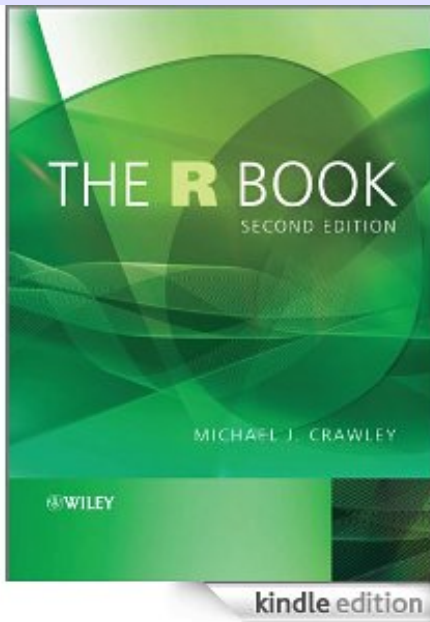


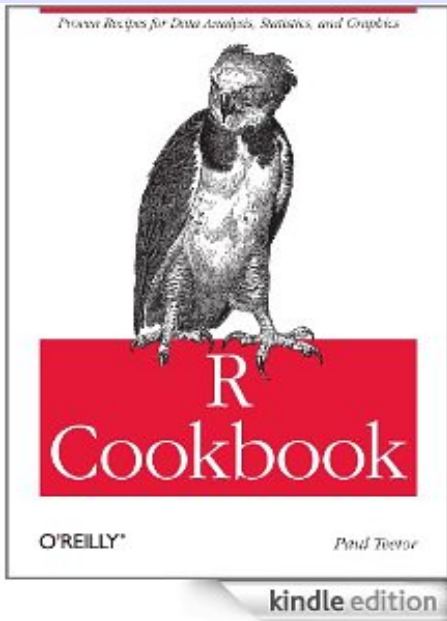
# Learning *R*

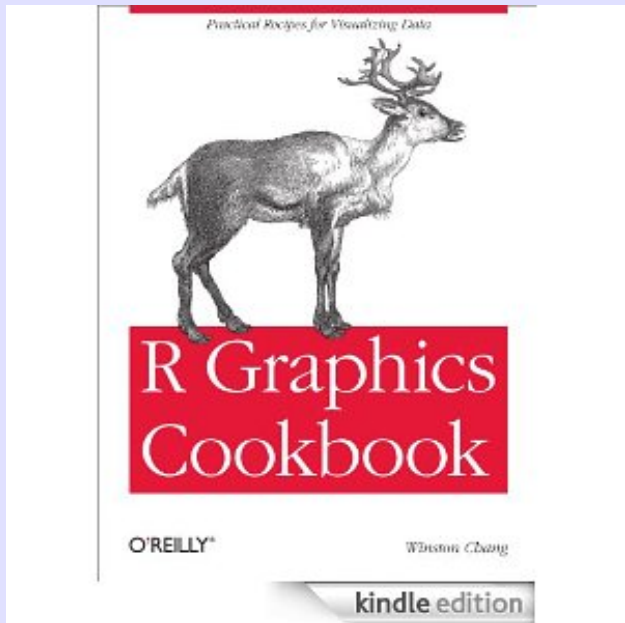
Carl James Schwarz

Department of Statistics and Actuarial Science  
Simon Fraser University  
Burnaby, BC, Canada  
cschwarz @ stat.sfu.ca

*R* is a free, open source statistical package that is increasingly being used in many fields. While *R* is free, it is not cheap – meaning that mastering *R* requires some time investment. It is particularly helpful to have some guidance in the more complex applications of *R*.









## The R Manuals

*edited by the R Development Core Team.*

The following manuals for R were created on Debian Linux and may differ from the manuals for Mac or Windows on platform-specific pages, but most parts will be identical for all platforms. The correct version of the manuals for each platform are part of the respective R installations. The manuals change with R, hence we provide versions for the most recent released R version (R-release), a very current version for the patched release version (R-patched) and finally a version for the forthcoming R version that is still in development (R-devel). Here they can be downloaded as PDF files or directly browsed as HTML:

Manual	R-release	R-patched	R-devel
<b>An Introduction to R</b> is based on the former "Notes on R", gives an introduction to the language and how to use R for doing statistical analysis and graphics.	<a href="#">HTML</a>   <a href="#">PDF</a>	<a href="#">HTML</a>   <a href="#">PDF</a>	<a href="#">HTML</a>   <a href="#">PDF</a>
<b>R Data Import/Export</b> describes the import and export facilities available either in R itself or via packages which are available from CRAN.	<a href="#">HTML</a>   <a href="#">PDF</a>	<a href="#">HTML</a>   <a href="#">PDF</a>	<a href="#">HTML</a>   <a href="#">PDF</a>
<b>R Installation and Administration</b>	<a href="#">HTML</a>   <a href="#">PDF</a>	<a href="#">HTML</a>   <a href="#">PDF</a>	<a href="#">HTML</a>   <a href="#">PDF</a>
<b>Writing R Extensions</b> covers how to create your own packages, write R help files, and the foreign language (C, C++, Fortran, ...) interfaces.	<a href="#">HTML</a>   <a href="#">PDF</a>	<a href="#">HTML</a>   <a href="#">PDF</a>	<a href="#">HTML</a>   <a href="#">PDF</a>
A draft of <b>The R language definition</b> documents the language <i>per se</i> . That is, the objects that it works on, and the details of the expression evaluation process, which are useful to know when programming R functions.	<a href="#">HTML</a>   <a href="#">PDF</a>	<a href="#">HTML</a>   <a href="#">PDF</a>	<a href="#">HTML</a>   <a href="#">PDF</a>
<b>R Internals</b> : a guide to the internal structures of R and coding standards for the core team working on R itself.	<a href="#">HTML</a>   <a href="#">PDF</a>	<a href="#">HTML</a>   <a href="#">PDF</a>	<a href="#">HTML</a>   <a href="#">PDF</a>
<b>The R Reference Index</b> : contains all help files of the R standard and recommended packages in printable form. (9MB, approx. 3500 pages)	<a href="#">PDF</a>	<a href="#">PDF</a>	<a href="#">PDF</a>

<http://cran.r-project.org/manuals.html>.

## Contributed Documentation

[English](#) --- [Other Languages](#)

Manuals, tutorials, etc. provided by users of R. The R core team does not take any responsibility for contents, but we appreciate the effort very much and encourage everybody to contribute to this list! To submit, follow the submission instructions on the [CRAN main page](#). All material below is available directly from CRAN, you may also want to look at the list of [other R documentation](#) available on the Internet.

**Note:** Please use the [directory listing](#) to sort by name, size or date (e.g., to see which documents have been updated lately).

### English Documents

Documents with more than 100 pages:

- **“Using R for Data Analysis and Graphics - Introduction, Examples and Commentary”** by John Maindonald ([PDF](#), data sets and scripts are available at [JM's homepage](#)).
- **“Practical Regression and Anova using R”** by Julian J. Faraway ([PDF](#), data sets and scripts are available at the [book homepage](#)).
- The [Web Appendix](#) to the book “An R and S-PLUS Companion to Applied Regression” by John Fox contains information about using R (and S-PLUS) to fit a variety of regression models.
- **“An Introduction to S and the Hmisc and Design Libraries”** by Carlos Alzola and Frank E. Harrell, especially of interest to SAS users of the Hmisc or Design packages, or R users interested in data manipulation, recoding, etc. ([PDF](#)).
- **“Statistical Computing and Graphics Course Notes”** by Frank E. Harrell, includes material on S, LaTeX, reproducible research, making graphs, brief overview of computer languages, etc. ([PDF](#)).
- **“An Introduction to R: Software for Statistical Modelling & Computing”** by Petra Kuhnert and Bill Venables ([ZIP 3.8MB](#)): A 360 PDF document of lecture notes in combination with the data sets and R scripts used in the manuscript.
- **“Introduction to the R Project for Statistical Computing for Use at the ITC”** by David Rossiter ([PDF](#), 2012-08-20, 141 pages).
- **“Analysis of Epidemiological Data Using R and Epicalc”** by Virasakdi Chongsuvivatwong ([PDF](#)).
- **“Statistics Using R with Biological Examples”** by Kim Seefeld and Ernst Linder ([PDF](#)).
- **“IcebreakerR”** by Andrew Robinson ([PDF](#), 2008-05-08).
- **“Applied Statistics for Bioinformatics Using R”** by Wim Krijnen ([PDF](#), 2009-11-17, 278 pages).
- **“An Introduction to R”** by Longhow Lam ([PDF](#), 2010-10-28, 212 pages).
- **“R and Data Mining: Examples and Case Studies”** by Yanchang Zhao ([PDF](#), 2013-04-26, 160 pages).

<http://cran.r-project.org/other-docs.html>.



Author: [Carl James Schwarz](#) [P.Stat.](#)  
Phone: (778) 782-3376  
Office: K 10559

[Professional Statistician \(P.Stat.\)](#)  
[More information on Accreditation of Statisticians in Canada](#)

Department of Statistics & Actuarial Science  
Simon Fraser University

# Course Notes for Beginning and Intermediate Statistics

This web page contains course notes for a variety of courses that I have taught at SFU and elsewhere. The goals of these courses are to:

- Learn how to design and analyse observational and experimental data using common statistical tools such as survey sampling, regression analysis, logistic regression, poisson regression, experimental design and analysis, and categorical (chi-square) analyses. This website is orientated towards examples in the biological sciences, but should be accessible to all users.
- Provide sample code on how to do the analyses using JMP, R, or SAS.
- [Enjoy the subject of statistics rather than dreading it.](#)

## Course Notes

- [FAQ](#) - How do I download copies of scientific papers from the web? How do I print the notes with two pages per sheet?
- [How do I get R, JMP, or SAS?](#)
- [SAS tricks and tips](#)
- [Chapter 0](#) - A review of introductory statistics and probability.
  - [A Bad Graph Hall of Shame](#)

The course notes below illustrate methods of analysis using JMP, R, or SAS. The programs are located in the [Sample Program Library](#). Note that even though a chapter may not have a version for a package, the program code for the example is often available – I just haven't had time yet to update the notes to include the code and output directly in the notes.

Package	Chapter and sections
---------	----------------------

**JMP R SAS 1 In the beginning...**

- 1.1 Introduction
- 1.2 Effective note taking strategies
- 1.3 It's all  $\Delta$   $\rho$   $\epsilon$   $\epsilon$   $\kappa$   $\delta$  to me
- 1.4 Which computer package?
- 1.5 FAQ - Frequently Asked Question

JMP R SAS 2 Introduction to Statistics

- 2.1 TRRGET - An overview of statistical inference
- 2.2 Parameters, Statistics, Standard Deviations, and Standard Errors
- 2.3 Confidence Intervals

# Online Resources on R



Department of Statistics & Actuarial Science  
Simon Fraser University

Author: [Carl James Schwarz](#) P.Stat.  
Phone: 778.782.3376  
Office: K 10559

[Professional Statistician \(P.Stat.\)](#)  
[More information on Accreditation of Statisticians in Canada](#)

## Datasets; SAS, R, and JMP programs; Output

Use the **find** feature of your browser to locate the datafile of interest. Refer to the [course notes](#) for details on the analysis of the datasets listed here.

A complete zip archive of the entire contents is available [here](#). An Excel workbook that contains all of the raw data is also available ( [ALLofDATA.xls](#)) **Being renovated-**

Description	Topics covered	SAS Program and output	JMP/Excel	R
Barley Yields	Simple Random Sample, Basic descriptive Statistics, Side-by-side box plots; dot-plots; histograms	<a href="#">ddt.sas</a> , <a href="#">ddt.csv</a> , <a href="#">ddt.pdf</a> <a href="#">ddt2g.sas</a> , <a href="#">ddt2g.csv</a> , <a href="#">ddt2g.pdf</a> <a href="#">druglib.sas</a> , <a href="#">druglib.csv</a> , <a href="#">druglib.pdf</a> <a href="#">barleyyields.sas</a> <a href="#">barleyyields.pdf</a>	<a href="#">ddt.jmp</a> <a href="#">ddt2g.jmp</a> NO JMP file for DrugLib <a href="#">barleyyields.txt</a> <a href="#">barleyyields.jmp</a>	<a href="#">ddt.r</a> , <a href="#">ddt.csv</a> <a href="#">ddt2g.r</a> , <a href="#">ddt2g.csv</a> <a href="#">druglib.r</a> , <a href="#">druglib.csv</a> No R code for barleyyields
Creel survey	Simple Random Sample	<a href="#">creel.sas</a> <a href="#">creel.lst</a>	<a href="#">creel.jmp</a>	<a href="#">creel.csv</a> <a href="#">creel.r</a>
Survey Sample Size Determination	Simple Random Sample	<a href="#">SurveySampleSize.sas</a> <a href="#">SurveySampleSize.lst</a>	<a href="#">SurveySampleSize.xls</a> Also available in the AllOfData workbook.	
Sockeye survey	Stratified Design, Simple Random Sample (SRS) in each stratum	<a href="#">sockeye.sas</a> <a href="#">sockeye.lst</a>	<a href="#">sockeye.jmp</a>	<a href="#">sockeye.csv</a> <a href="#">sockeye.r</a>
Caribou survey	Sample allocation in Stratified Design with Simple Random Sampling in each strata		<a href="#">caribou.xls</a> - See the caribou tab in the AllOfData workbook	
Estimating number of tundra swans	Stratified Design; Simple Random Sample in each stratum	<a href="#">tundra.sas</a> <a href="#">tundra.lst</a>	<a href="#">tundra.jmp</a>	<a href="#">tundra.csv</a> <a href="#">tundra.r</a>
Estimating number of grubs	Post-stratified Design after Simple Random Sample	<a href="#">post-strata-example.sas</a> <a href="#">post-strata-example.lst</a>	<a href="#">post-strata-example.jmp</a>	<a href="#">post-stratify.csv</a> <a href="#">post-stratify.r</a>
Wolf/moose ratio	Simple Random Sample with ratio estimator	<a href="#">wolf.sas</a> <a href="#">wolf.lst</a>	<a href="#">wolf.jmp</a> This includes an example of sample size determination for a ratio estimator.	<a href="#">wolf.csv</a> <a href="#">wolf.r</a>
Grouse numbers	Simple Random Sample with ratio estimator of total	<a href="#">grouse.sas</a> <a href="#">grouse.lst</a>	<a href="#">grouse.jmp</a>	<a href="#">grouse.csv</a> <a href="#">grouse.r</a>

<http://www.stat.sfu.ca/~cschwarz/CourseNotes> ->  
SampleProgramLibrary.

[Questions](#)[Tags](#)[Users](#)[Badges](#)[Unanswered](#)**Tag Info**[info](#)[newest](#)[featured](#)[frequent](#)[votes](#)[active](#)[unanswered](#)**About [r](#)****R**

[R](#) is an open source programming language and software environment for statistical computing and graphics. R is an implementation of the [S programming language](#) combined with lexical scoping semantics inspired by [Scheme](#). R was created by [Ross Ihaka](#) and [Robert Gentleman](#) and is now developed by the [R Development Core Team](#). The R environment is easily extended through a packaging system on [CRAN](#).

**Official CRAN Documentation**

<http://stackoverflow.com/tags/r/info>

## Cookbook for R

[index](#)

## Cookbook for R

Welcome to the Cookbook for R (formerly named *R Cookbook*). The goal of the cookbook is to provide solutions to common tasks and problems in analyzing data.

Most of the code in these pages can be copied and pasted into the R command window if you want to see them in action.

1. [Basics](#)
2. [Numbers](#)
3. [Strings](#)
4. [Formulas](#)
5. [Data input and output](#)
6. [Manipulating data](#)
7. [Statistical analysis](#)
8. [Graphs](#)
9. [Scripts and functions](#)
10. [Tools for experiments](#)

<http://www.cookbook-r.com>

$R$  is free, but not cheap!

- Download and Install BASE *R* from CRAN.
  - You will need administrative access to install and update *R*.
- Download and Install *Rstudio* from Web.
  - A Integrated Development Environment that simplifies using *R*.
  - You will also need the *knitr* package to create HTML notebooks
- Download and Install PACKAGES via *Rstudio*.
  - Packages are user-written extensions to *R*.
  - **ALWAYS** create a personal library for packages. Refer to <http://www.stat.sfu.ca/~cschwarz/CourseNotes/HowGetSoftware.html>
    - Much easier to upgrade to new version of *R*.
    - Much easier to upgrade packages on a regular basis (i.e. monthly). I check for upgrades every time I launch *Rstudio*.



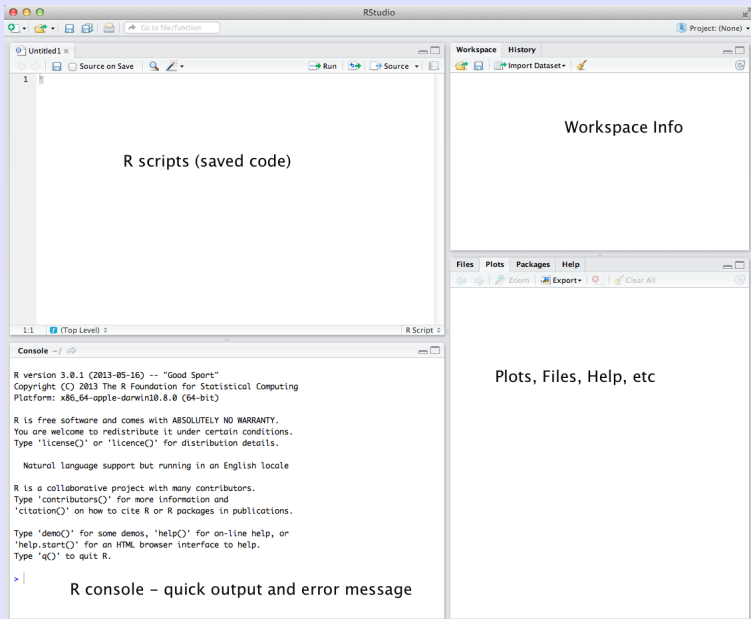
## Updating packages

- About once/month, use *Update packages* under *Rstudio*

## Updating versions of *R*

- *Macintosh*:. Usual download and install.
  - Automatically replaces and deletes older versions of *R*.
  - If you have a personal library, it is left unchanged.
  - Be sure to Update Packages after upgrae.
- *Windoze*:. Download and install *installr* pacakge.
  - Run this in native *R*, i.e. NOT under *Rstudio* using *installr()* or *updateR()* call on console.

# Using Rstudio



Familiarize yourself with the various functions in *Rstudio*

- Setting the directory - lower right window
- **IMPORTANT** Setting the directory where work stored
  - Session → Set Working Directory OR
  - Files → More → Set as Working Directory
- Scripting window (either File → New Script, or File → Open)
  - Run code by placing cursor on line or selecting line, and then  
Code → Run Lines (or use the shortcut, OS dependent)
- Console
- Workspace and History

Quick demo of using *Rstudio*. Enter and run the following code

```
1 # This is a quick demo of using Rstudio
2 x <- 1:10
3 x
4 plot(x,x)
```

# Using Rstudio

The screenshot displays the RStudio application window. The top toolbar includes icons for saving, running, and other functions, along with a 'Go to file/function' search bar. The 'Project: (None)' dropdown is visible in the top right.

The main editor window shows a file named 'Rcode-examples.r' with the following R code:

```
1 # This is a quick demo of using Rstudio-  
2 x <- 1:10-  
3 x-  
4 plot(x,x)
```

The console window at the bottom left shows the output of the R script, indicating that the environment is set up for running R code in an English locale. It also provides instructions on how to use R, including how to get help and how to quit.

The file explorer on the right side shows the current directory structure, including files like 'all.summary.txt', 'defs.tex', 'Images', 'listings.pdf', 'R-manuals', 'Rcourse.aux', 'Rcourse.log', and 'Rcourse.pdf'.

Console output:

```
Natural language support but running in an English locale  
  
R is a collaborative project with many contributors.  
Type 'contributors()' for more information and  
'citation()' on how to cite R or R packages in publications.  
  
Type 'demo()' for some demos, 'help()' for on-line help, or  
'help.start()' for an HTML browser interface to help.  
Type 'q()' to quit R.  
  
> setwd("~/Dropbox/CMI-R-Course")  
> setwd("~/Dropbox/CMI-R-Course")  
> |
```

File Explorer contents:

Name	Size	Modified
..		
all.summary.txt	132.4 KB	Oct 8, 2013, 7:09 PM
defs.tex	368 bytes	Oct 8, 2013, 9:13 PM
Images		
listings.pdf	724.7 KB	Oct 9, 2013, 8:34 PM
R-manuals		
Rcourse.aux	2.8 KB	Oct 9, 2013, 8:40 PM
Rcourse.log	43 KB	Oct 9, 2013, 8:40 PM
Rcourse.pdf	1.6 KB	Oct 9, 2013, 8:40 PM

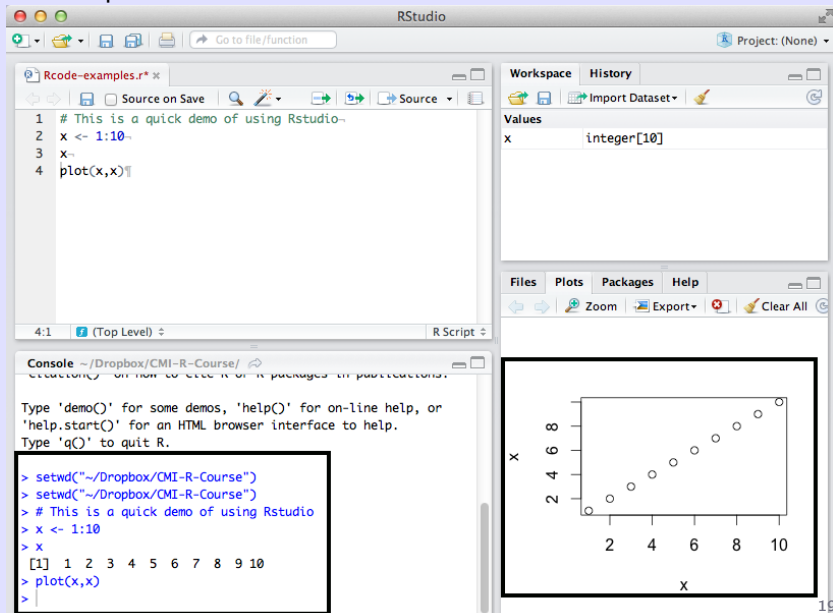
Executing code:

- All lines. Use *Run* button in menu or CNTRL-A. CNTRL-R/CNTRL-Enter
- Selected lines. Select, then *Run* button or CNTRL-R/CNTRL-Enter
- One line; Put cursor in line; then *Run* or CNTRL-R/CNTRL-Enter
- Look at Code → Run Region for other options

In all cases code get copied to the console and is then executed.

# Using Rstudio

Code copied to console and executed.



The screenshot displays the RStudio environment with three main panels:

- Source Panel:** Contains a script file named `Rcode-examples.r*` with the following code:

```
1 # This is a quick demo of using Rstudio~
2 x <- 1:10~
3 x~
4 plot(x,x)~
```
- Console Panel:** Shows the execution of the script. The first three lines are commented out. The fourth line, `plot(x,x)`, has been executed, resulting in the following output:

```
> setwd("~/Dropbox/CMU-R-Course")
> setwd("~/Dropbox/CMU-R-Course")
> # This is a quick demo of using Rstudio
> x <- 1:10
> x
[1] 1 2 3 4 5 6 7 8 9 10
> plot(x,x)
>
```
- Plots Panel:** Displays a scatter plot of `x` versus `x`. The x-axis and y-axis both range from 0 to 10, with major ticks every 2 units. The plot shows 10 data points forming a perfect linear relationship, represented by open circles.

# Using Rstudio- installing the *knitr* package

The screenshot shows the RStudio interface with the 'Install Packages' dialog box open. The dialog box has a tab for 'Repository (CRAN)' and a text field for 'Packages (separate multiple with space or comma):' containing 'knitr'. The 'Install' button is highlighted with a black box. The background shows the R console with the following code:

```
# This is a quick demo of using Rstudio-
x <- 1:10
x
plot(x,x)
```

The console output shows:

```
> setwd("~/Dropbox/CMU-R-Course")
> # This is a quick demo of using Rstudio
> x <- 1:10
> x
[1] 1 2 3 4 5 6 7 8 9 10
> plot(x,x)
>
```

The 'Workspace' pane on the right shows the variable 'x' as an 'integer[10]'. The 'Packages' pane on the right shows a list of installed and available packages, including 'abind', 'bdsmatrix', 'bitops', 'boot', 'boot', 'BSDA', and 'car'.



# Using Rstudio- installing the knitr package

The screenshot shows the RStudio interface with the following components:

- Source Editor:** Contains an R script with the following code:

```
1 # This is a quick demo of using Rstudio~
2 x <- 1:10~
3 x~
4 plot(x,x)~
```
- Console:** Shows the execution of the script and the installation of the knitr package. The output is as follows:

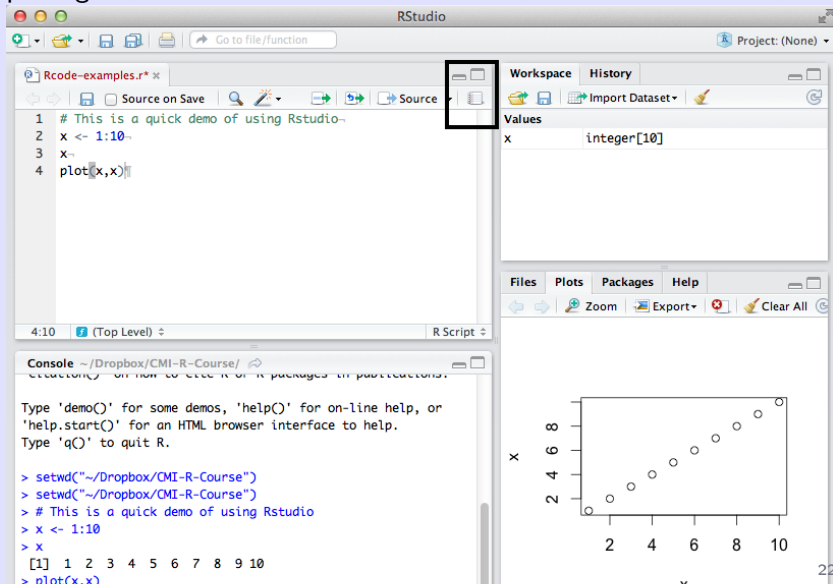
```
> # this is a quick demo of using Rstudio
> x <- 1:10
> x
[1] 1 2 3 4 5 6 7 8 9 10
> plot(x,x)
> install.packages("knitr")
Installing package into '/Users/cschwarz/Rlibs'
(as 'lib' is unspecified)
trying URL 'http://cran.rstudio.com/bin/macosx/contrib/3.0/knitr_1.5.tgz'
Content type 'application/x-gzip' length 862422 bytes (842 Kb)
opened URL
#####
downloaded 842 Kb

The downloaded binary packages are in
/var/folders/mk/xtb91k8m8xjgbx001s7bcdxh0000gp/T//RtmpgsDFIY/downloaded_p
s
> |
```
- Workspace:** Shows the variable 'x' with the value 'integer[10]'.
- Files:** Shows the 'Install Packages' button and a list of available packages.

Package	Description	Version
<a href="#">abind</a>	Combine multi-dimensional arrays	1.4-0
<a href="#">bdsmatrix</a>	Routines for Block Diagonal Symmetric matrices	1.3-1
<a href="#">bitops</a>	Bitwise Operations	1.0-6
<a href="#">boot</a>	Bootstrap Functions (originally by Angelo Canty for S)	1.3-9
<a href="#">boot</a>	Bootstrap Functions (originally by Angelo Canty for S)	1.3-9
<a href="#">BSDA</a>	Basic Statistics and Data Analysis	1.01
<a href="#">sar</a>	Companion to Applied Regression	2.0-19
<a href="#">caTools</a>	Tools: moving window statistics, GIF, Base64, ROC AUC, etc.	1.14
<a href="#">class</a>	Functions for	7.3-9

# Using Rstudio

Make an HTML notebook of code and output. Need the *knitr* package.



The screenshot displays the RStudio environment with the following components:

- Source Editor:** Contains an R script named `Rcode-examples.r*` with the following code:

```
1 # This is a quick demo of using Rstudio~
2 x <- 1:10~
3 x~
4 plot(x,x)
```
- Workspace:** Shows the variable `x` of type `integer[10]`.
- Console:** Displays the execution of the script, including the command `setwd("~/Dropbox/CMU-R-Course/")` and the output of `plot(x,x)`, which is a scatter plot of `x` against `x`.
- Plot:** A scatter plot showing the relationship between `x` and `x`, with data points ranging from 2 to 10 on both axes.

A black box highlights the 'Source' button in the top toolbar, which is used to open or edit R scripts.

# Using Rstudio

The screenshot displays the RStudio environment. The top toolbar includes icons for file operations and a 'Go to file/function' search bar. The 'Project: (None)' dropdown is visible on the right. The main editor window shows a script named 'Rcode-examples.r' with the following code:

```
1 # This is a quick demo of using Rstudio-
2 x <- 1:10-
3 x-
4 plot(x,x)
```

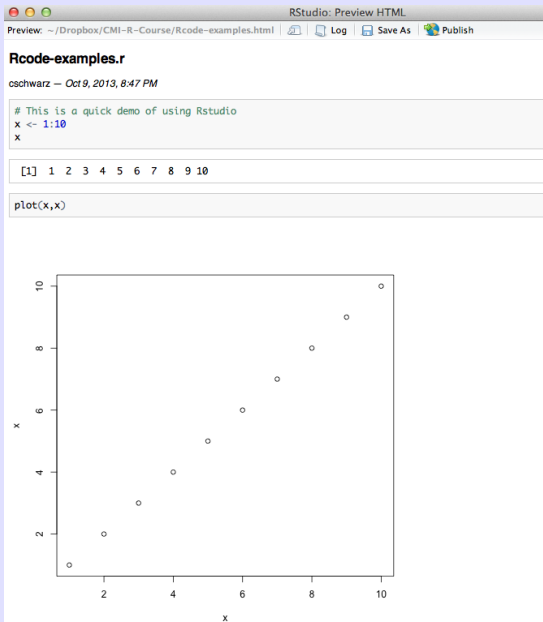
A modal dialog titled 'Compile Notebook from R Script' is open in the center. It explains that a notebook is a standalone HTML file and provides input fields for 'Title (optional): Rcode-examples.r' and 'Author (optional): cschwarz'. The 'Notebook type' is set to '(Default)'. 'Compile' and 'Cancel' buttons are at the bottom.

The console window at the bottom left shows the execution of the script:

```
> setwd("~/Dropbox/CMU-R-Course")
> setwd("~/Dropbox/CMU-R-Course")
> # This is a quick demo of using Rstudio
> x <- 1:10
> x
[1] 1 2 3 4 5 6 7 8 9 10
> plot(x,x)
>
```

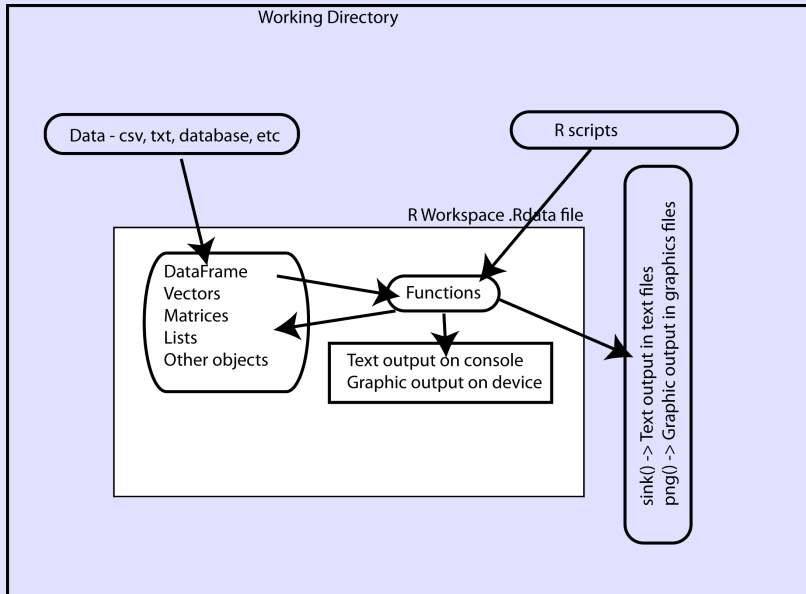
The 'Workspace' pane on the right shows the variable 'x' as an 'integer[10]'. The 'Plots' pane at the bottom right displays a scatter plot of 'x' versus 'x', showing a linear relationship with data points at (1,1), (2,2), (3,3), (4,4), (5,5), (6,6), (7,7), (8,8), (9,9), and (10,10).

# Using Rstudio



# *R* Basics

# The R environment - Overview



Rstudio manages all of the above

# Rstudio - Changing the working directory - I

The screenshot displays the RStudio application window. The top toolbar includes icons for file operations and a 'Go to file/function' search bar. The main editor shows a script named 'Rcode-examples.r' with the following code:

```
1 # This is a quick demo of using Rstudio~
2 x <- 1:10~
3 x~
4 plot(x,x)~
```

The console at the bottom shows the execution of the script:

```
> # This is a quick demo of using Rstudio
> x <- 1:10
> x
[1] 1 2 3 4 5 6 7 8 9 10
> plot(x,x)
> install.packages("knitr")
Installing package into '/Users/cschwarz/Rlibs'
(as 'lib' is unspecified)
trying URL 'http://cran.rstudio.com/bin/macosx/contrib/3.0/knitr_1.5.tar.gz'
Content type 'application/x-gzip' length 862422 bytes (842 Kb)
opened URL
#####
downloaded 842 Kb

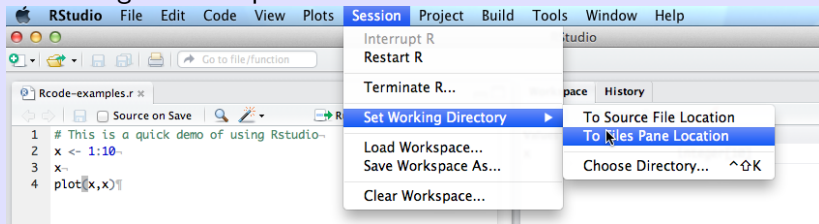
The downloaded binary packages are in
/var/folders/mk/xtb91k8m8xjgbx001s7bcdxh0000gp/T//RtmpgsDfIY/downloade
s
> |
```

The 'Values' pane on the right shows the variable 'x' as an 'integer[10]'. The 'Files' pane at the bottom right shows the file explorer with the following table:

Name	Size	Modified
..		
all.summary.txt	132.4 KB	Oct 8, 2013, 7:09
defs.tex	368 bytes	Oct 8, 2013, 9:13
Images		
listings.pdf	724.7 KB	Oct 9, 2013, 8:34
R-manuals		
R-course.aux	4 KB	Oct 9, 2013, 9:14
R-course.log	46.8 KB	Oct 9, 2013, 9:14
R-course.nav	2.4 KB	Oct 9, 2013, 9:14
R-course.out	0 bytes	Oct 9, 2013, 9:14
R-course.pdf	1.4 MB	Oct 9, 2013, 9:14
R-course.snm	0 bytes	Oct 9, 2013, 9:14
R-course.tex	10.4 KB	Oct 9, 2013, 9:14
R-course.toc	40 bytes	Oct 9, 2013, 9:14
R-course.vrb	117 bytes	Oct 9, 2013, 9:14
Rcode-examples.r	61 bytes	Oct 9, 2013, 8:47

# Rstudio - Changing the working directory - II

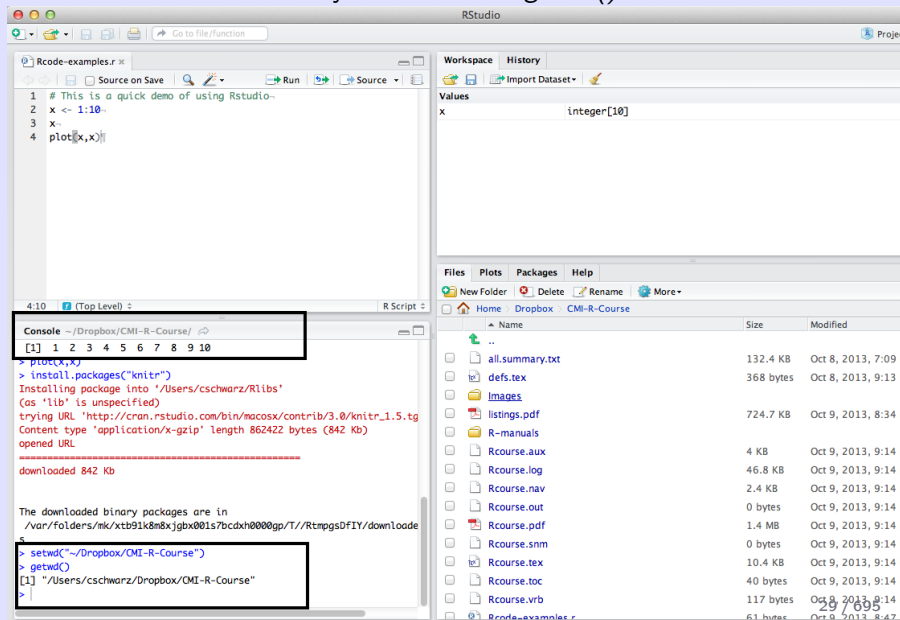
Don't forget this step!





# Rstudio - Changing the working directory - III

Check that correct directory selected. Use `getwd()`. Check console.



The screenshot shows the RStudio interface with the following components:

- Source Editor:** Contains a script named `Rcode-examples.r` with the following code:

```
1 # This is a quick demo of using Rstudio-  
2 x <- 1:10  
3 x  
4 plot(x,x)
```
- Console:** Shows the execution of the script. The prompt `[1]` is followed by the numbers 1 through 10. Below this, the output of `install.packages("knitr")` is shown, indicating the package was successfully installed from CRAN. The console also shows the current working directory as `~/Dropbox/CMi-R-Course/`. The last two lines of the console output are highlighted with a black box:

```
> setwd("~/Dropbox/CMi-R-Course")  
> getwd()  
[1] "~/Users/cschwarz/Dropbox/CMi-R-Course"
```
- Files Panel:** Displays the file explorer for the current project. The files listed include `all.summary.txt`, `defs.tex`, `Images`, `listings.pdf`, `R-manuals`, `Rcourse.aux`, `Rcourse.log`, `Rcourse.nav`, `Rcourse.out`, `Rcourse.pdf`, `Rcourse.snm`, `Rcourse.tex`, `Rcourse.toc`, `Rcourse.vrb`, and `Rcode-examples.r`. The file `Rcode-examples.r` is highlighted.

Set the working directory in *Rstudio* to the *SampleData* directory in the course notes.

Set the working directory in *Rstudio* to the *SampleData* directory in the course notes.

- Don't forget to check this by using the *getwd()* function or the file name at the top of the console pane.

Simple computations are evaluated immediately in the CONSOLE  
(try it)

```
1 3+4
2 6/18
3 4*3
4 4**3 or 4^3
5 sqrt(25)
6 log(100) vs. log10(100); exp(3) vs. 10**3
7 help(log)
8 help("+")
```

But this is BORING!

- R scripts are simple text files (usually with suffix .r or .R).
- *File* – > *New* or *File* – > *Open* in *Rstudio*.
- Use a suitable naming convention for directories and scripts.
- Use scripts for all but the simplest tasks
  - Ability to reuse code
  - Saves time dealing with typing errors (!)

# R scripts - Your first script

Create a NEW script and enter the following code (6 slides).

CASE is important in *R* code but not in comments.

You can break *R* code into more than one line and indentation not important.

```
1  # This script will read in the cereal data set,
2  #     make a simple listing, fit a regression line,
3  #     draw a scatter plot and add the line to the plot
4  #     do a single factor crd anova
5  #     get the compact letter display
6  #     make some plots of the results
7
8  # load required libraries
9  library(ggplot2)
10 library(lsmeans)
11
12 # Read in the cereal data from a csv file
13 cereal <- read.csv('cereal.csv',
14                   header=TRUE, as.is=TRUE, strip.white=TRUE)
```

## R scripts - Your first script - continued

```
15
16 # Define new variables and factors; check structure of data
17 cereal$shelfF <- factor(cereal$shelf)
18 cereal$Calories.fr.Protein <- cereal$protein * 4;
19
20 str(cereal)
21
22 # List the first few records
23 cereal[1:5,]
24
25 # List some variables
26 cereal$calories
27 cereal[, "calories"]
28 cereal$fat
29 cereal[1:5, c("name", "fat", "calories")]
```

## R scripts - Your first script - continued

```
32
33 # Make a nice scatter plot
34 plotbasic <- ggplot(data=cereal, aes(x=Fat, y=Calories))+
35     ggtitle("Calories vs Fat in cereals")+
36     xlab("Grams of Fat")+ylab("Calories/serving")+
37     geom_point()
38 plotbasic
39 ggsave(plotbasic, file='cal-vs-fat1.png',
40     h=4, w=6, units="in", dpi=300)
41
42 plotbasic2 <- ggplot(data=cereal, aes(x=Fat, y=Calories))+
43     ggtitle("Calories vs Fat in cereals")+
44     xlab("Grams of Fat")+ylab("Calories/serving")+
45     geom_jitter()
46 plotbasic2
47 ggsave(plotbasic, file='cal-vs-fat2.png',
48     h=4, w=6, units="in", dpi=300)
```



## R scripts - Your first script - continued

```
49
50 # Fit a regression between calories and grams of fat
51 fit.calories.fat <- lm( calories ~ fat, data=cereal)
52 summary(fit.calories.fat)
53 anova(fit.calories.fat) # careful Type I SS
54 coef(fit.calories.fat)
55 sqrt(diag(vcov(fit.calories.fat))) # extract the SE
56 confint(fit.calories.fat) # confidence intervals on parameters
57
58 names(summary(fit.calories.fat))
59 summary(fit.calories.fat)$r.squared
60 summary(fit.calories.fat)$sigma
61
62 class(fit.calories.fat)
63 methods(class=class(fit.calories.fat))
```

```
64
65 # Add the fitted line to the scatter plot
66 plotline <- plotbasic2 +
67   geom_abline(intercept=coef(fit.calories.fat)[1],
68               slope      =coef(fit.calories.fat)[2])
69 plotline
70 ggsave(plot=plotline, file="cal-vs-fat3.png",
71         h=4, w=6, units="in", dpi=300)
```

## R scripts - Your first script - continued

```
73
74 # Do a simple single factor ANOVA
75 # Is the mean number of calories the same for all shelves
76 # Need to use a FACTOR variable for the categorical variable
77 fit.sugars.shelf <- lm( sugars ~ shelfF, data=cereal)
78 anova(fit.sugars.shelf)
79
80 # Estimate the marginal means along with confidence limits
81 fit.sugars.shelf.lsmo <- lsmeans::lsmeans(fit.sugars.shelf,
82                                           ~shelfF)
83 fit.sugars.shelf.cld <- cld(fit.sugars.shelf.lsmo,
84                             adjust='tukey')
85 fit.sugars.shelf.cld
86 sf.cld.plot.bar(fit.sugars.shelf.cld, "shelfF",
87                 order=FALSE)
88
89 # Estimate the pairwise differences
90 pairs(fit.sugars.shelf.lsmo)
```

# R scripts - Running your first script

**SAVE** your script.

**RUN** the script one line at a time, or a block of lines at a time.

- Place cursor on line to run; or Highlight block of code to run.
  - Press the **RUN** button; or
  - *Code* → *Run Lines*; or
  - Use keyboard short cuts (depends on OS)
- Create an HTML notebook with code and output intermixed

**SAVE** your script (in case you made any changes.

**REVIEW** the *R* commands and the output.

# R scripts - Basic features of the script

Many R scripts have the same basic features:

- Comments describing the script
- Read in the data (*read.csv()*)
- Define factors (categorical variables, *factor()*) and derived variables.
- Check the data.frame (*str()*); list first few records
- Some basic plots (*ggplot()*)
- Fit a model (*lm()*)
- Use *methods* to extract information from model object (*anova()*, *summary()*)
- Make prediction (not shown), or follow-up analyses (*lsmeans::lsmeans()*)
- Make some final plots displaying results.

Reading data with  $R$

*R* is fairly flexible.

- \*.csv files - easiest
- Excel spreadsheets directly
- tables with white space delimiters
- Reading tables from URLs
- Internal data
- Querying most database systems (not part of this course)
- Scraping web pages (not part of this course)
- Fixing variable names to be valid *R* names.

Dealing with Dates and Times is always a pain.

*R* often converts character data to factors (a pain)

# Reading data - csv files

## Simple format in text format

- observations in rows; variables in columns
- separate values by a comma; enclose values in quotes if contain a comma
- variable names in first row
- Excel and most database packages can generate

```
1 cereal <- read.csv('../..'/SampleData/cereal.csv',  
2                   header=TRUE, as.is=TRUE, strip.white=TRUE)
```

- Data is “disconnected” from database
- *as.is=TRUE* stops automatic conversion - especially true for date/times.
- *strip.white=TRUE* removes extra white space at front/end of values
- Lots of options (see help page)



## Reading data - csv files

```
> head(cereal)
```

		name	mfr	type	calories	protein	fat	so
1		100%_Bran	N	C	60	4	1	
2		100%_Natural_Bran	Q	C	110	3	5	

```
'data.frame': 77 obs. of 15 variables:
```

```
$ name      : chr  "100%_Bran" "100%_Natural_Bran" "All-Bran"
```

```
$ mfr       : chr  "N" "Q" "K" "K" ...
```

```
$ type      : chr  "C" "C" "C" "C" ...
```

```
$ calories: int   60 110 80 50 110 110 110 140 90 90 ...
```

```
$ protein  : int   4 3 4 4 2 2 2 3 2 3 ...
```

Notice that NO factors created.

Many packages to read Excel workbooks

Two most popular are:

- *xlsx* - requires java to be installed and working (!)
- *readxl*

Many other packages around with varying degree of flexibility and speed.

## Reading data - Excel workbooks

```
1 library(xlsx)
2 cereal2 <- read.xlsx('../SampleData/ALLOfDATA.xls',
3                       sheetName='cereal',
4                       header=TRUE,
5                       stringsAsFactors=FALSE,
6                       startRow=8)
7 head(cereal2)
8 str(cereal2)
```

- *stringsAsFactors=FALSE* - prevents conversion
- Be careful with dates and times

# Reading data - Excel workbooks

```
> head(cereal2)
```

	Name	Manufacturer	Mfr	Hot.C
1	100% Bran	Nabisco	N	
2	100% Nat. Bran Oats & Honey	Quaker Oats	Q	

```
> str(cereal2)
```

```
'data.frame': 76 obs. of 18 variables:
```

```
$ Name          : chr  "100% Bran" "100% Nat. Bran Oats &  
$ Manufacturer  : chr  "Nabisco" "Quaker Oats" "Quaker Oat  
$ Mfr           : chr  "N" "Q" "Q" "K" ...  
$ Hot.Cold      : chr  "C" "C" "C" "C" ...  
$ Calories      : num  80 230 210 80 50 210 120 120 250 20
```

Notice that NO factors created.

