to the normal BMI calculation performed on the heights added together and the weights added together. Now, sum is associated with the plus-sign (+), so let's define the plus-sign for the *bmi* class as follows:

```
setMethod("Arith",
    signature(e1="bmi",e2="bmi"),
    definition=function(e1,e2) {
                 return( 703*(e1@wt+e2@wt)/( (e1@ht+e2@ht)^2) )
                }
)
```

The *Arith* class is the class associated with +, -, etc. We need to modify that class to get our class addition to work. Here is an example:

S3/S4 Objects and Object Oriented Programming


```
> bmi1 <- new("bmi",ht=45,wt=150)
> bmi2 <- new("bmi",ht=35,wt=123)
> bmi1 + bmi2
[1] 29.98734
```

There's much more to this than what I have shown. Please see the appendix for references.

R Startup Parameters

R Startup Parameters

In this section, we discuss the R startup parameters available at the Windows/Linux command line as well as memory-related functions.

If you type R –help at the Windows or Linux command line, you will receive the following helpful information about how to run R:

```
Usage: Rterm [options] [< infile] [> outfile] [EnvVars]

Start R, a system for statistical computation and graphics, with the
specified options

EnvVars: Environmental variables can be set by NAME=value strings

Options:
  -h, --help            Print usage message and exit
  --version             Print version info and exit
  --encoding=enc        Specify encoding to be used for stdin
  --save                Do save workspace at the end of the session
  --no-save             Don't save it
  --no-environ          Don't read the site and user environment files
  --no-site-file        Don't read the site-wide Rprofile
  --no-init-file        Don't read the .Rprofile or ~/.Rprofile files
  --restore             Do restore previously saved objects at startup
  --no-restore-data     Don't restore previously saved objects
  --no-restore-history  Don't restore the R history file
  --no-restore          Don't restore anything
  --vanilla             Combine --no-save, --no-restore, --no-site-file,
                          --no-init-file and --no-environ
```

R Startup Parameters


```
  --min-vsize=N         Set vector heap min to N bytes; '4M' = 4 MegaB
  --max-vsize=N         Set vector heap max to N bytes;
  --min-nsize=N         Set min number of cons cells to N
  --max-nsize=N         Set max number of cons cells to N
  --max-mem-size=N      Set limit for memory to be used by R
  --max-ppsize=N        Set max size of protect stack to N
  -q, --quiet           Don't print startup message
  --silent              Same as --quiet
  --slave               Make R run as quietly as possible
  --verbose             Print more information about progress
  --internet2           Use Internet Explorer for proxies etc.
  --args                Skip the rest of the command line
  --ess                 Don't use getline for command-line editing
                          and assert interactive use
  -f file               Take input from 'file'
  --file=file           ditto
  -e expression         Use 'expression' as input

One or more -e options can be used, but not together with -f or --file

An argument ending in .RData (in any case) is taken as the path
to the workspace to be restored (and implies --restore)
```

R Startup Parameters


```
Or: R CMD command args

where 'command' is one of:
  INSTALL  Install add-on packages.
  REMOVE   Remove add-on packages.
  SHLIB    Make a DLL for use with dyn.load.
  BATCH    Run R in batch mode.
  build    Build add-on packages.
  check    Check add-on packages.
  Rprof    Post process R profiling files.
  Rdconv   Convert Rd format to various other formats.
  Rdiff    difference R output files.
  Rd2dvi   Convert Rd format to DVI.
  Rd2pdf   Convert Rd format to PDF.
  Rd2txt   Convert Rd format to pretty text.
  Sd2Rd    Convert S documentation to Rd format.
  Stangle  Extract S/R code from Sweave documentation.
  Sweave   Process Sweave documentation.
  config   Obtain configuration information about R.
  open     Open a file via Windows file associations.

Use
  R CMD command --help
for usage information for each command.
```

As you see, there are a lot of start up flags. We've talked a little about the `R CMD command args` concept when building our own package, so we'll skip these. We will concentrate on the six memory-related command line flags: `min-vsize=N`, `max-vsize=N`, `min-nsize=N`, `max-nsize=N`, `max-mem-size=N` and `max-ppsize=N`.

R Startup Parameters

From ?Memory-limits:

R holds all objects in memory, and there are limits based on the amount of memory that can be used by all objects. Error messages beginning with "cannot allocate vector of size" indicate a failure to obtain memory, either because the size exceeded the address-space limit for a process or, more likely, because the system was unable to provide the memory. Note that on 32-bit OS there may well be enough free memory available, but not a large enough contiguous block of address space into which to map it.

There are also limits on individual objects. On all versions of R, the maximum length (number of elements) of a vector is 2^31-1 = 2,147,483,647, as lengths are stored as signed integers. In addition, the storage space cannot exceed the address limit, and if you try to exceed that limit, the error message begins "cannot allocate vector of length". The number of characters in a character string is in theory only limited by the address space.