# Reading data - Excel workbooks

```
1 library(readxl)
2 cereal3 <- read_excel('../SampleData/ALLofDATA.xls',
3                       sheet='cereal',
4                       col_names=TRUE,
5                       trim_ws=TRUE,
6                       skip=7)
7 head(cereal3)
8 str(cereal3)
```

- *col\_types* is automatically set to “guess” which works most of the time.
- Be careful of dates and time.

## Reading data - Excel workbooks

```
> head(cereal3)
```

```
# A tibble: 6 x 18
```

	Name	Manufacturer	Mfr	'Hot/Cold'
	<chr>	<chr>	<chr>	
1	100% Bran	Nabisco	N	
2	100% Nat. Bran Oats & Honey	Quaker Oats	Q	

```
> str(cereal3)
```

```
Classes 'tbl_df', 'tbl' and 'data.frame': 76 obs. of 18 variables:
```

```
$ Name          : chr  "100% Bran" "100% Nat. Bran Oats & Honey" ...
$ Manufacturer   : chr  "Nabisco" "Quaker Oats" "Quaker Oats" ...
$ Mfr            : chr  "N" "Q" "Q" "K" ...
$ Hot/Cold       : chr  "C" "C" "C" "C" ...
$ Calories       : num  80 230 210 80 50 210 120 120 250 200 ...
```

Notice that NO factors created.

Notice class of object is a *tibble* as well as a data frame.

# Reading data - White space delimited

White space delimited data.

- similar to csv files
- careful with values that contain white space

## Reading data - White space delimited

```
1 cereal4 <- read.table("http://lib.stat.cmu.edu/datasets/1993
2                       header=FALSE, as.is=TRUE, strip.white=
3 names(cereal4) <- c('Name','mfr','type','Calories','protein'
4                       'sugars','shelf','potass','vitamins','we
5 head(cereal4)
6 str(cereal4)
```

- Notice that I specified a URL
- Notice how column names are specified if data does not contain them in first row



## Reading data - White space delimited

```
> head(cereal4)
```

	Name	mfr	type	Calories	protein	Fat	s
1	100%_Bran	N	C	70	4	1	
2	100%_Natural_Bran	Q	C	120	3	5	

```
> str(cereal4)
```

```
'data.frame': 77 obs. of 15 variables:
```

```
$ Name      : chr  "100%_Bran" "100%_Natural_Bran" "All-Bran"
```

```
$ mfr       : chr  "N" "Q" "K" "K" ...
```

```
$ type      : chr  "C" "C" "C" "C" ...
```

```
$ Calories: int  70 120 70 50 110 110 110 130 90 90 ...
```

Data used underscores to prevent breaking values at white space.

Often require small amounts of data that should be stored with the script.

- *textConnection()* function useful.
- similar to reading \*.csv file.

## Reading data - Internal data

```
1 type.code.csv <- textConnection("  
2 type, code  
3 C , Cold Cereal  
4 H , Hot Cereal  ")  
5  
6 type.code <- read.csv(type.code.csv, header=TRUE,  
7       strip.white=FALSE, as.is=TRUE)  
8 head(type.code)  
9 str(type.code)  
10 type.code$type == "C"
```

- Can only read it “once” without redefining it.
- Connection name is arbitrary, but I adopt a simple convention.
- Notice that connection name NOT in quotes in *read.csv()*
- Notice how column names are specified if data does not contain them in first row

## Reading data - Internal data

```
> head(type.code)
  type      code
1    C    Cold Cereal
2    H    Hot Cereal

> str(type.code)
'data.frame': 2 obs. of  2 variables:
 $ type: chr  "C " "H "
 $ code: chr  " Cold Cereal" " Hot Cereal  "

> type.code$type == "C"
[1] FALSE FALSE
```

CAUTION: Notice extra white space around variable values.

Remove extra white space in variable values!!

```
1 type.code.csv <- textConnection("  
2 type, code  
3 C , 'Cold Cereal'  
4 H , 'Hot Cereal'  ")  
5  
6 type.code <- read.csv(type.code.csv, header=TRUE,  
7     strip.white=TRUE, as.is=TRUE)  
8 head(type.code)  
9 str(type.code)  
10 type.code$type == "C"
```

## Reading data - Internal data

```
> head(type.code)
  type      code
1    C Cold Cereal
2    H  Hot Cereal

> str(type.code)
'data.frame': 2 obs. of  2 variables:
 $ type: chr  "C" "H"
 $ code: chr  "Cold Cereal" "Hot Cereal"

> type.code$type == "C"
[1]  TRUE FALSE
```

Notice extra white space around variable values has been removed.

# Reading data - Adjusting variable names

It is sometime necessary to adjust variable names after reading

- Variable name has an misspelling
- Variable name is not a valid *R* variable name
  - Must start with a letter
  - Contain letters, numbers, periods (.), underscores (\_), but not blanks or other characters

CAUTION: Some functions do automatic “correction” of variable names and others do not.

See *make.names()* for more details.

## Reading data - Adjusting variable names

```
1 sample.csv <- textConnection("  
2 Bird #, Wiegth, Length mm, Mass (g)  
3 1, 100, 101, 102  
4 2, 200, 201, 202")  
5  
6 sample <- read.csv(sample.csv, header=TRUE,  
7                     strip.white=TRUE, as.is=TRUE)  
8 head(sample)  
9 str(sample)  
10 sample$Bird..
```



## Reading data - Adjusting variable names

```
> head(sample)
  Bird.. Wiegth Length.mm Mass..g.
1      1    100      101    102
2      2    200      201    202

> str(sample)
'data.frame': 2 obs. of  4 variables:
 $ Bird..    : int  1 2
 $ Wiegth    : int  100 200
 $ Length.mm: int  101 201
 $ Mass..g.  : int  102 202

> sample$Bird..
[1] 1 2
```

Notice how variable names are converted to valid *R* names.

## Reading data - Adjusting variable names

```
1 sample.csv <- textConnection("  
2 Bird #, Wieght, Length mm, Mass (g)  
3 1, 100, 101, 102  
4 2, 200, 201, 202")  
5 sample <- read.csv(sample.csv, header=TRUE,  
6                     strip.white=TRUE, as.is=TRUE,  
7                     check.names=FALSE)  
8 head(sample)  
9 str(sample)  
10 sample$Bird..  
11 sample$"Bird #"
```

## Reading data - Internal data

```
> head(sample)
  Bird # Wieght Length mm Mass (g)
1      1    100     101     102

> str(sample)
'data.frame': 2 obs. of  4 variables:
 $ Bird #    : int  1 2
 $ Wieght    : int  100 200
 $ Length mm: int  101 201

> sample$Bird..
NULL

> sample$"Bird #"
[1] 1 2
```

It is awkward (and sometime very difficult) to deal with irregular variable names.

## Reading data - Adjusting variable names

The *names()* function allows you access to variable names.

```
1 sample2 <- sample
2 names(sample2)
3 names(sample2) <- c("Bird","Weight","Length","Mass")
4 head(sample2)
```

```
> head(sample2)
```

	Bird	Weight	Length	Mass
1	1	100	101	102
2	2	200	201	202

## Reading data - Adjusting variable names

The *names()* function allows you access to variable names.

```
1 sample2 <- sample
2 names(sample2)
3 names(sample2) <- c("Bird","Weight","Length","Mass")
4 head(sample2)
```

```
> head(sample2)
  Bird Weight Length Mass
1    1    100    101  102
2    2    200    201  202
```

# Reading data - Adjusting variable names

The *names()* function allows you access to variable names.

Selective changing of names:

```
1 sample2 <- sample
2 names(sample2)
3 names(sample2)[2] <- c("Weight")
4 head(sample2)
```

```
> names(sample2)
[1] "Bird #"      "Wiegth"      "Length mm"  "Mass (g)"
```

```
> names(sample2)[2] <- c("Weight")
```

```
> head(sample2)
  Bird # Weight Length mm Mass (g)
1      1    100      101    102
2      2    200      201    202
```

## Reading data - Adjusting variable names

The *names()* function allows you access to variable names.  
Selective changing of names that is more robust

```
1 sample2 <- sample
2 names(sample2)
3
4 select <- grepl("Wieght", names(sample2))
5 select
6 sum(select)
7 names(sample2)[select]
8
9 names(sample2)[select] <- c("Weight")
10 head(sample2)
```

## Reading data - Adjusting variable names

The *names()* function allows you access to variable names.

Selective changing of names that is more robust

```
> names(sample2)
[1] "Bird #"      "Wieght"      "Length mm"  "Mass (g)"
>
> select <- grepl("Wieght", names(sample2))
> select
[1] FALSE  TRUE FALSE FALSE
> sum(select)
[1] 1
> names(sample2)[select]
[1] "Wieght"
>
> names(sample2)[select] <- c("Weight")
> head(sample2)
  Bird # Weight Length mm Mass (g)
1      1    100      101    102
2      2    200      201    202
```



Consider the Birds 'n Butts dataset.

- Save the *Correlational* worksheet as *csv* and read it.
- Read the *Correlational* worksheet directly.
- Change the error in the variable name.

# Reading data - Exercise

```
1 library(readxl)
2 butts <- read_excel('../sampledata/bird-butts-data.xlsx', sheet = 1)
3                       col_names=TRUE, skip=1)
4 butts[1:5,]
5 dim(butts)
6 str(butts)
7
8 # Or, save the sheet from the Excel file and read the csv file
9 butts <- read_csv("../sampledata/bird-butts-data-correlation.csv")
10 butts[1:5,]
11 dim(butts)
12 str(butts)
13
14 # Fix the names
15 select <- grep('weight', names(butts))
16 select
17 sum(select)
18 names(butts)[select]
19
20 names(butts)[ select] <- "Butts.weight"
```

# Reading data - Exercise

Changing the variable name:

```
> select <- grepl('wiegth', names(butts))
```

```
> select
```

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

```
> sum(select)
```

```
[1] 1
```

```
> names(butts)[select]
```

```
[1] "Butts.wiegth"
```

```
>
```

```
> names(butts)[ select] <- "Butts.weight"
```

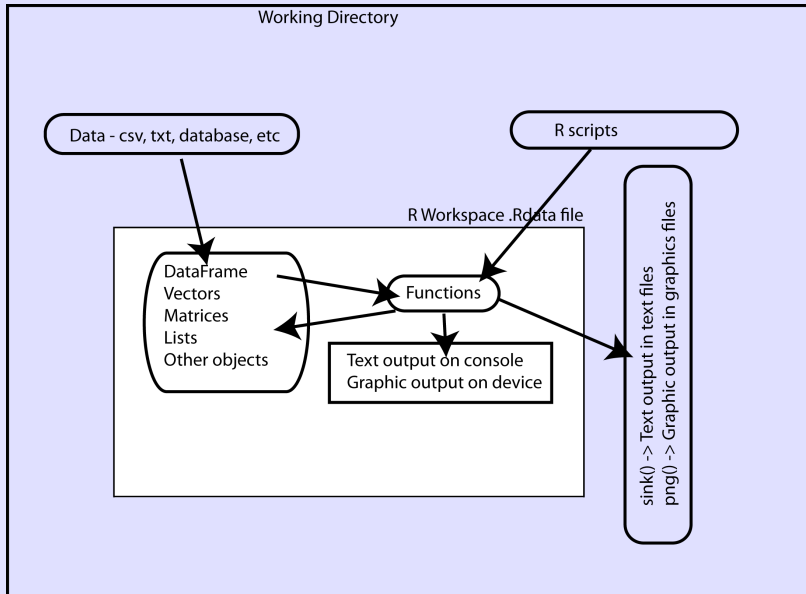
```
> butts[1:5,]
```

	Nest	Species	Nest.content	Butts.weight	Number.of.mites
1	1	HOSP	empty	6.13	4
2	2	HOSP	empty	3.73	30

Fairly rich set of functions to read data. Most common is to read rectangular structure into a data frame.

- *read.csv()* is easiest followed by reading Excel sheet directly.
- Able to access data bases as well - see *R* manuals.
- Use *textConnection()* for small tables so that data kept with script.
- CAUTION: Extra white space around variable values.
- CAUTION: Do NOT let *R* convert strings to factors.
- CAUTION: Dates and times
- CAUTION: Non-standard variable names.

# The R environment - Overview



Rstudio manages all of the above

## Comments and whitespace

- An octothorpe (#) indicates a comment. Everything after this is ignored

```
1 # This is a comment for the entire line
2 years <- 10 # number of years in study that must be s
```

- Use lots of comments to make your program readable
- Use whitespace (blank lines and indentation) to make your program more readable.
- Avoid using TABS as script may look different on another machine.
- Keep lines short (around 80 characters or less)

## Objects

- Everything in *R* is a named OBJECT.
- Examples of objects are vectors, matrices, lists, dataframes, functions.
- Object names start with a letter, then can contain letters, numbers, periods (.) or underscores (\_) arbitrary length.
  - *this.is.an.object.name*
  - *x*
  - *X*
  - *x47*
  - *thisisaverylongnamethatishardtoread*
- Object names are case sensitive so *camel*  $\neq$  *CaMeL*  $\neq$  *caMeL*
- Objects are not explicitly *typed* and the type is determined by examining the object, i.e. is an object a number, a character, a function etc. The *type* of an object can be dynamically changed at any time.

What objects are in my workspace? Seeing value of an object?

```
1 objects()    # displays all objects in my workspace
2 ls()         # alternate method for UNIX gurus
3 print(objectname) # displays contents of object
4 str(objectname)  # display "structure" of object (important)
5 objectname      # acts like print() in most cases (but not all)
6
7 rm(objectname) # removes an object from workspace
```

Try it.

*Rstudio* also shows objects in your workspace in upper right corner of screen.



# R language elements

Object assignment. Either a value, expression, another object, or a function.

Both `=` or `<-` can be used for assignment.

```
1 x <- 10  # numeric (integer or real or complex)
2 w = 15   # alternate way to assign
3 qq = x * w # product of two objects values
4 z <- 14.7
5 y <- TRUE # notice upper case; also use T and F
6 name <- 'Carl' # use matching ' or "
7 second.name <- "James"
8 ww <- sqrt(x)
9
10 ww <- sqrt(name) # oops not a valid operation
11 ww <- 15 + TRUE  # what do you get?
```

CAUTION: DO NOT USE `<=` or `<<-` for assignment - these have special meanings.

Types of values (principal types that you will encounter – there are more).

- numeric (integers or real numbers), 16 digits of precision
- character (a.k.a. string) - length 0+, delineated by matching ' or ". Either symbol can be used, but they must match to enclose the string.
- logical (TRUE or FALSE which are interpreted as 1 or 0 in arithmetic operations)
- function (you can create your own functions - very useful)
- factor (you will see this later - CAUTION)

Data is stored in workspace in 5 data structures (MOST COMMON)

- SCALAR, i.e. single value. Typically used to store a single value.
- VECTOR, i.e. set of values of SAME TYPE. Typically used as part of a data.frame.
- DATAFRAME, i.e. collection of VECTORS of same length of DIFFERENT TYPES. Typically used to store your data.
- LIST, i.e. general collection of ANY objects (data and others) of ANY type. Typically used to store results of model fits.
- matrix, i.e. rectangular set of values of SAME TYPE
- array, i.e. multidimensional set of values of SAME TYPE

Note the difference between a matrix and a DATAFRAME.  
You will deal mostly with DATAFRAMES.

# R language elements - Vectors

VECTORS is a set of values ALL OF THE SAME TYPE.

The `c()` concatenates objects together to make a vector.

```
1 age <- c(56, 56, 28, 23, 22)
2 height <- c(185, 162, 185, 167, 190)
3 f.names <- c('Carl', "Lois", 'Matthew', 'Marianne', 'David')
4 over.30 <- c(TRUE, TRUE, FALSE, FALSE, FALSE)
5
6 odd <- c(2.3, 'Carl')  # surprising, but look at result!
7
8 length(age)
9 length(f.names)
10 str(age)      # what is the structure of age?
11 str(f.names)
```

VECTORS is a set of values ALL OF THE SAME TYPE

```
1 age <- c(56, 56, 28, 23, 22)
2 height <- c(185, 162, 185, 167, 190)
3 f.names <- c('Carl', "Lois", 'Matthew', 'Marianne', 'David')
4 over.30 <- c(T, T, F, F, F) # AVOID using T/F for TRUE/FALSE
5
6 # The c() function is very versatile
7 ah <- c(age, height)
8 ah
9 age0age <- c(age, 0, age)
10 age0age
11 length(age0age)
12
13 odd <- c(f.names, over.30) # ??
14 odd
```

# R language elements - Dataframes

DATAFRAMES are collections of VECTORS of SAME length, but DIFFERENT types.

Not the same as a matrix (contents must be same type).

Commonly used to store your data for analysis or plotting.

```
1 age <- c(56, 56, 28, 23, 22)
2 height <- c(185, 162, 185, 167, 190)
3 f.names <- c('Carl', "Lois", 'Matthew', 'Marianne', 'David')
4 over.30 <- c(T, T, F, F, F)
5
6 schwarz <- data.frame( f.names, age, height, over.30,
7                       stringsAsFactors=FALSE)
8 schwarz
9 str(schwarz)
10 length(schwarz) # number of vectors, not length of vectors
11 dim(schwarz)
12 nrow(schwarz)
13 ncol(schwarz)
14 names(schwarz)
```

# R language elements - Reading Dataframes

DATAFRAMES are collections of VECTORS of SAME length, but DIFFERENT types.

Dataframes commonly created by reading in data from external files.

```
1 df.name <- read.csv( filename, header=TRUE,  
2                       as.is=TRUE, strip.white=TRUE)  
3 df.name <- read.table( filename, header=TRUE,  
4                        as.is=TRUE)
```

- Many more arguments; refer to *help(read.csv)*
- *header=TRUE* - first line has field names; R will convert to valid object names as needed
- *as.is=TRUE* - same as *stringsAsFactors=FALSE* - does NOT convert strings to factors (see later). RECOMMENDED!
- *strip.white=TRUE* - remove excess white space at front/end of fields. RECOMMENDED

## R language elements - Reading Dataframes - II

DATAFRAMES are collections of VECTORS of SAME length, but DIFFERENT types.

Dataframes are most commonly created by reading in data from external files rather than using the *data.frame()* function.

```
1 library(xlsx)
2 df.name <- read.xlsx(filename, header=TRUE, as.is=TRUE,
3                       sheetname='blah', startRow=xx) # Excel
4
5 library(readxl)
6 df.name <- read.excel(filename, header=TRUE, as.is=TRUE,
7                       sheetname='blah', startRow=xx) # hand
8
9 cereal2 <- read.table("http://lib.stat.cmu.edu/datasets/1993
10                      header=FALSE, as.is=TRUE)
11 names(cereal) <- c('name', 'mfr', 'type', 'Calories', 'protein')
```

Additional information available in Import/Export manual on reading from database systems at

<http://cran.r-project.org/doc/manuals/r-release/>



DATAFRAMES are collections of VECTORS of SAME length, but DIFFERENT types.

Back to our script ....

```
1 cereal <- read.csv('cereal.csv',
2                     header=TRUE, as.is=TRUE,
3                     strip.white=TRUE)
4
5 str(cereal)  # this function is VERY useful when things seem
6 length(cereal) # number of vectors, not length of vectors
7 dim(cereal)
8 nrow(cereal)
9 ncol(cereal)
10 names(cereal)
```

## R language elements - Dataframes - accessing - II

DATAFRAMES are collections of VECTORS of SAME length, but DIFFERENT types.

Cereal dataframe

Name	Calories	Fat		...		
------	----------	-----	--	-----	--	--

```
1 names(cereal)
2 cereal$name
3 cereal$calories
4 cereal[ , "calories"] # first index missing = ALL rows
5 calories # doesn't work because vector is hidden
6 with(cereal, calories) # careful of case.
```

DATAFRAMES are collections of VECTORS of SAME length, but DIFFERENT types.

Cereal dataframe

Name	Calories	Fat		...		
------	----------	-----	--	-----	--	--

```
1 cereal[1,]  
2 cereal[1:5,]  
3 cereal[, 1]  
4 cereal[, 1:5]  
5 cereal[1:4, 1:5]  
6 cereal[, "calories"]  
7 cereal[1:5, c("name", "calories", "fat")]
```

DATAFRAMES are collections of VECTORS of SAME length, but DIFFERENT types.

How to add and remove variables to data.frames

```
1 cereal$CalPerGramFat <- cereal$calories / cereal$Fat
2
3 cereal$CalPerGramFat  # some interesting values!
4
5 cereal$CalPerGramFat <- NULL # removes this variable from d.
```

VECTORS is a set of values ALL OF THE SAME TYPE

```
1 # operations on vectors
2 age <- c(56, 56, 28, 23, 22)
3 age.next.year <- age + 1
4 yob <- 2013 - age # element by element addition if same length
5
6 schwarz
7 schwarz$yob <- 2013 - schwarz$age
8 schwarz
```

# R language elements - Discourse on Vectors

VECTORS is a set of values ALL OF THE SAME TYPE

```
1  # functions on vectors
2  x <- c(0.5, 1, 1.5, 2, 4, 6, 8, 9, 10, 12)
3  length(x)
4  str(x)
5
6  sqrt(x)  # function applied to EACH element
7
8  # Other useful functions
9  range(x)
10 mean(x)
11 sd(x)
12 median(x)
13 summary(x)
14 sum(x)
15
16 # CAUTION: Compare min() and pmin() functions
17 min(x, 3)
18 pmin(x, 3)
```

Back to the cereal DATAFRAME ...

- Find average calories/serving?
- Find max, min, range of grams of fat/serving?
- Find average weight/serving? What happened?
  - Read `help(mean)` and use `na.rm=TRUE` argument to drop missing values

# R language elements - more fun with vectors

VECTORS is a set of values ALL OF THE SAME TYPE.

Generating “patterned vectors”

```
1  # Simple increments
2  help(":")  # be sure to put operators in quotes for the help
3  5:10
4  10.2:3
5  5:10-1  # careful : is evaluated prior to arithmetic
6  seq(1, 100, 10)
7  seq(to=100, from=1, by=10)
8
9
10 # replicate things
11 x <- c(5, 6, 7)
12 help(rep)
13 rep(TRUE, 10)
14 rep(x, times=2)
15 rep(x, length.out=8)
16 rep(x, each=2)
```



# R language elements - indexing

VECTORS is a set of values ALL OF THE SAME TYPE.

Indexing elements (extracting) from a vector

```
1  # Indexing
2  x <- c(5:9, 12:15, 34:37)
3  x
4
5  # Simple indexing
6  x[2]    # note the use of SQUARE brackets for indexing
7  x[c(2,3,7)] # the index can also be a vector
8  x[2,3,7]    # oops, not a proper index for a vector
9
10 n <- 10
11 x[n]    # indices can be variables. What does this mean?
12 inx <- c(3, 5, 7)
13 x(inx)   # Oops wrong types of brackets
14 x[inx]
```

VECTORS is a set of values ALL OF THE SAME TYPE.  
Indexing elements (extracting) from a vector using a selection vector

```
1 # Indexing
2 x <- c(5:9, 12:15, 34:37)
3 x
4
5 # using selection vector
6 select <- x >6 & x < 10
7 select
8 sum(select)
9 x[select]
```

- CAUTION: Careful about order of operations in selections
- CAUTION: *grep()* uses UNIX style pattern matching. This is especially important when searching strings.

# R language elements - indexing

VECTORS is a set of values ALL OF THE SAME TYPE.

Indexing elements INTO a vector (replacing)

```
1 x <- c(5:9, 12:15, 34:37)
2 x
3 x[2] <- 100  # note the use of SQUARE brackets for indexing
4 x
5
6 x[c(2,3,7)] <- 200 # the index can also be a vector
7 x
8 x[c(2,3,7)] <- c(500, 501) # notice the warning message
9 x
10
11 n <- 10
12 x[n] <- 500 # indices can be variables. What does this me
13 inx <- c(8,9,10)
14 x[inx] <- c(300, 400)
15 x
```

## R language elements - indexing

VECTORS is a set of values ALL OF THE SAME TYPE.

Indexing elements INTO a vector (replacing)

```
1 # using selection vector
2 select <- x >6 & x < 10
3 select
4 sum(select)
5 x[select] <- -1
6 x
```

```
> select <- x >6 & x < 10
```

```
> select
```

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

```
> sum(select)
```

```
[1] 2
```

```
> x[select] <- -1
```

```
> x
```

```
[1] 5 200 201 -1 -1 12 200 300 400 300 35 36 37
```

VECTORS is a set of values ALL OF THE SAME TYPE.

Dropping elements from a vector

```
1  # Indexing
2  x <- c(5:9, 12:15, 34:37)
3  x
4
5  # Dropping elements
6  x
7  x[-2]  # note the use of SQUARE brackets for indexing
8  x[-c(2,3,7)]  # the index can also be a vector
9
10 n <- -10
11 x[n]    # indices can be variables. What does this mean?
12 inx <- c(3, 5, 7)
13 x[-inx]
```

VECTORS is a set of values ALL OF THE SAME TYPE.

Logical conditions to select elements

```
1  # Using logical vectors to select elements
2  x <- c(5:9, 12:15, 34:37)
3  x
4
5  # Selecting elements where entry is TRUE
6  x > 10
7  x[ x>10 ]
8  x[ x > 10 & x < 20]
9  x[ x %% 2 == 0]   # %note the use of == to test for equality
10 x[ x>10 ] <- 500
11 x
```

Logical conditions to select elements - useful functions - *grepl()*

```
1 # We want to rename "Fiber" to "Fibre"
2 # Avoid using explicit index (i.e. names(cereal)[9] <- "Fibre")
3 names(cereal)
4 names(cereal)[grepl("fiber",names(cereal))] <- "fibre"
5 names(cereal)
```

## R language elements - indexing - %in%

Logical conditions to select elements - useful functions - %in%

```
1 # Select certain cereal manufacturers
2 cereal[ cereal$mfr %in% c("P","A"),] # don't forget the last
3
4 cereal[grep("Bran", cereal$name),] # don't forget the last
```



Back to the cereal dataframe ...

- Print *Calories* for ALL cereals.
- Print *Name*, *Calories*, *Fat* for ALL cereals.
- Print *Name*, *Calories*, *Fat* for first 5 cereals.
- Print *Name*, *Calories*, *Fat* for cereals 1, 3, 5, 7, 9, 11, ... 20.
- Print *Name*, *Calories*, *Fat* for cereals where calories > 150.
- Print *Name*, *Calories*, *Fat* for cereals with max calories.
- Print *Name*, *Calories*, *Fat* for cereals with fat > 2 sd from the mean fat content.
- Recode *Fiber* as "low" (below or at the median) or "high" (above the median)
- Cross tab *Fiber* and *FiberClass*.

FUNCTIONS are complex operations on one or more objects.  
FUNCTIONS can return any type of object

- `sqrt(x)` returns an object same dim as `x` with  $\sqrt{\text{elements}}$
- `ggplot(...)` creates a plot
- `lm(y ~ x, data=blah)` returns a linear model object (a list) with many different components

Help files have list of arguments and default values if an argument is not specified, e.g. `help(mean)`.

```
1 mean(cereal$calories)
2 mean(cereal$calories, trim=0.2)
3 mean(cereal$weight)
4 mean(cereal$weight, na.rm=TRUE)
5 mean(cereal$weight, na.rm=TRUE, trim=0.3)
```

FUNCTIONS are complex operations on one or more objects.

FUNCTIONS can return any type of object

Regression/ANOVA in *R* is done using the *lm()* function

```
1 result <- lm( response ~ x1 + x2 + x3 ... , data=blah)
2 str(result)  # Yikes!
3 names(result)
```

If *X* variables are continuous, this gives REGRESSION.

if *X* variables are categorical (factors), this gives ANOVA.

Model objects in R.

Regression/ANOVA in R is done using the *lm()* function

```
1 result <- lm( response ~ x1 + x2 + x3 ... , data=blah)
2 str(result)    # Yikes!
3 names(result)
```

Use specialized functions (called METHODS) to extract information from model objects. Look at help pages

```
1 summary(result)
2 anova(result)
3 coef(result)
4 confint(result)
5 methods(class=class(result))    # shows methods available
```

Many other useful functions - refer to reference card and packages

- `c()` - combine arguments into a vector
- `seq(from, to, by)` - generate a sequence
- `expand.grid(v1, v2, ...)` - generate all possible combinations
- `is.na()` - test for missing values
- `str()` - display structure of object
- `length()`, `dim()`, `nrow()`, `ncol()` - size of objects
- `max()`, `pmax()` - max and parallel max of objects
- `unique()` - list unique values in object
- `subset()` - make a subset
- `xtabs()` - make a cross-tabulation
- `rbind()`, `cbind()` - combine rows and columns
- `paste` - paste together strings
- `substr` - string extraction and replacement
- `grepl()` - pattern matching

# R language elements - Functions

Some examples of useful functions:

```
1  # concatenating objects together, especially to make a vector
2  limits <- c(0, 100)
3
4  ggplot(data=cereal, aes(x=fat, y=calories))+
5      geom_point()+
6      ylim(c(0,100))
7
8  # generating sequence
9  seq(1,10,2)
10 seq(1, by=2, length.out=10 )
11
12 # generating all possible combinations
13 expand.grid( sex=c("m","f"), age=c(10,20,30), stringsAsFactors=FALSE)
```

# R language elements - Functions

Some examples of useful functions:

```
1  # checking for and counting number of missing values; select  
2  is.na(cereal$weight)  
3  sum(is.na(cereal$weight))  
4  sum(!is.na(cereal$weight))  
5  select <- is.na(cereal$weight)  
6  cereal[ !select,]  
7  expand.grid( sex=c("m","f"), age=c(10,20,30), stringsAsFactors=FALSE)  
8  
9  # max and parallel maximum  
10 x <- c(1,2,3,4,5,6)  
11 max(x)  
12 pmax(3, x)  
13  
14 # finding the set of unique values  
15 unique(cereal$type)
```

Some examples of useful functions:

```
1 # xtabs - counting and checking
2 xtabs(~type, data=cereal, exclude=NULL, na.action=na.pass)
3 xtabs(~type+cups, data=cereal, exclude=NULL, na.action=na.pass)
4
5 # pasting together text
6 paste("Analysis of ", nrow(cereal), ' breakfast cereals', sep=" ")
7 ggplot(data=cereal, aes(x=fat, y=calories))+
8   geom_point()+
9   ggtitle(paste("Analysis of ", nrow(cereal), ' breakfast cereals', sep=" "))
```



Some examples of useful functions:

```
1  # pattern matching - Google is your friend
2  select <- grepl("bran", cereal$name) # exact match
3  cereal[select,]
4
5  select <- grepl("bran", cereal$name, ignore.case=TRUE)
6  cereal[select,]
7
8  select <- grepl("^bran", cereal$name, ignore.case=TRUE)
9  cereal[select,] # start with bran
10
11 select <- grepl("bran$", cereal$name, ignore.case=TRUE)
12 cereal[select,] # end with bran
```

Some examples of useful functions:

```
1  # dealing with strings
2  toupper(cereal$name)
3  tolower(cereal$name)
4  trimws(cereal$name)
5
6  substr(cereal$name, 1, 4)
7  substring(cereal$name, 4)
8
9  substr(cereal$name, 1, -1 +regexpr("_", cereal$name, fixed=TRUE))
10 substr(cereal$name, 1, pmax(5,-1 +regexpr("_", cereal$name, fixed=TRUE)))
11
12 gsub("-", "_", cereal$name)
```

Some examples of useful functions:

```
1  # sorting
2  sort(cereal$name)
3  cereal[ order(cereal$name), ] # reorder a data frame
4  cereal[ order(cereal$calories, cereal$name),]
5
6  # file path that is device independent
7  file.path("../sampledata")
```

# R language elements - Functions

```
1  # merging and combining data frames
2  byear <- data.frame(name =c('Carl', 'Lois', 'Matthew', 'Mar
3                          byear=c( 1956,    1956,    1986,    1990
4  bcity <- data.frame(name =c('Carl', 'Lois', 'Matthew'),
5                      city =c('Wpg', 'Brandon', 'Wpg'))
6  wcity <- data.frame(name =c('Matthew', 'Marianne', 'David'),
7                      city =c('Ottawa', 'Vancouver', 'Victoria
8
9  # cbind must be used with caution
10 cbind(bcity, wcity)
11
12 # merge - careful of non-matches
13 merge(byear, bcity)
14 merge(byear, bcity, all=TRUE)
15 merge(byear, bcity, all.y=TRUE)
16
17 # multiple merges - Google is your friend
18 Reduce(function(...){merge(..., all=TRUE)}),
19         list(byear, bcity, wcity))
```

Base graphics follow a “pen-on-paper” paradigm.

- Begin with a base command (a.k.a. a *high level plotting command*) such as *plot()* which establishes plot limits, axes titles, etc.
  - *plot()* - scatter plots
  - *hist()* - histogram
  - *barchart()* - bar charts
  - *stripchart()* - dot plots
  - etc.
- Embellish the plot with additional commands (a.k.a. *low level plotting command*)
  - *abline()* - add a line to a plot
  - *lines()* - join points with line segments
  - *segment()* - separate line segments
  - etc.
- By default, graphics are “sent” to a graphics-window that is non-permanent.
  - *dev.copy()* and *dev.off()* pair send window a file
  - bracket the code with *png()* and *dev.off()* commands

*Grammar of Graphics* plotting routines don't display plot until the end after it is completely constructed. So, axes can be adjusted after the fact (unlike the pen-and-paper paradigm).

*ggplot2* package gives much nicer graphics than Base R, but takes some skill to learn.

See <http://www.cookbook-r.com/Graphs/> for examples.

```
1 library(ggplot2)
2 plot.calories.fat <- ggplot(data=cereal,
3     aes(x=fat, y=calories)) + # specify a scatter plot
4     geom_point(shape=1) +     # Use hollow circles
5     geom_abline(intercept=..., slope=...) # add a line
6 plot.calories.fat
```

R packages.

- Extensions to base R contributed by users (almost 5000 packages!).
- Commonly distributed via CRAN (<http://cran.r-project.org>).
- Download, install, and update package using *Rstudio*.
  - If your computer is locked down, i.e. no administrative access, install packages in local directory rather than in system directories.
- Before using a package, use *library(package name)*, e.g. *library(ggplot2)*.
- Use the syntax *package::function()* to make it clear where a function is coming from and to avoid name conflicts.
- Documentation available:
  - *help(package='package name')*
  - On the CRAN website (use google)
  - On the CRAN website (task view)

*R* packages - Good news.

- Large user community == Large number of packages
- Many different application of *R*

*R* packages - Bad news.

- NO quality control
- NO consistency in naming conventions across packages
- NO central indexing (but google is your friend)
- NO support unless by package author (but google is your friend)



R packages - some useful packages.

- *boot* - bootstrapping (distributed with base R)
- *car* - correction to *anova()*; recoding
- *ggplot2* - grammar of graphics plotting (much superior to base R plotting)
- *Hmisc* - miscellaneous functions by Harrel
- *knitr* - *Rstudio* generates html and other reports
- *lme4*, *lmerTest* - linear mixed models
- *lsmeans* - marginal means and multiple comparisons for models
- *nlme* - nonlinear models and *lme()*.
- *plyr* - split-apply-combine paradigm
- *reshape2* - melting and casting data frames.
- *xlsx*, *readxl* - read/write Excel tables

Plus many others

What happens when you upgrade *R* (e.g. 3.0.1 to 3.1.0)?

- New version is created in NEW directory
- Standard base packages installed in new library
- Need to upgrade packages in your PERSONAL library using *Rstudio* → Packages → Update.
- Occasionally, need `update.packages(checkBuilt=TRUE)`

If you don't have a personal library, you must REINSTALL every package every time you update your version of *R*!

If using *Windoze*, use `updateR()` in *installr* package.

Missing Values - NA's, NaN, Infs

Missing values (NA) can occur

- Data lacking values (are these MCAR, MAR, or IM)?
- Illegal computations ("Carl"/3) or 10/0
- Over/Under flow (e.g.  $\exp(1000)$ )

Default action of *R* is to propagate missing values, i.e.  $3 + \text{NA}$  is also NA.

You can change this action in several ways (the `na.actions`).

# R Missing Values

Look at *cereal* dataframe and the *weight* variable.

Some cereals have missing values.

Compare the following results.

Read *help(is.na)* and *help(na.omit)*

```
1 mean(cereal$weight)
2 mean(cereal$weight, na.rm=TRUE)
3 mean(na.omit(cereal$weight))
4
5 length(cereal$weight) # includes missing values
6 length(na.omit(cereal$weight))
7 is.na(cereal$weight)
8 sum( is.na(cereal$weight)) # count num missing
9
10 dim(cereal)
11 dim(na.omit(cereal)) # drop row with missing data
12
13 complete.cases(cereal)
14 dim(cereal[complete.cases(cereal),])
```

Look at cereal dataframe and the *weight* variable.  
Compute the % weight of fat, i.e. grams of fat/serving size.  
Compute the protein:fat ratio.

```
1 # Using NA in operations leads to NA's
2 cereal$prop.fat <- cereal$fat /cereal$weight
3 cereal$prop.fat
4 is.na(cereal$protein.fat) # Inf is a value
5
6
7 # NA is different from Inf;   protein/fat ratio
8 cereal$protein.fat <- cereal$protein / cereal$fat
9 cereal$protein.fat
10 iis.na(cereal$protein.fat) # Inf is a value
11 is.infinite(cereal$protein.fat)
```

# R Missing Values - Caution with plotting functions

Missing values typically don't show up on plots!

```
1 ggplot(data=cereal, aes(x=weight, y=calories))+  
2   ggtitle("calories vs. Weight per serving")+  
3   xlab("Weight per serving")+ylab("calories")+  
4   geom_jitter()+  
5   geom_smooth(method="lm",se=FALSE)
```

Warning messages:

```
1: Removed 2 rows containing missing values (stat_smooth).  
2: Removed 2 rows containing missing values (geom_point).
```

## R Missing Values - Caution with modeling functions

Look at cereal dataframe and the weight variable.  
Regress calories against serving size.

```
1 fit.cal.serving <- lm(calories ~ weight,  
2                       data=cereal)  
3 summary(fit.cal.serving) # note missingness
```

Residual standard error: 12.89 on 73 degrees of freedom  
(2 observations deleted due to missingness)



## R Missing Values - Caution with modeling functions

Look at cereal dataframe and the weight variable.

Regress calories against serving size.

```
1 fitted(fit.cal.serving) # only length 75 despite having ind  
2 length(fitted(fit.cal.serving))
```

So the following fails:

```
1 cereal$fitted <- fitted(fit.cal.serving)
```

```
Error in '$<-.data.frame'('*tmp*', "fitted", value = c(100.6  
  replacement has 75 rows, data has 77
```

## R Missing Values - Caution with modeling functions

Look at cereal dataframe and the weight variable.

Regress calories against serving size.

It is possible to have *lm()* deal *nicely* with NA, but this approach is not implemented consistently in *R*.

```
1 fit.cal.serving2 <- lm(calories ~ weight,  
2                       na.action=na.exclude,  
3                       data=cereal)  
4 summary(fit.cal.serving2)  
5 fitted(fit.cal.serving2) # now padded with NA  
6 length(fitted(fit.cal.serving2))  
7 cereal$fitted <- fitted(fit.cal.serving2)
```

## R Missing Values - Caution with modeling functions

Look at cereal dataframe and the weight variable.

Regress calories against serving size.

I prefer to use the predict with new data that propagates missing values properly.

Compare:

```
1 predict(fit.cal.serving)
2 predict(fit.cal.serving, newdata=cereal)
3
4 predict(fit.cal.serving2)
```

- ALWAYS check for NA's in data and ask if MCAR, MAR, or IM

- Sometime special codes are used, e.g. -1 means missing

```
1 mydf [ is.na(mydf$var), ] # list rows where var is missing
2 mydf [ mydf == -1 ] <- NA # common code
```

- NA and Inf's propagate through computation but R is NOT consistent.
  - `mean(var)` is NA if var contains any missing
  - `mean(var, na.rm=TRUE)` drops the NAs
  - `lm( Y ~ X, data=blah )` drops missing rows automatically
- Different results from methods with `na.action=na.exclude` vs. `na.action = na.omit`.

Whew!

Basic structure of many scripts:

- header with comments describing the purpose, date, etc
- read data into a **DATAFRAME** which is a collection of **VECTORS**
- review data to ensure that read properly
- preliminary plots; check for outliers; problem points, etc
- fit a model
- extract information from *R* model objects using **METHODS**
- final plots and reports

## Data Structures in R

- **Vectors** - collect of values of SAME TYPE
- **Dataframe** - collection of vectors of SAME LENGTH, but DIFFERENT TYPES
- **Lists** - collection of arbitrary objects (e.g. results from a model fit)

## Data Types in R

- **Numbers** - up to 16 digits of precision
- **Logical** - TRUE or FALSE
- **Character** - 'strings' of arbitrary length
- and other such as *factors*, *functions*, etc.

## Operations on data in R

- direct computations - vectorize as much as possible; seldom need for “loops” in R

```
1 df$newvar <- df$x + df$y
2 df$log.Y <- log(df$Y)
3 df$new <- recode(df$old, 'recode specifications')
```

- functions; builtin simple (*log()*); builtin complex (*lm()*); user written

```
1 df$log.Y <- log(df$Y)
2 calories.mean <- mean(df$calories,
3                       na.rm=TRUE, trim=0.1)
4 fit.calories.fat <- lm( calories ~ fat, data=cereal)
5 my.result <- my.function(data1, data2, data3)
```



biology  
**letters**

[rsbl.royalsocietypublishing.org](http://rsbl.royalsocietypublishing.org)

## Animal behaviour

Incorporation of cigarette butts into nests  
reduces nest ectoparasite load in urban  
birds: new ingredients for an old recipe?

Raw data is in *bird-butts-data.xlsx*

Is there a relationship between number of butts and parasite load?

Create an R script to

- Read *Correlational* tab into a data frame
  - Read *xls* sheet directly and/or create a \*.csv file
  - What are variable names?
  - Do you need to **skip** the first record?
- List the first few records
- Check the dimensions of the data.frame.
- Check the variable names of the data.frame.
- Check the structure of the data.frame
- ....

## R language elements - Exercise - Birds 'n Butts.

Is there a relationship between number of butts and parasite load?

```
> butts[1:5,]
```

	Nest	Species	Nest.content	Butts.weight	Number.of.mites
1	1	HOSP	empty	6.13	4
2	2	HOSP	empty	3.73	30
3	3	HOSP	eggs	0.06	84
4	4	HOSP	eggs	8.30	2
5	5	HOSP	eggs	0.00	12

```
> dim(butts)
```

```
[1] 57  5
```

```
> str(butts)
```

```
'data.frame': 57 obs. of  5 variables:
```

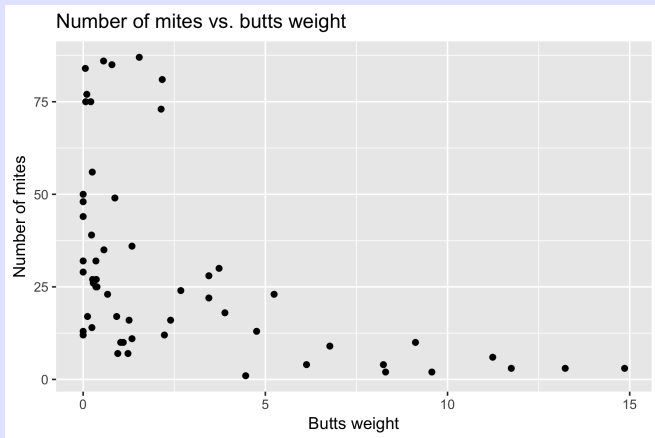
```
$ Nest      : int  1 2 3 4 5 6 7 8 9 10 ...
$ Species   : chr   "HOSP" "HOSP" "HOSP" "HOSP" ...
$ Nest.content : chr   "empty" "empty" "eggs" "eggs" ...
$ Butts.weight : num  6.13 3.73 0.06 8.3 0 1.23 1.03 0 2
$ Number.of.mites: int  4 30 84 2 12 7 10 44 16 32 . 135 / 695
```

Is there a relationship between number of butts and parasite load?

Create an *R* script to

- ...
- Preliminary plot of mites vs. butts weight

Is there a relationship between number of butts and parasite load?

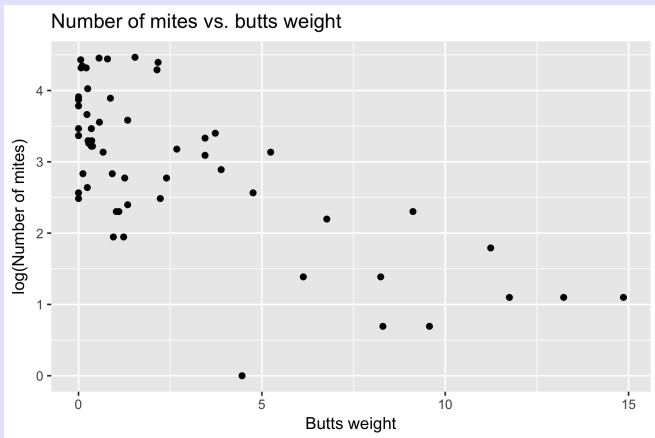


Is there a relationship between number of butts and parasite load?

- ....
- Is a transformation needed to get a straight line?
- Create a derived variable for the  $\log()$  transformation of number of mites.
- Plot the  $\log$  *mites* vs. butts weight

## R language elements - Exercise - Birds 'n Butts.

Is there a relationship between number of butts and parasite load?



Is there a relationship between number of butts and parasite load?

- ...
- Identify the outlier with 1 mite
- Use a logical vector to select rows of the data frame with 1 mite and print it

	Nest	Species	Nest.content	Butts.weight	Number.of.mites	l
56	56	HOFI	eggs	4.46		1



Is there a relationship between number of butts and parasite load?

- ...
- Fit straight line of log *mites* vs. *butts weight*
- Get the ANOVA table.
- Get the summary table.
- Extract information from *summary()* method.

## R language elements - Exercise - Birds 'n Butts.

Is there a relationship between number of butts and parasite load?

```
> anova(fit.log.mites)
```

Response: log.mites

	Df	Sum Sq	Mean Sq	F value	Pr(>F)	
Butts.weight	1	32.394	32.394	52.295	1.574e-09	***
Residuals	55	34.069	0.619			

```
> summary(fit.log.mites)
```

Coefficients:

	Estimate	Std. Error	t value	Pr(> t )	
(Intercept)	3.49066	0.12981	26.891	< 2e-16	***
Butts.weight	-0.20276	0.02804	-7.232	1.57e-09	***

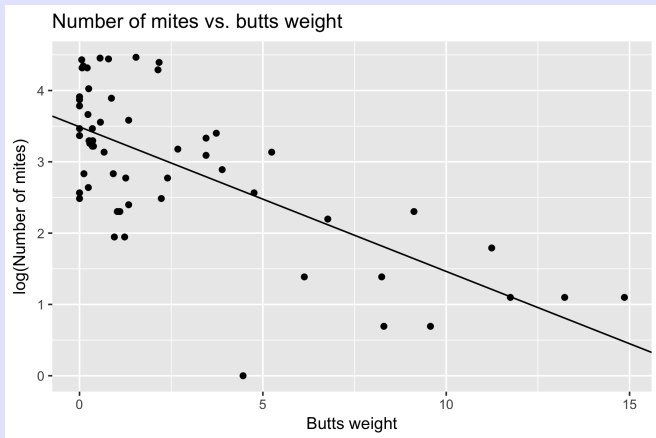
```
> names(summary(fit.log.mites))
```

```
> summary(fit.log.mites)$r.squared
```

Is there a relationship between number of butts and parasite load?

- ...
- Add the line to the previous plot
  - `geom_abline(intercept=..., slope=..)` vs.
  - `geom_smooth(method="lm", SE=FALSE)`
- Save the graphic to your directory.

Is there a relationship between number of butts and parasite load?



Is there a relationship between number of butts and parasite load?

- ...
- Create an notebook (HTML or Word or PDF) of entire script and output

Plotting using *ggplot2*

# Plotting with *ggplot2*

Plotting paradigms:

❶ Base *R* - AVOID

- Pen-on-paper; once plotted, cannot be changed;
- You need to do everything, e.g. legends; grouping data

❷ *lattice* graphics - AVOID

- Extension to Base *R* plotting routines to handle some features (e.g. legends) automatically.
- Lack of formal graphical models makes it hard to extend

❸ *ggplot2* graphics - RECOMMENDED

- Based on Grammar of Graphs by Cleveland.
- Build up a graph piece by piece
- Display the graph only once all parts are added. Axes, etc. automatically adjust once all layers added

<http://ggplot2.org> is the main reference site.

*R* Graphics Cookbook by Winston Chang.

*ggplot2*: Elegant Graphics for Data Analysis by Hadley Wickham

Elements of a graph:

- **data** and **aesthetics**: color, size, title, etc.
- **geometry**: what you see: points, lines, polygons, etc.
- **statistics**: transforming (summarizing) data: bins, fitting lines
- **scales**: map data space to aesthetic space: which points get which color; mpg to distance from origin, etc.
- **co-ordinate system**: cartesian co-ordinates or polar co-ordinates, etc.
- **faceting**: multiple plots on same graph

Only suitable for static graphs - no interactive or dynamic graphs.

Visit <http://r-statistics.co/>

Top50-Ggplot2-Visualizations-MasterList-R-Code.html for some nice graphic templates.



## Plotting with *ggplot2* - Scatterplot

Create a plot of calories vs. grams of fat; draw a smoother; fit a line; add confidence bands to the fit

```
1
2 library(ggplot2)
3
4 cereal <- read.csv('cereal.csv',
5                   header=TRUE, as.is=TRUE,
6                   strip.white=TRUE)
7 cereal[1:5,]
8 cereal$shelfF <- factor(cereal$shelf)  # categorical variab
```

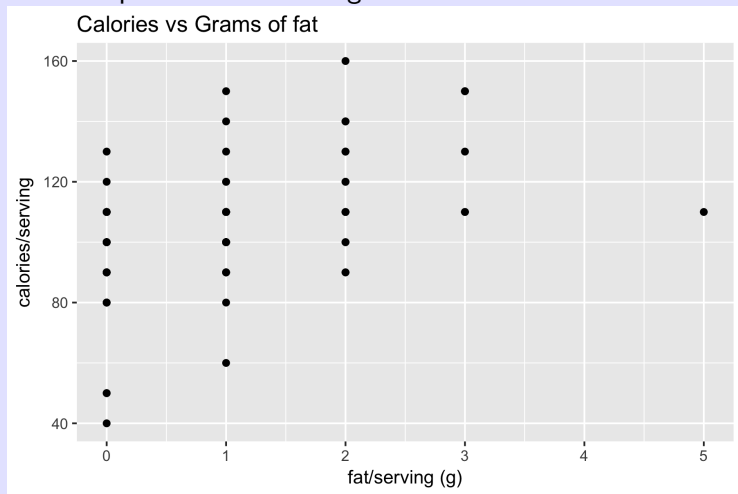
# Plotting with *ggplot2* - Scatterplot

Create a plot of calories vs. grams of fat

```
1 library(ggplot2)
2
3 # Start with basic plotting
4 newplot <- ggplot(cereal, aes(x=fat, y=calories))+
5   ggtitle("Calories vs Grams of Fat")+
6   xlab("Fat/serving (g)") + ylab("Calories/serving")+
7   geom_point()
8 newplot
9 ggsave(plot=newplot, file='cereal-calories-fat-base.png',
10        h=4, w=6, units="in", dpi=300)
```

# Plotting with *ggplot2* - Scatterplot

Create a plot of calories vs. grams of fat



# Plotting with *ggplot2* - Scatterplot

Create a plot of calories vs. grams of fat; jittering

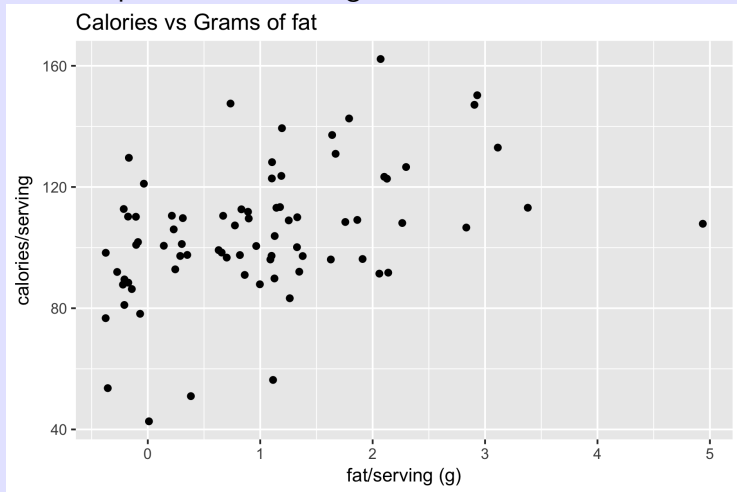
```
1  
2 # jittering  
3 newplot <- ggplot(cereal, aes(x=fat, y=calories))+  
4   ggtitle("Calories vs Grams of Fat")+  
5   xlab("Fat/serving (g)") + ylab("Calories/serving")+  
6   geom_jitter() #too much
```

You can control the amount of jittering, color, size of points

```
1   ... geom_jitter(position=position_jitter(w=.1, h=.1))  
2   ... geom_jitter(position=position_jitter(w=.1, h=.1),  
3     color='red', size=4))
```

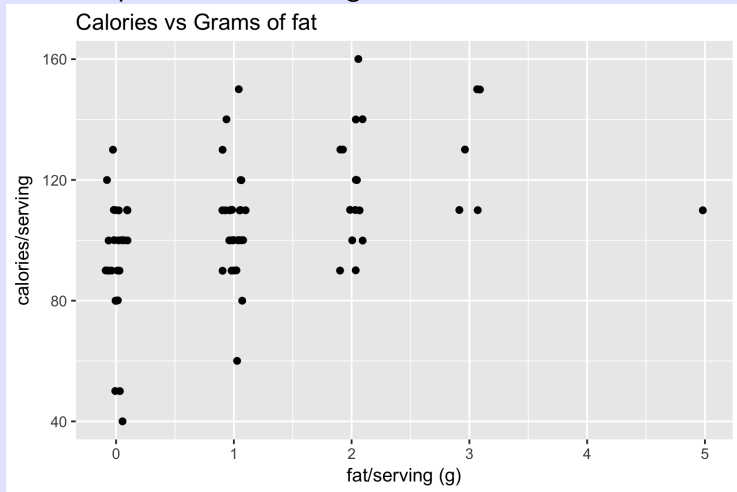
# Plotting with *ggplot2* - Scatterplot

Create a plot of calories vs. grams of fat



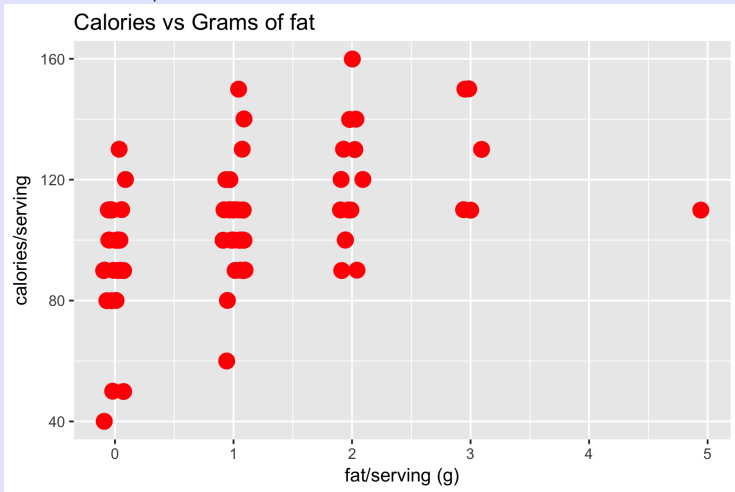
# Plotting with *ggplot2* - Scatterplot

Create a plot of calories vs. grams of fat -  $w=.1$ ,  $h=.1$



# Plotting with *ggplot2* - Scatterplot

Create a plot of calories vs. grams of fat - `w=.1`, `h=.1`,  
`color="red"`, `size=4`



# Plotting with *ggplot2* - Dodgine

Apply a consistent shift across groups

position=position\_dodge(w=..., h=...)

```
1 report <- plyr::ddply(cereal, c("shelf","fiber.grp"),
2   sf.simple.summary, variable="calories", crd=TRUE)
3 report
```

	shelf	fiber.grp	n	nmiss	mean	sd	se	lcl	ucl
1	1	high	2	0	85.00	7.07	5.00	21.47	148.53
2	1	low	18	0	102.22	10.60	2.50	96.95	107.49
3	2	high	1	0	120.00	NA	NaN	NaN	NaN
4	2	low	20	0	107.00	12.18	2.72	101.30	112.70
5	3	high	9	0	102.22	34.20	11.40	75.94	128.51
6	3	low	27	0	107.41	27.68	5.33	96.46	118.36



## Plotting with *ggplot2* - Dodging

Apply a consistent shift across groups

`position=position_dodge(w=.., h=...)`

```
1 newplot <- ggplot2::ggplot(report, aes(x=shelf, y=mean,
2     color=fiber.grp, shape=fiber.grp))+
3   ggtitle("Illustrating dodging - none applied")+
4   geom_point()+
5   geom_line()+
6   geom_errorbar(aes(ymin=lcl, ymax=ucl), width=.05)+
7   ylab("Mean calories/serving (95% ci)")
8 newplot
```

# Plotting with *ggplot2* - Dodging

Plot of results ignoring dodging.



# Plotting with *ggplot2* - Dodging

Apply a consistent shift across groups

`position=position_dodge(w=.., h=...)`

```
1 newplot <- ggplot2::ggplot(report, aes(x=shelf, y=mean,
2     color=fiber.grp, shape=fiber.grp))+
3     ggtitle("Illustrating dodging - dodging applies")+
4     geom_point( position=position_dodge(w=0.1))+
5     geom_line(position=position_dodge(w=0.1))+
6     geom_errorbar(aes(ymin=lcl, ymax=ucl), width=.1, position
7     ylab("Mean calories/serving (95% ci)")
8 newplot
```

Keep the dodging consistent among layers.

It is possible to have different amounts of dodging but unusual.

# Plotting with *ggplot2* - Dodging

Dodging applied.



## Plotting with *ggplot2* - Dodging

Apply a consistent shift across groups

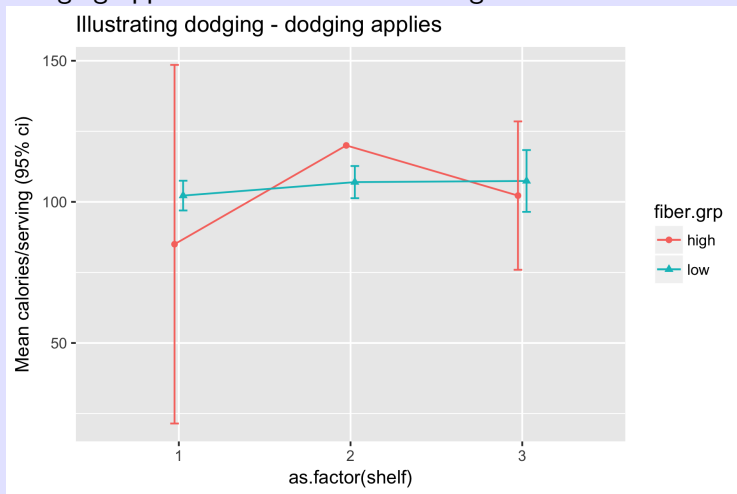
Force *shelf* to be character and not numeric. Note use of *group*  
`position=position_dodge(w=..., h=...)`

```
1 newplot <- ggplot2::ggplot(report, aes(x=as.factor(shelf),
2     y=mean,
3     group=fiber.grp, color=fiber.grp, shape=fiber.grp))+
4   ggtitle("Illustrating dodging - dodging applies")+
5   geom_point( position=position_dodge(w=0.1))+
6   geom_line(position=position_dodge(w=0.1))+
7   geom_errorbar(aes(ymin=lcl, ymax=ucl), width=.1,
8     position=position_dodge(w=0.1))+
9   ylab("Mean calories/serving (95% ci)")
10 newplot
```

Not clear why *group=* is needed??

# Plotting with *ggplot2* - Dodging

Dodging applied with  $X$  axis as a categorical variable.



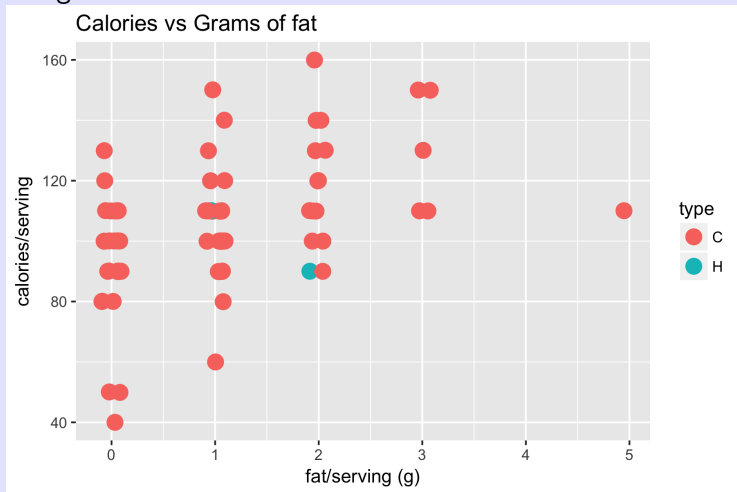
# Plotting with *ggplot2* - Scatterplot

Create a plot of calories vs. grams of fat - color/shape by categorical variable

```
1 newplot <- ggplot(cereal, aes(x=fat, y=calories,  
2                               color=type))+  
3   ggtitle("Calories vs Grams of Fat")+  
4   xlab("Fat/serving (g)") + ylab("Calories/serving")+  
5   geom_jitter(position=position_jitter(w=.1, h=.1), size=4)  
6  
7 newplot <- ggplot(cereal, aes(x=fat, y=calories,  
8                               color=shelfF))+  
9   ggtitle("Calories vs Grams of Fat")+  
10  xlab("Fat/serving (g)") + ylab("Calories/serving")+  
11  geom_jitter(position=position_jitter(w=.1, h=.1), size=4)  
12  
13 newplot <- ggplot(cereal, aes(x=fat, y=calories,  
14                               color=shelfF, shape=shelfF))+  
15  ggtitle("Calories vs Grams of Fat")+  
16  xlab("Fat/serving (g)") + ylab("Calories/serving")+  
17  geom_jitter(position=position_jitter(w=.1, h=.1), size=4)
```

# Plotting with *ggplot2* - Scatterplot

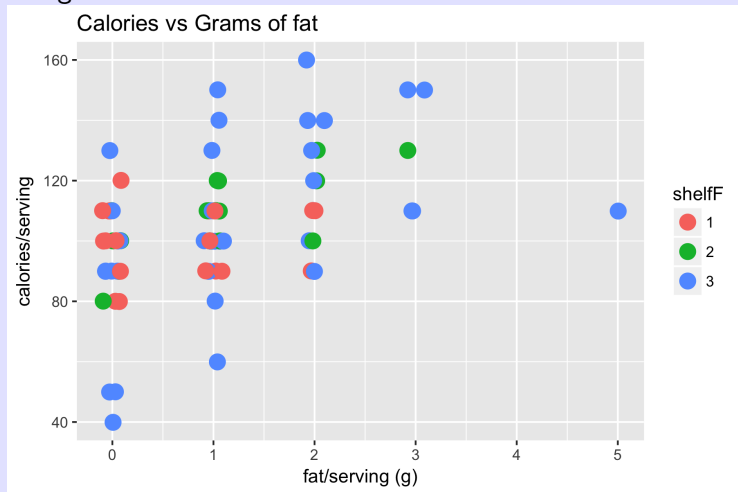
Create a plot of calories vs. grams of fat - color/shape by categorical variable





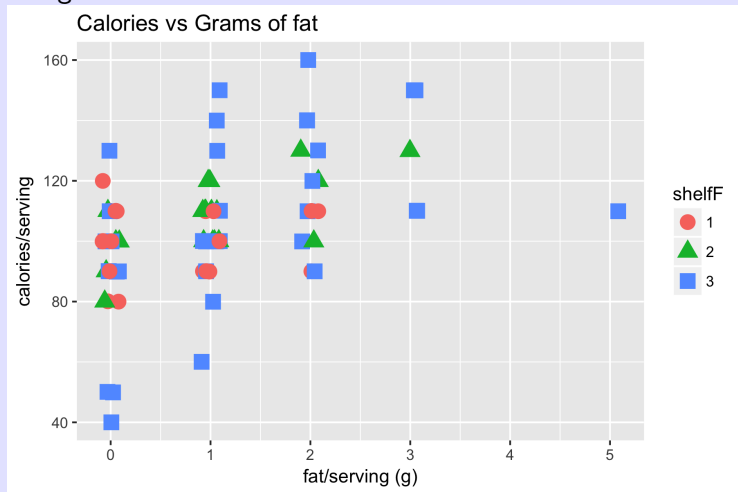
# Plotting with *ggplot2* - Scatterplot

Create a plot of calories vs. grams of fat - color/shape by categorical variable



# Plotting with *ggplot2* - Scatterplot

Create a plot of calories vs. grams of fat - color/shape by categorical variable



## Plotting with *ggplot2* - Scatterplot

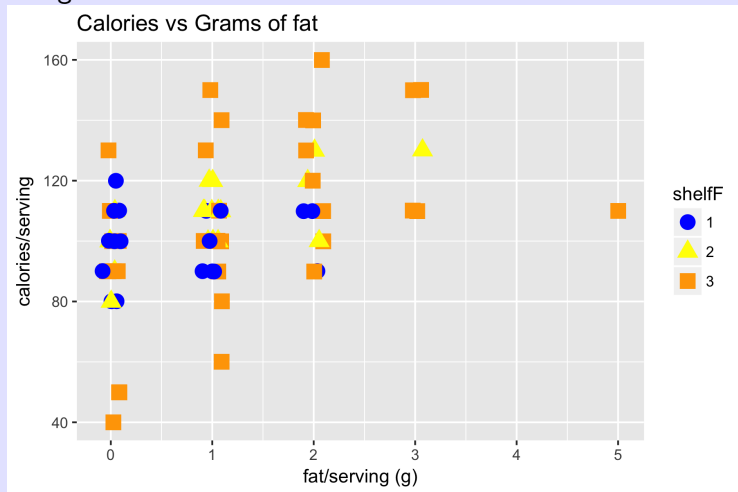
Create a plot of calories vs. grams of fat - color/shape by categorical variable.

Specify the colors directly.

```
1 newplot <- ggplot(data=cereal, aes(x=fat, y=calories,  
2   color=shelfF, shape=shelfF))+  
3   ggtitle("Calories vs Grams of fat")+  
4   xlab("fat/serving (g)") + ylab("calories/serving")+  
5   geom_jitter(position=position_jitter(w=.1, h=.1), size=4)  
6   scale_color_manual( values=c("blue","yellow","orange"))  
7 newplot
```

# Plotting with *ggplot2* - Scatterplot

Create a plot of calories vs. grams of fat - color/shape by categorical variable



## Plotting with *ggplot2* - Scatterplot

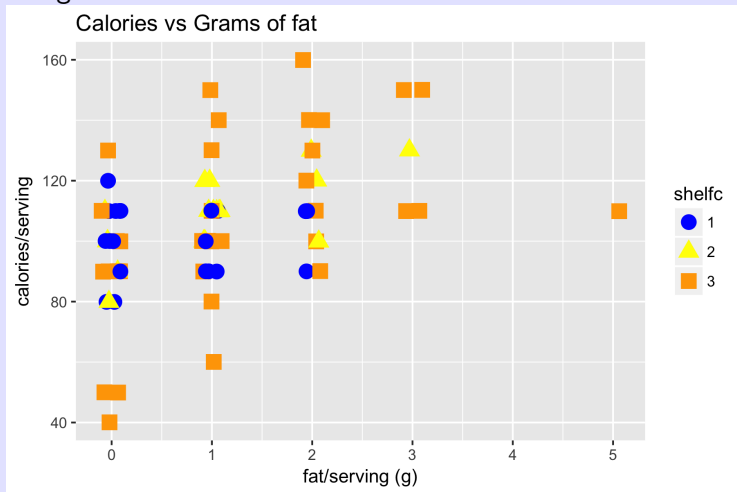
Create a plot of calories vs. grams of fat - color/shape by categorical variable.

Specify the colors using a named vector - preferred method rather than relying on alphabetical ordering, but need to create character variable in place of any numeric codes.

```
1 cereal$shelfc <- as.character(cereal$shelf)
2 shelf_colors=c("1"="blue", "2"="yellow", "3"="orange")
3 newplot <- ggplot(data=cereal, aes(x=fat, y=calories,
4   color=shelfc, shape=shelfc))+
5   ggtitle("Calories vs Grams of fat")+
6   xlab("fat/serving (g)") + ylab("calories/serving")+
7   geom_jitter(position=position_jitter(w=.1, h=.1), size=4)
8   scale_color_manual( values=shelf_colors)
9 newplot
```

# Plotting with *ggplot2* - Scatterplot

Create a plot of calories vs. grams of fat - color/shape by categorical variable



# Plotting with *ggplot2* - Scatterplot

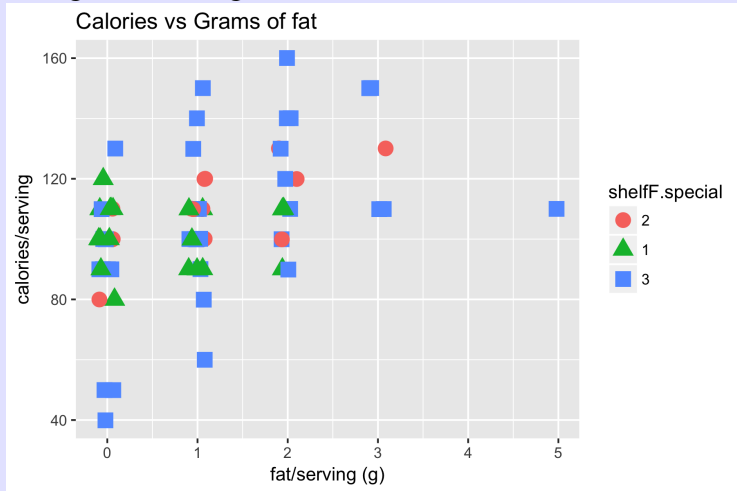
Change order of legend.

(1) Define the factor with the ordering you want

```
1 cereal$shelfF.special <- factor(cereal$shelf,  
2   levels=c(2,1,3))  
3 newplot <- ggplot(data=cereal, aes(x=fat, y=calories,  
4   color=shelfF.special,  
5   shape=shelfF.special))+  
6   ggtitle("Calories vs Grams of fat")+  
7   xlab("fat/serving (g)") + ylab("calories/serving")+  
8   geom_jitter(position=position_jitter(w=.1, h=.1), size=4)  
9 newplot
```

# Plotting with *ggplot2* - Scatterplot

## Change order of legend





# Plotting with *ggplot2* - Scatterplot

Change order of legend.

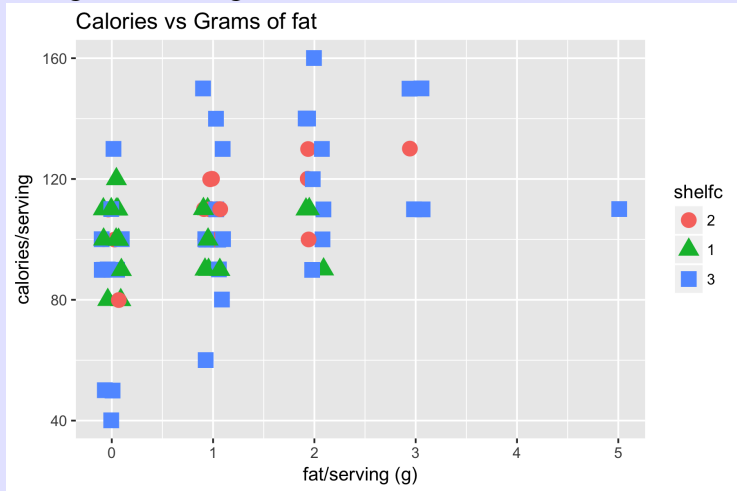
(2) Define the order in the plot

```
1 cereal$shelfc <- as.character(cereal$shelf)
2 newplot <- ggplot(data=cereal, aes(x=fat, y=calories, color=
3   ggtitle("Calories vs Grams of fat")+
4   xlab("fat/serving (g)") + ylab("calories/serving") +
5   geom_jitter(position=position_jitter(w=.1, h=.1), size=4)
6   scale_color_discrete( limits=c("2","1","3")) +
7   scale_shape_discrete( limits=c("2","1","3")))
8 newplot
```

Must be a character variable that defines the groups.

# Plotting with *ggplot2* - Scatterplot

## Change order of legend



# Plotting with *ggplot2* - Scatterplot

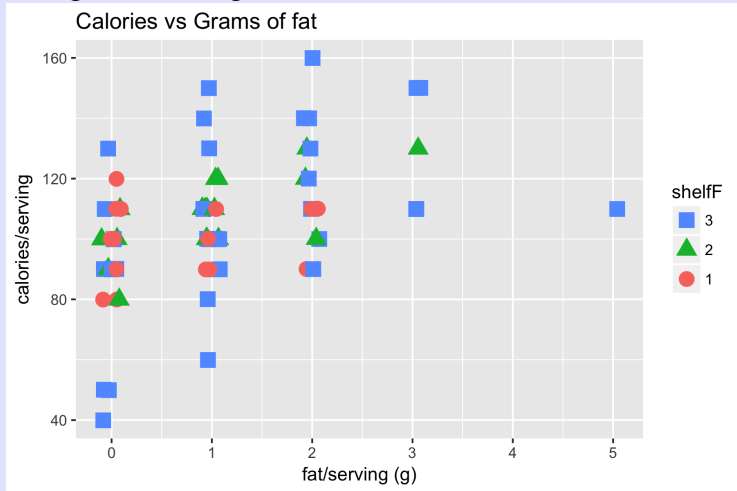
Reversing order of the legend

```
1 newplot <- ggplot(data=cereal, aes(x=fat, y=calories, color=
2   ggtitle("Calories vs Grams of fat")+
3   xlab("fat/serving (g)") + ylab("calories/serving") +
4   geom_jitter(position=position_jitter(w=.1, h=.1), size=4)
5   scale_color_discrete(guide = guide_legend(reverse=TRUE))
6   scale_shape_discrete(guide = guide_legend(reverse=TRUE))
7 newplot
```

Need both `scale__discrete`.

# Plotting with *ggplot2* - Scatterplot

## Change order of legend



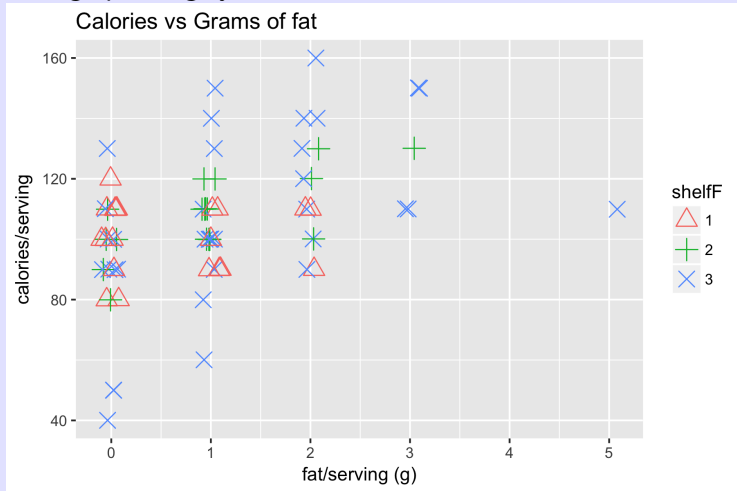
## Plotting with *ggplot2* - Scatterplot

Change the plotting symbol - similar to changing color

```
1 newplot <- ggplot(data=cereal, aes(x=fat, y=calories, color=
2   ggtitle("Calories vs Grams of fat")+
3   xlab("fat/serving (g)") + ylab("calories/serving") +
4   geom_jitter(position=position_jitter(w=.1, h=.1), size=4)
5   scale_shape_manual(values=c(2,3,4))
6 newplot
```

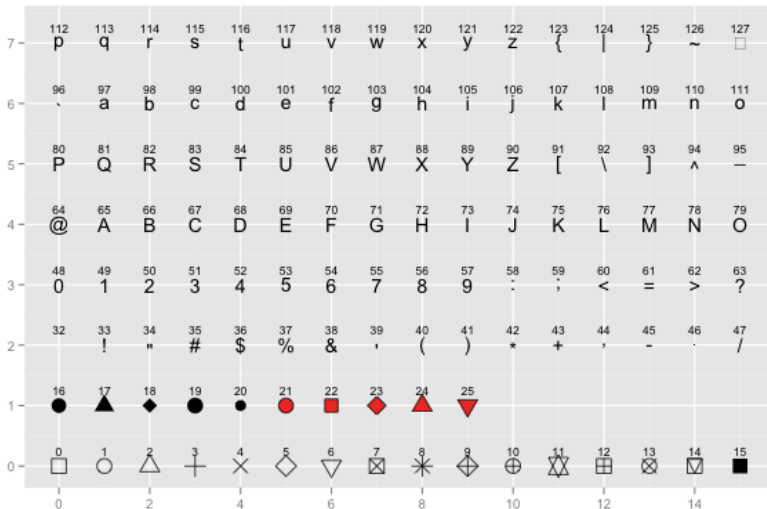
# Plotting with *ggplot2* - Scatterplot

## Change plotting symbols



# Plotting with *ggplot2* - Scatterplot

Different plotting symbols (see *help(pch)*)



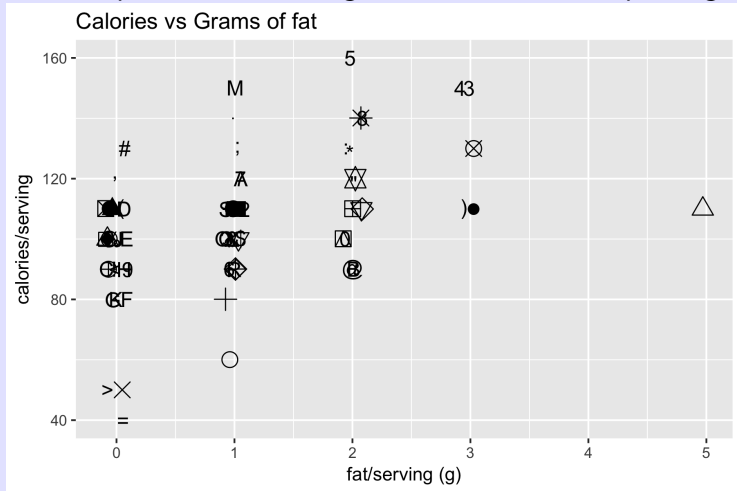
## Plotting with *ggplot2* - Scatterplot

```
1  # Different symbol for each point
2  newplot <- ggplot(cereal, aes(x=fat, y=calories))+
3    ggtitle("calories vs Grams of fat")+
4    xlab("fat/serving (g)") + ylab("calories/serving")+
5    geom_jitter(position=position_jitter(w=.1, h=.1),
6               shape=c(1:25,32:83))
7  newplot
```



## Plotting with *ggplot2* - Scatterplot

## Create a plot of calories vs. grams of fat - different plotting symbols



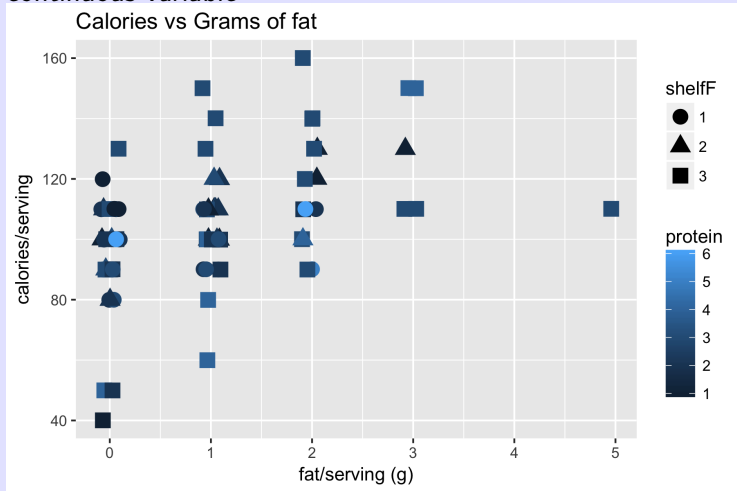
## Plotting with *ggplot2* - Scatterplot

Create a plot of calories vs. grams of fat - color by continuous variable

```
1 newplot <- ggplot(cereal, aes(x=fat, y=calories,  
2                               color=protein, shape=shelfF))+  
3   ggtitle("Calories vs Grams of Fat")+  
4   xlab("Fat/serving (g)") + ylab("Calories/serving")+  
5   geom_jitter(position=position_jitter(w=.1, h=.1), size=4)  
6  
7 newplot <- ggplot(cereal, aes(x=fat, y=calories,  
8                               color=protein, shape=shelfF))+  
9   ggtitle("Calories vs Grams of Fat")+  
10  xlab("Fat/serving (g)") + ylab("Calories/serving")+  
11  geom_jitter(position=position_jitter(w=.1, h=.1), size=4)-  
12  scale_color_gradient(high="black", low="blue")
```

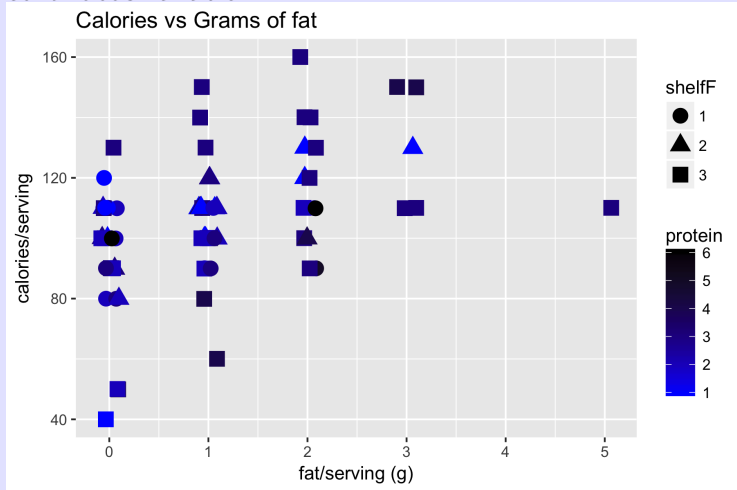
# Plotting with *ggplot2* - Scatterplot

Create a plot of calories vs. grams of fat - color/shape by continuous variable



# Plotting with *ggplot2* - Scatterplot

Create a plot of calories vs. grams of fat - color/shape by continuous variable



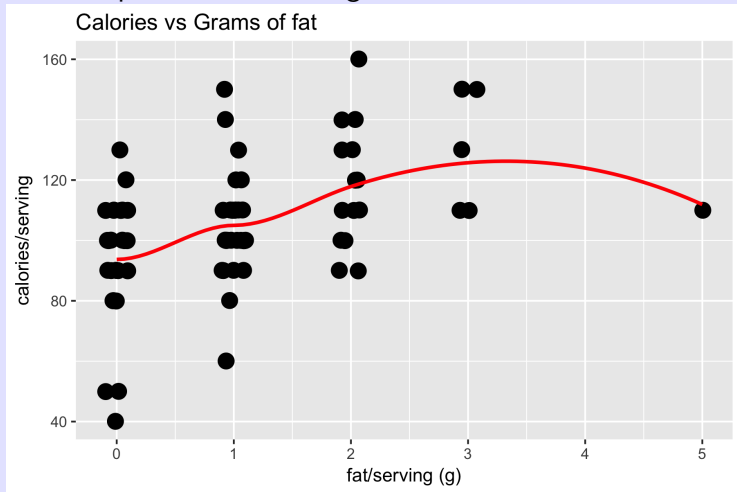
# Plotting with *ggplot2* - Scatterplot

Create a plot of calories vs. grams of fat; add a smoother

```
1 newplot <- ggplot(cereal, aes(x=fat, y=calories))+
2   ggtitle("calories vs Grams of fat")+
3   xlab("fat/serving (g)") + ylab("calories/serving")+
4   geom_jitter(position=position_jitter(w=.1, h=.1))+
5   geom_smooth(method="loess", se=FALSE, color="red")
6 newplot
7
8 newplot <- ggplot(cereal, aes(x=fat, y=calories))+
9   ggtitle("calories vs Grams of fat")+
10  xlab("fat/serving (g)") + ylab("calories/serving")+
11  geom_jitter(position=position_jitter(w=.1, h=.1))+
12  geom_smooth(method="loess", se=FALSE, color="red")+
13  geom_smooth(method="lm", color="black")
14 newplot
```

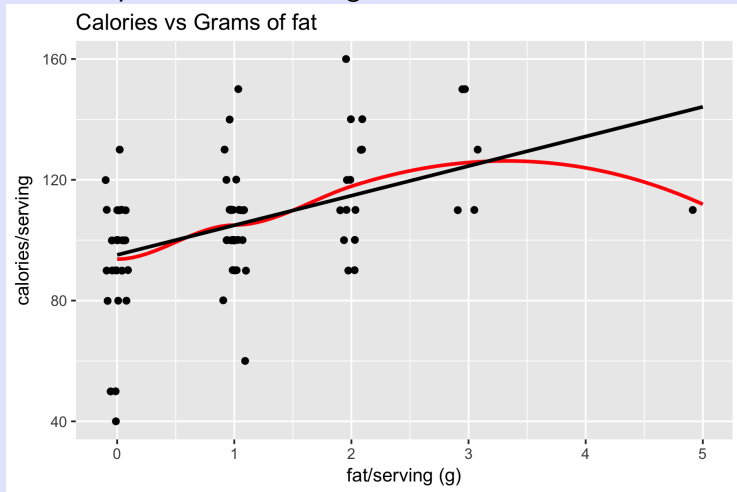
# Plotting with *ggplot2* - Scatterplot

Create a plot of calories vs. grams of fat - smoothers



# Plotting with *ggplot2* - Scatterplot

Create a plot of calories vs. grams of fat - smoothers



# Plotting with *ggplot2* - Modifying the legend

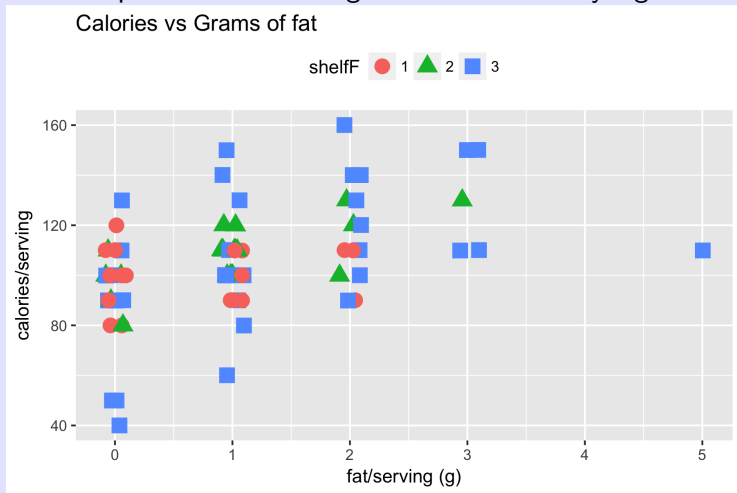
[http://www.cookbook-r.com/Graphs/Legends\\_\(ggplot2\)](http://www.cookbook-r.com/Graphs/Legends_(ggplot2))

```
1 newplot <- ggplot(cereal, aes(x=fat, y=calories,  
2                             shape=shelfF, color=shelfF))+  
3   ggtitle("Calories vs Grams of fat")+  
4   xlab("fat/serving (g)") + ylab("calories/serving")+  
5   geom_jitter(position=position_jitter(w=.1, h=.1), size=4)-  
6   theme(legend.position="top")  
7 newplot  
8  
9 # Set the "anchoring point" of the legend  
10 # (bottom-left is 0,0; top-right is 1,1)  
11 # Numeric positions are relative to the entire area,  
12 # including titles and labels, not just the plotting area.  
13 newplot <- ggplot(cereal, aes(x=fat, y=calories,  
14                             shape=shelfF, color=shelfF))+  
15   ggtitle("Calories vs Grams of fat")+  
16   xlab("fat/serving (g)") + ylab("calories/serving")+  
17   geom_jitter(position=position_jitter(w=.1, h=.1), size=4)-  
18   theme(legend.justification=c(1,0),  
19         legend.position=c(1,0))
```



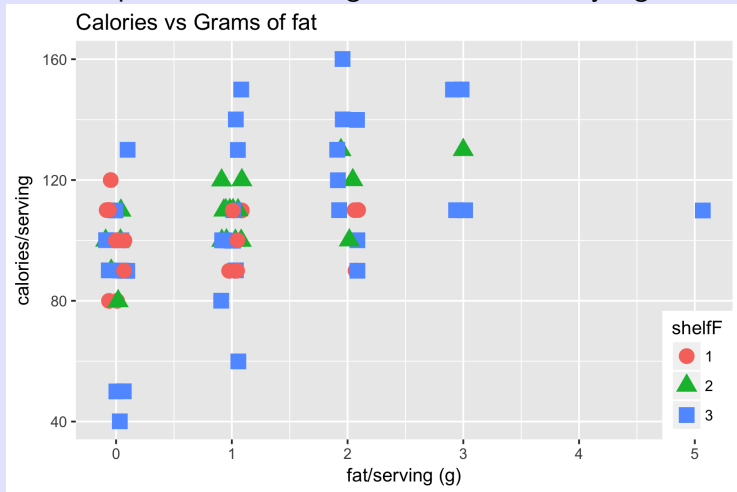
# Plotting with *ggplot2* - Scatterplot

Create a plot of calories vs. grams of fat - modify legend



# Plotting with *ggplot2* - Scatterplot

Create a plot of calories vs. grams of fat - modify legend



## Plotting with *ggplot2* - Modifying the legend

[http://www.cookbook-r.com/Graphs/Legends\\_\(ggplot2\)](http://www.cookbook-r.com/Graphs/Legends_(ggplot2))

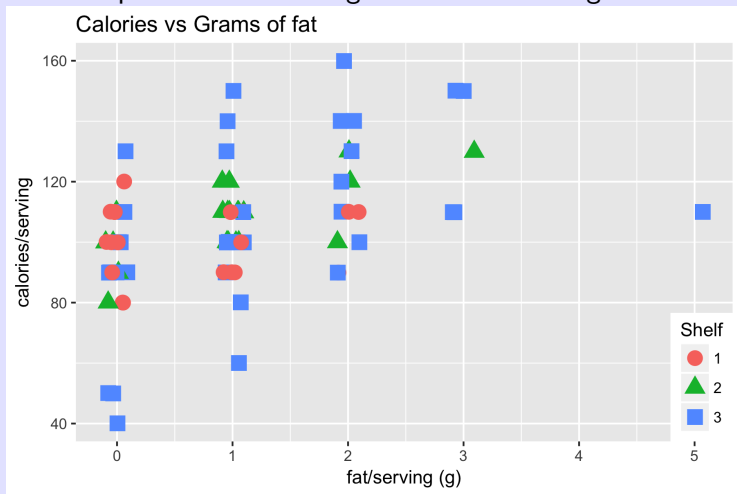
Change legend title.

```
1 newplot <- ggplot(data=cereal, aes(x=fat, y=calories,  
2                               shape=shelfF, color=shelfF))+  
3   ggtitle("Calories vs Grams of fat")+  
4   xlab("fat/serving (g)") + ylab("calories/serving")+  
5   geom_jitter(position=position_jitter(w=.1, h=.1), size=4)+  
6   theme(legend.justification=c(1,0), legend.position=c(1,0))  
7   scale_color_discrete(name="Shelf")+  
8   scale_shape_discrete(name="Shelf")  
9 newplot
```

Notice you need BOTH scale commands.

# Plotting with *ggplot2* - Scatterplot

Create a plot of calories vs. grams of fat - set legend title



## Plotting with *ggplot2* - Modifying the legend

[http://www.cookbook-r.com/Graphs/Legends\\_\(ggplot2\)](http://www.cookbook-r.com/Graphs/Legends_(ggplot2))

Use long legend title.

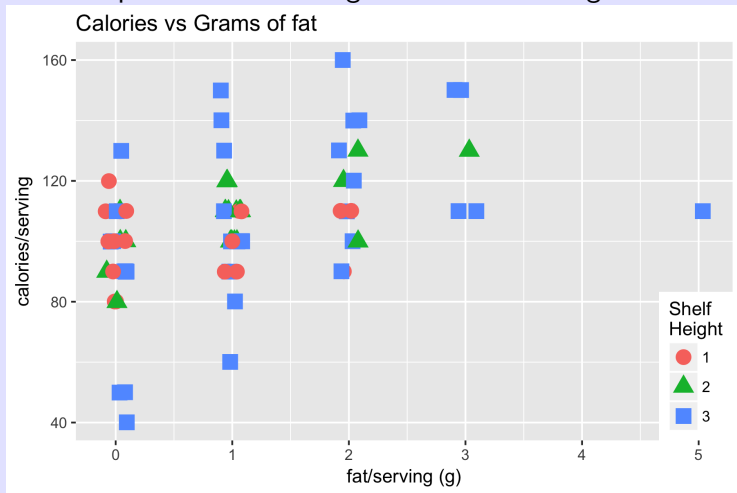
```
1 newplot <- ggplot(data=cereal, aes(x=fat, y=calories,  
2                                shape=shelfF, color=shelfF))+  
3   ggtitle("Calories vs Grams of fat")+  
4   xlab("fat/serving (g)") + ylab("calories/serving")+  
5   geom_jitter(position=position_jitter(w=.1, h=.1), size=4)+  
6   theme(legend.justification=c(1,0), legend.position=c(1,0))+  
7   scale_color_discrete(name="Shelf\nHeight")+  
8   scale_shape_discrete(name="Shelf\nHeight")  
9 newplot
```

Notice you need BOTH scale commands.