Unix

The address-space limit is system-specific: 32-bit OSes imposes a limit of no more than 3Gb: it is often 3Gb of less. Running 32-bit executables on a 64-bit OS will have similar limits: 64-bit executables will have an essentially infinite system-specific limit. See the commands limit or ulimit for how to impose limitations on the resources available to a single process. For example: ulimit –t 600 –m 20000000 limits a process to 10 minutes of CPU time and approx. 2GB of memory.

R Startup Parameters

Windows

The address space limit is 2Gb under 32-bit Windows unless the OS's default has been changed to allow more (up to 3Gb). See how to change the PAE setting on Windows. Under most 64-bit versions of Windows, the limit is 4Gb. Unlike Unix ulimit, Windows provides no analog, so R imposes its own limits. See ?memory.size and ?memory.limit in the help documentation. These are explained below.

The `memory.size()` and `memory.limit()` Functions

The `memory.size()` function reports the current or maximum memory allocation of the malloc function used in this version of R. The `memory.limit()` function reports, or increases, the limit in force on the total allocation.

`memory.size(max=TRUE|FALSE)` – If TRUE, then reports the *maximum* amount of memory obtained from the OS; if FALSE, reports the amount *currently* in use; if NA, the memory limit.

memory.limit(size=NA|n) – If NA, reports the memory limit in Mb; if N, values up to 4095 are allowed (4095Mb=4Gb).

The command-line flag `--max-mem-size` sets the maximum value of obtainable memory. Note that this cannot exceed 3Gb on 32-bit Windows, and most versions of Windows are limited to 2Gb.

R Startup Parameters

For example, on Windows Vista 64-bit with 4GB RAM:

```
> memory.size()
[1] 39.39
> memory.limit()
[1] 3583
> memory.size(max=NA)
[1] 3583.88
> memory.size(max=TRUE)
[1] 41.25
> memory.size(max=FALSE)
[1] 39.39
>
```

To understand the command line flags, you need to know that R maintains separate areas for fixed- and variable-sized objects. The first of these is allocated as an array of *cons cells*, and the second are thrown on a *heap of Vcells* of 8bytes each. Each *cons cell* occupies 28 bytes on a 32-bit machine, and usually 56 bytes on a 64-bit machine. The --min-nsize/--max-nsize command line flags can be used to specify the number of cons cells, and the --min-vsize/--max-vsize can be used to specify the size of the vector heap in bytes. Both options must be integers, or integers followed by G, M, K or k. The "min" flags set the minimal sizes for the number of cons cells and for the vector heap. These values are also the initial values, but thereafter R will grow or shrink the areas depending on usage, but never exceeding the limits set by the "max" flags.

The default values are currently minima of 350k cons cells, 6Mb of vector heap

R Startup Parameters

and no maxima (other than machine resources). The maxima can be changed during an R session by calling `mem.limits(nsize=#,vsize=#)`.

You can find out the current memory consumption (the heap and cons cells used as numbers and megabytes), by typing `gc()` at the R prompt:

```
> gc()
          used (Mb) gc trigger (Mb) max used (Mb)
Ncells  379267 10.2     667722 17.9   667722 17.9
Vcells 1890506 14.5    3125593 23.9  2900563 22.2
>
```

Note that if you delete a large object, you may want to call `gc()` as this may prompt R to return memory to the OS. The command-line option `--max-ppsize` controls the maximum size of the pointer protection stack. This defaults to 50000, but can be increased to allow deep recursion or large and complicated calculations to be done. Note that parts of the garbage collection process goes through the full reserved pointer protection stack and hence becomes slower when the size is increased. Currently, the maximum value accepted is 500000.

You can list the counts of the cons cells by types using the `memory.profile()` function:

```
> memory.profile()
      NULL     symbol   pairlist    closure environment    promise   language    special    builtin       char
logical    integer     double    complex
         1       8322     149377       3716         913       5325      45791        118       1191     122755
4238       6224       4309          2
  character        ...        any       list expression   bytecode externalptr    weakref        raw         S4
      23407          0          0       2538          1          0        679        173        174        834
>
```

R Startup Parameters

You can determine an estimate of the memory (in bytes) that is being used to store an R object by using the object.size(x) function, where x is the name of an object such as a vector or data frame. For example, to size how much memory in bytes being used up by the dfDJIA data frame:

```
> object.size(dfDJIA)
329816 bytes
```

To get this number reported in megabytes, try this:

```
> print(object.size(dfDJIA),quote=FALSE,units="Mb")
0.3 Mb
```

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.