Same method used for long axis labels, title, etc.

# Plotting with *ggplot2* - Scatterplot

Create a plot of calories vs. grams of fat - set legend title



## Plotting with *ggplot2* - Modifying the legend

[http://www.cookbook-r.com/Graphs/Legends\\_\(ggplot2\)](http://www.cookbook-r.com/Graphs/Legends_(ggplot2))

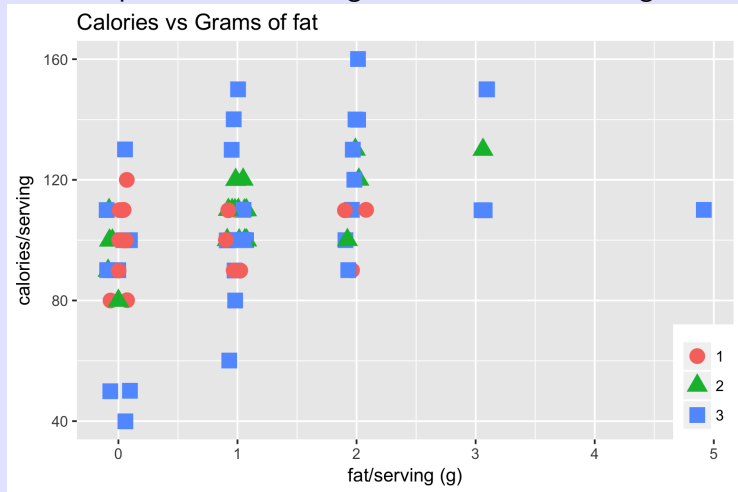
Remove legend title.

```
1 newplot <- ggplot(data=cereal, aes(x=fat, y=calories,  
2                                   shape=shelfF, color=shelfF))+  
3   ggtitle("Calories vs Grams of fat")+  
4   xlab("fat/serving (g)") + ylab("calories/serving")+  
5   geom_jitter(position=position_jitter(w=.1, h=.1), size=4)+  
6   theme(legend.justification=c(1,0), legend.position=c(1,0))  
7   scale_color_discrete(name=NULL)+  
8   scale_shape_discrete(name=NULL)  
9 newplot
```

Notice you need BOTH scale commands.

# Plotting with *ggplot2* - Scatterplot

Create a plot of calories vs. grams of fat - remove legend title



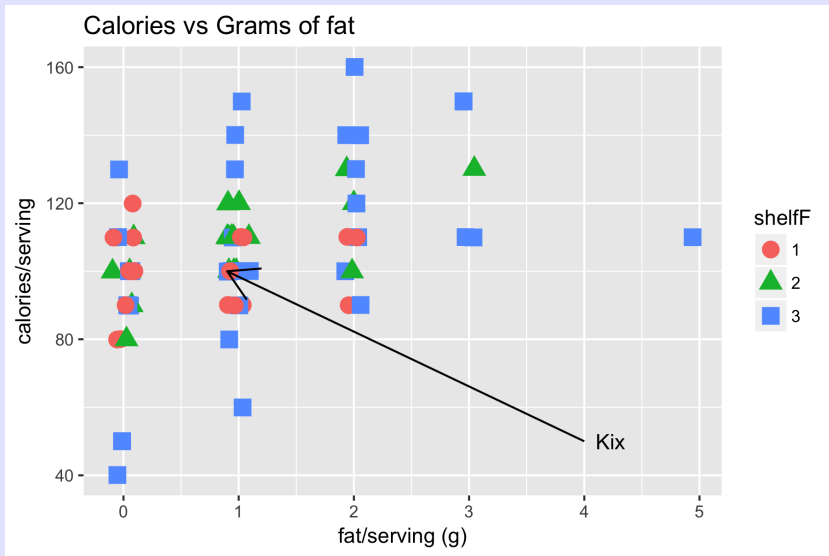


# Plotting with *ggplot2* - Scatterplot

Add a call out

```
1 fc <- cereal[ cereal$name == "Kix",]
2 fc
3
4 newplot <- ggplot(cereal, aes(x=fat, y=calories,
5                               shape=shelfF, color=shelfF))+
6   ggtitle("Calories vs Grams of fat")+
7   xlab("fat/serving (g)") + ylab("calories/serving")+
8   geom_jitter(position=position_jitter(w=.1, h=.1), size=4)+
9   annotate("segment",
10          x=4, y=50,
11          xend=fc$fat-.1, yend=fc$calories,
12          arrow=arrow(ends="last")
13          )+
14   annotate("text",
15          x=4.1, y=50,
16          label="Kix",
17          hjust=0)
18 newplot
```

# Plotting with *ggplot2* - Scatterplot



# Plotting with *ggplot2* - Cheat sheets

- [`http://www.cookbook-r.com/Graphs/Legends\_\(ggplot2\)`](http://www.cookbook-r.com/Graphs/Legends_(ggplot2))
- [`http://zevross.com/blog/2014/08/04/beautiful-plotting-in-r-a-ggplot2-cheatsheet-3/`](http://zevross.com/blog/2014/08/04/beautiful-plotting-in-r-a-ggplot2-cheatsheet-3/)
- *Rstudio* → Help → Cheatsheets

Return to the Birds 'n Butts dataset.

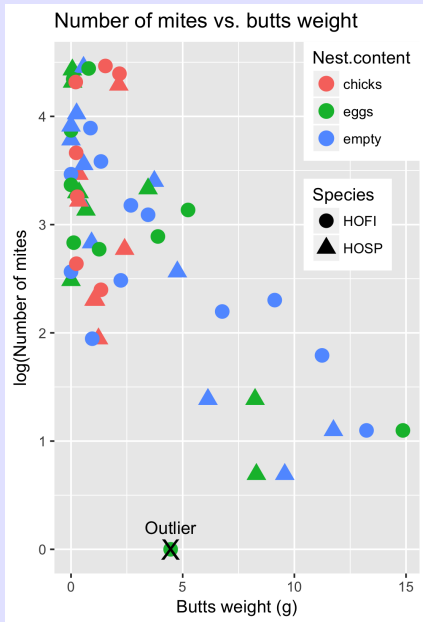
Plot the  $\log(\text{number of parasites})$  vs. butts weight in nest with a different symbol/colour for a point depending on the species and nest content.

- Use different symbol for each species
- Use different color for each nest content type
- Don't forget the legend.
- Add a callout for the outlier.

## Plotting with *ggplot2* - Exercise

```
1  prelimplotlog <- ggplot(data=butts,  
2                          aes(x=Butts.weight, y=log.mites,  
3                              shape=Species, color=Nest.content))+  
4  ggtitle("Number of mites vs. butts weight")+  
5  xlab("Butts weight (g)") + ylab("log(Number of mites)')+  
6  geom_point(size=4)+  
7  annotate("point",  
8          x=outlier$Butts.weight,  
9          y=outlier$log.mites,  
10         shape="X",size=6)+  
11  annotate("text",  
12         x=outlier$Butts.weight,  
13         y=outlier$log.mites+.2,  
14         label="Outlier")+  
15  theme(legend.justification=c(1,1), legend.position=c(1  
16  prelimplotlog
```

# Plotting with *ggplot2* - Exercise



Return to the Birds 'n Butts dataset.

Compute the mean number of mites by combination of species and egg contents along with the 95% confidence intervals (use the examples previously)

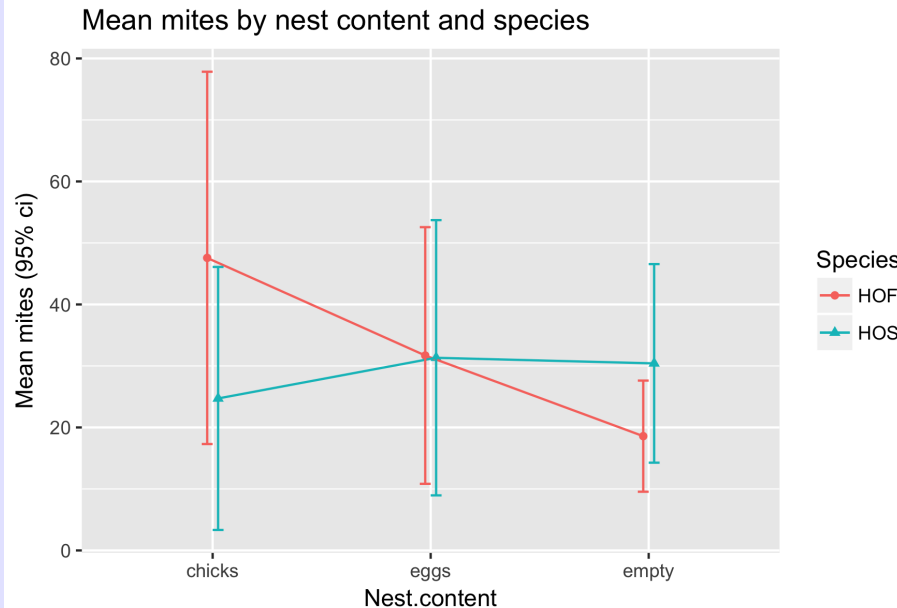
- Create a profile plot (mean by nest contents by species) with and without dodging.

## Plotting with *ggplot2* - Exercise

```
1 report <- plyr::ddply(butts, c("Species","Nest.content"),
2   sf.simple.summary, variable="Number.of.mites", crd=TRUE)
3 report
4
5 newplot <- ggplot2::ggplot(report, aes(x=Nest.content, y=mean,
6   group=Species, color=Species))
7   ggtitle("Mean mites by nest content and species")+
8   geom_point( position=position_dodge(w=0.1))+
9   geom_line(position=position_dodge(w=0.1))+
10   geom_errorbar(aes(ymin=lcl, ymax=ucl), width=.1, position=position_dodge(w=0.1))
11   ylab("Mean mites (95% ci)")
12 newplot
```



# Plotting with *ggplot2* - Exercise

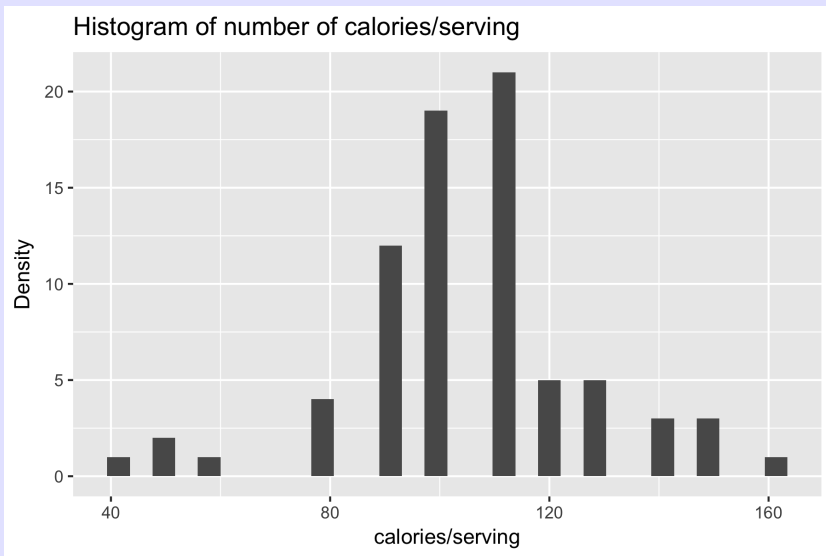


# Plotting with *ggplot2* - Histograms

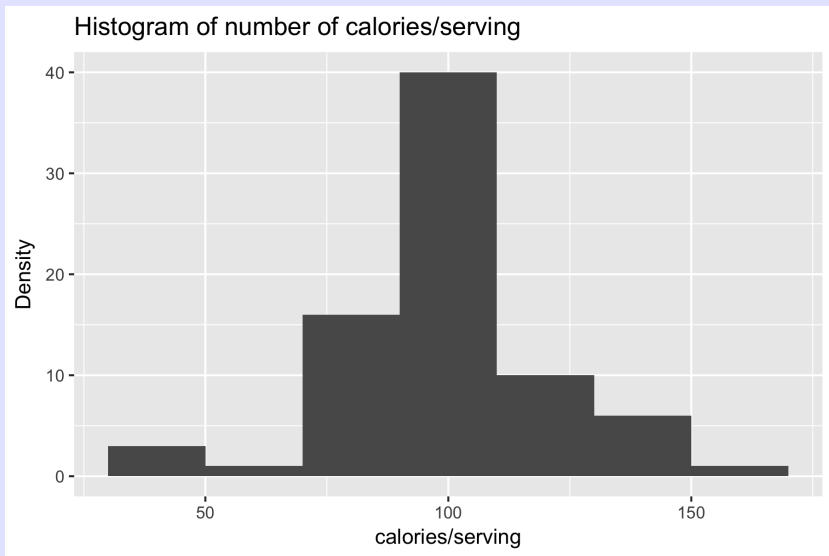
Histograms use the *geom\_hist()* geometry. Note only x aesthetics used.

```
1 histplot <- ggplot(data=cereal, aes(x=calories))+  
2   ggtitle("Histogram of number of calories/serving")+  
3   xlab("calories/serving")+ylab("Density")+  
4   geom_histogram()  
5 histplot  
6  
7 histplot <- ggplot(data=cereal, aes(x=calories))+  
8   ggtitle("Histogram of number of calories/serving")+  
9   xlab("calories/serving")+ylab("Density")+  
10  geom_histogram(binwidth=20 )  
11 histplot
```

# Plotting with *ggplot2* - Histograms



# Plotting with *ggplot2* - Histograms



# Plotting with *ggplot2* - Histograms

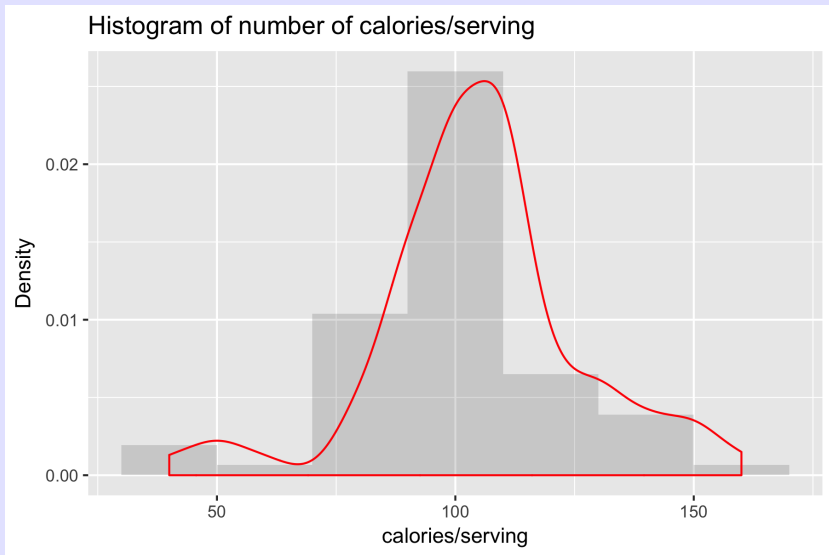
Histograms use the *geom\_hist()* geometry. Note only x aesthetics used. Adding a non-parametric density estimate.

```
1 histplot <- ggplot(data=cereal, aes(x=calories))+  
2   ggtitle("Histogram of number of calories/serving")+  
3   xlab("calories/serving")+ylab("Density")+  
4   geom_histogram(aes(y=..density..),binwidth=20,alpha=0.2)+  
5   geom_density(color='red')  
6 histplot
```

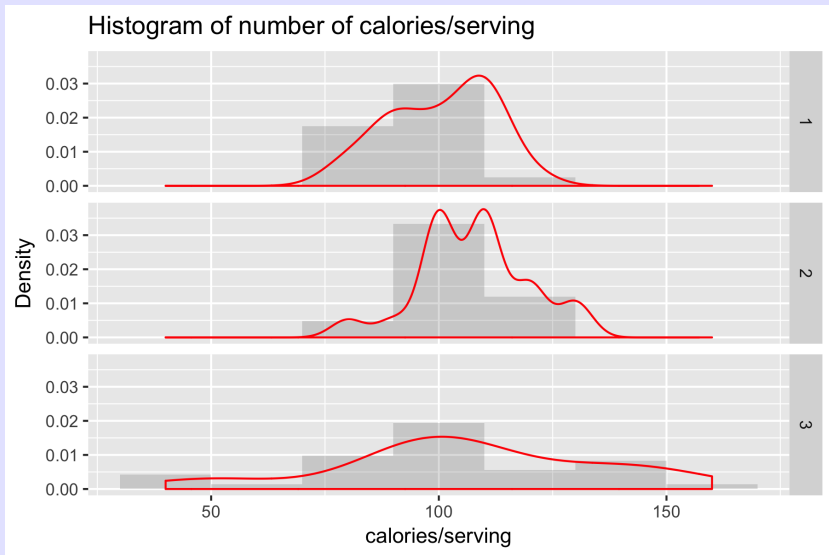
Example of using *facets* - more will be covered later

```
1 histplot <- ggplot(data=cereal, aes(x=calories))+  
2   ggtitle("Histogram of number of calories/serving")+  
3   xlab("calories/serving")+ylab("Density")+  
4   geom_histogram(aes(y=..density..),binwidth=20,alpha=0.2)+  
5   geom_density(color='red')+  
6   facet_grid(shelfF~ .)  
7 histplot
```

# Plotting with *ggplot2* - Histograms



# Plotting with *ggplot2* - Histograms



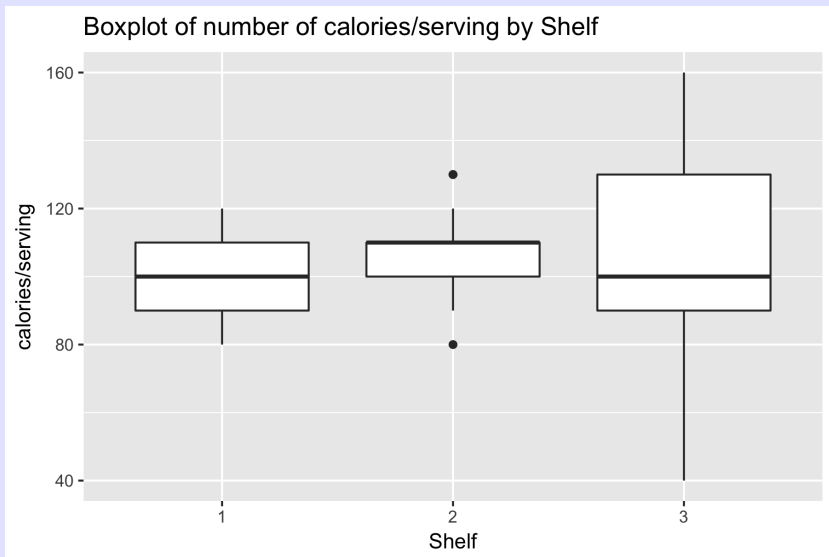
# Plotting with *ggplot2* - Boxplots

## Side-by-side box plots

```
1 boxplot <- ggplot(data=cereal, aes(x=shelfF, y=calories))+  
2   ggtitle("Boxplot of number of calories/serving by Shelf")-  
3   ylab("calories/serving")+xlab("Shelf")+  
4   geom_boxplot()  
5 boxplot
```



# Plotting with *ggplot2* - Boxplots



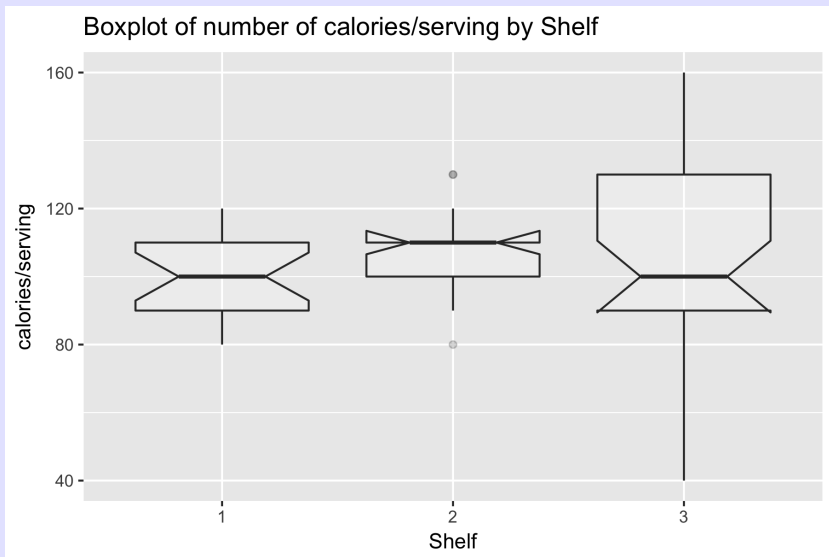
## Plotting with *ggplot2* - Boxplots

Side-by-side box plots. Change transparency and add notches.

```
1 boxplot <- ggplot(data=cereal, aes(x=shelfF, y=calories))+  
2   ggtitle("Boxplot of number of calories/serving by Shelf")-  
3   ylab("calories/serving")+xlab("Shelf")+  
4   geom_boxplot(alpha=0.2, notch=TRUE)  
5 boxplot
```

Notched box-plots provide a way to compare if population  
MEDIANs are approximately equal across multiple groups.

# Plotting with *ggplot2* - Boxplots



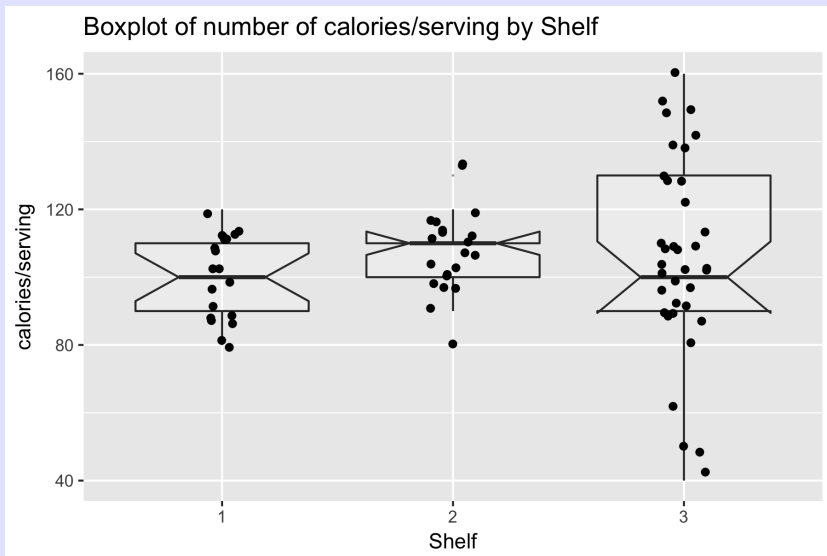
## Plotting with *ggplot2* - Boxplots

Side-by-side box plots with data points overlaid

```
1 boxplot <- ggplot(data=cereal, aes(x=shelfF, y=calories))+  
2   ggtitle("Boxplot of number of calories/serving by Shelf")-  
3   ylab("calories/serving")+xlab("Shelf")+  
4   geom_boxplot(alpha=0.2, notch=TRUE, outlier.size=0)+  
5   geom_point(position=position_jitter(width=0.1))  
6 boxplot
```

'CAUTION: If there are outliers, you might get double plotting. Set *outlier.size=0* in *geom\_boxplot()*

# Plotting with *ggplot2* - Boxplots



Usually only used for screening data for outliers.

## Plotting with *ggplot2* - Boxplots

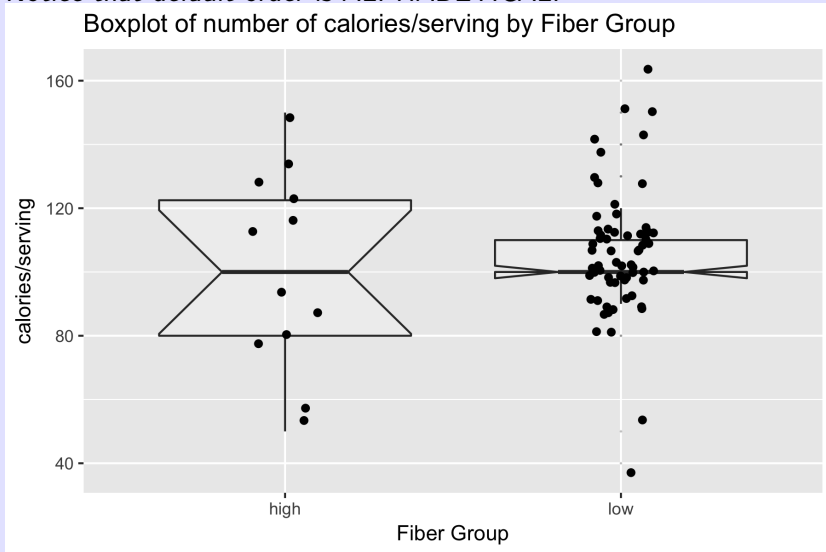
Side-by-side box plots with data points overlaid and order of boxes changed.

Usual to set up a **FACTOR** (more on this later) with the correct ordering

```
1 cereal$fiber.grp <- recode(cereal$fiber,  
2                           "lo:3='low'; 3:hi='high'")  
3 xtabs(~fiber.grp, data=cereal)  
4 cereal$fiber.grpF <- factor(cereal$fiber.grp)  
5  
6 boxplot <- ggplot(data=cereal,  
7                   aes(x=fiber.grpF, y=calories))+  
8   ggtitle("Boxplot of number of calories/serving by Fiber Gr  
9   xlab("calories/serving")+ylab("Fiber Group")+  
10  geom_boxplot(alpha=0.2, notch=TRUE, outlier.size=0)+  
11  geom_point(position=position_jitter(width=0.1))  
12 boxplot
```

# Plotting with *ggplot2* - Boxplots

Notice that default order is ALPHABETICAL.



Box plots usually only used for screening data for outliers.

## Plotting with *ggplot2* - Boxplots

Side-by-side box plots with data points overlaid and order of boxes changed.

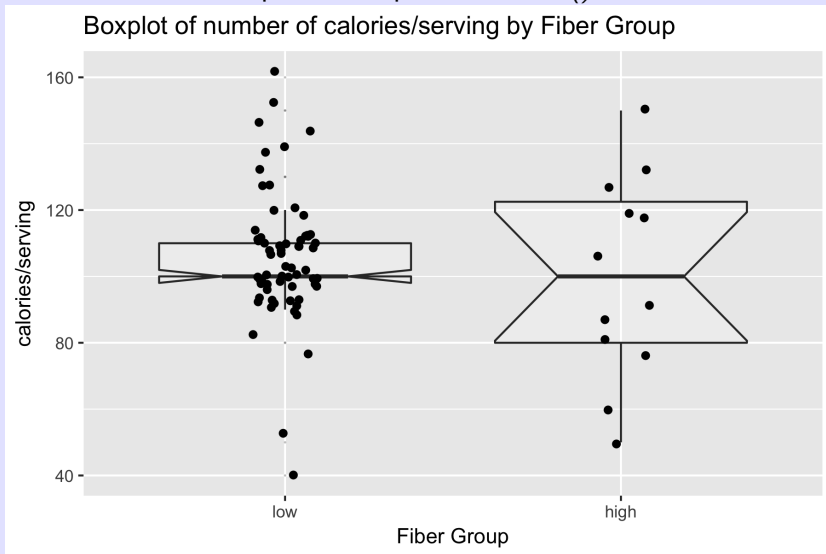
Usual to set up a **FACTOR** (more on this later) with the correct ordering

```
1 cereal$fiber.grpF <- factor(cereal$fiber.grp,  
2                             levels=c("low","high"), order=TRUE)  
3  
4 boxplot <- ggplot(data=cereal, aes(x=fiber.grpF, y=calories))  
5   ggtitle("Boxplot of number of calories/serving by Fiber Gr")  
6   xlab("calories/serving")+ylab("Fiber Group")+  
7   geom_boxplot(alpha=0.2, notch=TRUE, outlier.size=0)+  
8   geom_point(position=position_jitter(width=0.1))  
9 boxplot
```



# Plotting with *ggplot2* - Boxplots

Notice that order is specified in previous *factor()* function.



Box plots usually only used for screening data for outliers.

## Plotting with *ggplot2* - Exercise

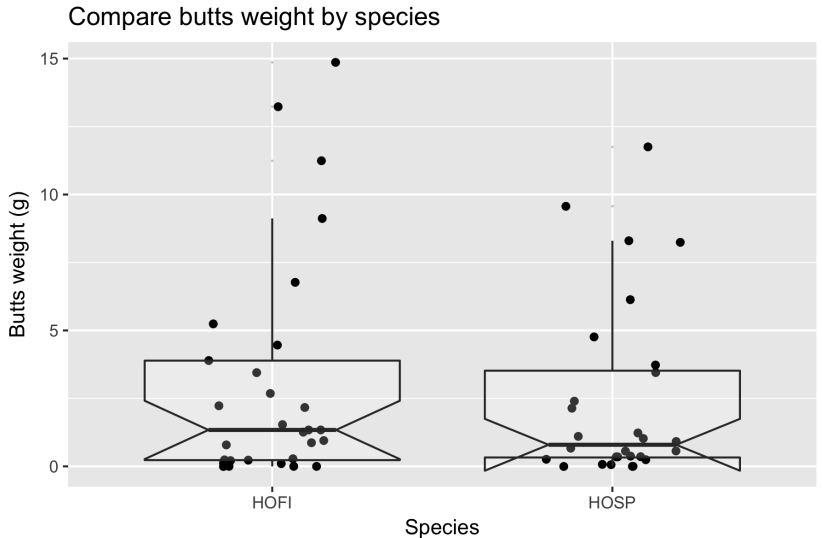
Return to the Birds 'n Butts dataset.

Compare the number of parasites and butts weight by species and/or nest content using box-plots. Try both the regular and `log()` scales. What do you conclude?

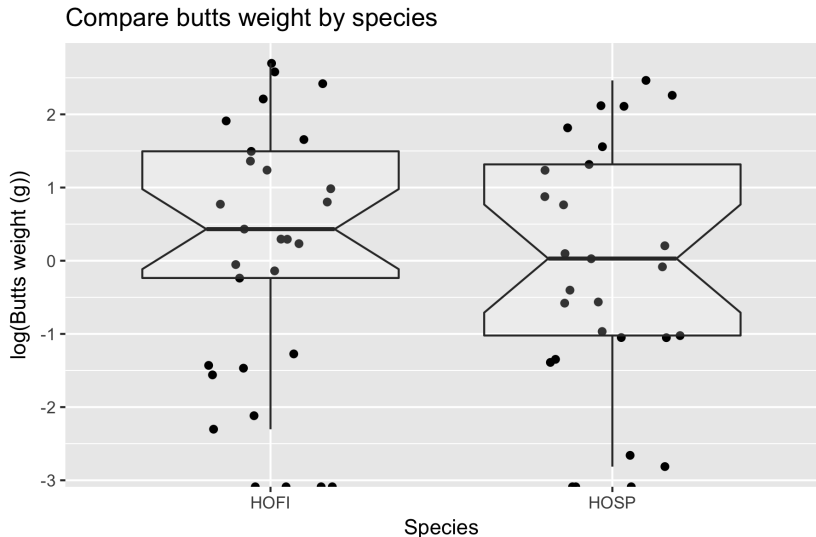
## Plotting with *ggplot2* - Exercise

```
1 birdbox <- ggplot(data=butts,  
2                   aes(x=Species, y=Butts.weight))+  
3                   ggtitle("Compare butts weight by species")+  
4                   xlab("Species")+ylab("Butts weight (g)")+  
5                   geom_point(position=position_jitter(w=0.2))+  
6                   geom_boxplot(notch=TRUE, alpha=0.2,outlier.size=10)  
7 birdbox
```

# Plotting with *ggplot2* - Exercise



# Plotting with *ggplot2* - Exercise



Perhaps on the log-scale is better comparison (except for  $\log(0)$ ?)

## Plotting with *ggplot2* - Bar/line charts with error bars

```
1 source("schwarz.functions.r") # load some helper functions
2
3 # Compute the mean calories/serving by fiber group with se
4 sumstat.all <- sf.simple.summary(cereal, "calories",
5                                 crd=TRUE)
6 sumstat.all
7
8 library(plyr)
9 sumstat.shelf <- ddply(cereal, "shelfF", sf.simple.summary,
10                       variable="calories", crd=TRUE)
11 sumstat.shelf
```

## Plotting with *ggplot2* - Bar/line charts with error bars

```
> sumstat.all
```

	n	nmiss	mean	sd	se	
	77.000000	0.000000	105.064935	21.620128	2.463842	100.1

```
> sumstat.shelf
```

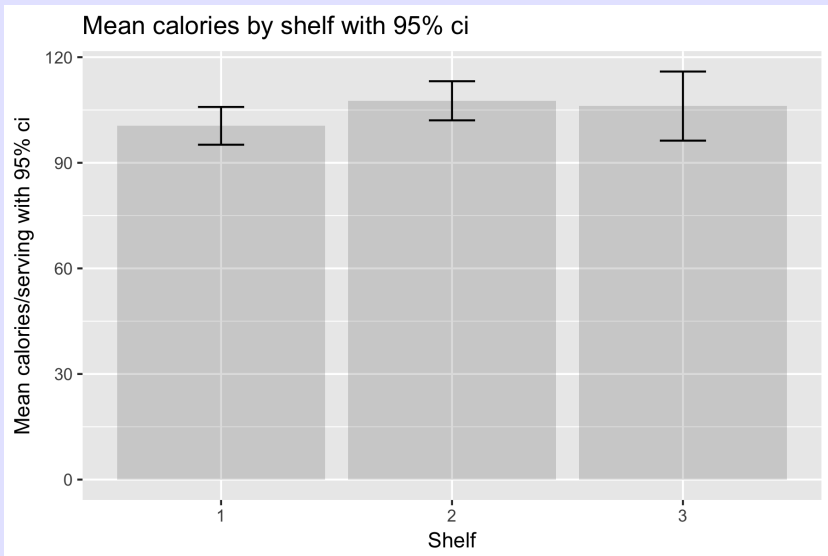
	shelfF	n	nmiss	mean	sd	se	lcl	
1	1	20	0	100.5000	11.45931	2.562380	95.13688	105.8
2	2	21	0	107.6190	12.20851	2.664114	102.06180	113.1
3	3	36	0	106.1111	29.01012	4.835021	96.29550	115.9

## Plotting with *ggplot2* - Bar/line charts with error bars

```
1 # Side-by-side bar charts
2 cerealbar <- ggplot(data=sumstat.shelf,
3                     aes(x=shelfF, y=mean))+
4   ggtitle("Mean calories by shelf with 95% ci")+
5   xlab("Shelf")+ylab("Mean calories/serving with 95% ci")+
6   geom_bar(stat="identity", alpha=0.2)+
7   geom_errorbar(aes(ymin=lcl, ymax=ucl), width=0.2)
8 cerealbar
```



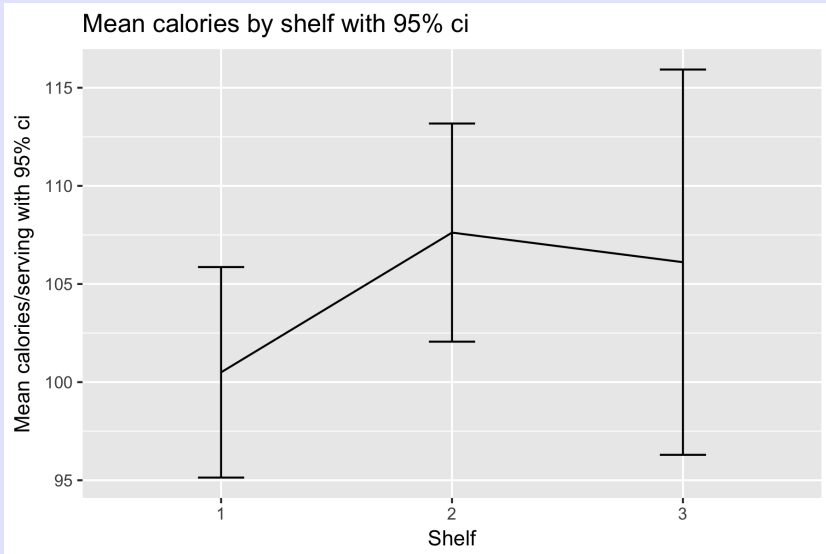
# Plotting with *ggplot2* - Bar/line charts with error bars



## Plotting with *ggplot2* - Bar/line charts with error bars

```
1 # A line chart is similar, except you don't need to start
2 # the bars at zero
3 cerealline <- ggplot(data=sumstat.shelf,
4                      aes(x=shelfF, y=mean))+
5   ggtitle("Mean calories by shelf with 95% ci")+
6   xlab("Shelf")+ylab("Mean calories/serving wth 95% ci")+
7   geom_line(aes(group=1))+
8   geom_errorbar(aes(ymin=lcl, ymax=ucl), width=0.2)
9 cerealline
```

# Plotting with *ggplot2* - Bar/line charts with error bars



## Plotting with *ggplot2* - Exercise

Return to the Birds 'n Butts dataset.

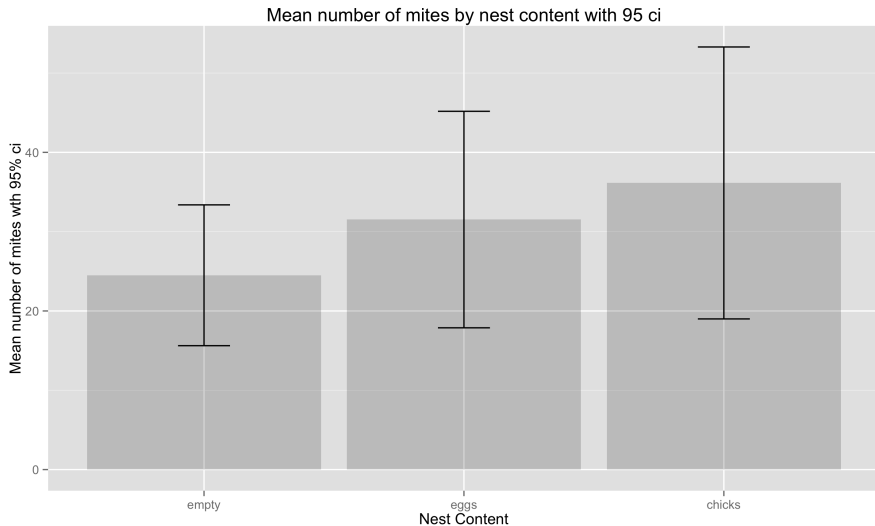
Summarize the number of parasites and butts weight by nest content using the code given in the notes.

Compare the means using bar/line plots with confidence intervals. Try both the regular and `log()` scales. What do you conclude?

## Plotting with *ggplot2* - Exercise

```
1 library(plyr)
2 sumstat <- ddply(butts, "Nest.content", sf.simple.summary,
3                 variable="Number.of.mites", crd=TRUE)
4 sumstat$Nest.contentF <- factor(sumstat$Nest.content,
5                                 levels=c("empty", "eggs", "chicks"),
6                                 order=TRUE)
7 sumstat
8
9 nestbar <- ggplot(data=sumstat,
10                  aes(x=Nest.contentF, y=mean))+
11   ggtitle("Mean number of mites by nest content with 95% c")
12   xlab("Nest Content")+ylab("Mean number of mites with 95%")
13   geom_bar(stat="identity", alpha=0.2)+
14   geom_errorbar(aes(ymin=lcl, ymax=ucl), width=0.2)
15 nestbar
```

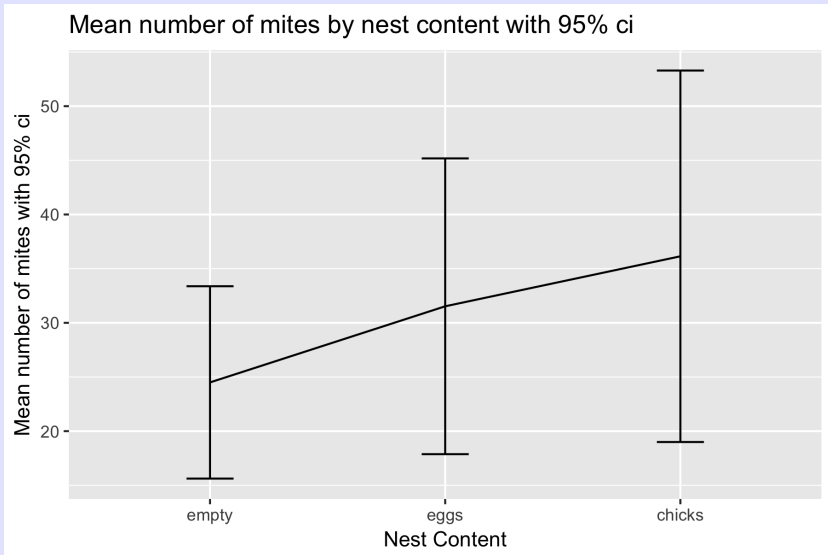
# Plotting with *ggplot2* - Exercise



## Plotting with *ggplot2* - Exercise

```
1 nestline <- ggplot(data=sumstat,  
2                   aes(x=Nest.contentF, y=mean))+  
3   ggtitle("Mean number of mites by nest content with 95% c  
4   xlab("Nest Content")+ylab("Mean number of mites with 95%  
5   geom_line(aes(group=1))+  
6   geom_errorbar(aes(ymin=lcl, ymax=ucl), width=0.2)  
7 nestline
```

## Plotting with *ggplot2* - Exercise





## Plotting with *ggplot2* - Facetting

Make separate panels by one or two variables.

```
... facet_wrap( ~ v2 , ncol=2)
```

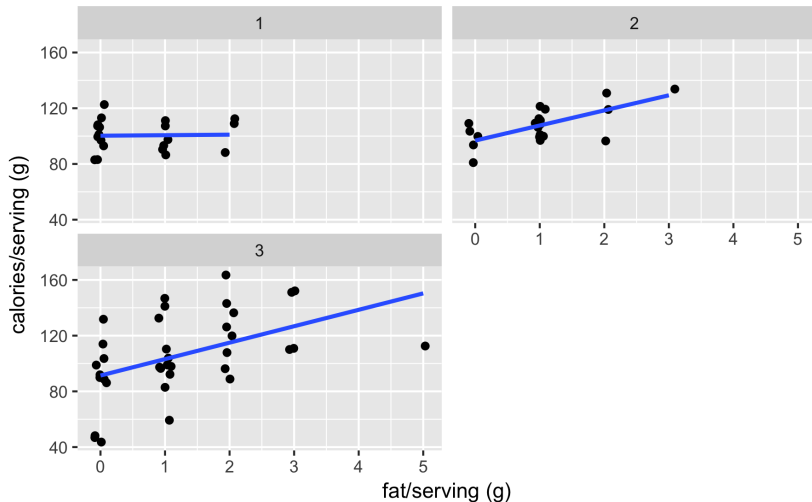
```
... facet_grid( v1 ~ v2)
```

Scales can be free or fixed on both *X* and *Y* axes.

```
1 cerealplot <- ggplot(data=cereal, aes(x=fat, y=calories))+  
2   ggtitle("Calories vs fat in each serving")+  
3   xlab("Fat/serving (g)") + ylab("Calories/serving (g)") +  
4   geom_jitter() +  
5   geom_smooth(method="lm", se=FALSE) +  
6   facet_wrap(~shelfF, ncol=2)  
7   cerealplot
```

# Plotting with *ggplot2* - Facetting

Calories vs fat in each serving



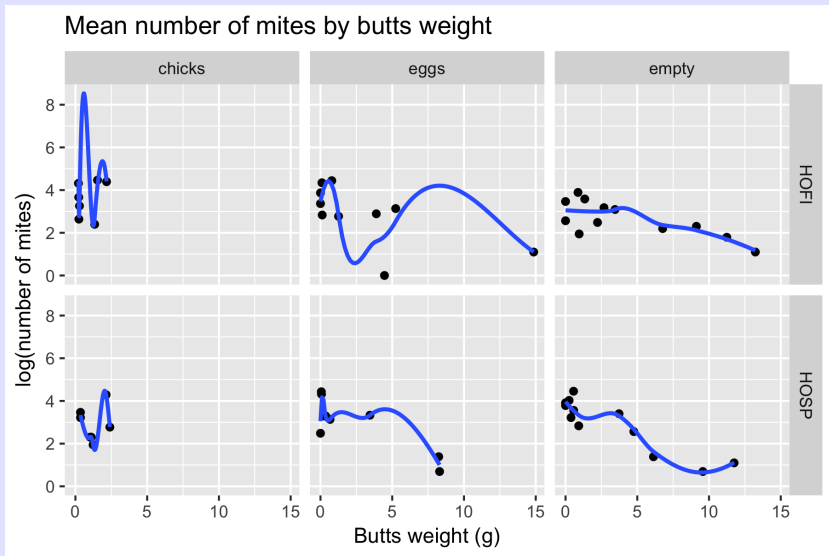
## Plotting with *ggplot2* - Facetting - Exercise

Plot the  $\log(\text{number mites})$  vs. butts weight with fitted line for each combination of species and nest content.

## Plotting with *ggplot2* - Facetting - Exercise

```
1 mitesfacet <- ggplot(data=butts, aes(x=Butts.weight, y=log.n
2   ggtitle("Mean number of mites by butts weight")+
3   xlab("Butts weight (g)")+ylab("log(number of mites)")+
4   geom_point()+
5   geom_smooth(method="loess", se=FALSE)+
6   facet_grid(Species ~ Nest.content)
7 mitesfacet
```

# Plotting with *ggplot2* - Facetting - Exercise

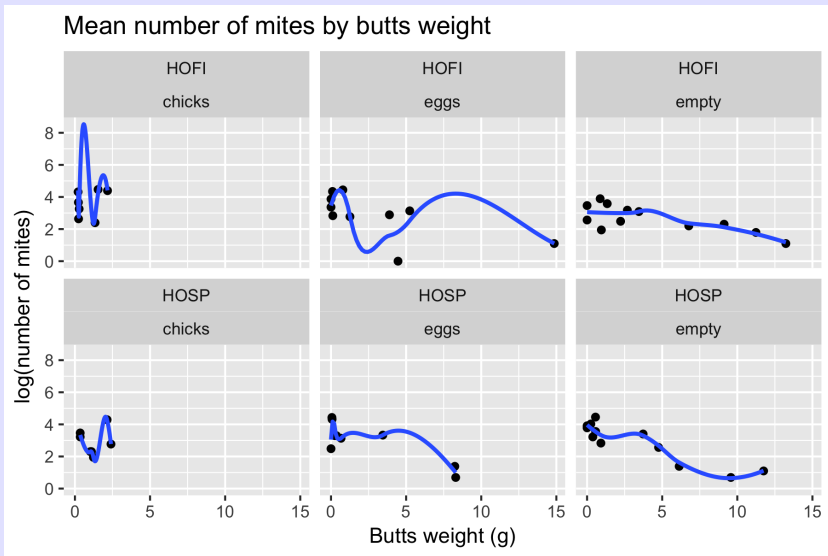


# Plotting with *ggplot2* - Facetting

More than one variable can be used in a facet dimension.

```
1 mitesfacet <- ggplot(data=butts, aes(x=Butts.weight, y=log.n
2   ggtitle("Mean number of mites by butts weight")+
3   xlab("Butts weight (g)") + ylab("log(number of mites)") +
4   geom_point() +
5   geom_smooth(method="loess", se=FALSE) +
6   facet_wrap( ~Species + Nest.content, ncol=3)
7 mitesfacet
```

# Plotting with *ggplot2* - Facetting - Exercise

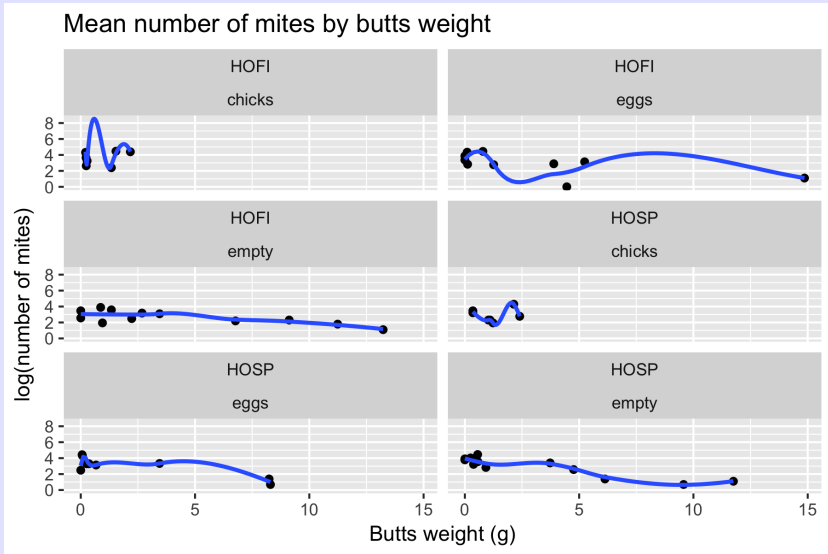


## Plotting with *ggplot2* - Facetting

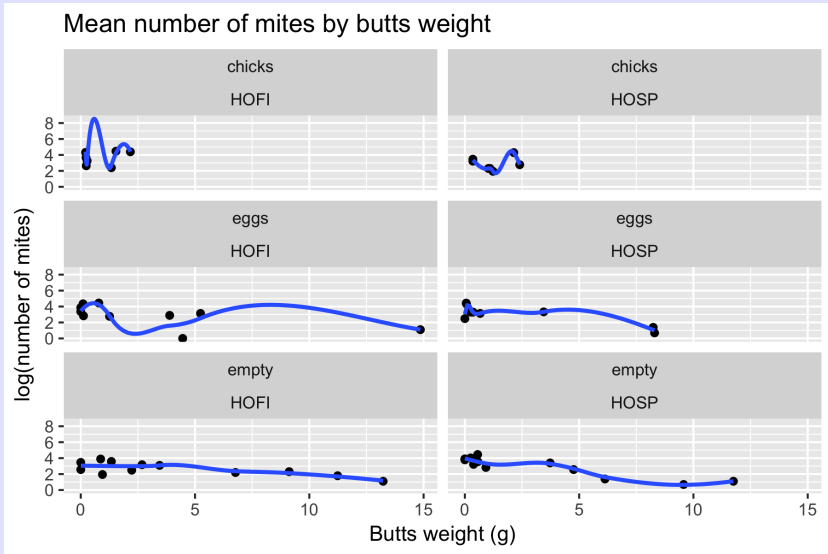
More than one variable can be used in a facet dimension.  
Repeat with different values for *ncol* and different order of facet variables.



# Plotting with *ggplot2* - Facetting - Exercise



# Plotting with *ggplot2* - Facetting - Exercise



# Plotting with *ggplot2* - Facetting - Modifying facet labels

More than one variable can be used in a facet dimension.

How is the display modified?

... `facet_wrap( v1 ~ . , nrow=2, labeller=....)`

`label_value` - display value of factor

`label_both` - display variable name and value

`label_parsed` - useful for math expressions

....

See

<http://ggplot2.tidyverse.org/reference/labellers.html>

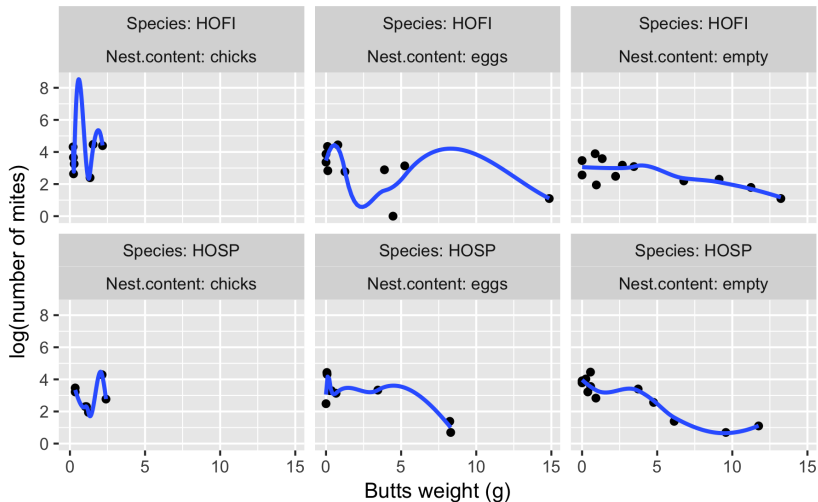
for more details

# Plotting with *ggplot2* - Facetting - Modifying facet labels

```
1 mitesfacet <- ggplot(data=butts, aes(x=Butts.weight, y=log.n
2   ggtitle("Mean number of mites by butts weight")+
3   xlab("Butts weight (g)") + ylab("log(number of mites)") +
4   geom_point() +
5   geom_smooth(method="loess", se=FALSE) +
6   facet_wrap( ~Species + Nest.content, ncol=3,
7     labeller=label_both)
8 mitesfacet
```

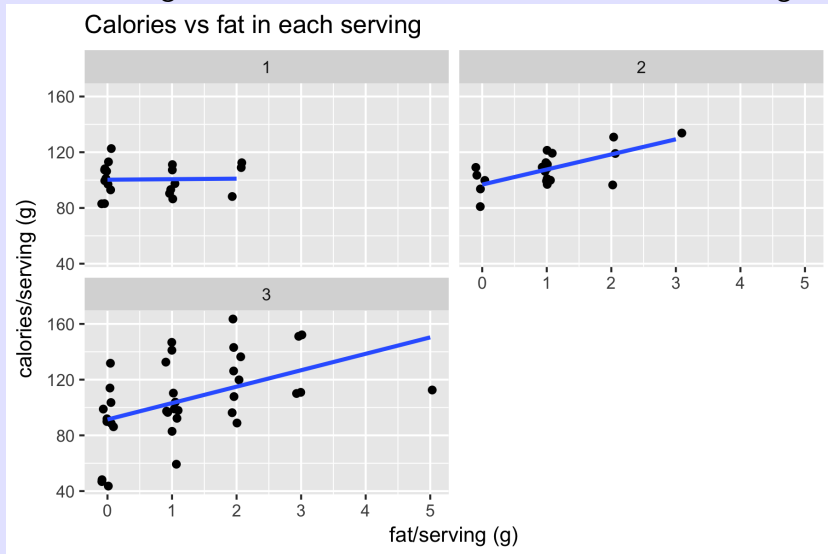
# Plotting with *ggplot2* - Facetting - Exercise

Mean number of mites by butts weight



# Plotting with *ggplot2* - Facetting - Modifying facet labels

How to change facet values, i.e. 1="low", 2="medium", 3="high"



# Plotting with *ggplot2* - Facetting - Modifying facet labels

Three methods (different complexity)

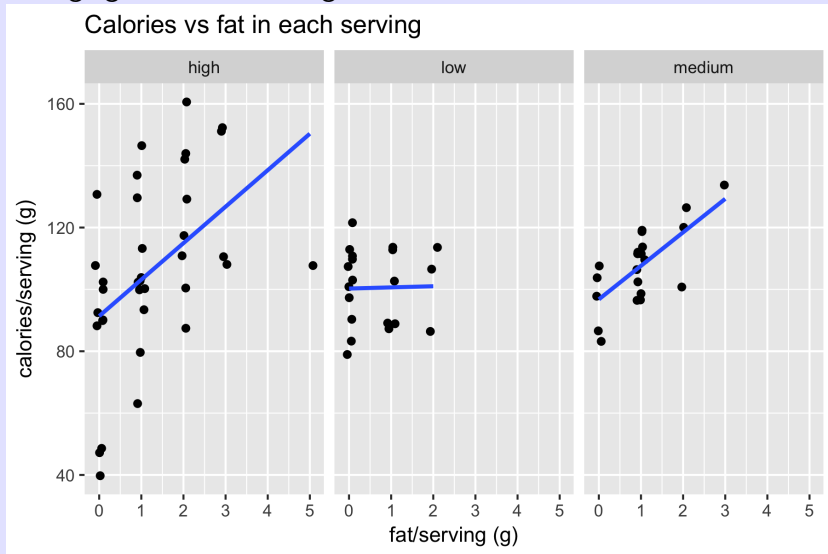
(1) Use *recode()* to make new variable and use it.

- But ... order of facets may change.

```
1 cereal$shelf2 <- car::recode(cereal$shelf,
2   "  1='low';
3     2='medium';
4     3='high'")
5 cerealplot <- ggplot(data=cereal, aes(x=fat, y=calories))+
6   ggtitle("Calories vs fat in each serving")+
7   xlab("fat/serving (g)")+ylab("calories/serving (g)")+
8   geom_point(position=position_jitter(width=0.1))+
9   geom_smooth(method="lm", se=FALSE)+
10  facet_wrap(~shelf2, ncol=3)
11 cerealplot
```

# Plotting with *ggplot2* - Facetting - Modifying facet labels

Changing facet labels using *recode*





# Plotting with *ggplot2* - Facetting - Modifying facet labels

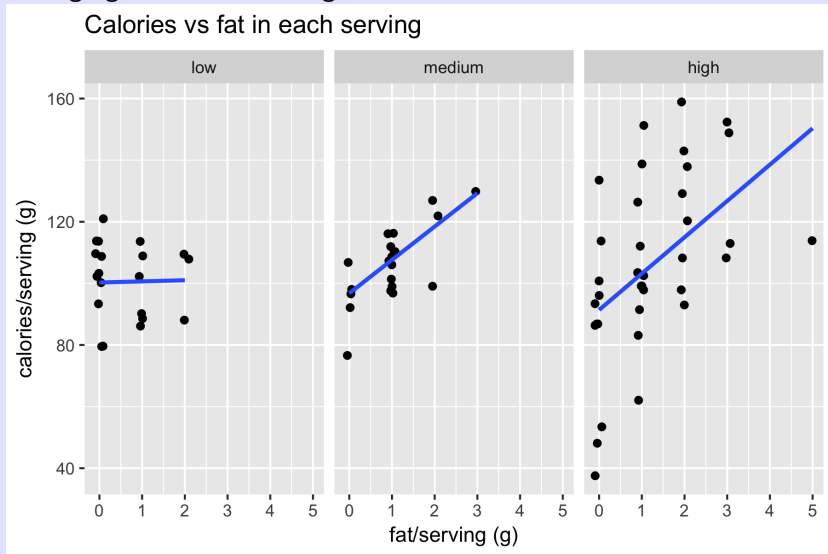
Three methods (different complexity)

(2) Create appropriate factor labels

```
1 cereal$shelfF2 <- factor(cereal$shelf, labels=c("low","medium","high"))
2
3 cerealplot <- ggplot(data=cereal, aes(x=fat, y=calories))+
4   ggtitle("Calories vs fat in each serving")+
5   xlab("fat/serving (g)") + ylab("calories/serving (g)") +
6   geom_point(position=position_jitter(width=0.1)) +
7   geom_smooth(method="lm", se=FALSE) +
8   facet_wrap(~shelfF2, ncol=3)
9 cerealplot
```

# Plotting with *ggplot2* - Facetting - Modifying facet labels

Changing facet labels using *factor*



# Plotting with *ggplot2* - Facetting - Modifying facet labels

Three methods (different complexity)

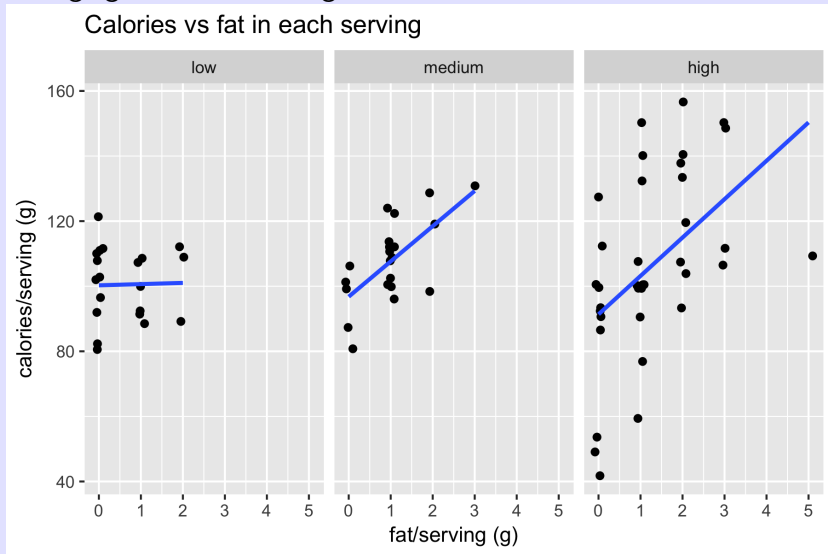
(3) Use a labeller function

```
1 cereal$shelfc <- as.character(cereal$shelf)
2 shelf_names <- c('1' = "low",
3                 '2' = "medium",
4                 '3' = "high")
5 cerealplot <- ggplot(data=cereal, aes(x=fat, y=calories))+
6   ggtitle("Calories vs fat in each serving")+
7   xlab("fat/serving (g)") + ylab("calories/serving (g)") +
8   geom_point(position=position_jitter(width=0.1)) +
9   geom_smooth(method="lm", se=FALSE) +
10  facet_wrap(~shelfc, ncol=3, labeller=labeller(shelfc=as.character(shelf_names)))
11 cerealplot
```

Faceting variable **MUST** be character.

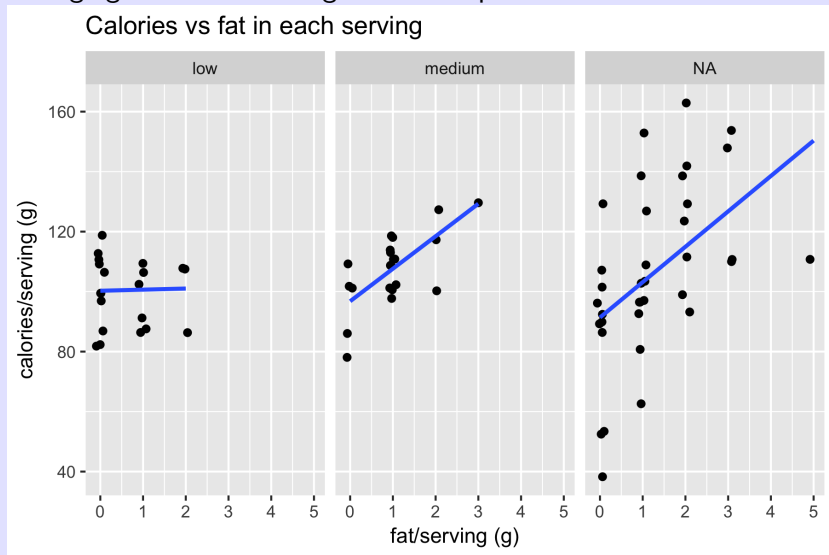
# Plotting with *ggplot2* - Facetting - Modifying facet labels

Changing facet labels using *factor*



# Plotting with *ggplot2* - Facetting - Modifying facet labels

Changing facet labels using *factor* - impact of mismatch.



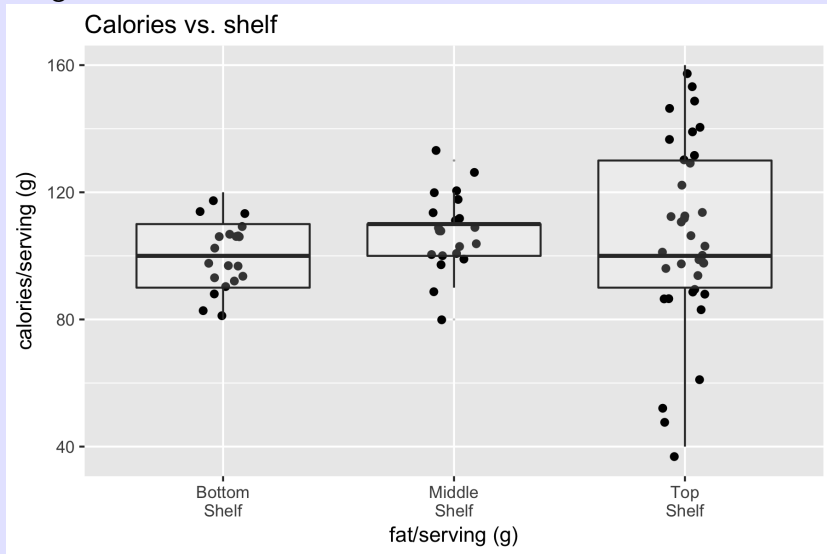
# Plotting with *ggplot2* - Tick mark labels

## Long tick mark labels

```
1 cereal$shelf.special <- car::recode(cereal$shelf,  
2                                   "1='Bottom\nShelf';  
3                                   2='Middle\nShelf';  
4                                   3='Top\nShelf'")  
5 cereal$shelf.special <- gsub(" ", "\n", cereal$shelf.special)  
6 cerealplot <- ggplot(data=cereal, aes(x=shelf.special, y=calories))  
7   ggtitle("Calories vs fat in each serving")+  
8   xlab("fat/serving (g)") + ylab("calories/serving (g)") +  
9   geom_point(position=position_jitter(width=0.1)) +  
10  geom_boxplot(alpha=0.2, outlier.size=0)  
11 cerealplot
```

# Plotting with *ggplot2* - Facetting - Tick mark labels

Long tick mark labels



## Plotting with *ggplot2* - Tick mark labels

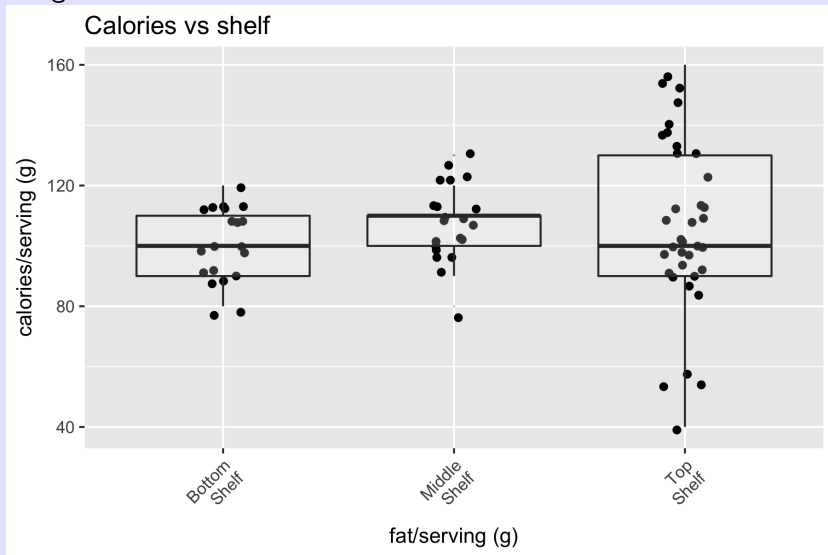
Long tick mark labels that are rotated

```
1 cereal$shelf.special <- car::recode(cereal$shelf,  
2                                   "1='Bottom\nShelf';  
3                                   2='Middle\nShelf';  
4                                   3='Top\nShelf'")  
5 cereal$shelf.special <- gsub(" ", "\n", cereal$shelf.special)  
6 cerealplot <- ggplot(data=cereal, aes(x=shelf.special, y=calories))  
7   ggtitle("Calories vs shelf")+  
8   xlab("fat/serving (g)") + ylab("calories/serving (g)") +  
9   geom_point(position=position_jitter(width=0.1)) +  
10  geom_boxplot(alpha=0.2, outlier.size=0) +  
11  theme(axis.text.x = element_text(angle = 45, hjust = 1))  
12 cerealplot
```



# Plotting with *ggplot2* - Facetting - Tick mark labels

Long tick mark labels that are rotated



## Plotting with *ggplot2* - Additional packages

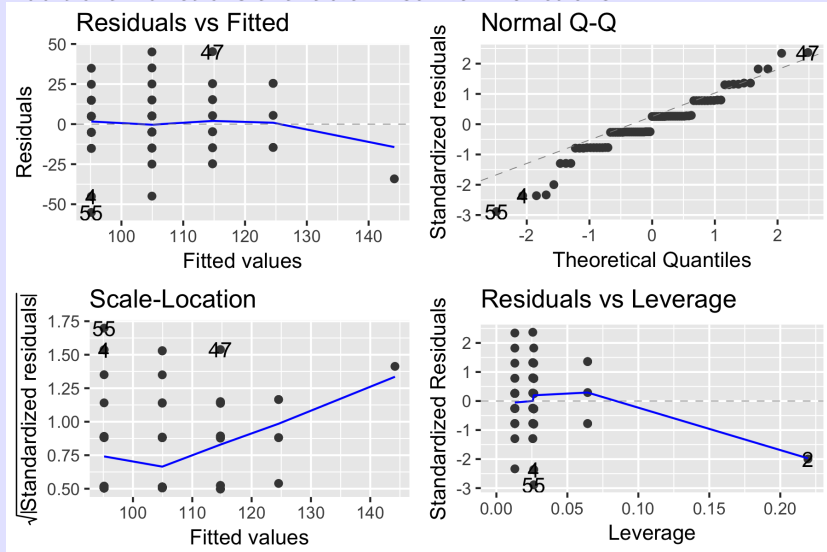
*ggfortify* - for diagnostic plots after model fits in place of *plot()*

```
1 library(ggfortify)
2 lm.fit <- lm(calories ~ fat, data=cereal)
3 diagplot <- autoplot(lm.fit)
4 diagplot
```

# Plotting with *ggplot2* - additional packages

Diagnostic plots from *lm()* and *glm*

Additional functions available in *schwarz.functions.r*



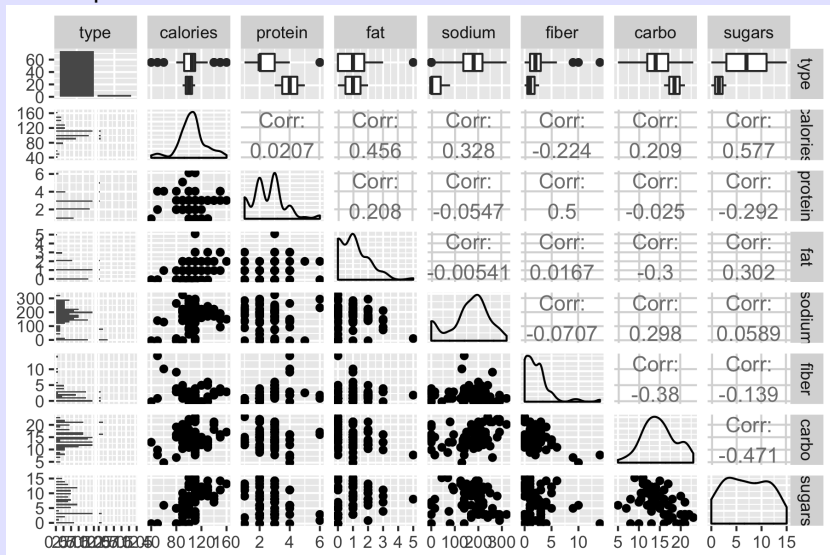
## Plotting with *ggplot2* - Additional packages

*GGally* - for scatterplot matrix and grouping multiple plots together (see later)

```
1 library(GGally)
2 spm <- ggpairs(cereal, col=c(3,4,5,6,7,8,9,10))
3 spm
```

# Plotting with *ggplot2* - additional packages

## Scatterplot matrix



## Plotting with *ggplot2* - Additional packages

*ggmap* - map data

```
1 library(ggmap)
2 sfu.coord <- c(-122.917957, 49.276765 )
3
4 my.drive.csv <- textConnection("
5 long, lat
6 -122.84378900000002, 49.290091999999999
7 -122.82799615332033, 49.28426960031931
8 -122.82696618505861, 49.27755059244836
9 -122.86679162451173, 49.27676664856581
10 -122.88790597387697, 49.26276555269492
11 -122.90833367773439, 49.26534205263451
12 -122.92532815405275, 49.273518748310764
13 -122.91434182592775, 49.27766258341439")
14 my.drive <- read.csv(my.drive.csv, header=TRUE, as.is=TRUE,
```

# Plotting with *ggplot2* - Additional packages

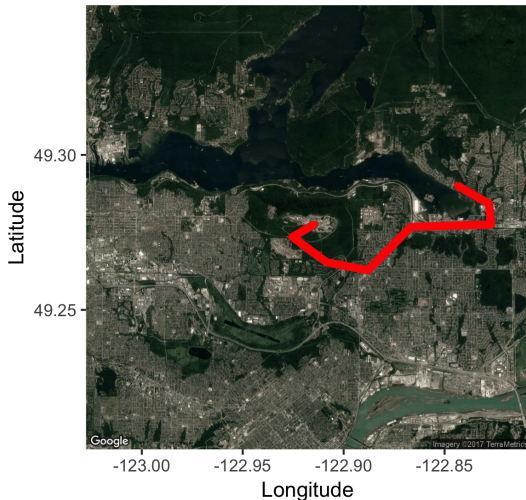
*ggmap* - map data

```
1  # get the map from google. You can fiddle with the zoom to
2  ##cc <- get_map(sfu.coord, maptype="terrain", source="goog
3
4  google.map <- get_map(sfu.coord, maptype="satellite",
5                        source="google", zoom=12)
6
7  my.map <- ggmap(google.map)
8
9  plot1 <- my.map +
10         ggtitle("My drive to work")+
11         geom_point(data=my.drive, aes(x=long, y=lat),
12                   size=1, color='red' )+
13         geom_path(data=my.drive, aes(x=long, y=lat),
14                  color="red", size=2)+
15         xlab("Latitude")+ylab("Longitude")
16  plot1
```

# Plotting with *ggplot2* - additional packages

## Plotting map data

My drive to work





## Plotting with *ggplot2* - Additional packages

Visit <http://www.ggplot2-exts.org/gallery/> for more packages.

# Plotting with *ggplot2* - Saving plots

Preferred:

```
1 ggsave(plot=cerealplot, file="blah.png",  
2       h=4, w=6, units="in", dpi=300)  
3  
4 ggsave(plot=cerealplot, file="blah.jpg",  
5       h=4, w=6, units="in", dpi=300)
```

Old fashioned methods:

```
1 png("file.png", h=4, w=6, units="in", res=300)  
2   ggplot ...  
3   def.off()
```

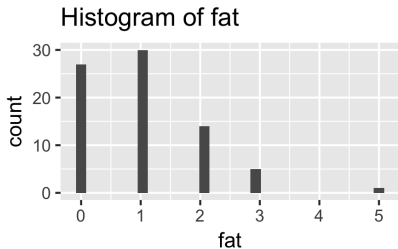
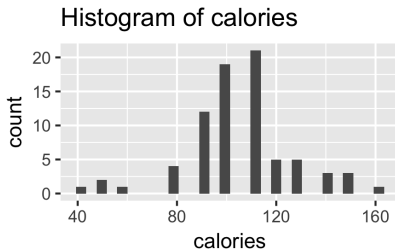
## Plotting with *ggplot2* - Multiple different plots/page

```
1 library(gridExtra)
2 p1 <- ggplot(data=cereal, aes(x=fat, y=calories))+
3       ggtitle("Calories vs fat")+ geom_point()
4
5 p2 <- ggplot(data=cereal, aes(x=Calories))+
6       ggtitle("Histogram of Calories")+geom_histogram()
7
8 p3 <- ggplot(data=cereal, aes(x=fat))+
9       ggtitle("Histogram of Fat")+ geom_histogram()
10
11 p4 <- ggplot(data=cereal, aes(x=shelfF, y=calories))+
12       ggtitle("Box plot of calories")+geom_boxplot()
13
14 allplot <- arrangeGrob(p1,p2,p3,p4, nrow=2,
15                         top="A medley of plots")
16 plot(allplot)
```

Refer to vignettes in *gridExtra* package.

# Plotting with *ggplot2* - Multiple-plots/page

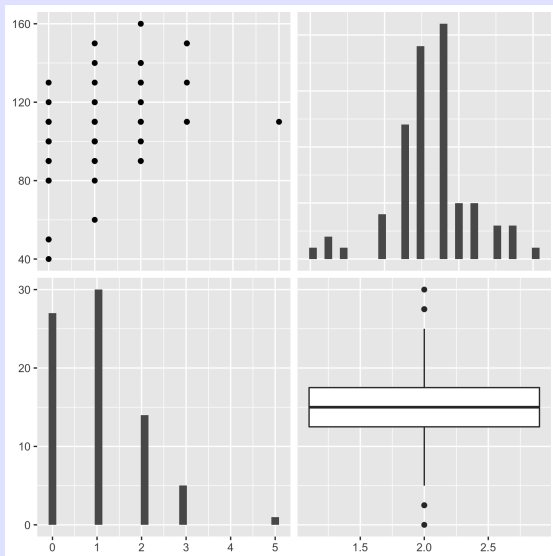
A medley of plots



## Plotting with *ggplot2* - Multiple different plots/page

```
1 library(GGally)
2 plot.list <- list(p1, p2, p3, p4)
3 allplot <- GGally::ggmatrix(plot.list,
4                             ncol=2, nrow=2)
5 allplot
```

# Plotting with *ggplot2* - Multiple-plots/page



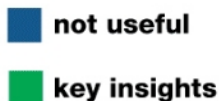
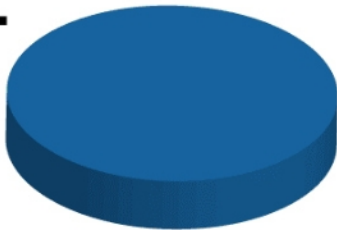
# *ggplot()* Plotting - Summary

Avoid Base *R* graphics and use *ggplot()*!

- Build up the graph using bits and pieces
  - *ggplot(data=..., aes(x=..., y=..., shape=..., color=..., group=...))* +
  - *ggtitle()* + *xlab()* + *ylab()* +
  - *geom\_point()* and *geom\_jitter()*
  - *geom\_histogram()*
  - *geom\_boxplot()*
  - *geom\_bar()*, *geom\_line()*, *geom\_errorbar()*
  - *facet\_grid()* and *facet\_wrap()*
  - *theme()*
- Once all pieces are together, then create the plot and adjust scales etc.

NEVER, NEVER, NEVER, NEVER, NEVER produce a pie chart.

# THE AMOUNT OF USEFUL INFORMATION IN A PIE CHART



<http://www.perceptualedge.com/articles/08-21-07.pdf>



Transformations, derived variables, and  
recoding data

Three standard actions depending on variable.

- Continuous variable → continuous variable

```
df$logvar <- log(df$var)
```

- Categorical variable → categorical variable

```
df$newvar <- recode(df$oldvar, " oldv1=newv1; oldv2=newv2")
```

- Continuous variable → categorical variable

```
df$newvar <- recode(df$oldvar, " l1:u1=newv1; l2:u2=newv2")
```

ALWAYS CHECK YOUR RECODES using *xtabs()* or *ggplot()*.

The *recode()* is found in the *car* library.

- ALWAYS recompute derived/recoded variables on the fly rather than doing externally to keep data and derived variables in sync.

This protects you from data being changed but transformed data not being updated in the original database.

How to add new variables to a data.frame.

```
1 cereal$ProteinFatRatio <- cereal$Protein/ cereal$Fat
2 cereal$ProteinFatRatio
3
4 cereal$TotalCarbs <- cereal$carbo + cereal$sugars
```

Avoid looping for recoding.

```
1 # bad way to check if fatality occurred
2 for(i in 1:length(accidents$Accidents_Severity)){
3     if( accidents$Accidents_Severity[i] ==1)
4         {accidents$Fatality[i] <- 'Fatal'} else
5         {accidents$Fatality[i] <- 'Not Fatal'}
6     accidents$logY[i]<- log(accidents$Y[i])
7 }
```

## Recodes - *recode()* function

Categorical to Categorical.

Watch the quotes in the recode string!

```
1 library(car)
2 accidents$Type <- recode(accidents$Accident_Severity,
3   " 1='Fatal'; 2='Serious'; 3='Minor';
4   else='Error'")
5 accidents$Type <- factor(accidents$Type,
6   levels=c("Minor","Serious","Fatal"),
7   ordered=TRUE)
8 xtabs( ~Accident_Severity+Type, data=accidents) # check rec
```

	Type		
Accident_Severity	Minor	Serious	Fatal
1	0	0	1731
2	0	20440	0
3	132243	0	0

ALWAYS CHECK YOUR RECODES!

# Recodes - *recode()* function

Categorical to Categorical.

```
1 unique(accidents$Number_of_Vehicles)
2 accidents$Nveh <- recode(accidents$Number_of_Vehicles,
3   "1='Single'; 2='Two'; 3:6='Multi'; 7:hi='Mess' ")
4 accidents$Nveh <- factor(accidents$Nveh,
5   levels=c("Single","Two","Multi","Mess"),
6   ordered=TRUE)
7 xtabs( ~Nveh+Number_of_Vehicles, data=accidents)
```

	Number_of_Vehicles							
Nveh	1	2	3	4	5	6	7	8
Single	47319	0	0	0	0	0	0	0
Two	0	91870	0	0	0	0	0	0
Multi	0	0	11908	2500	541	155	0	0
Mess	0	0	0	0	0	0	60	34

ALWAYS CHECK YOUR RECODES!

## Recodes - *recode()* function

Continuous to Categorical. Change the time-of-day to hours since midnight

```
1 accidents[1:5, c("Date","Time")]
2 accidents$mydt <- as.POSIXct(paste(accidents$Date," ",accidents$Time,
3                                 format="%d/%m/%Y %H:%M", tz="UCT"))
4 accidents$my.time <- as.numeric(format(accidents$mydt,"%H"))
5                     as.numeric(format(accidents$mydt, "%M"))/60
6 accidents$TimePer <- recode(accidents$my.time,
7                             "lo:6='Night'; 6:12='Morning';
8                             12:16='Aft'; 16:20='Evening';
9                             20:hi='Night'" )
10 accidents$TimePer <- factor(accidents$TimePer,
11                             levels=c("Night","Morning","Aft","Evening"),
12                             ordered=TRUE)
```

ALWAYS CHECK YOUR RECODES!



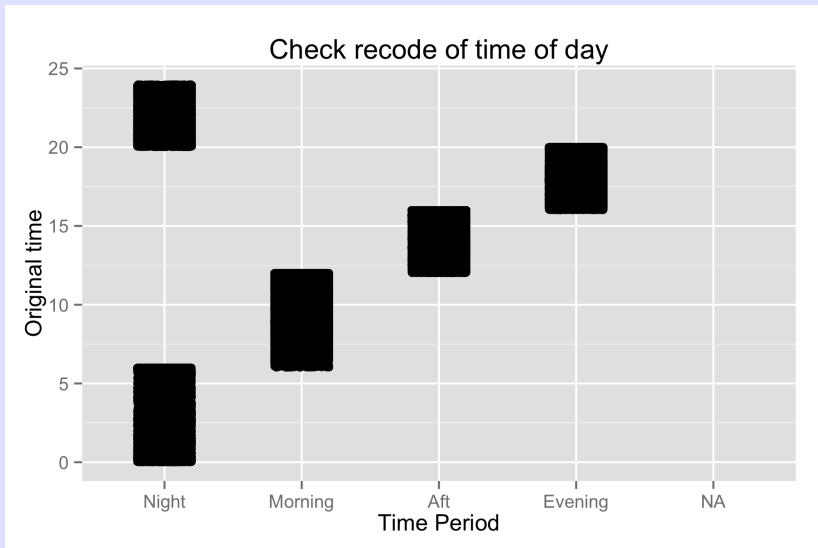
## Recodes - *recode()* function

Continuous to Categorical. Change the time-of-day to hours since midnight

```
1 recodetod <- ggplot(data=accidents, aes(x=TimePer, y=my.time
2   ggtitle("Check recode of time of day")+
3   xlab("Time Period")+ylab("Original time")+
4   geom_point(position=position_jitter(w=0.2))
5 recodetod
```

ALWAYS CHECK YOUR RECODES!

# Vectorizing Recodes



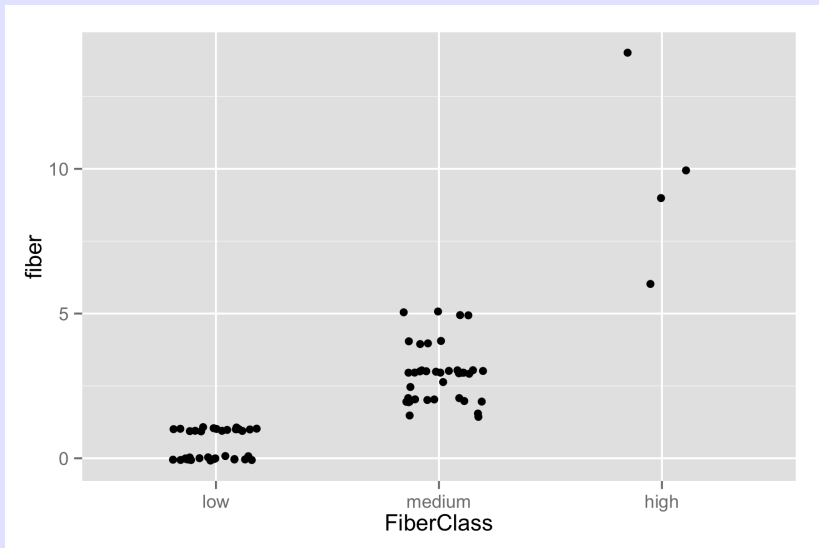
ALWAYS CHECK YOUR RECODES!

## Recodes - *recode()* function

Continuous to Categorical. Change the grams of fiber to a fiber class.

```
1 library(car) # load the recode function
2 # Watch the quotes!
3 cereal$FiberClass <- recode(cereal$fiber,
4                             ' 10:1="low";
5                             1:5="medium";
6                             5:hi="high"  ')
7 cereal$FiberClass <- factor(cereal$FiberClass,
8                             levels=c("low","medium","high"),
9                             ordered=TRUE)
10
11 # Always check your work
12 xtabs(~FiberClass+fiber, data=cereal)
13 ggplot(data=cereal, aes(x=FiberClass, y=fiber))+
14   geom_point()
```

## Recodes - *recode()* function



ALWAYS CHECK YOUR RECODES!

## Recodes - `recode()` function - Exercise I

Refer to the cereal dataset.

- Recode type of cereal (*C* or *H*) to *Cold* and *Hot*.
- Recode grams of protein to *Above Median* and *Below Median*
  - HINT: The values in a recode must be static. Hence you must construct the recode string using the `paste()` before calling `recode()`

CAUTION: Always check your recodes using `xtabs()` or `ggplot()`

## Recodes - *recode()* function - Exercise I

Recode cereal type.

```
1 cereal$type.long <- car::recode(cereal$type,  
2                               " 'C'='Cold'; 'H'='Hot'  ")  
3 xtabs(~type+type.long, data=cereal,  
4       exclude=NULL, na.action=na.pass)
```

```
> xtabs(~type+type.long, data=cereal, exclude=NULL, na.action=na.pass)  
      type.long  
type Cold Hot  
  C     74   0  
  H      0   3
```

## Recodes - *recode()* function - Exercise I

Recode protein level.

```
1 median(cereal$protein)
2 rec.string <- paste("lo:", median(cereal$protein),
3                     "='At or below median'; ",
4                     median(cereal$protein),
5                     ":hi='Above median'", sep="")
6 rec.string
7 cereal$protein.class <- car::recode(cereal$protein,
8                                     rec.string)
9 plot <- ggplot(data=cereal, aes(x=protein.class, y=protein))
10   ggtitle("Checking protein recode")+
11   geom_point(position=position_jitter(w=0.3, h=0))+
12   geom_hline(yintercept=median(cereal$protein), color="red")
13 plot
```

Note use of computed value in recode string and ggplot.

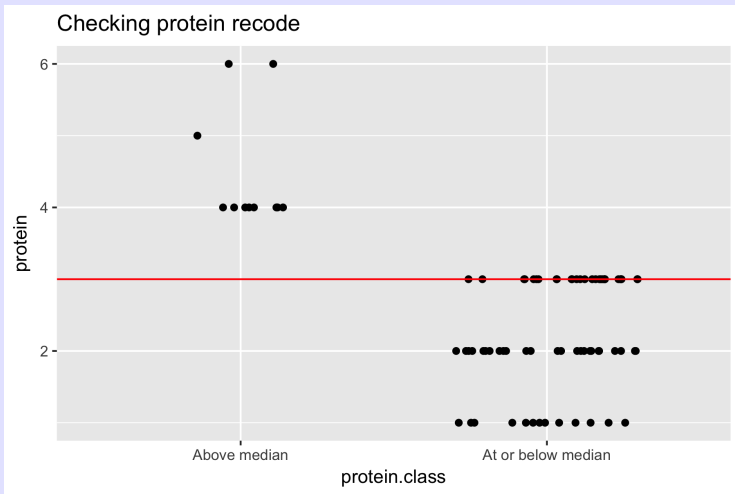
## Recodes - *recode()* function - Exercise I

Recode protein level.

```
> median(cereal$protein)
[1] 3
> rec.string <- paste("lo:", median(cereal$protein), "'At or
> rec.string
[1] "lo:3='At or below median'; 3:hi='Above median'"
```



## Recodes - *recode()* function - Exercise I



What is the impact of high winds on the fatality rate in the Accident database.

- Examine *Weather* variable.
  - Split into 2 variables (fine/rain/snow + not.high.wind/high.wind)
  - Discard accidents not in codes 1:6
- Recode *Accident\_Severity* variable as Fatal/Not Fatal (1/0)
- Compute the proportion of fatality for the 3 x 2 combinations using *ddply()* and *summarize*.
- Use *group=*, *color=* options in *aes()* in *ggplot()*. What do you conclude?

What is the impact of high winds on fatality rate?

```
1 unique(accidents$Accident_Severity)
2 accidents$Fatality <- accidents$Accident_Severity==1
3 xtabs(~Fatality + Accident_Severity, data=accidents)
```

## Recodes - *recode()* function - Exercise II

What is the impact of high winds on fatality rate?

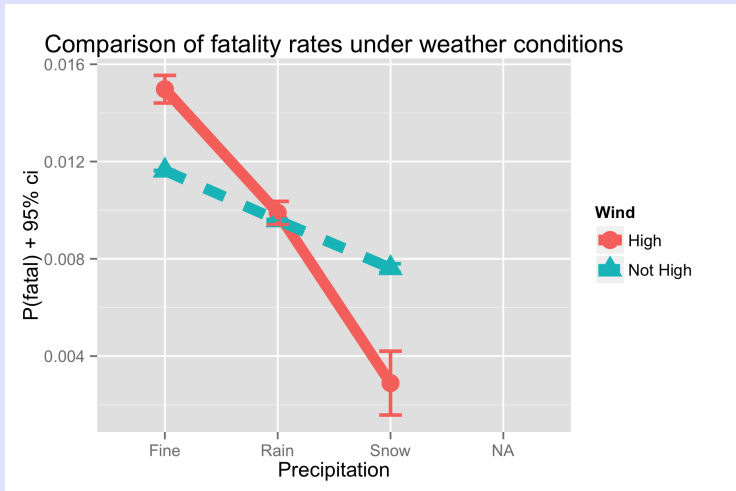
```
1 accidents$Conditions <-  
2     recode(accidents$Weather_Conditions,  
3     "c(1,4)='Fine'; c(2,5)='Rain'; c(3,6)='Snow';  
4     else=NA")  
5 xtabs(~Conditions + Weather_Conditions,  
6     data=accidents)  
7  
8 accidents$Wind <- recode(accidents$Weather_Conditions,  
9     "c(1,2,3)='Not High'; 4:6='High'; else=NA")  
10 xtabs(~Wind + Weather_Conditions, data=accidents)
```

## Recodes - *recode()* function - Exercise II

What is the impact of high winds on fatality rate?

```
1 library(plyr)
2 pfatal.df <- ddply(accidents, c("Wind","Conditions"),
3                     summarize,
4                     pfatal=mean(Fatality))
5 pfatal.df
6
7 plotCondWind <- ggplot(data=pfatal.df,
8                         aes(x=Conditions, y=Fatality.mean,
9                             shape=Wind, color=Wind,
10                             linetype=Wind, group=Wind))+
11   geom_point(size=5)+
12   geom_line(size=3)
13 plotCondWind
```

# Recodes - `recode()` function - Exercise II



Extra code on website to compute and add confidence limits.

- Seldom necessary to use *for()* loops and *if()* statements to do recoding.
- Use *recode()* from the *car* package.
- Check your results
  - Use *xtabs()* for discrete  $\rightarrow$  discrete recoding.
  - Use *ggplot()* with *geom\_point()* for continuous  $\rightarrow$  discrete recoding.
  - Plot  $f(x)$  vs.  $x$  for continuous  $\rightarrow$  continuous recoding.

## Reshaping Data

Wide  $\leftrightarrow$  Long formats



# Reshaping Data

Wide data format commonly found with many variables or longitudinal data

```
1 > chick.wide <- read.csv("../sampledata/chickweight.csv",  
2 +                       header=TRUE, as.is=TRUE,  
3 +                       strip.white=TRUE)  
4 > head(chick.wide)
```

```
> head(chick.wide)
```

	Chick	Diet	Day01	Day02	Day04	Day06	Day08	Day10	Day12	Day14
1	1	Diet1	42	51	59	64	76	93	106	125
2	2	Diet1	40	49	58	72	84	103	122	135
3	3	Diet1	43	39	55	67	84	99	115	132

We would like a plot of the mean weight over time for each diet.

# Reshaping Data

Long data format transposes each row of data into a long format

```
> head(chick.long)
```

	Chick	Diet	Time	Weight
1	1	1	Day01	42
2	1	1	Day02	51
3	1	1	Day04	59
4	1	1	Day06	64
5	1	1	Day08	76
6	1	1	Day10	93

## DO NOT USE *reshape* in Base R

Do use *reshape2* package and *melt()* and *dcast* functions.