Appendix C: Useful Functions

The following commands are used to determine R-related environment variables:

Environment Variable: R_HOME

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

Environment Variable: R_LIBS (the library subdirectory of R_HOME)

```
> .Library
[1] "C:/PROGRA~2/R/R-210~1.1/library"
```

Session Information

```
> sessionInfo()
R version 2.10.1 (2009-12-14)
i386-pc-mingw32

locale:
[1] LC_COLLATE=English_United States.1252  LC_CTYPE=English_United States.1252
LC_MONETARY=English_United States.1252 LC_NUMERIC=C
[5] LC_TIME=English_United States.1252

attached base packages:
[1] stats     graphics  grDevices datasets  grid      utils     methods   base

other attached packages:
[1] digest_0.4.2   reshape_0.8.3  plyr_0.1.9     proto_0.3-8    rcom_2.2-3     rscproxy_1.3-1

loaded via a namespace (and not attached):
[1] ggplot2_0.8.7 tools_2.10.1
```

Appendix C: Useful Functions

It is impossible to list all the environment variables which can affect an R session: some affect the OS system functions which R uses, and others will affect add-on packages. But here are notes on some of the more important ones. Those that set the defaults for options are consulted only at startup (as are some of the others).

- HOME:The user's 'home' directory.
- LANGUAGE:Optional. The language(s) to be used for message translations. This is consulted when needed.
- LC_ALL:(etc) Optional. Use to set various aspects of the locale – see Sys.getlocale. Consulted at startup.
- R_BATCH:Optional – set in a batch session.
- R_BROWSER:The path to the default browser. Used to set the default value of options("browser").
- R_COMPLETION:Optional. If set to FALSE, command-line completion is not used. (Not used by Mac OS GUI.)
- R_DEFAULT_PACKAGES:A comma-separated list of packages which are to be loaded in every session. See options.
- R_DOC_DIR:The location of the R 'doc' directory. Set by R.
- R_DVIPSCMD:The path to dvips. Defaults to the value of DVIPS, and if that is unset to a value determined when R was built. Used by R CMD Rd2dvi.
- R_ENVIRON:Optional. The path to the site environment file: see Startup. Consulted at startup.
- R_GSCMD:Optional. The path to GhostScript, used by dev2bitmap, bitmap and embedFonts. Consulted when those functions are invoked.
- R_HISTFILE:Optional. The path of the history file: see Startup. Consulted at startup and when the history is saved.
- R_HISTSIZE:Optional. The maximum size of the history file, in lines. Exactly how this is used depends on the interface. For Rgui it controls the number of lines saved to the history file: the size of the history used in the session is controlled by the console customization: see Rconsole.
- R_HOME:The top-level directory of the R installation: see R.home. Set by R.
- R_INCLUDE_DIR:The location of the R 'include' directory. Set by R.
- R_LATEXCMD:The path to latex. Defaults to the value of LATEX, and if that is unset to a value determined when R was built. Used by R CMD Rd2dvi.
- R_LIBS:Optional. Used for initial setting of .libPaths.
- R_LIBS_SITE:Optional. Used for initial setting of .libPaths.
- R_LIBS_USER:Optional. Used for initial setting of .libPaths.

…continued on next slide…

Appendix C: Useful Functions

- R_MAKEINDEXCMD:The path to makeindex. Defaults at startup to the value of MAKEINDEX, and if that is unset to a value determined when R was built. Used by R CMD Rd2dvi.
- R_PAPERSIZE:Optional. Used to set the default for options("papersize"), e.g. used by pdf and postscript.
- R_PDFLATEXCMD:The path to pdflatex. Defaults at startup to the value of PDFLATEX, and if that is unset to a value determined when R was built. Used by R CMD Rd2dvi.
- R_PDFVIEWER:The path to the default PDF viewer. Used by R CMD Rd2dvi.
- R_PLATFORM:The platform – a string of the form cpu-vendor-os, see R.Version.
- R_PROFILE:Optional. The path to the site profile file: see Startup. Consulted at startup.
- R_RD4DVI:Options for latex processing of Rd files. Used by R CMD Rd2dvi.
- R_RD4PDF:Options for pdflatex processing of Rd files. Used by R CMD Rd2dvi.
- R_SHARE_DIR:The location of the R 'share' directory. Set by R.
- R_TEXI2DVICMD:The path to texi2dvi. Defaults to the value of TEXI2DVI, and if that is unset to a value determined when R was built.
- R_UNZIPCMD:The path to unzip. Sets the iniital value for options("unzip") on a Unix-alike when package utils is loaded.
- R_ZIPCMD:The path to zip. Only used in R itself by R CMD INSTALL --build on Windows.
- TMPDIR, TMP, TEMP:Consulted (in that order) when setting the temporary directory for the session: see tempdir. TMPDIR is also used by some of the utilities see the help for build.
- TZ:Optional. The current timezone. See Sys.timezone for the system-specific formats. Consulted as needed.
- GSC:Optional: the path to GhostScript, used if R_GSCMD is not set.
- R_USER:The user's 'home' directory. Set by R. (HOME will be set to the same value if not already set.)
- TZDIR:Optional. The top-level directory of the timezone database. See Sys.timezone.

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`.