Why reshape?

- Many statistical models require repeated measure data to be in long format.
- *ggplot()* expects most data to be in long format.
- Quick and dirty way to get plots of multiple variables for screening etc. using faceting in *ggplot()*

## Reshaping Data - melting - wide → long

```
reshape2::melt(df,  
  id.vars=c(...),  
  measure.vars=c( ),  
  variable.name="xxxx",  
  value.name="yyyyy")
```

Separate variables into:

- *id.vars* that will remain fixed for all values in long format to group the values.
- *measure.vars* - variables to be transposed into long format.

Need to define:

- *variable.name* - variable to hold the names of the transposed variables
- *value.name* - variable to hold the value of the transposed variables.

## Reshaping Data - melting - wide → long

Example:

```
1 chick.long <- reshape2::melt(chick.wide,  
2                               id.vars=c("Chick","Diet"),  
3                               measure.vars=c("Day01","Day02","Day04",  
4                                                "Day06","Day08","Day10",  
5                                                "Day12","Day14","Day16",  
6                                                "Day18","Day20","Day21"),  
7                               variable.name="Time",  
8                               value.name="Weight")
```

If you leave out *measure.var*, then all other variables not in *id.vars* will be transposed.

## Reshaping Data - melting - wide $\rightarrow$ long

Example:

```
> head(chick.wide)
```

	Chick	Diet	Day01	Day02	Day04	Day06	Day08	Day10	Day12	Day14
1	1	Diet1	42	51	59	64	76	93	106	120

```
> head(chick.long)
```

	Chick	Diet	Time	Weight
1	1	1	Day01	42
2	1	1	Day02	51
3	1	1	Day04	59
4	1	1	Day06	64
5	1	1	Day08	76
6	1	1	Day10	93

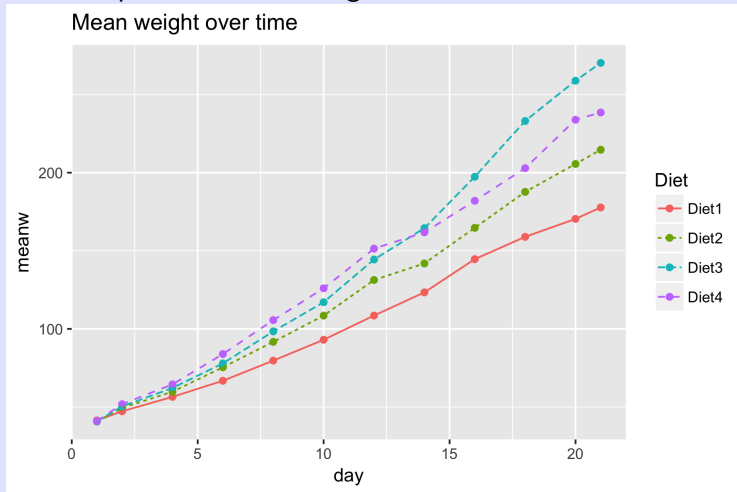
## Reshaping Data - melting - wide → long

Now we can compute means for each diet x day combination

```
1 meanw <- plyr::ddply(chick.long, c("Time","Diet"),
2                       summarize,
3                       day = as.numeric(substring(Time[1],4)),
4                       meanw=mean(Weight, na.rm=TRUE))
5 meanw
6
7 plot.meanw <- ggplot2::ggplot(data=meanw,
8                               aes(x=day, y=meanw,
9                                   color=Diet, linetype=Diet))+
10   geom_point()+
11   geom_line()+
12   ggtitle("Mean weight over time")
13 plot.meanw
```

# Plotting with *ggplot2* - Scatterplot

Create a plot of calories vs. grams of fat





## Reshaping Data - melting - wide → long

Example: melting different variables for plotting purposes:  
Calories from fat, protein, carbohydrates across shelves.

```
1 head(cereal[,c("name","fat","protein","carbo")])
2
3 cereal$calories.fat      <- cereal$fat * 9
4 cereal$calories.protein <- cereal$protein *4
5 cereal$calories.carbo   <- cereal$carbo * 4
6
7 cereal.long <- reshape2::melt(cereal,
8                               id.var=c("name","shelfF"),
9                               measure.var=c("calories.fat",
10                                              "calories.protein",
11                                              "calories.carbo"),
12                               variable.name="Source",
13                               value.name="Calories")
14 head(cereal.long)
```

# Reshaping Data - melting - wide → long

Example: comparing calories from different sources.

```
> head(cereal[,c("name","fat","protein","carbo")])
```

	name	fat	protein	carbo
1	100%_Bran	1	4	5.0
2	100%_Natural_Bran	5	3	8.0
3				

```
> head(cereal.long)
```

	name	shelfF	Source	Calories
1	100%_Bran	3	calories.fat	9
2	100%_Bran	3	calories.protein	16
3	100%_Bran	3	calories.carbo	20
4	100%_Natural_Bran	3	calories.fat	45
5	100%_Natural_Bran	3	calories.protein	12
6	100%_Natural_Bran	3	calories.carbo	32

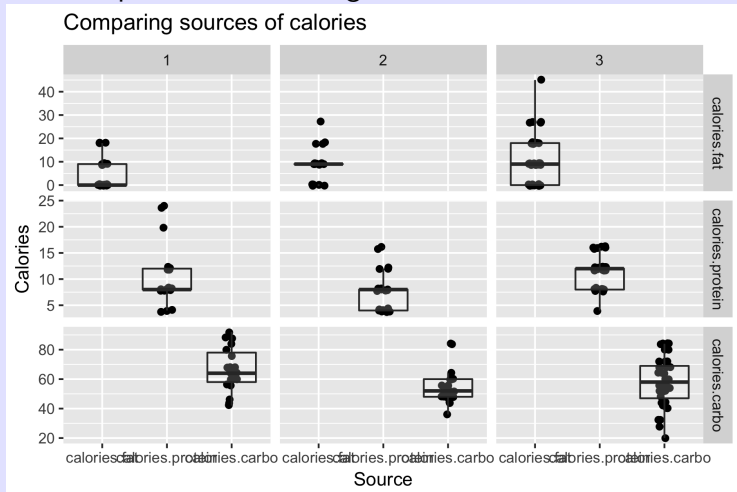
## Reshaping Data - melting - wide $\rightarrow$ long

Example: comparing calories from different sources.

```
1 plot1 <- ggplot(data=cereal.long, aes(x=Source, y=Calories))
2   ggtitle("Comparing sources of calories")+
3   geom_point(position=position_jitter(w=0.1))+
4   geom_boxplot(alpha=0.2, outlier.size=0)+
5   facet_grid(Source ~ shelfF, scales="free")
6 plot1
```

# Plotting with *ggplot2* - Scatterplot

Create a plot of calories vs. grams of fat



## Reshaping Data - melting - wide $\rightarrow$ long

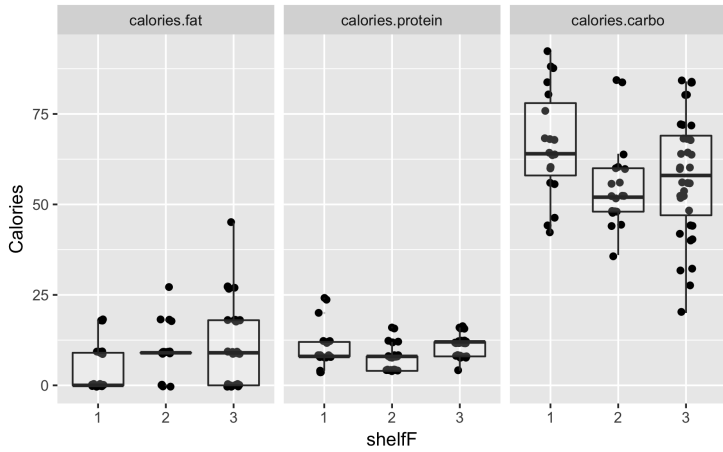
Example: comparing calories from different sources.

```
1 plot2 <- ggplot(data=cereal.long, aes(x=shelfF, y=Calories))
2   ggtitle("Comparing sources of calories")+
3   geom_point(position=position_jitter(w=0.1))+
4   geom_boxplot(alpha=0.2, outlier.size=0)+
5   facet_wrap(~Source )
6 plot2
```

# Plotting with *ggplot2* - Scatterplot

Create a plot of calories vs. grams of fat

Comparing sources of calories



## Reshaping Data - casting - long → wide

Less common:

```
reshape2::dcast(df,  
  id.vars ~ measure.vars,  
  value.var="yyyyy")
```

Example:

```
1 chick.wide2 <- reshape2::dcast(chick.long,  
2                               Chick+Diet ~ Time,  
3                               value.var="Weight")  
4 head(chick.wide2)  
5
```

# Reshaping Data - melting - wide $\rightarrow$ long

Example:

```
> head(chick.long)
```

	Chick	Diet	Time	Weight
1	1	1	Day01	42
2	1	1	Day02	51
3	1	1	Day04	59
4	1	1	Day06	64
5	1	1	Day08	76
6	1	1	Day10	93

```
> head(chick.wide2)
```

	Chick	Diet	Day01	Day02	Day04	Day06	Day08	Day10	Day12	Day14
1	1	Diet1	42	51	59	64	76	93	106	120
2	2	Diet1	40	49	58	72	84	103	122	135



Teeth dataset - number of types of teeth for mammals

- Read the data
- Sum the top and bottom teeth classification.
- Melt the 4 types of teeth
- Make a nice plot comparing the distribution of teeth by mammal classification (H or C)

## Reshaping data - Exercise

```
1 teeth.wide <- read.csv("../sampledata/Teeth.csv",
2                           header=TRUE, strip.white=TRUE,
3                           as.is=TRUE)
4 teeth.wide
5
6 teeth.wide$Incisors <- teeth.wide$Top.incisors + teeth.wide$
7 teeth.wide$Canines <- teeth.wide$Top.canines + teeth.wide$
8 teeth.wide$Premolars<- teeth.wide$Top.premolars+ teeth.wide$
9 teeth.wide$Molars <- teeth.wide$Top.molars + teeth.wide$
10
11 head(teeth.wide[,c("Mammal", "Class", "Incisors", "Canines", "Pr

> head(exp.long)
> head(teeth.wide[,c("Mammal", "Class", "Incisors", "Canines", "
      Mammal Class Incisors Canines Premolars Molars
1      BADGER      c         6         2         6         3
2      COUGAR      c         6         2         5         2
3 ELEPHANT SEAL      c         3         2         8         2
```

## Reshaping data - Exercise

```
1 teeth.long <- reshape2::melt(teeth.wide,  
2                             id.vars=c("Mammal","Class"),  
3                             measure.vars=c("Incisors","Canines","Premolars"),  
4                             value.name="Teeth",  
5                             variable.name="Tooth.Type")  
6 head(teeth.long)  
7
```

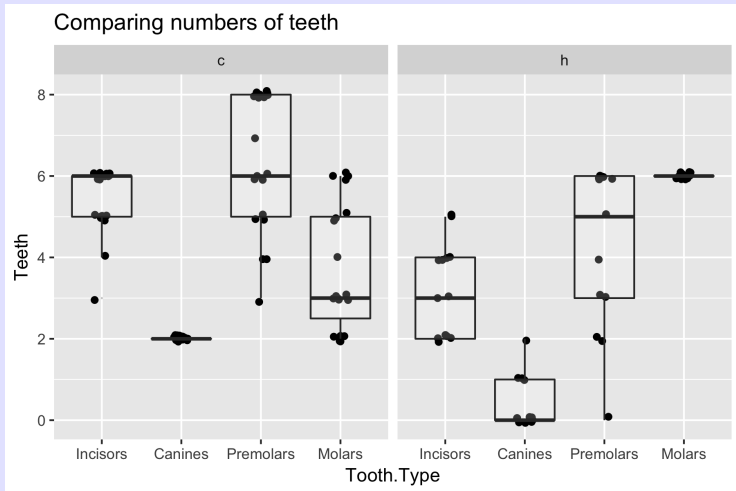
```
> head(teeth.long)
```

	Mammal	Class	Tooth.Type	Teeth
1	BADGER	c	Incisors	6
2	COUGAR	c	Incisors	6
3	ELEPHANT SEAL	c	Incisors	3

## Reshaping data - Exercise

```
1 plot1 <-ggplot(data=teeth.long, aes(x=Tooth.Type, y=Teeth))-  
2   ggtitle("Comparing numbers of teeth")+  
3   geom_point(position=position_jitter(h=.1, w=.1))+  
4   geom_boxplot(alpha=0.2, outlier.size=0)+  
5   facet_wrap(~Class)  
6 plot1  
7
```

# Reshaping data - Exercise

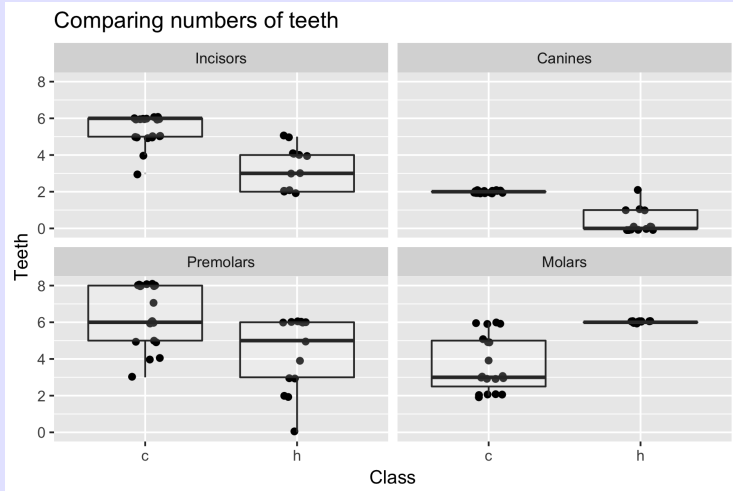


What do you conclude?

## Reshaping data - Exercise

```
1 plot2 <- ggplot(data=teeth.long, aes(x=Class, y=Teeth))+  
2   ggtitle("Comparing numbers of teeth")+  
3   geom_point(position=position_jitter(h=.1, w=.1))+  
4   geom_boxplot(alpha=0.2, outlier.size=0)+  
5   facet_wrap(~Tooth.Type, ncol=2)  
6 plot2  
7
```

# Reshaping data - Exercise



What do you conclude?

Return to the Birds 'n Butts dataset.

Look at the Experimental tab in the Excel file.

Paired design with two halves of 1 nest receiving treatments.

- Read directly in from Excel spreadsheet.
  - How do you skip the first row?
  - How do you fix the variable names?
- Cast to put both values of number of mites on same record
- Compute the difference in the number of mites
- Make a plot to decide if there is an effect?



Return to the Birds 'n Butts dataset.

Look at the Experimental tab in the Excel file.

Paired design with two halves of 1 nest receiving treatments.

- Read directly in from Excel spreadsheet.
  - How do you skip the first row?
  - How do you fix the variable names?
- Cast to put both values of number of mites on same record
- Compute the difference in the number of mites
- Make a plot to decide if there is an effect?

## Reshaping data - Exercise

```
1 exp.long <- read_excel("../sampledata/bird-butts-data.xlsx",
2                           sheet="Experimental", skip=1)
3 names(exp.long) <- make.names(names(exp.long))
4 head(exp.long)
5
```

```
> head(exp.long)
```

```
# A tibble: 6 x 5
```

	Nest	Species	Nest.content	Number.of.mites	Treatment
	<dbl>	<chr>	<chr>	<dbl>	<chr>
1	1	HOSP	empty	0	control
2	1	HOSP	empty	0	experimental
3	2	HOSP	empty	1	control
4	2	HOSP	empty	0	experimental

## Reshaping data - Exercise

```
1 exp.wide <- reshape2::dcast(exp.long,  
2                             Nest+Species+Nest.content ~ Treatment  
3                             value.var="Number.of.mites")  
4 exp.wide$c.minus.e <- exp.wide$control - exp.wide$experimental  
5 head(exp.wide)  
6
```

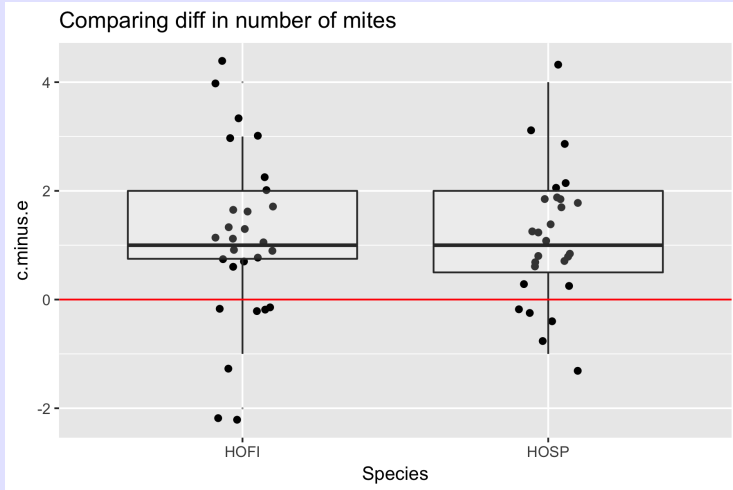
```
> head(exp.wide)
```

	Nest	Species	Nest.content	control	experimental	c.minus.e
1	1	HOSP	empty	0	0	0
2	2	HOSP	empty	1	0	1
3	3	HOSP	eggs	3	1	2

## Reshaping data - Exercise

```
1 plot1 <- ggplot(data=exp.wide, aes(x=Species, y=c.minus.e))-  
2   ggtitle("Comparing diff in number of mites")+  
3   geom_point(position=position_jitter(w=0.1))+  
4   geom_boxplot(alpha=0.2, outlier.size=0)+  
5   geom_hline(yintercept=0,color="red")  
6 plot1  
7
```

# Reshaping data - Exercise



What do you conclude?

Sometimes necessary to convert from wide  $\leftrightarrow$  long formats.  
Use *reshape2* package.

- Careful of spelling, e.g. *value.var* vs. *value.name*
- Wide to long is often used prior to making a plot using *ggplot()*.
- Long to wide is often used to bring elements of paired data together.

Factor data type

## Two broad classification of data

- **categorical** - values that do not have (theoretical) intermediate values. E.g. sex, size (large vs. small), shelf. Can be coded using character (e.g. 'small' vs. 'large') (PREFERRED); but numerical codes can also be used (e.g. 1=fatality, 0=not fatal).
- **continuous** - values can have (theoretical) intermediate values. E.g. height, weight, length, temperature. Of course, these must always be measured to some interval (e.g. height to the nearest cm). Invariably coded using a numerical value.



Two broad classification of data.

In some cases, variables could fall in to either category.

You must think about the data and how it will be used.

- `lm( $y \sim var$ )` has different meaning if *var* is continuous or *var* is categorical.
- `ggplot(data=xxx, aes(x=var, y=y, ))` .. affects behavior of subsequent layers depending if *var* is continuous or *var* is categorical.

*R* made some design decisions when created that have unexpected consequence today.

- Because of lack of storage space, all character strings were stored as factors. Reduced space needed to store duplicate values.
- Because Base *R* plotting functions only used numerical values, needed a numerical code for character data.
- Because Base *R* model functions could not access variables inside a formula easily, needed an external way to define a variable.
- Because factors are defined on a data structure, it is difficult to add new levels to the variable.

*R* made some design decisions when created that have unexpected consequence today.

This leads to the default (POOR) behavior of *R*

- *read.csv()* and the like automatically converted strings to factors.

Specify *as.is=TRUE* or *stringsAsFactors=FALSE* to prevent this.

- *data.frame()* and the like automatically converted strings to factors.

Specify *stringsAsFactors=FALSE* to prevent this.

- Need to specify that categorical variables (esp. numeric variables) are factors explicitly.

Some functions can automatically classify character data as factors, but not consistent in *R*.

# Factors - Part of a Factor

A *factor* data type consists to two parts:

- a (hidden) vector of **levels** in character format (numbers are converted to characters)
- an index into the vector of levels

```
1 x <- rep(c("low","medium","high"), times=1:3)
2 x
3 str(x)
4
5 xF <- factor(x)
6 xF
7 str(xF)
8 as.numeric(xF)  # likely not what you want
```

## Factors - Part of a Factor

```
> x
[1] "low"      "medium" "medium" "high"    "high"    "high"
> str(x)
chr [1:6] "low" "medium" "medium" "high" "high" "high"

> xF
[1] low      medium medium high    high    high
Levels: high low medium
> str(xF)
Factor w/ 3 levels "high","low","medium": 2 3 3 1 1 1
> as.numeric(xF)
[1] 2 3 3 1 1 1
```

# Factors - Ordering levels

Levels are sorted alphabetically, by default, but this can be changed.

```
1 xF0 <- factor(x, levels=c("low","medium","high"),
2               ordered=TRUE)
3 xF0
4 str(xF0)
5 as.numeric(xF0)  # likely not what you want
```

```
> xF0
[1] low    medium medium high    high    high
Levels: low < medium < high
> str(xF0)
Ord.factor w/ 3 levels "low"<"medium"<...: 1 2 2 3 3 3
> as.numeric(xF0)  # likely not what you want
[1] 1 2 2 3 3 3
```

This is useful in plotting when you want a certain order.

## Factors - Retrieving original values

No easy way to retrieve the original values except using code fragments such as

```
1 # get back the levels from before converting to a factor
2 levels(xF)[xF]
3 levels(xF0)[xF0]
4 as.character(xF)
```

```
> levels(xF)[xF]
[1] "low"      "medium" "medium" "high"    "high"    "high"
> levels(xF0)[xF0]
[1] "low"      "medium" "medium" "high"    "high"    "high"
> as.character(xF)
[1] "low"      "medium" "medium" "high"    "high"    "high"
```

## Factors - Retrieving original values - numerical codes

Numerical values will still be returned as character, so you may need another conversion:

```
1 xx <- c(3,1,3,3,3,1,2,2,3,2,1)
2 xxF <- factor(xx)
3 xxF
4 levels(xxF)[xxF]
5 as.numeric(levels(xxF)[xxF])
6 as.numeric(as.character(xxF))
```

```
> xxF <- factor(xx)
> xxF
[1] 3 1 3 3 3 1 2 2 3 2 1
Levels: 1 2 3
> levels(xxF)[xxF]
[1] "3" "1" "3" "3" "3" "1" "2" "2" "3" "2" "1"
> as.numeric(levels(xxF)[xxF])
[1] 3 1 3 3 3 1 2 2 3 2 1
> as.numeric(as.character(xxF))
[1] 3 1 3 3 3 1 2 2 3 2 1
```



CAUTION: reading character data using *read.csv()*

```
1 cereal <- read.csv('.././../sampledata/cereal.csv',
2                     header=TRUE,
3                     strip.white=TRUE)
4 cereal[1:5,]
5 str(cereal)
6
7 cereal <- read.csv('.././../sampledata/cereal.csv',
8                     header=TRUE, as.is=TRUE,
9                     strip.white=TRUE)
10 cereal[1:5,]
11 str(cereal)
```

## Factors - Implications - Reading data

CAUTION: reading character data using *read.csv()*

```
> cereal <- read.csv('../..sampledata/cereal.csv',  
+                   header=TRUE)
```

```
> cereal[1:2,]
```

	name	mfr	type	calories	protein	fat	so
1	100%_Bran	N	C	60	4	1	
2	100%_Natural_Bran	Q	C	110	3	5	

```
> str(cereal)
```

```
'data.frame': 77 obs. of 15 variables:
```

```
$ name      : Factor w/ 77 levels "100%_Bran","100%_Natural_Bran",...  
$ mfr       : Factor w/ 7 levels "A","G","K","N",...: 4 6 3 3 ...  
$ type      : Factor w/ 2 levels "C","H": 1 1 1 1 1 1 1 1 1 1 ...  
$ calories: int  60 110 80 50 110 110 110 140 90 90 ...
```

# Factors - Implications - Reading data

CAUTION: reading character data using *read.csv()*

```
> cereal <- read.csv('.././sampledata/cereal.csv',  
+                   header=TRUE, as.is=TRUE,  
+                   strip.white=TRUE)  
> cereal[1:2,]
```

	name	mfr	type	calories	protein	fat	sc
1	100%_Bran	N	C	60	4	1	
2	100%_Natural_Bran	Q	C	110	3	5	

```
> str(cereal)
```

```
'data.frame': 77 obs. of 15 variables:
```

```
$ name      : chr  "100%_Bran" "100%_Natural_Bran" "All-Bran"  
$ mfr       : chr  "N" "Q" "K" "K" ...  
$ type      : chr  "C" "C" "C" "C" ...
```

# Factors - Implications - Reading data

ALWAYS use

*as.is=TRUE* or *stringsAsFactors=FALSE*

when reading data! Either replace or make new variable.

Always explicitly set factors AFTER you read in the data

```
1 cereal <- read.csv('../..sampledata/cereal.csv',
2                     header=TRUE, as.is=TRUE,
3                     strip.white=TRUE)
4 cereal$type <- factor(cereal$type)
5 cereal$shelfF <- factor(cereal$shelf)
6 str(cereal)
```

```
> str(cereal)
```

```
'data.frame': 77 obs. of 16 variables:
```

```
$ name      : chr  "100%_Bran" "100%_Natural_Bran" "All-Bran"
```

```
$ mfr       : chr  "N" "Q" "K" "K" ...
```

```
$ type      : Factor w/ 2 levels "C","H": 1 1 1 1 1 1 1 1 1 1
```

```
$ shelf     : int   3 3 3 3 3 1 2 3 1 3 ...
```

```
$ shelfF    : Factor w/ 3 levels "1","2","3": 3 3 3 3 3 1 2 3
```

## Factors - Implications - *data.frame()*

CAUTION: *data.frame()* automatically converts to factors.

```
1 name <- c("a","b","c","d","e")
2 weight <- c(50,75,50,75, 80)
3 fiber <- c('low',"low","high","high","high")
4 cereal2 <- data.frame(name, weight, fiber)
5 cereal2
6 str(cereal2)
```

```
> cereal2
  name weight fiber
1    a     50  low
2    b     75  low
```

```
> str(cereal2)
'data.frame': 5 obs. of 3 variables:
 $ name: Factor w/ 5 levels "a","b","c","d",...: 1 2 3 4 5
 $ weight: num 50 75 50 75 80
 $ fiber : Factor w/ 2 levels "high","low": 2 2 1 1 1
```

## Factors - Implications - *data.frame()*

CAUTION: *data.frame()* automatically converts to factors.  
Specify *stringsAsFactors=FALSE* – watch CASE and SPELLING

```
1 cereal3 <- data.frame(name, weight, fiber,  
2                       stringsAsFactors=FALSE)  
3 cereal3$fiberF <- factor(cereal3$fiber)  
4 cereal3  
5 str(cereal3)
```

```
> cereal3
```

```
  name weight fiber fiberF
```

```
1     a     50   low    low  
2     b     75   low    low
```

```
> str(cereal3)
```

```
'data.frame': 5 obs. of  4 variables:
```

```
$ name: chr  "a" "b" "c" "d" ...
```

```
$ weight: num  50 75 50 75 80
```

```
$ fiber : chr  "low" "low" "high" "high" ...
```

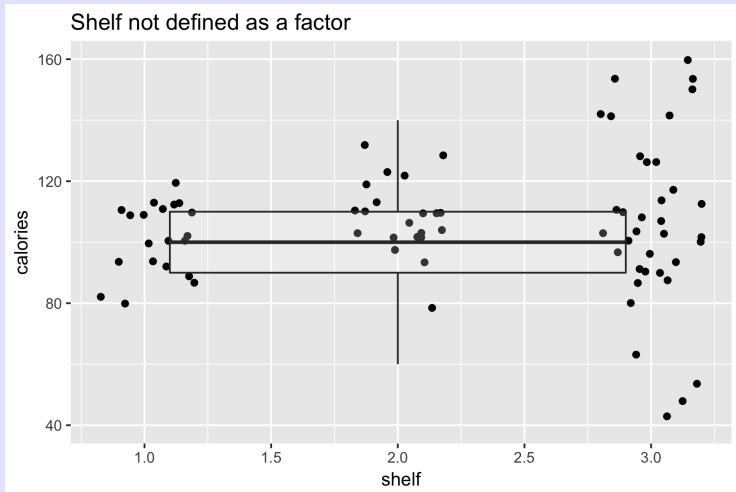
```
$ fiberF: Factor w/ 2 levels "high","low": 2 2 1 1 1
```

## Factors - Implications - *ggplot()*

CAUTION: *ggplot()* looks to see if a numerical variable is a factor or continuous

```
1  # Notice the difference in ggplot
2  plot1 <- ggplot(data=cereal, aes(x=shelf, y=calories))+
3    ggtitle("Shelf not defined as a factor")+
4    geom_jitter(position=position_jitter(w=0.2))+
5    geom_boxplot(alpha=0.2)
6  plot1
7
8
9  plot2 <- ggplot(data=cereal, aes(x=shelfF, y=calories))+
10    ggtitle("Shelf defined as a factor")+
11    geom_jitter(position=position_jitter(w=0.2))+
12    geom_boxplot(alpha=0.2)
13  plot2
```

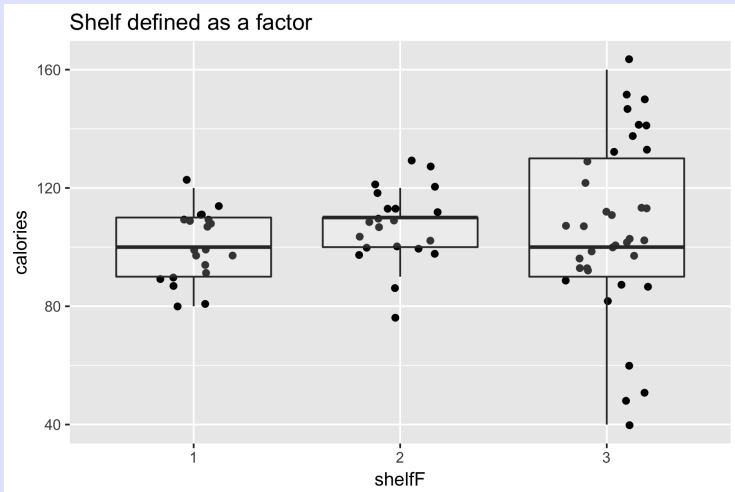
# Factors - Implications - *ggplot()*



Notice how *X* axis is formatted.



# Factors - Implications - *ggplot()*



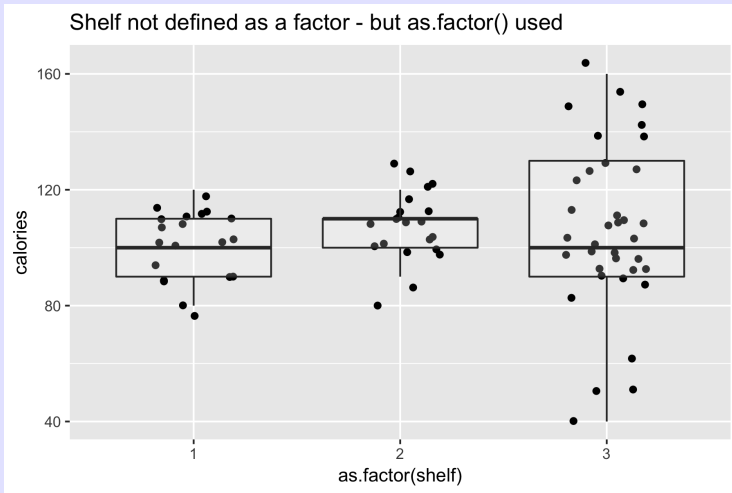
Notice how *X* axis is formatted.

CAUTION: *ggplot()* looks to see if a numerical variable is a factor or continuous.

This can be changed on the fly, but notice the label on the *X* axis.

```
1 plot3 <- ggplot(data=cereal,  
2               aes(x=as.factor(shelf), y=calories))+  
3   ggtitle("Shelf not defined as a factor - but as.factor() r  
4   geom_jitter(position=position_jitter(w=0.2))+  
5   geom_boxplot(alpha=0.2)  
6 plot3
```

# Factors - Implications - *ggplot()*



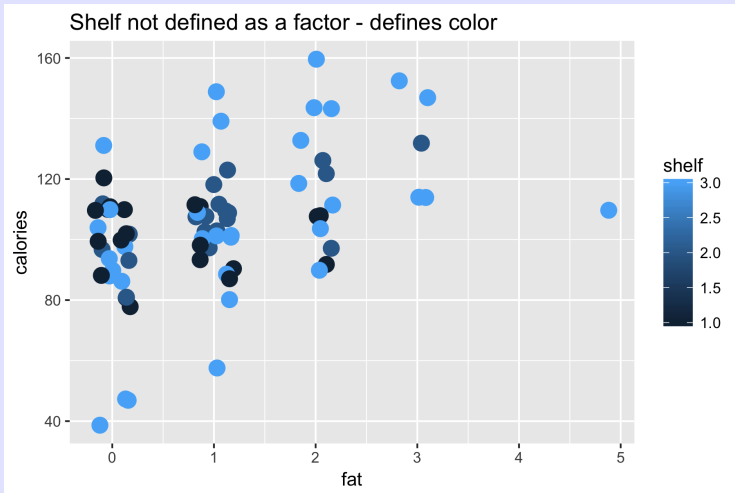
## Factors - Implications - *ggplot()*

CAUTION: *ggplot()* looks to see if a numerical variable is a factor or continuous.

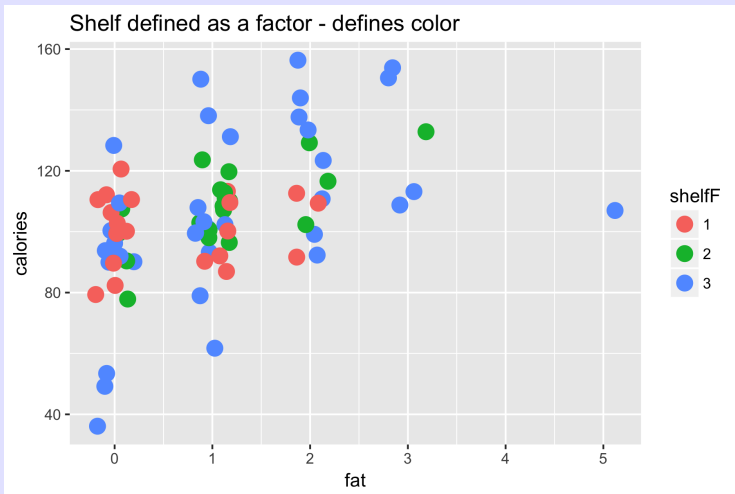
This affect coloring and size attributes.

```
1 plot4 <- ggplot(data=cereal,  
2               aes(x=fat, y=calories, color=shelf))+  
3   ggtitle("Shelf not defined as a factor - defines color")+  
4   geom_jitter(position=position_jitter(w=0.2), size=4)+  
5 plot4  
6  
7 plot5 <- ggplot(data=cereal,  
8               aes(x=fat, y=calories, color=shelfF))+  
9   ggtitle("Shelf not defined as a factor - defines color")+  
10  geom_jitter(position=position_jitter(w=0.2), size=4)+  
11 plot5
```

# Factors - Implications - *ggplot()*



# Factors - Implications - *ggplot()*



## Factors - Implications - *lm()*

CAUTION: *lm()* looks to see if a numerical variable is a factor or continuous to decide if ANOVA or REGRESSION

```
1 result1 <- lm(calories ~ shelf, data=cereal) # regression
2 anova(result1)
3 summary(result1)
4
5
6 result2 <- lm(calories ~ shelfF, data=cereal) # anova
7 anova(result2)
8 summary(result2)
```

## Factors - Implications - *lm()*

CAUTION: *lm()* looks to see if a numerical variable is a factor or continuous to decide if ANOVA or REGRESSION

```
> result1 <- lm(calories ~ shelf, data=cereal) # regression
> anova(result1)
```

Response: calories

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
shelf	1	316	315.73	0.6725	0.4148
Residuals	75	35209	469.45		

```
> summary(result1)
```

Coefficients:

	Estimate	Std. Error	t value	Pr(> t )
(Intercept)	99.660	7.038	14.16	<2e-16 ***
shelf	2.448	2.985	0.82	0.415



## Factors - Implications - *lm()*

CAUTION: *lm()* looks to see if a numerical variable is a factor or continuous to decide if ANOVA or REGRESSION

```
> result2 <- lm(calories ~ shelfF, data=cereal) # anova  
> anova(result2)
```

Response: calories

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
shelfF	2	593	296.58	0.6283	0.5363
Residuals	74	34932	472.05		

```
> summary(result2)
```

Coefficients:

	Estimate	Std. Error	t value	Pr(> t )
(Intercept)	100.500	4.858	20.687	<2e-16 ***
shelfF2	7.119	6.788	1.049	0.298
shelfF3	5.611	6.059	0.926	0.357

Return to the Birds 'n Butts dataset.

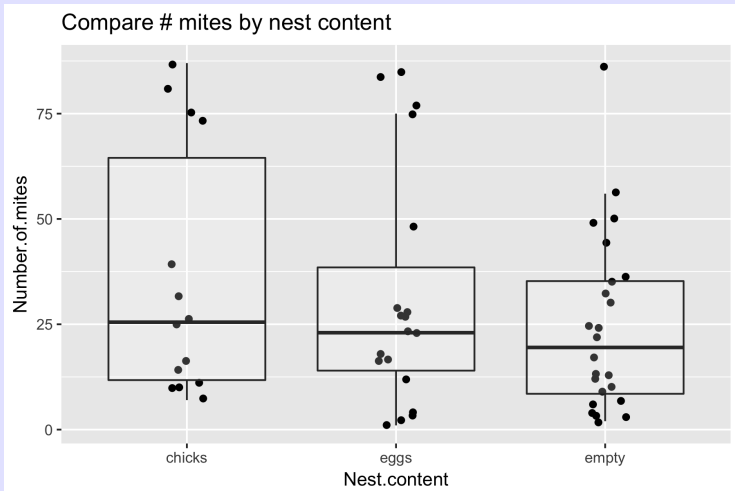
Look at correlational data

- Read data
- Make boxplot of number of mites by nest content colored by species.
- Define factor to order nest content from full to empty in boxplot.

# Factors - Exercise

```
1 library(xlsx)
2 butts <- read.xlsx('../sampledata/bird-butts-data.xlsx', sheet=1,
3                   header=TRUE, stringsAsFactors=FALSE, startRow=1,
4                   butts[1:5,])
5
6 plot1 <- ggplot(data=butts, aes(x=Nest.content, y=Number.of.mites)) +
7   ggtitle("Compare # mites by nest content") +
8   geom_point(position=position_jitter(w=0.1)) +
9   geom_boxplot(alpha=0.2, outlier.size=-1)
10 plot1
```

# Factors - Exercise

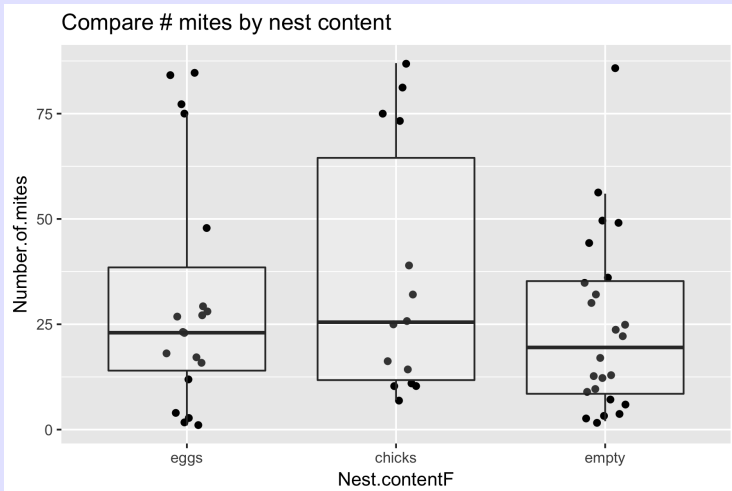


Notice that nest content is ordered alphabetically.

## Factors - Exercise

```
1 butts$Nest.contentF <- factor(butts$Nest.content,  
2     levels=c("eggs","chicks","empty"),  
3     order=TRUE)  
4  
5 plot2 <- ggplot(data=butts, aes(x=Nest.contentF, y=Number.of  
6     ggtitle("Compare # mites by nest content")+  
7     geom_point( position=position_jitter(w=0.1))+  
8     geom_boxplot(alpha=0.2, outlier.size=-1)  
9 plot2
```

# Factors - Exercise



Notice the order for nest content.

## Factors - Summary

- RECOMMENDED: use *as.is=TRUE* or *stringsAsFactors=FALSE* when reading data or creating data.frames.
- RECOMMENDED: explicitly create factor variables *df\$varF <- factor(df\$var)* after data.frame is created.
- CAUTION: *ggplot()* looks at numerical variables to see if factors
- CAUTION: *lm()* and others give different analyses if *X* variable is factor or numerical.

## Dealing with Dates and Times



# Date and Times in *R*

## Types of Dates and Times

- Date values, e.g. 2013-01-12
- Time values within a day, e.g. 23:14
- DateTime (time stamp) values, e.g. 2013-01-12 23:14
- Duration values, e.g. 65:14 is 65 hours and 14 minutes and not a clock time; 9 months and 3 days for a pregnancy is a duration.
- Period values, e.g. 1 month can be 28, 29, 30 or 31 days long. (not covered in this course)

## Some summary articles:

- <http://www.noamross.net/blog/2014/2/10/using-times-and-dates-in-r---presentation-code.html>
- <http://www.statmethods.net/input/dates.html>
- <http://www.jstatsoft.org/v40/i03/paper> *lubridate* package

Dealing with Dates only is 3 step process but is straightforward.

- Input Date values as a character string (e.g. "23/01/2013") (use the `as.is=TRUE` option on `read.table()` or `read.csv()`)
- Convert to internal form (number of days since 1970-01-01)
  - Use the `as.Date(string, format="format codes" )` function with the format codes described in `help(strptime)`.
  - (Internal) negative values indicate days before the origin.
  - Allows arithmetic in the usual fashion.
- Convert to display format (default is yyyy-mm-dd) and/or extract parts of the date using `format(datevar, "format codes")`.

Leap years are properly handled (unlike Excel where 1900 is treated incorrectly).

Create the textConnection and then read in the following data.

```
1 testDates <- textConnection("  
2 d1c,    d2c,    d3c  
3 23aug01, 1970-07-10, 13/07/1956  
4 14sep01, 1972-07-11, 14/07/1956  
5 30feb01, 1972-13-12, 15/07/1956  
6 1mar2001,72-08-14,    16/07/1956") # example of inputting data  
7  
8 my.dates <- read.csv(testDates, header=TRUE,  
9                       as.is=TRUE, strip.white=TRUE)  
10 my.dates  
11 str(my.dates)
```

At this point, all of the variables are CHARACTER data type.

# Dates in *R* - Converting to internal format

Convert to internal format using codes from *help(strptime)*

```
1 my.dates$d1 <- as.Date(my.dates$d1c,  
2                       format="%d%b%y")  
3 my.dates$d1  
4 as.numeric(my.dates$d1)  
5  
6 my.dates$d2 <- as.Date(my.dates$d2c,  
7                       format="%Y-%m-%d")  
8 my.dates$d2  
9 as.numeric(my.dates$d2)  
10  
11 my.dates$d3 <- as.Date(my.dates$d3c,  
12                      format="%d/%m/%y")  
13 my.dates$d3  
14 as.numeric(my.dates$d3)
```

- CAUTION - century implied for %y format character. Always use 4-digit yyyy in input data.
- CAUTION - cannot mix yy and yyyy when using %y or %Y. Always use 4-digit yyyy in input data

# Dates in R - Arithmetic on Dates

The usual arithmetic operations deal with the internal Dates in sensible ways

```
1 my.dates$d3 + 20  # adds 20 days
2
3 mean(my.dates$d3)
4 as.numeric(mean(my.dates$d3)) # correct average stored here
5
6 seq(from=my.dates$d3[1], by="3 weeks", length.out=3) #sequence
7 seq(from=my.dates$d3[1], by="2 months", length.out=3)
8 seq(from=my.dates$d3[1], by="1 year", length.out=3)
```

# Dates in R - Arithmetic on Dates

Extracting parts of a Date. Use the format codes in *help(strptime)*  
CAUTION: results of *format()* are always CHARACTER and may need conversion to numeric values.

```
1 # Extract the day of the month
2 format(my.dates$d3, "%d")
3 format(my.dates$d3, "%d") + 1 # oops
4 as.numeric(format(my.dates$d3, "%d"))
5
6 # Extract day of the week (0=Sunday)
7 as.numeric(format(my.dates$d3, "%w"))
8
9 # Julian day (number of days since 1 Jan of that year) 001-
10 as.numeric(format(my.dates$d3, "%j"))
```

## Dates in *R* - Exercise II

Refer to *road-accidents-summary-2010.csv* file in SampleData.

- Read data into *R*.
- Convert input date to internal *R* dates.
- Plot # accidents/day by day of year.
- Fit a *lowess()* smoother to data using *geom\_smooth()*
- Find proportion of fatalities by day of year, plot, and fit lowess curve.
- Can create both plots in the same window (Hint: melt the two variables and use facetting).

Look at number of accident by day of the week

- Extract day of the week using *format()* or *weekdays()* functions.
- Use *geom\_boxplot()* as seen earlier

# Dates in R - Exercise I

```
1 # The accident data
2 naccidents <- read.csv('../sampledata/road-accidents-2010-sv
3                       header=TRUE,
4                       as.is=TRUE, strip.white=TRUE)
5 naccidents[1:5,]
6 str(naccidents)
```

```
> naccidents[1:5,]
```

	Date	naccidents	nfatal
1	01/01/2010	282	4
2	01/02/2010	657	5
3	01/03/2010	608	2

...

```
> str(naccidents)
```

```
'data.frame': 365 obs. of 3 variables:
```

```
 $ Date      : chr  "01/01/2010" "01/02/2010" "01/03/2010" "
```

...



# Dates in R - Exercise I

```
1 # Convert date to internal date format
2 naccidents$mydate <- as.Date(naccidents$Date,
3                               format="%d/%m/%Y")
4 sum(is.na(naccidents$mydate))
5 naccidents[1:5,]
6 str(naccidents)
```

```
> naccidents[1:5,]
```

	Date	naccidents	nfatal	mydate
1	01/01/2010	282	4	2010-01-01
2	01/02/2010	657	5	2010-02-01

```
> str(naccidents)
```

```
'data.frame': 365 obs. of 4 variables:
```

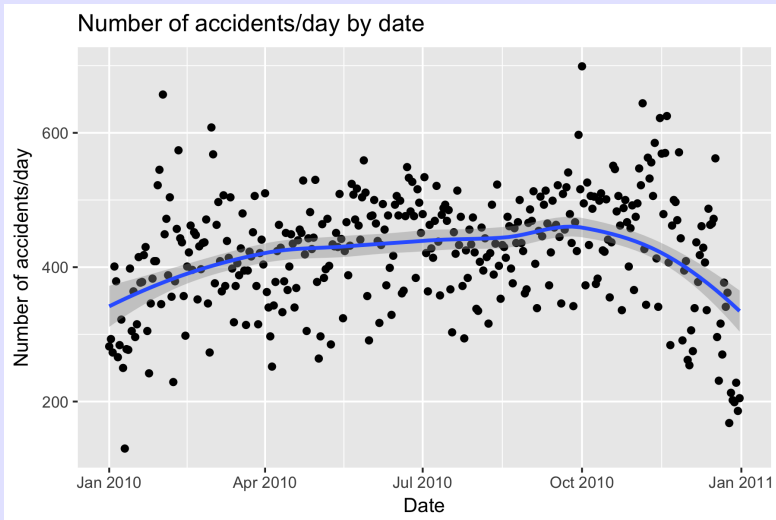
```
$ Date      : chr  "01/01/2010" "01/02/2010" "01/03/2010" "
```

```
$ mydate    : Date, format: "2010-01-01" "2010-02-01" "2010
```

```
>
```

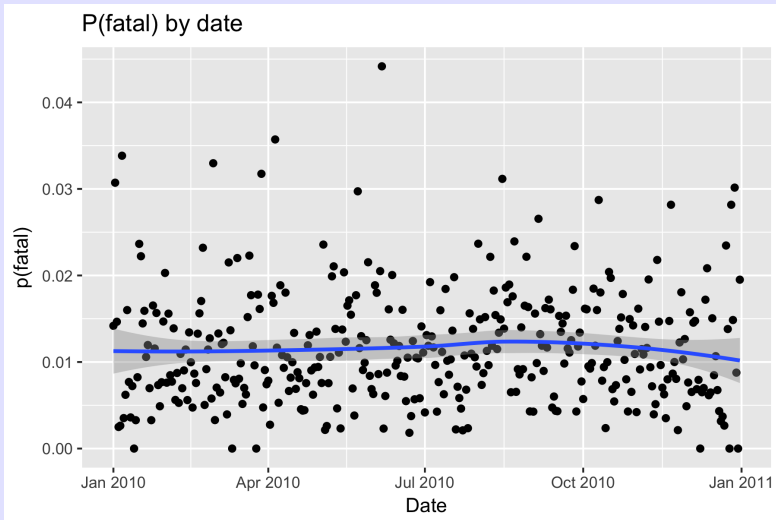
```
1 plotnacc <- ggplot(data=naccidents, aes(x=mydate, y=naccidents))
2   ggtitle("Number of accidents/day by date")+
3   xlab("Date")+ylab("Number of accidents/day")+
4   geom_point()+
5   geom_smooth()
6 plotnacc
```

# Dates in *R* - Exercise I



```
1 # look at proportion of fatalities
2 naccidents$pfatal <- naccidents$nfatal /
3     naccidents$naccidents
4 plotpfatal <- ggplot(data=naccidents, aes(x=mydate, y=pfatal))
5     ggtitle("P(fatal) by date")+
6     xlab("Date")+ylab("p(fatal)")+
7     geom_point()+
8     geom_smooth()
9 plotpfatal
```

# Dates in *R* - Exercise I



## Dates in R - Exercise I

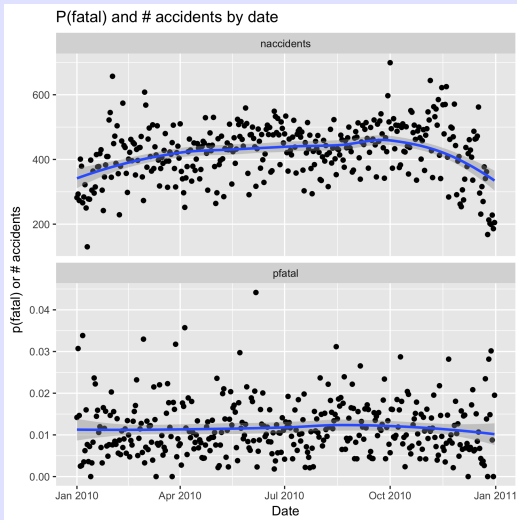
```
1 # melt the data set and plot
2 plotdata <- reshape2::melt(naccidents,
3                             id.var="mydate",
4                             measure.var=c("naccidents","pfatal"),
5                             variable.name="Measure",
6                             value.name="value")
7 head(plotdata,n=2)
8 tail(plotdata,n=2)
```

```
> head(plotdata,n=2)
      mydate      Measure value
1 2010-01-01 naccidents    282
2 2010-02-01 naccidents    657
> tail(plotdata,n=2)
      mydate      Measure      value
729 2010-10-31    pfatal 0.01092896
730 2010-12-31    pfatal 0.01951220
```

# Dates in R - Exercise I

```
1 plotboth <- ggplot(data=plotdata, aes(x=mydate, y=value))+  
2   ggtitle("P(fatal) and # accidents by date") +  
3   xlab("Date")+ylab("p(fatal) or # accidents") +  
4   geom_point() +  
5   geom_smooth() +  
6   facet_wrap(~Measure, ncol=1, scales="free_y")  
7 plotboth
```

# Dates in *R* - Exercise I

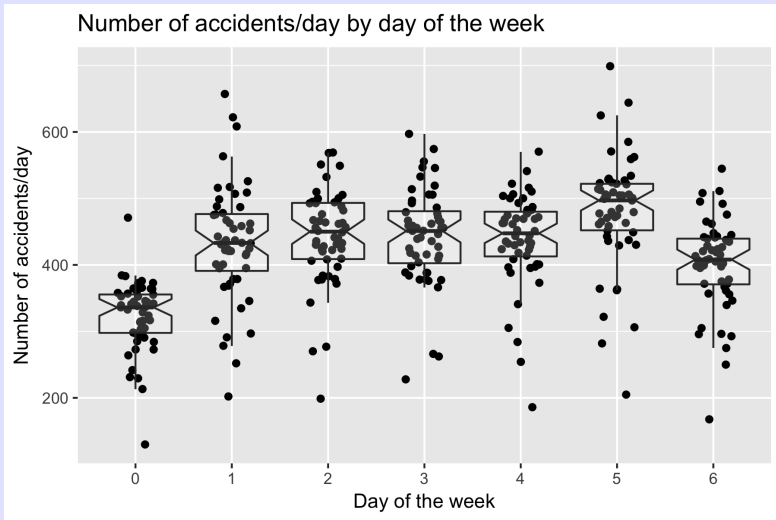




## Dates in *R* - Exercise I

```
1 naccidents$weekday <- format(naccidents$mydate, format="%w")
2 naccidents[1:10,]
3
4 plotnacc2 <- ggplot(data=naccidents, aes(x=weekday, y=naccidents$mydate))
5   ggtitle("Number of accidents/day by day of the week")+
6   xlab("Day of the week")+ylab("Number of accidents/day")+
7   geom_point(position=position_jitter(w=0.2))+
8   geom_boxplot(notch=TRUE, alpha=0.2, outlier.size=-1, outlier.shape=1)
9 plotnacc2
```

# Dates in *R* - Exercise I



Refer to *road-accidents-2010.csv* file in *SampleData*.

- Read data into *R*.
- Convert input date to internal *R* dates.
- Find number of accidents by day of year (use *ddply()* and *summarize()* in *plyr* package)
- Plot # accidents/day by day of year.
- Fit a *lowess()* smoother to data using *geom\_smooth()*

Look at number of accident by day of the week

- Extract day of the week using *format()* or *weekdays()* functions.
- Use *geom\_boxplot()* as seen earlier

## Dates in *R* - Exercise II

```
1 # The accident data
2 accidents <- read.csv('../SampleData/road-accidents-2010.csv',
3                       header=TRUE,
4                       as.is=TRUE, strip.white=TRUE)
5 accidents[1:5,]
6 str(accidents)
```

```
> accidents[1:5,]
```

```
.....
```

	Accident_Severity	Number_of_Vehicles	Number_of_Casualties
1	3	2	1
2	3	1	1

```
> str(accidents)
```

```
'data.frame': 154414 obs. of 33 variables:
```

```
...
```

```
$ Date      : chr  "11/01/2010" "11/01/2010" "12/01/2010" "02/01/2011"
```

```
...
```

## Dates in R - Exercise II

```
1
2 # Convert date to internal date format
3 accidents$mydate <- as.Date(accidents$Date,
4                             format="%d/%m/%Y")
5 sum(is.na(accidents$mydate))
6 accidents[1:5,]
7 str(accidents)

> accidents[1:5,]
...
  Urban_or_Rural_Area Did_Police_Officer_Attend_Scene_of_Acc
1                   1
2                   1
> str(accidents)
'data.frame': 154414 obs. of  33 variables:
 $ Date      : chr  "11/01/2010" "11/01/2010" "12/01/2010"
 $ mydate    : Date, format: "2010-01-11" "2010-01-11" "2010-01-11"
>
```

## Dates in *R* - Exercise II

```
1 # Summarize number of accidents by date
2 library(plyr)
3 naccidents <- ddply(accidents, "mydate", summarize,
4                     freq=length(Accident_Index))
5 naccidents[1:5,]
6 str(naccidents)
```

```
> naccidents[1:5,]
      mydate freq
1 2010-01-01  282
2 2010-01-02  293
...
```

```
> str(naccidents)
```

```
'data.frame': 365 obs. of 2 variables:
```

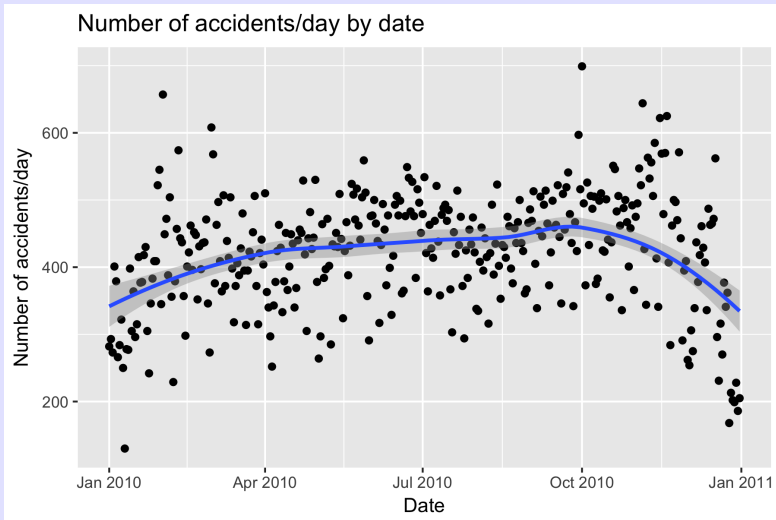
```
$ mydate: Date, format: "2010-01-01" "2010-01-02" "2010-01-03" ...
```

```
$ freq : int 282 293 273 401 379 266 284 322 250 130 ...
```

## Dates in *R* - Exercise II

```
1 plotnacc <- ggplot(data=naccidents, aes(x=mydate, y=freq))+  
2   ggtitle("Number of accidents/day by date")+  
3   xlab("Date")+ylab("Number of accidents/day")+  
4   geom_point()+  
5   geom_smooth()  
6 plotnacc
```

# Dates in *R* - Exercise II





## Dates in R - Exercise II

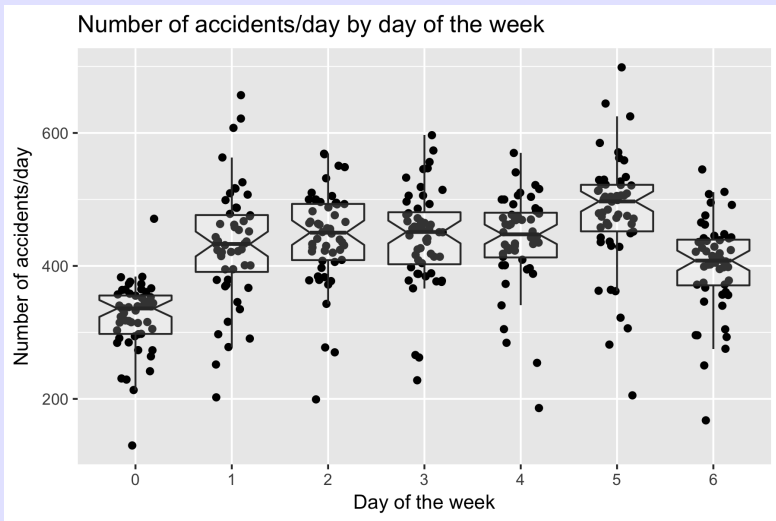
```
1 # Extract day of the week - leave as character values
2 naccidents$weekday <- format(naccidents$mydate, format="%w")
3 naccidents[1:10,]
```

```
> naccidents[1:10,]
      mydate freq weekday
1 2010-01-01  282        5
2 2010-01-02  293        6
3 2010-01-03  273        0
4 2010-01-04  401        1
5 2010-01-05  379        2
...
```

## Dates in *R* - Exercise II

```
1 plotnacc2 <- ggplot(data=naccidents, aes(x=dow, y=freq))+  
2   ggtitle("Number of accidents/day by day of the week")+  
3   xlab("Day of the week")+ylab("Number of accidents/day")+  
4   geom_point(position=position_jitter(w=0.2))+  
5   geom_boxplot(notch=TRUE, alpha=0.2)  
6 plotnacc2
```

# Dates in *R* - Exercise II



Refer to *road-accidents-2010.csv* file in *SampleData*.

- Create 0/1 variable if fatality occurs (no or yes; check codebook for *Accident\_Severity*).  
Use the magic incantation of *recode()* function in *car* package.
- Find proportion of accidents with fatality by day of year
  - The mean of a 0/1 variable is the proportion.  
Use the magic incantation of *ddply()* and *summarize()* in the *plyr* package.
- Plot proportion of fatalities by day of year.
- Fit a *lowess()* smoother to data from *geom\_smooth()*
- Plot proportion of fatalities by day of the week
  - Hint: Extract weekday using *format()*.
  - Hint: Use *geom\_boxplot()* as seen earlier with some jittering and notches.

## Dates in *R* - Exercise III

```
1 names(accidents)
2 unique(accidents$Accident_Severity)
3 library(car)
4 accidents$Fatality <- recode(accidents$Accident_Severity,
5                             ' 1=1; 2:hi=0')
6 accidents[1:5, c("Accident_Severity", "Fatality")]
7 xtabs(~Fatality + Accident_Severity, data=accidents)
```

```
> accidents[1:5, c("Accident_Severity", "Fatality")]
  Accident_Severity Fatality
```

1	3	0
2	3	0

```
> xtabs(~Fatality + Accident_Severity, data=accidents)
      Accident_Severity
```

Fatality	1	2	3
0	0	20440	132243
1	1731	0	0

## Dates in R - Exercise III

The *summarize()* and *ddply()* functions in *plyr* package is quite useful for simple summaries by groups.

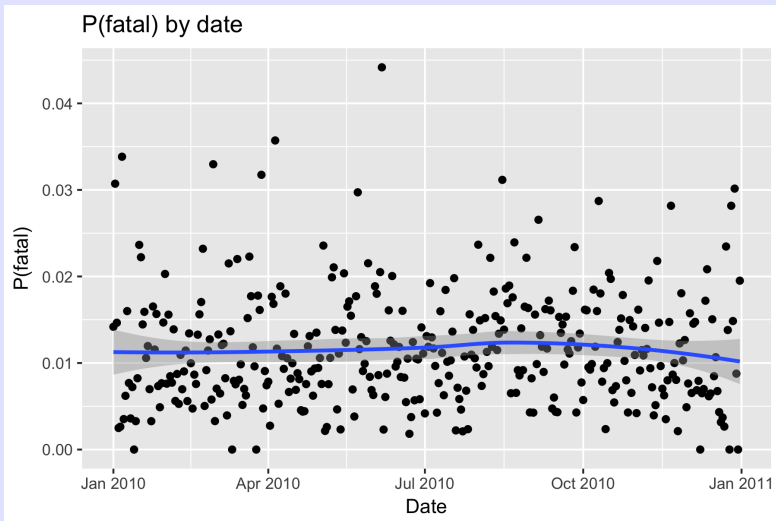
Example of the Split-Apply-Combine paradigm to be explained later.

```
1 library(plyr)
2 pfatal.df <- ddply(accidents, "mydate", summarize,
3                     freq=length(mydate),
4                     pfatal=mean(Fatality))
5 pfatal.df[1:5,]
```

```
> pfatal.df[1:5,]
      mydate freq    pfatal
1 2010-01-01  282 0.014184397
2 2010-01-02  293 0.030716724
3 2010-01-03  273 0.014652015
4 2010-01-04  401 0.002493766
5 2010-01-05  379 0.002638522
```

```
1 plotpfatal <- ggplot(data=pfatal.df,  
2                       aes(x=mydate, y=pfatal))+  
3       ggtitle("P(fatal) by date")+  
4       xlab("Date")+ylab("P(fatal)")+  
5       geom_point()+  
6       geom_smooth()  
7 plotpfatal
```

# Dates in *R* - Exercise III

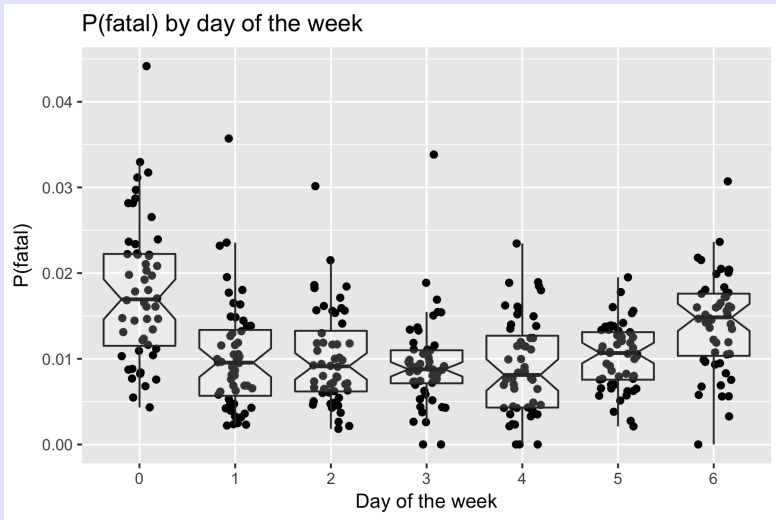




## Dates in R - Exercise II

```
1  # Extract day of the week - leave as character
2  pfatal.df$weekday <- format(pfatal$mydate, format="%w") # 1
3  pfatal.df[1:10,]
4
5  plotpfatal2 <- ggplot(data=pfatal.df, aes(x=weekday, y=pfatal
6    ggtitle("P(fatal) by day of the week")+
7    xlab("Day of the week")+ylab("P(fatal)")+
8    geom_point(position=position_jitter(w=0.2))+
9    geom_boxplot(notch=TRUE, alpha=0.2)
10 plotpfatal2
```

# Dates in *R* - Exercise III



# Date+Times (Time Stamps) in *R*

## NOT A SIMPLE TASK (not a fault of *R*)

- Local time vs Constant Time? Suggest you work in Constant Time to avoid problems when you change locations of computer.
- Time zone - your machine vs. where collected? Use UTC to avoid these problems.
- Daylight savings time? Always measure in Standard Time if possible.
- Leap seconds?

Dealing with Dates+Times is 3 step process:

- Input DateTime values as a character string (e.g. "2013-10-01 10:23") (*as.is=TRUE* on *read.table()* or *read.csv()*). You may need to *paste()* separate date and time.
- Convert to internal form (number of seconds since origin)
  - Use the *as.POSIXct(string, format= )* function where the codes are found in *help(strptime)*
  - Allows arithmetic in the usual fashion
- Convert to display format (default is yyyy-mm-dd hh:mm:ss)

Create the textConnection and then read in the following data.

```
1 testDateTimes <- textConnection("  
2 dt1c,    dt2c  
3 2013-10-23 10:23, 2012-07-13 1:23:00  
4 2013-11-23 25:23, 2012-07-14 3:23:00  
5 2013-12-23 13:23, 2012-07-15 5:23:10  
6 2013-12-24 10:62, 2013-07-15 7:23:23 ") # example of input  
7  
8 my.dt <- read.csv(testDateTimes, header=TRUE,  
9                   as.is=TRUE, strip.white=TRUE)  
10 my.dt  
11 str(my.dt)
```

At this point, all of the variables are CHARACTER data type.

# Date+Times in R

Convert to internal format using codes from *help(strptime)*

```
1 # Convert from character to internal DateTime representation
2 my.dt$dt1 <- as.POSIXct(my.dt$dt1c,
3                          format="%Y-%m-%d %H:%M", tz="UTC")
4 my.dt$dt1
5 as.numeric(my.dt$dt1)
6 str(my.dt1)
```

```
> my.dt$dt1
```

```
[1] "2013-10-23 10:23:00 UTC" NA      "2013-12-23 13:23:00 UTC"
```

```
[4] NA
```

```
> as.numeric(my.dt$dt1)
```

```
[1] 1382523780      NA 1387804980      NA
```

```
> str(my.dt)
```

```
'data.frame': 4 obs. of 5 variables:
```

```
$ dt1c: chr  "2013-10-23 10:23" "2013-11-23 25:23" "2013-12-23 13:23"
```

```
$ dt2c: chr  "2012-07-13 1:23:00" "2012-07-14 3:23:00" "2012-07-14 01:23:00"
```

```
$ dt1 : POSIXct, format: "2013-10-23 10:23:00" NA "2013-12-23 13:23:00"
```

```
$ dt2 : POSIXct, format: "2012-07-13 01:23:00" "2012-07-14 03:23:00" "2012-07-14 01:23:00"
```

Convert to internal format using codes from *help(strptime)*

```
1 # Look what happens if you don't specify a time zone
2 my.dt$dt1b <- as.POSIXct(my.dt$dt1c,
3                           format="%Y-%m-%d %H:%M")
4 my.dt$dt1b
5 as.numeric(my.dt$dt1b)
6 str(my.dt)
```

```
> my.dt$dt1b
[1] "2013-10-23 10:23:00 PDT" NA    "2013-12-23 13:23:00 PST"
[4] NA
```

It only know the time zone from where your machine is currently located.

Arithmetic and sequence operations are allowed

```
1  # read in the the other data values
2  my.dt$dt2 <- as.POSIXct(my.dt$dt2c,
3                          format="%Y-%m-%d %H:%M", tz="UTC")
4  my.dt$dt2
5  as.numeric(my.dt$dt2)
6
7  # Arithmetic aoperations allowed
8  mean(my.dt$dt2)
9  as.numeric(mean(my.dt$dt2))
10
11 # sequences of dates etc
12 seq(from=my.dt$dt2[1], by="3 weeks", length.out=3)
```

## Date+Times in R

Arithmetic and sequence operations are allowed

```
> my.dt$dt2
```

```
[1] "2012-07-13 01:23:00 UTC" "2012-07-14 03:23:00 UTC" "2012-07-15 05:23:00 UTC"
```

```
[4] "2013-07-15 07:23:00 UTC"
```

```
> as.numeric(my.dt$dt2)
```

```
[1] 1342142580 1342236180 1342329780 1373872980
```

```
>
```

```
> # Arithmetic operations allowed
```

```
> mean(my.dt$dt2)
```

```
[1] "2012-10-13 16:23:00 UTC"
```

```
> as.numeric(mean(my.dt$dt2))
```

```
[1] 1350145380
```

```
>
```

```
> # sequences of dates etc
```

```
> seq(from=my.dt$dt2[1], by="3 weeks", length.out=3)
```

```
[1] "2012-07-13 01:23:00 UTC" "2012-08-03 01:23:00 UTC" "2012-08-13 01:23:00 UTC"
```

```
>
```



Extracting parts of DateTime using codes from *help(strptime)*

```
1 # extract the various bits from the date-time format
2 as.numeric(format(my.dt$dt2, "%d"))
3 as.numeric(format(my.dt$dt2, "%H"))
```

```
> as.numeric(format(my.dt$dt2, "%d"))
```

```
[1] 13 14 15 15
```

```
> as.numeric(format(my.dt$dt2, "%H"))
```

```
[1] 1 3 5 7
```

Convert to internal format using codes from *help(strptime)*  
CAUTION - beware of changes due to Daylight savings times.

```
1 # Beware of daylight saving changes
2 # In BC, this occurred at 2013-11-03 at 2:00 am
3 # Compare the behaviour of
4 mytime <- as.POSIXct("2013-11-03 01:57:00", tz="UTC")
5 mytime
6 seq(mytime, by='1 min', length.out=10)
7 as.numeric(seq(mytime, by='1 min', length.out=10))
8
9 mytime <- as.POSIXct("2013-11-03 01:57:00")
10 mytime
11 seq(mytime, by='1 min', length.out=10)
12 as.numeric(seq(mytime, by='1 min', length.out=10))
```

# Date+Times in R

Daylight savings time problems.

```
> mytime
```

```
[1] "2013-11-03 01:57:00 UTC"
```

```
> seq(mytime, by='1 min', length.out=10)
```

```
[1] "2013-11-03 01:57:00 UTC" "2013-11-03 01:58:00 UTC" "2013-11-03 01:59:00 UTC"
```

```
[4] "2013-11-03 02:00:00 UTC" "2013-11-03 02:01:00 UTC" "2013-11-03 02:02:00 UTC"
```

```
[7] "2013-11-03 02:03:00 UTC" "2013-11-03 02:04:00 UTC" "2013-11-03 02:05:00 UTC"
```

```
[10] "2013-11-03 02:06:00 UTC"
```

```
> mytime
```

```
[1] "2013-11-03 01:57:00 PDT"
```

```
> seq(mytime, by='1 min', length.out=10)
```

```
[1] "2013-11-03 01:57:00 PDT" "2013-11-03 01:58:00 PDT" "2013-11-03 01:59:00 PDT"
```

```
[4] "2013-11-03 01:00:00 PST" "2013-11-03 01:01:00 PST" "2013-11-03 01:02:00 PST"
```

```
[7] "2013-11-03 01:03:00 PST" "2013-11-03 01:04:00 PST" "2013-11-03 01:05:00 PST"
```

```
[10] "2013-11-03 01:06:00 PST"
```

CAUTION - beware of changes due to Daylight savings times.  
In BC, this occurred at 2013-11-03 at 2:00 am

```
1 mytime <- as.POSIXct("2013-11-03 01:57:00", tz="UTC")
2 as.numeric(format(seq(mytime, by='1 min', length.out=10),
3                     "%H"))
4
5 mytime <- as.POSIXct("2013-11-03 01:57:00")
6 as.numeric(format(seq(mytime, by='1 min', length.out=10),
7                     "%H"))
```

## Daylight savings time problems

```
> as.numeric(format(seq(mytime, by='1 min', length.out=10),  
[1] 1 1 1 2 2 2 2 2 2 2  
>  
> mytime <- as.POSIXct("2013-11-03 01:57:00")  
> as.numeric(format(seq(mytime, by='1 min', length.out=10),  
[1] 1 1 1 1 1 1 1 1 1 1
```

# Dates+Times in R - Exercise I

Refer to *road-accidents-2010.csv* file in `SampleData`.

- Read data into *R*.
- Combine Date and Time values into a `DateTime` value
- Convert the `DateTime` character string into internal format
  - Hint: use *paste()* to put Date and Time together
  - Hint: check for missing values of the converted `DateTime` values. Why did these occur?
- Find the minute of the accident and draw a histogram - explanation?
  - Hint: use the *binwidth=* option of *geom\_histogram()* to get individual bar by minute
- Find proportion of fatalities by hour of the day and plot - explanation?

## Dates+Times in *R* - Exercise I

```
1 accidents <- read.csv('../..sampledata/road-accidents-2010
2                       as.is=TRUE, strip.white=TRUE)
3 accidents$DateTime <- paste(accidents$Date, accidents$Time)
4 accidents[1:5,c("Date","Time","DateTime")]
```

## Dates+Times in *R* - Exercise I

```
> accidents[1:5,c("Date","Time","DateTime")]
```

	Date	Time	DateTime
1	11/01/2010	07:30	11/01/2010 07:30
2	11/01/2010	18:35	11/01/2010 18:35
3	12/01/2010	10:22	12/01/2010 10:22
4	02/01/2010	21:21	02/01/2010 21:21
5	04/01/2010	20:35	04/01/2010 20:35



## Dates+Times in *R* - Exercise I

```
1 accidents$mydt <- as.POSIXct(accidents$DateTime,  
2                             format="%d/%m/%Y %H:%M", tz="UTC")  
3 accidents[1:5,c("Date","Time","DateTime","mydt")]  
4 str(accidents)
```

## Dates+Times in R - Exercise I

```
> accidents[1:5,c("Date","Time","DateTime","mydt")]
      Date   Time      DateTime      mydt
1 11/01/2010 07:30 11/01/2010 07:30 2010-01-11 07:30:00
2 11/01/2010 18:35 11/01/2010 18:35 2010-01-11 18:35:00
3 12/01/2010 10:22 12/01/2010 10:22 2010-01-12 10:22:00
4 02/01/2010 21:21 02/01/2010 21:21 2010-01-02 21:21:00
5 04/01/2010 20:35 04/01/2010 20:35 2010-01-04 20:35:00
> str(accidents)
'data.frame': 154414 obs. of  34 variables:
 $ Date      : chr  "11/01/2010" "11/01/2010" "12/01/2010" "0
 $ Time      : chr  "07:30" "18:35" "10:22" "21:21" ...
 $ DateTime  : chr  "11/01/2010 07:30" "11/01/2010 18:35" "
 $ mydt      : POSIXct, format: "2010-01-11 07:30:00" "2010-01-1
```

## Dates+Times in *R* - Exercise I

```
1 sum(is.na(accidents$mydt))
2 accidents[is.na(accidents$mydt),c("Date","Time","mydt")]
```

## Dates+Times in R - Exercise I

```
> sum(is.na(accidents$mydt))
```

```
[1] 8
```

```
> accidents[is.na(accidents$mydt),c("Accident_Index", "Date",
```

	Accident_Index	Date	Time	mydt
144656	201091NP08355	24/07/2010		<NA>
144823	2010921000796	17/02/2010		<NA>
145079	2010921002018	20/05/2010		<NA>
145082	2010921002035	20/05/2010		<NA>
145561	2010921004092	20/10/2010		<NA>
149625	2010961001411	26/07/2010		<NA>
149755	2010961001975	14/10/2010		<NA>
149780	2010961002066	25/10/2010		<NA>

## Dates+Times in R - Exercise I

```
1 # Extract the minute of the accident
2 accidents$min <- as.numeric(format(accidents$mydt, "%M"))
3 accidents[1:5, c("Date", "Time", "mydt", "min")]
```

## Dates+Times in R - Exercise I

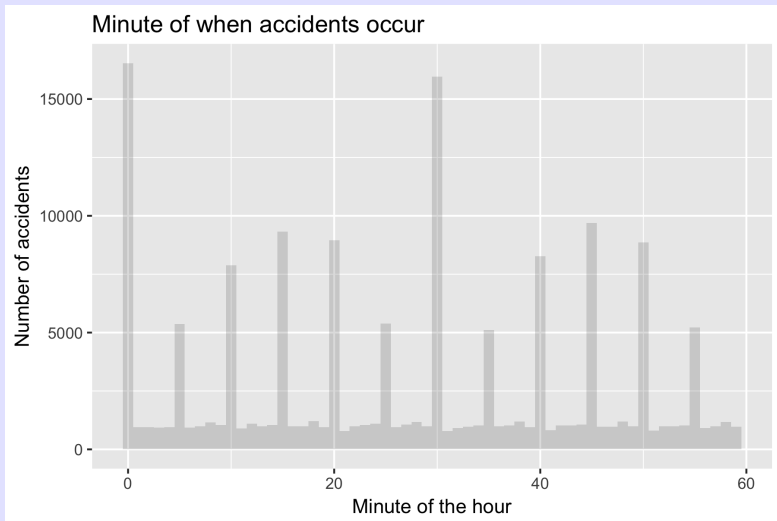
```
> accidents[1:5, c("Date", "Time", "mydt", "min")]
```

	Date	Time		mydt	min
1	11/01/2010	07:30	2010-01-11	07:30:00	30
2	11/01/2010	18:35	2010-01-11	18:35:00	35
3	12/01/2010	10:22	2010-01-12	10:22:00	22
4	02/01/2010	21:21	2010-01-02	21:21:00	21
5	04/01/2010	20:35	2010-01-04	20:35:00	35

## Dates+Times in *R* - Exercise I

```
1 plotaccmin <- ggplot(data=accidents, aes(x=min))+  
2   ggtitle("Minute of when accidents occur")+  
3   xlab("Minute of the hour")+ylab("Number of accidents")+  
4   geom_histogram(binwidth=1, alpha=0.2)  
5 plotaccmin
```

# Dates+Times in R - Exercise I





Find the proportion of fatalities by hour of the day.

```
1 accidents$Fatality <- accidents$Accident_Severity==1
2 xtabs(~Fatality + Accident_Severity, data=accidents)
3
4 accidents$hour <- as.numeric(format(
5     accidents$mydt, "%H"))
6 accidents[1:5, c("Date","Time","mydt","Fatality","hour")]
```

## Dates+Times in R - Exercise I

```
> accidents[1:5, c("Date", "Time", "mydt", "Fatality", "hour")]
```

	Date	Time	mydt	Fatality	hour
1	11/01/2010	07:30	2010-01-11 07:30:00	FALSE	7
2	11/01/2010	18:35	2010-01-11 18:35:00	FALSE	18
3	12/01/2010	10:22	2010-01-12 10:22:00	FALSE	10
4	02/01/2010	21:21	2010-01-02 21:21:00	FALSE	21
5	04/01/2010	20:35	2010-01-04 20:35:00	FALSE	20

Find the proportion of fatalities by hour of the day.

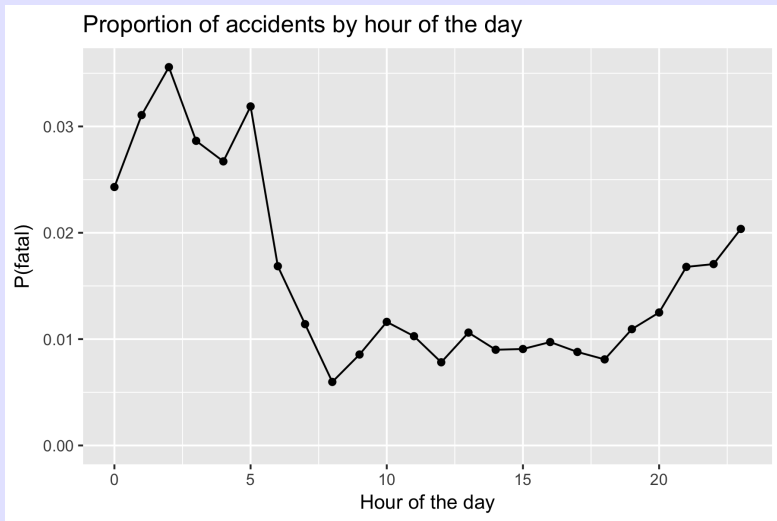
```
1 library(plyr)
2 pfatal.df <- ddply(accidents, "Hour", summarize,
3                     pfatal=mean(Fatality))(
4 pfatal.df[1:5,]
```

```
> pfatal.df[1:5,]  
  hour      pfatal  
1     0 0.02430402  
2     1 0.03106682  
3     2 0.03557618  
4     3 0.02864583  
5     4 0.02671312
```

## Dates+Times in *R* - Exercise I

```
1 plotpfatalhour <- ggplot(data=pfatal.df, aes(x=hour, y=pfatal))
2   ggtitle("Proportion of accidents by hour of the day")+
3   xlab("Hour of the day")+ylab("P(fatal)")+
4   geom_point()+
5   geom_line()
6 plotpfatalhour
```

# Dates+Times in R - Exercise I



- Lengths of time not connected to a particular date.
- Key issues are reading in duration data (65:13) and converting to second.
- *lubridate* package with duration data
  - *hm()*, *ms()*, *hms()* functions available.
  - Longer durations involving days, hours, minutes, seconds more difficult as no standard representation.

Dealing with Durations only is 2 step process but is straightforward.

- Input Duration values as a character string (e.g. "25:13") (use the `as.is=TRUE` option on `read.table()` or `read.csv()`)
- Convert to internal form (number of seconds) using functions in *lubridate*
  - (Internal) negative values indicate "negative" durations (?)
  - Allows arithmetic but some care is needed to keep as duration values..



Create the textConnection and then read in the following data.

```
1 library(lubridate)
2 testDuration.csv <- textConnection("
3 du1c,    du2c
4 10:23, 1:23:00
5 25:23, 3:23:00
6 13:23, 5:23:10
7 10:62, 7:23:23 ") # example of inputting data inline
8
9 my.du <- read.csv(testDuration.csv, header=TRUE, # notice t
10                  as.is=TRUE, strip.white=TRUE)
11 my.du
12 str(my.du)
```

At this point, all of the variables are CHARACTER data type.

# Dates in *R* - Converting to internal format

Convert to internal format using functions from *lubridate*

```
1 # First duration is ambiguous
2 my.du$du1a <- ms(my.du$du1c)
3 my.du$du1a
4 as.numeric(my.du$du1a)
5
6 my.du$du1b <- hm(my.du$du1c)
7 my.du$du1b
8 as.numeric(my.du$du1b)
9
10 my.du$du2 <- hms(my.du$du2c)
11 my.du$du2
12 as.numeric(my.du$du2)
13
14 mean(my.du$du1a)    #???????
15 mean(as.duration(my.du$du1a))
```

Once converted to seconds, standard arithmetic (e.g. means can be computed) but need to specify that `as.duration()`

## Dates in *R* - Converting to internal format

Convert to internal format using functions from *libridate*

```
> my.du$du1a <- ms(my.du$du1c)
```

```
> my.du$du1a
```

```
[1] "10M 23S" "25M 23S" "13M 23S" "10M 62S"
```

```
> as.numeric(my.du$du1a)
```

```
[1] 623 1523 803 662
```

```
> my.du$du1b <- hm(my.du$du1c)
```

```
> my.du$du1b
```

```
[1] "10H 23M 0S" "25H 23M 0S" "13H 23M 0S" "10H 62M 0S"
```

```
> as.numeric(my.du$du1b)
```

```
[1] 37380 91380 48180 39720
```

## Dates in *R* - Converting to internal format

Convert to internal format using functions from *lubridate*

```
> my.du$du2 <- hms(my.du$du2c)
> my.du$du2
[1] "1H 23M 0S"  "3H 23M 0S"  "5H 23M 10S" "7H 23M 23S"
> as.numeric(my.du$du2)
[1] 4980 12180 19390 26603

> mean(my.du$du1a)  #????????
[1] 32.75
> mean(as.duration(my.du$du1a))
[1] 902.75
```

Once converted to seconds, standard arithmetic (e.g. means can be computed) but need to specify that `as.duration()`

# Dates+Times in R - Summary

- Date only data vs. DateTime data
- *as.Date()* vs. *as.POSIXct()*
- Days since origin vs. Seconds since origin
- Careful of time zones/ daylight savings times/ etc
- Use *lubridate* package with pure time data or with duration data (e.g. 65:30 as 65 hours and 30 minutes)
- Use *lubridate* package with period data, e.g. 1 month doesn't have a standard length; what is 31 January + 1 month; does 31 January + 1 month = 28 February - 1 month?

# *R* Basics - Lists

LISTS are a more general data structure than vectors and dataframes.

A dataframe is a special kind of list where each element of the list is a vector of the same length.

LIST is a collection of elements of any type and of any size, including sub-lists.

A common way to report the results of a modelling function

```
1 fit.cal.fat <- lm( calories ~ fat, data=cereal)
2 str(fit.cal.fat)
```

Accessing elements of a list:

```
1 str(fit.cal.fat)
2
3 fit.cal.fat$coefficients  # better to use coef() function
4 fit.cal.fat$coefficients[1]
5 # subtle difference between [k] and [[k]]
6 x <- fit.cal.fat[1]
7 x; str(x)
8 x[1]
9
10 y <- fit.cal.fat[[1]]
11 y; str(y)
12 y[1]
13
14 # lists within lists.
15 fit.cal.fat$model$calories
```

Always try and use the \$ syntax for list elements.

Almost always use [[ k ]] to access list elements by subscripting.



Creating a list:

```
1 age <- c(56, 56, 28, 23, 22)
2 height <- c(185, 162, 185, 167, 190)
3 f.names <- c('Carl', "Lois", 'Matthew', 'Marianne', 'David')
4 people <- list(yob=2013-age, height=height, names=f.names)
5 str(people)
6
7 type <- c("dog", "cat")
8 p.names <- c("fido", "roger")
9 pets <- list(species=type, names=p.names)
10 str(pets)
11
12 schwarz <- list(humans=people, animals=pets)
13 str(schwarz)
14 schwarz$humans
15 schwarz$humans$yob
```

See *help(list)*

```
newlist <- list( elem1=..., elem2=..., elem3=...)
```

Return to the creals dataset.

Look at correlational data

- Read data into a data frame.
- Make a scatter plot of calories vs. grams of fat with a smoother.
- Run a regression of calories vs. grams of fat
- Put the data frame, the boxplot, and the regression fit into a single list structure with named elements.
- Practise access parts of the list.

```
1 cereal <- read.csv('.././sampledata/cereal.csv',
2                     header=TRUE, as.is=TRUE,
3                     strip.white=TRUE)
4 cereal[1:5,]
5 str(cereal)
6
7 # Make the plot
8 plot <- ggplot2::ggplot(data=cereal, aes(x=fat, y=calories))
9   ggtitle("Number of calories vs. grams of fat")+
10   geom_point( position=position_jitter(h=.5, w=.1))+
11   geom_smooth()
12 plot
```

```
1  # Do the fit
2  reg.fit <- lm(calories ~ fat, data=cereal)
3  reg.fit
4
5
6  # create a list structure
7  results <- list(data=cereal, plot=plot, fit=reg.fit)
```

```
1 names(results)
2
3 str(results[1])
4 str(results[[1]])
5
6 results[[1]][1:5,]
7 results$data[1:5,]
```

```
> names(results)
[1] "data" "plot" "fit"
>
> str(results[1])
List of 1
 $ data:'data.frame': 77 obs. of  15 variables:
  ..$ name      : chr [1:77] "100%_Bran" "100%_Natural_Bran" "
  ..$ mfr       : chr [1:77] "N" "Q" "K" "K" ...
  ..$ type      : chr [1:77] "C" "C" "C" "C" ...

> str(results[[1]])
'data.frame': 77 obs. of  15 variables:
 $ name      : chr  "100%_Bran" "100%_Natural_Bran" "All-Bran"
 $ mfr       : chr  "N" "Q" "K" "K" ...
 $ type      : chr  "C" "C" "C" "C" ...
```

## Lists - Exercise

```
> results[1][1:5,]
```

```
Error in results[1][1:5, ] : incorrect number of dimensions
```

```
> results[[1]][1:5,]
```

	name	mfr	type	calories	protein	fat	so
1	100%_Bran	N	C	60	4	1	
2	100%_Natural_Bran	Q	C	110	3	5	

```
> results$data[1:5,]
```

	name	mfr	type	calories	protein	fat	so
1	100%_Bran	N	C	60	4	1	
2	100%_Natural_Bran	Q	C	110	3	5	

```
1 results$plot
2
3 results[3]$coefficients
4 results[[3]]$coefficients
5 results$fit$coefficients
6 summary(results$fit)$r.squared
```



```
> results[3]$coefficients  
NULL
```

```
> results[[3]]$coefficients  
(Intercept)          fat  
  95.131579      9.806005
```

```
> results$fit$coefficients  
(Intercept)          fat  
  95.131579      9.806005
```

```
> summary(results$fit)$r.squared  
[1] 0.2083875
```

- Common output data structure from modelling functions.
- Useful for storing different types of data structures, e.g. plots + data.frame + model results
- Data frames are special type of list.
- Use `$` or `[[ ]]` to access list elements.

# Calling *R* Functions

A FUNCTION is a transformation from input to output.

It has the following parts:

- Function Name - how the function is invoked.
- Arguments - list of arguments to which will be supplied data from the call.  
Arguments can be any type or data structure.
- Body - the code that defines what the function does.  
No limit on what a function can do.
- Output - value of the function. Any data type or structure.

Examples of function calls.

```
1 mean.calories <- mean(cereal$calories)
2 result.lm <- lm( calories ~ fat, data=cereal)
3 plot1 <- ggplot(data=cereal, aes(y=calories, x=fat))+
4       geom_point()
```

A FUNCTION is a transformation from input to output.

Internals of a function e.g. coefficient of variation =  $\text{sd}/\text{mean}$

```
1 my.cv <- function( x ){
2   # Compute the coefficient of variation
3     my.mean <- mean(x)
4     my.sd   <- sd(x)
5     cv      <- my.sd / my.mean
6     names(cv) <- 'cv'
7     return(cv)
8 } # end of my.cv
9 my.data <- c(1:10);  my.data
10 my.cv(my.data)
11 my.cv(x=my.data)
12 my.cv(y=my.data)
```

## R basics - Functions - Internals

A FUNCTION is a transformation from input to output.

Internals of a function e.g. coefficient of variation =  $\text{sd}/\text{mean}$

```
> my.data <- c(1:10)
```

```
> my.data
```

```
[1]  1  2  3  4  5  6  7  8  9 10
```

```
> my.cv(my.data)
```

```
      cv
```

```
0.5504819
```

```
> my.cv(x=my.data)
```

```
      cv
```

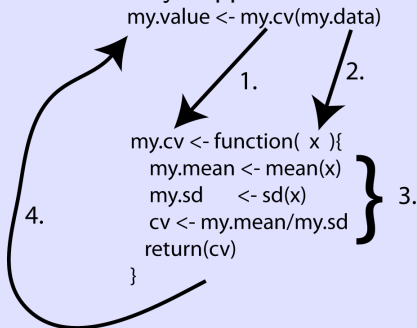
```
0.5504819
```

```
> my.cv(y=my.data)
```

```
Error in my.cv(y = my.data) : unused argument (y = my.data)
```

# R basics - Functions - Actions when called

What actually happens when a function is called?



- 1 See if function is defined in workspace?
- 2 Make a COPY of data objects, and pass COPY to arguments of function.
- 3 Do the computations. All NEW objects are local.
- 4 Take results of `return()`, and send back to calling statement. Destroy any LOCAL objects. Destroy COPY of initial data objects passed to arguments.

How do I find out how to use a function?

- *help(function name)*; e.g. `help(mean)`
- `??mean` - finds all keywords in function descriptions
- Google is your friend!



# R basics - Functions - Reading help

How do I find out how to use a function? *help(mean)*

Description

Generic function for the (trimmed) arithmetic mean.

Usage

```
mean(x, ...)
```

## Default S3 method:

```
mean(x, trim = 0, na.rm = FALSE, ...)
```

```
.....
```

```
1 mean(cereal$calories)
2 mean(cereal$calories, trim=.10)
3 mean(cereal$weight)
4 mean(cereal$weight, na.rm=TRUE)
5 mean(cereal$weight, na.rm=TRUE, trim=0.10)
6 mean(cereal$weight, na.rm=TRUE,
7       trim=5/length(cereal$weight))
```

## Types of arguments

```
help(mean)
```

```
mean(x, trim = 0, na.rm = FALSE, ...)
```

- Arguments with NO default, e.g. *x*.
- Arguments with default, e.g. *trim* or *na.rm*.
- Additional arguments, e.g. ... that are passed to function.

## ADVANCED USAGE

```
1 mean(cereal$calories)
2 mean(cereal$calories, trim=.10)
3 mean(cereal$weight)
4 mean(cereal$weight, na.rm=TRUE)
5 mean(cereal$weight, na.rm=TRUE, trim=0.10)
```

# R basics - Functions - Argument matching

How does *R* match calling arguments to function arguments

```
help(mean)
mean(x, trim = 0, na.rm = FALSE, ...)
```

```
1 mean(cereal$calories)
2 mean(cereal$weight, na.rm=TRUE)
3 mean(cereal$weight, na.rm=TRUE, trim=0.10)
4 mean(na.rm=TRUE, x=cereal$weight, .10) # AVOID
5 mean(na.rm=TRUE, x=cereal$weight, trim=.10, blahblah=3) #
```

- 1 Exact argument names are matched. Default values are replaced.
- 2 Partial match of argument names (CAUTION - AVOID)
- 3 Positional matching left to right of remaining arguments
- 4 Any other arguments go into the ....

Useful to pass functions as arguments to functions

```
1 ddply(cereal, "shelf", summarize, mean=mean(calories))  
2 aaply(matrix, 1, sum)  
3 optim(function, parms)
```

Key problems:

- passing arguments to the passed function via the ... argument.
- very common in more advanced usages such as bootstrapping or optimization

- Very common to automate actions - more general than a script.
- Function should be SELF CONTAINED and so ALL DATA must be passed to function.
- CAUTION: Avoid argument matching with partial matching or in strange order.
- Many possible data structures that can be returned. Most common are:
  - Single value
  - Vector
  - Dataframe
  - List

# *R* Writing *R* Functions

A FUNCTION is a transformation from input to output.

It has the following parts:

- Function Name - how the function is invoked.
- Arguments - list of arguments to which will be supplied data from the call.  
Arguments can be any type or data structure.
- Body - the code that defines what the function does.  
No limit on what a function can do.
- Output - value of the function. Any data type or structure.

Examples of function calls.

```
1 mean.calories <- mean(cereal$calories)
2 result.lm <- lm( calories ~ fat, data=cereal)
3 plot1 <- ggplot(data=cereal, aes(y=calories, x=fat))+
4       geom_point()
```

A FUNCTION is a transformation from input to output.

Defining a function e.g. coefficient of variation =  $sd/mean$

```
1 my.cv <- function( x ){
2   # Compute the coefficient of variation
3   my.mean <- mean(x)
4   my.sd   <- sd(x)
5   cv     <- my.sd / my.mean
6   names(cv) <- 'cv'
7   return(cv)
8 } # end of my.cv
9 my.data <- c(1:10); my.data
10 my.cv(my.data)
11 my.cv(x=my.data)
12 my.cv(y=my.data)
```

- INPUT - list of arguments to which will be supplied data from the call
- OUTPUT - value of *return()* - good form to put as last statement



## R basics - Functions - Internals

A FUNCTION is a transformation from input to output.

Internals of a function e.g. coefficient of variation =  $\text{sd}/\text{mean}$

```
> my.data <- c(1:10)
```

```
> my.data
```

```
[1]  1  2  3  4  5  6  7  8  9 10
```

```
> my.cv(my.data)
```

```
      cv
```

```
0.5504819
```

```
> my.cv(x=my.data)
```

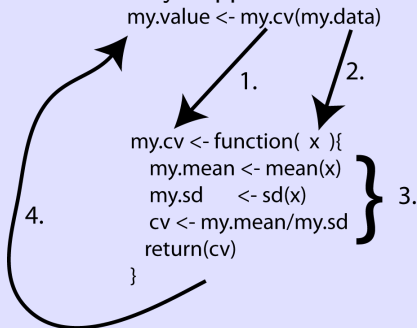
```
      cv
```

```
0.5504819
```

```
> my.cv(y=my.data)
```

```
Error in my.cv(y = my.data) : unused argument (y = my.data)
```

What actually happens when a function is called?



- 1 See if function is defined in workspace?
- 2 Make a COPY of data objects, and pass COPY to arguments of function.
- 3 Do the computations. All NEW objects are local.
- 4 Take results of `return()`, and send back to calling statement.  
Destroy any LOCAL objects.  
Destroy COPY of initial data objects passed to arguments.

Good programming practices:

- For all but trivial functions, use lots of comments describing the purpose and the arguments required for the function.
- Pass all necessary data to functions.  
Don't rely on GLOBAL variables in calling environment.
- Explicitly return final object.
- No side effects, i.e. do not modify any global variable using «-
- Try and handle missing values gracefully.
- ALWAYS test your function with known data.

Creating the function:

- Use text editor to define the function as shown above.
- Highlight and Run in *Rstudio* to compile the function.
- Debug the function.
- Use the function.

If you modify the function, you **MUST** recompile it before using it again.

Try

```
1 my.cv(cereal$calories)
2 my.cv(cereal$weight)
```

Modify the *my.cv()* to incorporate one more argument:

- Removing missing values before computations  
(*remove.na=TRUE*)

```
1 my.cv(cereal$calories)
2 my.cv(cereal$weight)
3 my.cv(cereal$weight, remove.na=TRUE)
```

More on arguments to function.

```
1  # Adding complexity. More arguments to the function
2  my.cv <- function( x ,remove.na=FALSE){
3  # Compute the coefficient of variation
4  # after removing na's
5      my.mean <- mean(x, na.rm=remove.na)
6      my.sd    <- sd(x,   na.rm=remove.na)
7      cv      <- my.mean / my.sd
8      names(cv) <- 'cv'
9      return(cv)
10 } # end of my.cv
11
12 my.cv(cereal$calories)
13 my.cv(cereal$weight)
14 my.cv(cereal$weight, remove.na=TRUE)
```

## R basics - Functions - Defining a function

“Reusing” argument names. Often done and is confusing until you get used to it.

```
1  # Adding complexity. More arguments to the function
2  my.cv <- function( x ,na.rm=FALSE){
3  # Compute the coefficient of variation
4  # after removing na's
5      my.mean <- mean(x, na.rm=na.rm)
6      my.sd    <- sd(x,  na.rm=na.rm)
7      cv      <- my.mean / my.sd
8      names(cv) <- 'cv'
9      return(cv)
10 } # end of my.cv
11
12 my.cv(cereal$calories)
13 my.cv(cereal$weight)
14 my.cv(cereal$weight, na.rm=TRUE)
```

Keep track of where the *na.rm* comes from and is going to with code such as *na.rm=na.rm*.

## R basics - Functions - Defining a function

Add a *chop* argument than trims the top 'chop' and bottom 'chop' fraction of observations before computing the mean and sd.

Hint: `low <- quantile(x, prob=chop, na.rm=TRUE)` returns the *chop* quantile.

Hint: `high <- quantile(x, prob=1-chop, na.rm=TRUE)` returns the  $1 - \text{chop}$  quantile.

Hint: `x[x >= low & x <= high]` selects middle observations .

```
> my.cv(cereal$calories)
```

```
4.859589
```

```
> my.cv(cereal$weight)
```

```
NA
```

```
> my.cv(cereal$weight, na.rm=TRUE)
```

```
6.760404
```

```
> my.cv(cereal$weight, na.rm=TRUE, chop=0.10)
```

```
22.98736
```



## R basics - Functions - Defining a function

```
1 my.cv <- function( x , chop=0, na.rm=FALSE){  
2   # Compute the coefficient of variation  
3   # after removing a fraction 'chop' from top and bottom and  
4   # dealing with na's  
5     newx <- x[ x ≥ quantile(x, probs=chop, na.rm=TRUE) &  
6               x ≤ quantile(x, probs=1-chop, na.rm=TRUE)]  
7     my.mean <- mean(newx, na.rm=na.rm)  
8     my.sd    <- sd(newx, na.rm=na.rm)  
9     cv      <- my.mean / my.sd  
10    names(cv) <- 'cv'  
11    return(cv)  
12 } # end of my.cv
```

Useful functions for debugging functions:

- *browser()* - stops when hit. *Rstudio* can also set a break point.

```
1 my.cv <- function( x , trimfrac=0, na.action=FALSE){  
2   # Compute the coefficient of variation  
3   # after removing trimfrac from top and bottom and  
4   # dealing with na's  
5     newx <- x[ x ≥ quantile(x, probs=trimfrac) &  
6               x ≤ quantile(x, probs=1-trimfrac)]  
7     browser()  
8     ....  
9   }  
10 my.data <- c(0, 50:60, 100); my.data
```

Don't forget to recompile function after inserting/removing *browser()*

Actions after the *browser()* is hit

- any *R* expression. Check out local variables etc.
- *n*, *c*, *Q* for *next*, *continue*, *Quit* respectively.
- CAUTION: blank lines will quit the *browser()* - GROAN

Write a function *my.summary(x)* which returns

- Total number of observations (including NAs)
- Total number of NAs
- Mean (ignoring NAs)
- Std dev (ignoring NAs)
- CV (ignoring NAs)

It should return a data.frame.

```
> my.summary(cereal$calories)
  nobs nmiss      mean      sd      cv
1   77     0 105.0649 21.62013 4.859589
> my.summary(cereal$weight)
  nobs nmiss  mean      sd      cv
1   77     2 1.0304 0.1524169 6.760404
```

## R basics - Functions - Exercise I

```
1 my.summary <- function(x){  
2   # Compute the number, number missing, mean, sd, cv  
3   # with all the NA removed  
4     nobs <- length(x)  
5     nmiss <- sum(is.na(x))  
6     mean <- mean(x, na.rm=TRUE)  
7     sd    <- sd(x, na.rm=TRUE)  
8     cv    <- sd/mean  
9     res   <- c(nobs,nmiss,mean,sd,cv)  
10    res   <- data.frame(nobs,nmiss,mean,sd,cv,  
11                        stringsAsFactors=FALSE)  
12    return(res)  
13 } # end of my.summary
```

Recall cereal data.

Write a function *my.lines(cereal.df)* which

- Estimates the regression line between calories and fat using *lm()*
- Returns the estimated slope, its se, and the 95% confidence interval as a data frame.

```
> my.line(cereal)
```

	slope	slope.se	lcl	ucl
fat	9.806005	2.206897	5.409642	14.20237

```
1 my.line <- function(df){
2   # do a regression of Calories vs Fat and return
3   # the slope, its se, and the 95% ci on the slope
4   fit <- lm(Calories ~ Fat, data=df)
5   slope <- coef(fit)[2]
6   slope.se <- sqrt(diag(vcov(fit)))[2]
7   slope.ci <- confint(fit)[2,]
8   lcl <- slope.ci[1]
9   ucl <- slope.ci[2]
10  res <- data.frame(slope, slope.se, lcl, ucl, stringsAsFactors=FALSE)
11  return(res)
12 } # end my.line
```

Modify your function to specify the names of the  $X$  and  $Y$  variables in the call.

Hint: Use

```
1  fit <- lm(df[,Yname] ~ df[,Xname], data=df)
```

in your function body to do the fit.

```
> my.line2(cereal, Xname="fat",      Yname="calories")
              X          Y      slope slope.se      lcl      ucl
df[, Xname] fat calories 9.806005 2.206897 5.409642 14.20237
> my.line2(cereal, Xname="protein", Yname="calories")
              X          Y      slope slope.se      lcl
df[, Xname] protein calories 0.4091816 2.279836 -4.132485 4
```



```
1 my.line2 <- function(df, Xname, Yname){
2   # do a regression of Yname vs Xname and return
3   # the slope, its se, and the 95% ci on the slope
4   fit <- lm(df[,Yname] ~ df[,Xname], data=df)
5   slope <- coef(fit)[2]
6   slope.se <- sqrt(diag(vcov(fit)))[2]
7   slope.ci <- confint(fit)[2,]
8   lcl <- slope.ci[1]
9   ucl <- slope.ci[2]
10  res <- data.frame(X=Xname, Y=Yname, slope,
11                    slope.se, lcl, ucl)
12  return(res)
13 } # end my.line2
```

Modify your function to specify the names of the  $X$  and  $Y$  variables in the call.

Return a list with the plot, the fitted object, and the summary.

Hint: You will use `aes_string()` in `ggplot()`.

```
> my.line3(cereal, Xname="fat",      Yname="calories")
```

returns the plot, the fit object, and the final result

## R basics - Functions - Exercise IV

```
1 my.line3 <- function(df, Xname, Yname){
2   # Get the plot
3   plot <- ggplot(data=df, aes_string(x=Xname, y=Yname))+
4     ggtitle(paste("Plot of ", Yname," vs ", Xname, sep=""))+
5     geom_point( position=position_jitter(h=.1, w=.1))+
6     geom_smooth(method="lm", se=FALSE)
7
8   # do a regression of Y vs X and return
9   # the slope, its se, and the 95% ci on the slope
10  fit <- lm(df[,Yname] ~ df[,Xname], data=df)
11  slope <- coef(fit)[2]
12  slope.se <- sqrt(diag(vcov(fit)))[2]
13  slope.ci <- confint(fit)[2,]
14  lcl <- slope.ci[1]
15  ucl <- slope.ci[2]
16  res <- data.frame(X=Xname, Y=Yname, slope, slope.se, lcl,
17
18    return(list(plot=plot, fit=fit, res=res))
19 } # end my.line3
```

Useful to pass functions as arguments to functions

- `report <- ddply(df, "byvar", function) # SAC paradigm`
- `bootres <- boot(data, function, reps) # bootstrapping`
- `max <- optim(function, parms)`

Key problems:

- passing arguments to the passed function, the `...` argument

Use your *my.line()* function and *ddply()* to fit a separate line between calories and fat for the 3 shelves.

```
> ddply(cereal, "shelf", my.line)
```

	shelf	slope	slope.se	lcl	ucl
1	1	0.3703704	3.581444	-7.153964	7.894705
2	2	10.8333333	2.626300	5.336424	16.330243
3	3	11.7948718	3.698559	4.278496	19.311248

Use your *my.line2()* function and *ddply()* to fit a separate line between *Xname* and *Yname* for the 3 shelves.

```
> ddpoly(cereal, "shelf", my.line2, Xname="fat", Yname="calo  
shelf    X          Y      slope slope.se      lcl      uc  
1        1 fat calories 0.3703704 3.581444 -7.153964  7.894705  
2        2 fat calories 10.8333333 2.626300  5.336424 16.330243  
3        3 fat calories 11.7948718 3.698559  4.278496 19.311248
```

Notice how the last two arguments pass values to the function — more on this later.

Use your *my.line3()* function and *ddply()* to fit a separate line between *Xname* and *Yname* for the 3 shelves and return the plots, the fits, and the results. [More on this later]

```
> ddpoly(cereal, "shelf", my.line2, Xname="fat", Yname="calo  
shelf    X          Y          slope slope.se      lcl      uc  
1      1 fat calories  0.3703704 3.581444 -7.153964  7.894705  
2      2 fat calories 10.8333333 2.626300  5.336424 16.330243  
3      3 fat calories 11.7948718 3.698559  4.278496 19.311248
```

Notice how the last two arguments pass values to the function — more on this later.

Use your *my.line3()* function and *ddply()* to fit a separate line between *Xname* and *Yname* for the 3 shelves and return the plots, the fits, and the results. [More on this later]

```
results <- dlply(cereal, "shelf", my.line3, Xname="fat", Yname="fat",  
names(results)  
results[[1]]
```

Notice how the last two arguments pass values to the function — more on this later.



### Bootstrapping

- In some cases, the SE is not easily found because the problem is non-standard or certain assumptions (e.g. normality) is violated.
- Bootstrapping provides a way to compute SE for statistics similar to “means”. Does not work well for statistics that are related to order statistics such as max, min, median, etc.

### Key Idea:

- Define a function that computes a statistic based on some data
- Create a boot-strap sample which is a sample with replacement from original data
- Compute the statistic on the boot-strap sample.
- Repeat previous two steps many (typically more than 1000) times.
- Look at SD of statistics and 2.5<sup>th</sup> and 97.5<sup>th</sup> percentiles for SE and CI

## Bootstrapping

```
1 library(boot)
2 help(boot)
```

```
boot(data, statisticfunction, R, sim = "ordinary", stype = c,
      strata = rep(1,n), L = NULL, m = 0, weights = NULL,
      ran.gen = function(d, p) d, mle = NULL,
      simple = FALSE, ...,                                     <= NOTICE
      parallel = c("no", "multicore", "snow"),
      ncpus = getOption("boot.ncpus", 1L), cl = NULL)
```

Allows you pass data (and other arguments) to your function.

## R basics - Functions - Passing functions as arguments

Bootstrapping Example: ratio of mean calories/serving / mean fat/serving

```
1 ratio.meanY.meanX <- function(df, ind, Y,X, na.rm=FALSE){  
2   # Compute the ratio of the mean of Y to  
3   # mean of X potentially removing missing values  
4   res <- mean(df[ind,X],na.rm=na.rm)/  
5           mean(df[ind,Y],na.rm=na.rm)  
6   names(res) <- "ratio"  
7   return(res)  
8 }
```

First argument to function is *data* and second argument is a vector of indices, indicating which rows of data frame to use.

```
> ratio.meanY.meanX(df=cereal,1:nrow(cereal), X="fat",Y="cal")  
      ratio  
0.009641533
```

# R basics - Functions - Passing functions as arguments

Bootstrapping Example: ratio of mean calories/serving / mean fat/serving

```
1 library(boot)
2 bootres <- boot(cereal, ratio.meanY.meanX, R=10,
3               X="Fat",Y="Calories", na.rm=TRUE )
4 bootres
```

```
> bootres
ORDINARY NONPARAMETRIC BOOTSTRAP
```

```
Call: boot(data = cereal, statistic = ratio.meanY.meanX, R =
  Y = "calories", na.rm = TRUE)
```

Bootstrap Statistics :

	original	bias	std. error
t1*	0.009641533	9.710211e-05	0.001037265

## R basics - Functions - Passing functions as arguments

Bootstrapping Example: ratio of mean calories / mean fat

Examine structure of *bootres* further:

```
1 str(bootres)
2 bootres$t0
3 bootres$t
4 quantile(bootres$t, prob=c(0.25, .975))

> bootres$t0
      ratio
0.009641533
> bootres$t
           [,1]
[1,] 0.008798017
[2,] 0.008663366
      .....
> quantile(bootres$t, prob=c(0.25, .975))
           25%          97.5%
0.009076569 0.011911880
```

Bootstrapping Example: ratio of mean calories/serving / mean fat/serving

Add browser to my function to see what happens, esp. the calling arguments

Find the SE and 95% CI for  $R^2$  on the regression of two variables from the cereal dataset

## R basics - Functions - Bootstrapping - Exercise

Find the SE and 95% CI for  $R^2$  on the regression of two variables from the cereal dataset

```
1 r2YX <- function(df, ind, Y,X){
2   # Compute the regression of Y on X and then find R2
3   fit <- lm( df[ind,Y] ~ df[ind, X])
4   res <- summary(fit)$r.squared
5   names(res) <- "R2"
6   return(res)
7 } # end r2YX
```

First argument to function is *data* and second argument is a vector of indices, indicating which rows of data frame to use.

```
> fit <- lm( calories ~ fat, data=cereal)
> summary(fit)$r.squared
[1] 0.2083875
```

```
> r2YX(cereal, 1:nrow(cereal), Y="Calories", X="Fat")
      R2
0.2083875
```



# R basics - Functions - Passing functions as arguments

Find the SE and 95% CI for  $R^2$  on the regression of two variables from the cereal dataset

```
1 bootres <- boot(cereal, r2YX, R=100,  
2               X="fat",Y="calories" )  
3 bootres  
4 str(bootres)  
5 bootres$t0  
6 bootres$t[1:10]  
7 quantile(bootres$t, prob=c(0.25, .975))
```

```
> bootres  
ORDINARY NONPARAMETRIC BOOTSTRAP
```

Bootstrap Statistics :

	original	bias	std. error
t1*	0.2083875	-0.004412033	0.06778598

Find the SE and 95% CI for  $R^2$  on the regression of two variables from the cereal dataset

```
> bootres
```

```
ORDINARY NONPARAMETRIC BOOTSTRAP
```

```
Call: boot(data = cereal, statistic = r2YX, R = 100, X = "Fa
```

	original	bias	std. error
t1*	0.2083875	-0.004412033	0.06778598

```
> quantile(bootres$t, prob=c(0.25, .975))
```

	25%	97.5%
	0.1553856	0.3548753

- Very common to automate actions - more general than a script.
- All functions should be SELF CONTAINED
  - Do not reference variables NOT in argument list
  - Do not create side effects
- Many possible data structures that can be returned. Most common are:
  - Single value
  - Vector
  - Dataframe
  - List
- Don't forget to use the ... argument to pass arguments to passed functions.

Split - Apply - Combine

Performing the same analysis  
to multiple chunks of your data

# Split - Apply - Combine

## Split-Apply-Combine

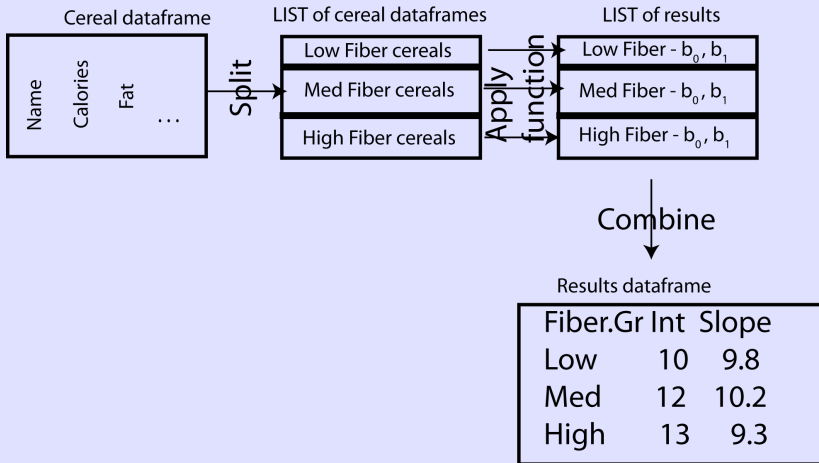
- **Split** up a big data frame
- **Apply** a function to each piece
- **Combine** the results together

## Examples

- Compute the mean calories/serving for each display shelf.
- Compute number of accidents and  $p(\text{fatality})$  for each day in the year.
- Fit a separate regression line to different fiber groups.
- Do a separate analysis for each year of accident data.

# Split - Apply - Combine - Schematic

Find the slope and intercept of the regression of Calories vs. Fat for each Fiber Group in the cereal dataframe.



# Split - Apply - Combine

## Base *R* procedures (AVOID)

- *by()* takes data.frame → list
- *split()* takes data.frame → list
- *lapply()* takes list → list
- *sapply()* takes list → vector or matrix

## *plyr* package (much more logically arranged) (RECOMMENDED)

- *xyply()* where x and y are d=data.frame, l=list, a=array, \_=nothing
- Hadley Wickham (2011).

The Split-Apply-Combine Strategy for Data Analysis.

Journal of Statistical Software, 40(1), 1-29.

<http://www.jstatsoft.org/v40/i01/>.

# Split - Apply - Combine

AVOID: Base R (input data structure (left); output data structure (top))

	array	data frame	list	nothing
array	apply			
data frame		<i>aggregate</i>	by	
list	sapply		lapply	
n replicates	replicate		replicate	
function arguments	mapply		mapply	



# Split - Apply - Combine

USE: *plyr* package (input data structure (left); output data structure (top))

	array	data frame	list	nothing
array	aapply	adply	alply	a_ply
data frame	dapply	<b>ddply</b>	<b>dlply</b>	d_ply
list	lapply	<b>ldply</b>	llply	l_ply
n replicates	rapply	rdply	<b>rlply</b>	r_ply
function arguments	maply	mdply	<b>mlply</b>	m_ply

## Split - Apply - Combine - *ddply()* + *summarize()*

Cereal dataset.

Find the number of cereals and the mean calories/serving for each shelf

```
1 cereal <- read.csv('../sampledata/cereal.csv',
2                     header=TRUE, as.is=TRUE,
3                     strip.white=TRUE)
4 cereal[1:5,]
5
6 library(plyr)
7 sumstats <- ddply(cereal, "shelf", summarize,
8                   ncereal=length(name),
9                   mean.calories=mean(calories))
10 sumstats
```

```
> sumstats
  shelf ncereal mean.calories
1     1      20    100.5000
2     2      21    107.6190
3     3      36    106.1111
```

Find the following quantities for each shelf:

- Standard deviation of calories/serving
- Mean number of calories from fat (1 g of fat has 9 calories)
- Mean proportion of calories from fat of total calories.
- Mean weight/serving

```
> sumstats
```

	shelf	std.calores	mean.fcal	mean.pcal.fat	mean.wt
1	1	11.45931	5.40	0.05404545	0.991500
2	2	12.20851	9.00	0.07986014	1.015714
3	3	29.01012	11.25	0.09859932	NA

## Split - Apply - Combine - *ddply()* + *summarize()* Exercise I

```
1 library(plyr)
2 sumstats <- ddply(cereal, "shelf", summarize,
3                   std.calores=sd(calories),
4                   mean.fcal = mean(fat*9),
5                   mean.pcal.fat = mean( fat*9 / calories),
6                   mean.wt=mean(weight))
7 sumstats
```

```
> sumstats
```

	shelf	std.calores	mean.fcal	mean.pcal.fat	mean.wt
1	1	11.45931	5.40	0.05404545	0.991500
2	2	12.20851	9.00	0.07986014	1.015714
3	3	29.01012	11.25	0.09859932	NA

Revise to account for missing values:

```
> sumstats
```

	shelf	std.calores	mean.fcal	mean.pcal.fat	mean.wt
1	1	11.45931	5.40	0.05404545	0.991500
2	2	12.20851	9.00	0.07986014	1.015714
3	3	29.01012	11.25	0.09859932	1.062353

# Split - Apply - Combine - *ddply()* + *summarize()* Exercise I

Revise to account for missing values:

```
1 library(plyr)
2 sumstats <- ddply(cereal, "shelf", summarize,
3                   std.calores=sd(calories),
4                   mean.fcal = mean(fat*9),
5                   mean.pcal.fat = mean( fat*9 / calories),
6                   mean.wt=mean(weight, na.rm=TRUE))
7 sumstats
```

```
> sumstats
```

	shelf	std.calores	mean.fcal	mean.pcal.fat	mean.wt
1	1	11.45931	5.40	0.05404545	0.991500
2	2	12.20851	9.00	0.07986014	1.015714
3	3	29.01012	11.25	0.09859932	1.062353

Fit a separate regression line between calories and fat and report the intercept and slope for each shelf.

Recall line for ALL of data is found as:

```
result <- lm(calories ~ fat, data=cereal)
summary(result)
coef(result)
coef(result)[1]
coef(result)[2]
```

```
> sumstats
  shelf intercept      slope
1      1 100.27778  0.3703704
2      2  96.78571 10.8333333
3      3  91.36752 11.7948718
```

## Split - Apply - Combine - *ddply()* + *summarize()* Exercise II

```
1 library(plyr)
2 sumstats <- ddply(cereal, "shelf", summarize,
3                   intercept=coef(lm(calories ~fat))[1],
4                   slope      =coef(lm(calories ~fat))[2])
5 sumstats
```

```
> sumstats
  shelf intercept      slope
1     1 100.27778  0.3703704
2     2  96.78571 10.8333333
3     3  91.36752 11.7948718
```



## Split - Apply - Combine - *ddply()* + *summarize()* Exercise III

Fit a separate regression line between calories and fat and report the intercept and slope for each shelf.

Recall line for ALL of data is found as:

```
result <- lm(calories ~ fat, data=cereal)
summary(result)
coef(result)
coef(result)[1]
coef(result)[2]
```

```
> sumstats
  shelf intercept      slope
1      1 100.27778  0.3703704
2      2  96.78571 10.8333333
3      3  91.36752 11.7948718
```

Refer to *road-accidents-2010.csv* file in *SampleData*.

- Read data into *R*.
- Convert input date to internal *R* dates.
- Find number of accidents by day of year (use *ddply()* and *summarize()* in *plyr* package)
- Plot # accidents/day by day of year.
- Fit a *lowess()* smoother to data using *geom\_smooth()*

Look at number of accident by day of the week

- Extract day of the week using *format()* or *weekdays()* functions.
- Use *geom\_boxplot()* as seen earlier

## Split - Apply - Combine - *ddply()* + *summarize()* Exercise IV

```
1 # The accident data
2 accidents <- read.csv('../SampleData/road-accidents-2010.csv',
3                       header=TRUE,
4                       as.is=TRUE, strip.white=TRUE)
5 accidents[1:5,]
6 str(accidents)
```

```
> accidents[1:5,]
```

```
.....
```

	Accident_Severity	Number_of_Vehicles	Number_of_Casualties
1	3	2	1
2	3	1	1

```
> str(accidents)
```

```
'data.frame': 154414 obs. of 33 variables:
```

```
...
```

```
$ Date      : chr  "11/01/2010" "11/01/2010" "12/01/2010" "02/01/2010"
```

```
...
```

## Split - Apply - Combine - *ddply()* + *summarize()* Exercise IV

```
1
2 # Convert date to internal date format
3 accidents$mydate <- as.Date(accidents$Date,
4                             format="%d/%m/%Y")
5 sum(is.na(accidents$mydate))
6 accidents[1:5,]
7 str(accidents)

> accidents[1:5,]
...
  Urban_or_Rural_Area Did_Police_Officer_Attend_Scene_of_Acc
1                   1
2                   1
> str(accidents)
'data.frame': 154414 obs. of  33 variables:
 $ Date           : chr  "11/01/2010" "11/01/2010" "12/01/2010"
 $ mydate         : Date, format: "2010-01-11" "2010-01-11" "2010-01-11"
>
```

## Split - Apply - Combine - *ddply()* + *summarize()* Exercise IV

```
1 # Summarize number of accidents by date
2 library(plyr)
3 naccidents <- ddply(accidents, "mydate", summarize,
4                     freq=length(Accident_Index))
5 naccidents[1:5,]
6 str(naccidents)
```

```
> naccidents[1:5,]
      mydate freq
```

```
1 2010-01-01  282
2 2010-01-02  293
```

```
...
```

```
> str(naccidents)
```

```
'data.frame': 365 obs. of  2 variables:
```

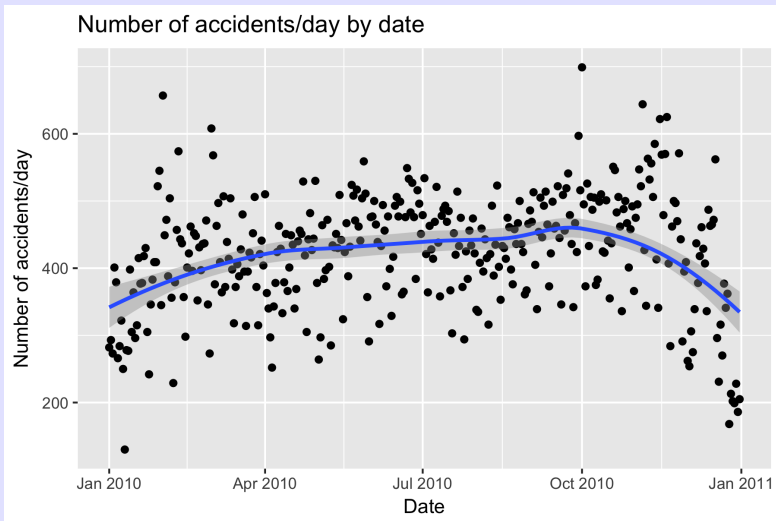
```
$ mydate: Date, format: "2010-01-01" "2010-01-02" "2010-01-
```

```
$ freq  : int  282 293 273 401 379 266 284 322 250 130 ...
```

## Split - Apply - Combine - *ddply()* + *summarize()* Exercise IV

```
1 plotnacc <- ggplot(data=naccidents, aes(x=mydate, y=freq))+  
2   ggtitle("Number of accidents/day by date")+  
3   xlab("Date")+ylab("Number of accidents/day")+  
4   geom_point()+  
5   geom_smooth()  
6 plotnacc
```

# Split - Apply - Combine - *ddply()* + *summarize()* Exercise IV



Refer to *road-accidents-2010.csv* file in *SampleData*.

- Create 0/1 variable if fatality occurs (no or yes; check codebook for *Accident\_Severity*).  
Use the magic incantation of *recode()* function in *car* package.
- Find proportion of accidents with fatality by day of year
  - The mean of a 0/1 variable is the proportion.  
Use the magic incantation of *ddply()* and *summarize()* in the *plyr* package.
- Plot proportion of fatalities by day of year.
- Fit a *lowess()* smoother to data from *geom\_smooth()*
- Plot proportion of fatalities by day of the week
  - Hint: Extract weekday using *format()*.
  - Hint: Use *geom\_boxplot()* as seen earlier with some jittering and notches.



# Split - Apply - Combine - *ddply()* + *summarize()* Exercise V

```
1 names(accidents)
2 unique(accidents$Accident_Severity)
3 library(car)
4 accidents$Fatality <- recode(accidents$Accident_Severity,
5                             ' 1=1; 2:hi=0')
6 accidents[1:5, c("Accident_Severity", "Fatality")]
7 xtabs(~Fatality + Accident_Severity, data=accidents)
```

```
> accidents[1:5, c("Accident_Severity", "Fatality")]
  Accident_Severity Fatality
1                   3       0
2                   3       0
```

```
> xtabs(~Fatality + Accident_Severity, data=accidents)
      Accident_Severity
Fatality    1      2      3
      0      0 20440 132243
      1 1731      0      0
```

## Split - Apply - Combine - *ddply()* + *summarize()* Exercise V

The *summarize()* and *ddply()* functions in *plyr* package is quite useful for simple summaries by groups.

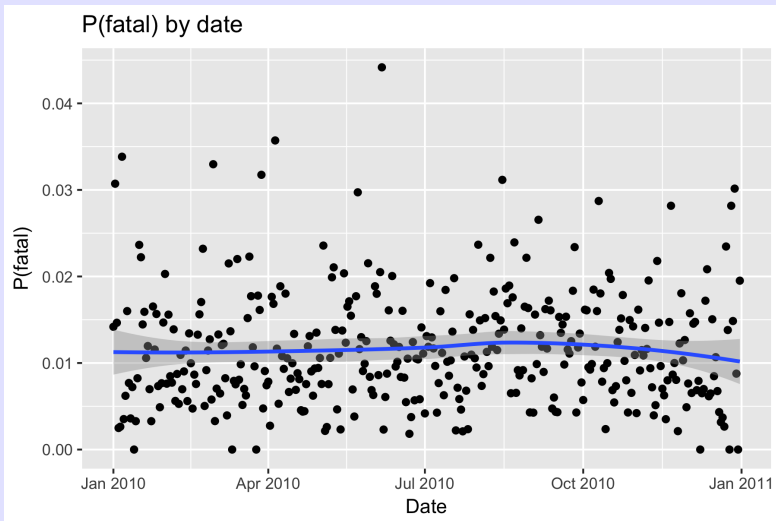
Example of the Split-Apply-Combine paradigm to be explained later.

```
1 library(plyr)
2 pfatal.df <- ddply(accidents, "mydate", summarize,
3                     freq=length(mydate),
4                     pfatal=mean(Fatality))
5 pfatal.df[1:5,]
```

```
> pfatal.df[1:5,]
      mydate freq    pfatal
1 2010-01-01  282 0.014184397
2 2010-01-02  293 0.030716724
3 2010-01-03  273 0.014652015
4 2010-01-04  401 0.002493766
5 2010-01-05  379 0.002638522
```

```
1 plotpfatal <- ggplot(data=pfatal.df,  
2                       aes(x=mydate, y=pfatal))+  
3       ggtitle("P(fatal) by date")+  
4       xlab("Date")+ylab("P(fatal)")+  
5       geom_point()+  
6       geom_smooth()  
7 plotpfatal
```

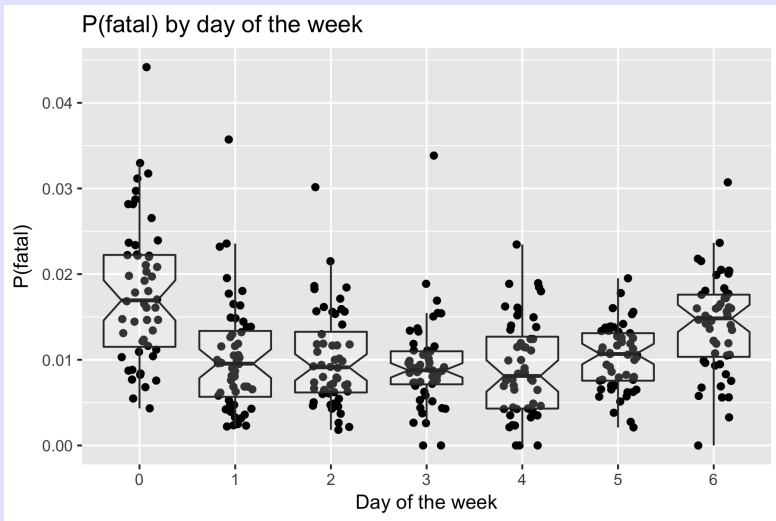
# Split - Apply - Combine - *ddply()* + *summarize()* Exercise V



## Split - Apply - Combine - *ddply()* + *summarize()* Exercise V

```
1  # Extract day of the week - leave as character
2  pfatal.df$weekday <- format(pfatal$mydate, format="%w") # l
3  pfatal.df[1:10,]
4
5  plotpfatal2 <- ggplot(data=pfatal.df, aes(x=weekday, y=pfatal
6    ggtitle("P(fatal) by day of the week")+
7    xlab("Day of the week")+ylab("P(fatal)")+
8    geom_point(position=position_jitter(w=0.2))+
9    geom_boxplot(notch=TRUE, alpha=0.2)
10 plotpfatal2
```

# Split - Apply - Combine - *ddply()* + *summarize()* Exercise V



## Split - Apply - Combine - *ddply()* + your own function

In many cases, a more complex function is applied to each chunk rather than simple summaries.

The chunk is passed as DATA.FRAME and traditionally called *x*.

```
ddply(dataframe, byvars, function(x)
  { # don't forget the opening brace
  func definition
  res1 <- ..
  res2 <- ...
  res <- data.frame(res1,res2,
                    stringsAsFactors=FALSE)
  return(res)
} # don't forget the closing brace
) # don't forget the closing )
```

## Split - Apply - Combine - *ddply()* + your own function

Fit a separate regression line for each shelf

```
1 library(plyr)
2 sumstats <- ddply(cereal, "shelf", function(x) {
3     result <- lm(calories ~ fat, data=x)    # notice use
4     intercept <- coef(result)[1]
5     slope <- coef(result)[2]
6     res <- data.frame(intercept, slope,
7                       stringsAsFactors=FALSE)
8     return(res)
9 })
10 sumstats
```

```
> sumstats
  shelf intercept      slope
1     1 100.27778  0.3703704
2     2  96.78571 10.8333333
3     3  91.36752 11.7948718
```



Refer back to the accidents dataset. For each day. compute

- Number of accidents
- Proportion of fatalities
- MEAN weather severity (*Weather\_Conditions*). Not really valid but a close approximation)
- Day of the week (0=Sunday)

Plot number of accident over the year with the SIZE of point related to mean weather conditions.

Add loess curve.

## Split - Apply - Combine - Exercise VI

```
1 accidents <- read.csv('../sampledata/road-accidents-2010.csv')
2               as.is=TRUE, strip.white=TRUE)
3 # Convert date to internal date format
4 accidents$mydate <- as.Date(accidents$Date,
5                             format="%d/%m/%Y")
6 # Create the fatality variable
7 accidents$Fatality <- accidents$Accident_Severity == 1
```

## Split - Apply - Combine - Exercise VI

Using *ddply()* and *summarize()*

```
1 naccidents <- ddply(accidents, "mydate", summarize,  
2                     freq=length(mydate),  
3                     pfatal=mean(Fatality),  
4                     mean.weather=mean(Weather_Conditions),  
5                     dow=format(mydate, "%w")[1])  
6 naccidents[1:5,]
```

```
> naccidents[1:5,]
```

	mydate	freq	pfatal	mean.weather	dow
1	2010-01-01	282	0.014184397	2.262411	5
2	2010-01-02	293	0.030716724	2.740614	6
3	2010-01-03	273	0.014652015	2.857143	0
4	2010-01-04	401	0.002493766	2.518703	1
5	2010-01-05	379	0.002638522	2.936675	2

## Split - Apply - Combine - Exercise VI

Using *ddply()* and an explicit function

```
1 naccidents <- ddply(accidents, "mydate", function(x){
2     freq <- nrow(x)
3     mean.weather <- mean(x$Weather_Conditions)
4     pfatal <- mean(x$Fatality)
5     dow=format(x$mydate, "%w")[1]
6     res <- data.frame(freq, mean.weather, pfatal,
7                       dow, stringsAsFactors=FALSE)
8     return(res)
9 })
10 naccidents[1:5,]
```

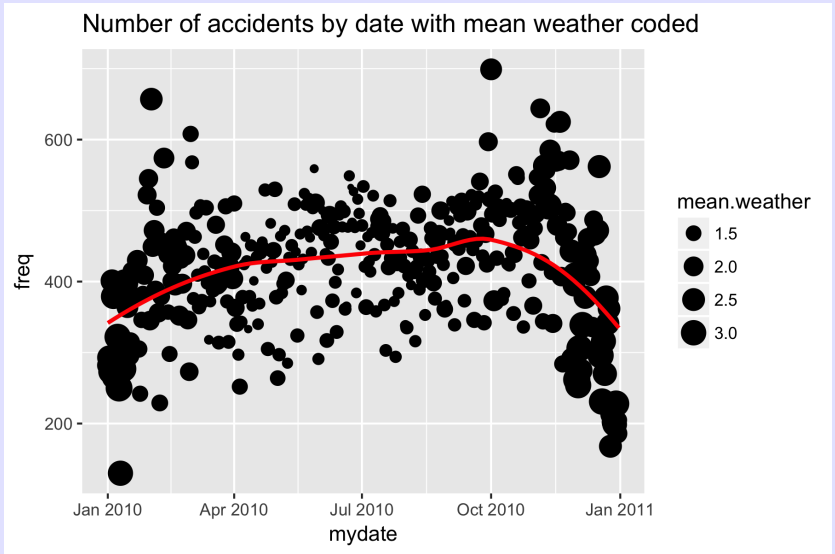
```
> naccidents[1:5,]
```

	mydate	freq	mean.weather	pfatal	dow
1	2010-01-01	282	2.262411	0.014184397	5
2	2010-01-02	293	2.740614	0.030716724	6
3	2010-01-03	273	2.857143	0.014652015	0
4	2010-01-04	401	2.518703	0.002493766	1
5	2010-01-05	379	2.936675	0.002638522	2

Make the plots

```
1 newplot <- ggplot(data=naccidents,  
2                   aes(x=mydate, y=freq ))+  
3   ggtitle("Number of accidents by date with mean weather coo  
4   geom_point( aes(size=mean.weather))+  
5   geom_smooth(method="loess", color="red", se=FALSE)  
6 newplot
```

# Split - Apply - Combine - Exercise VI



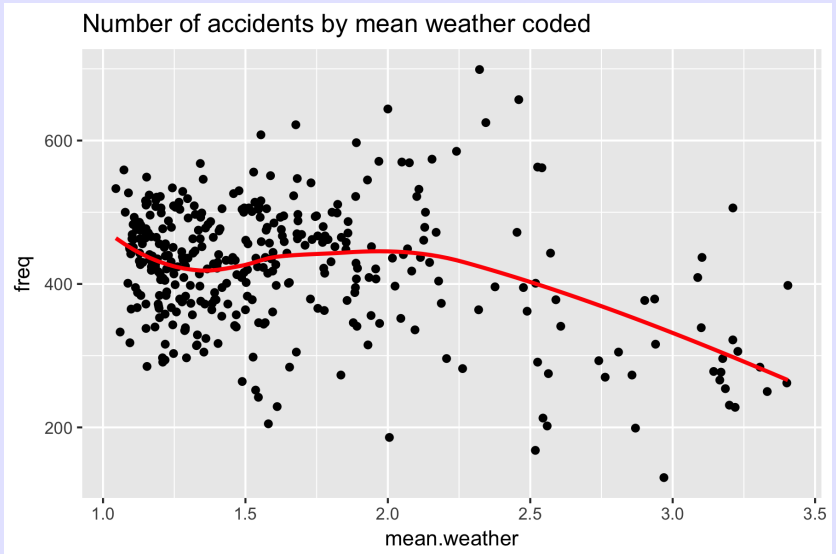
Plot number of accidents vs. mean weather conditions;  
Add loess curve

Plot number of accidents vs. mean weather conditions;

```
1 newplot <- ggplot(data=naccidents,  
2                   aes(x=mean.weather, y=freq))+  
3   geom_point( ) +  
4   geom_smooth(method="loess", color="red", se=FALSE) +  
5   ggtitle("Number of accidents by mean weather coded")  
6 newplot
```



# Split - Apply - Combine - Exercise VI



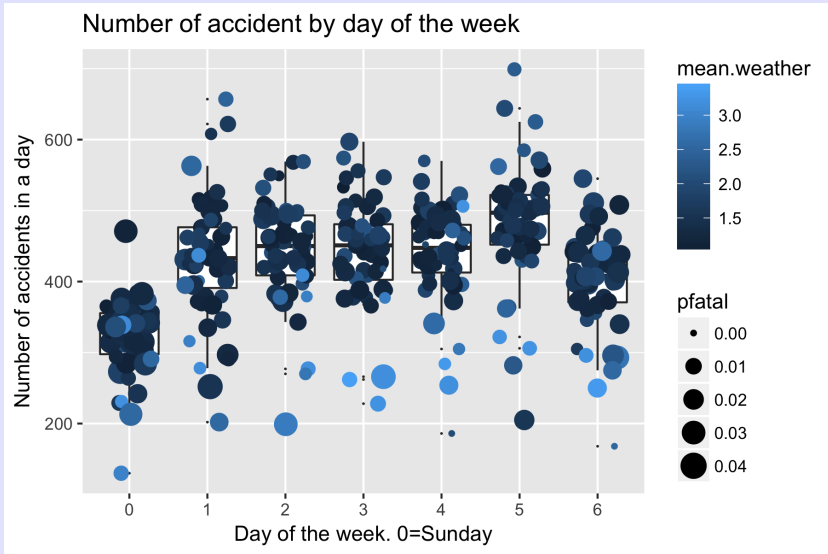
Accident data.

Make a box-plot of number of accident by day of the week coded using proportion of fatalities by the size of the symbol and the mean weather condition by a color gradient.

## Split - Apply - Combine - Exercise VII

```
1 newplot <- ggplot(data=naccidents, aes(x=dow, y=freq))+  
2   geom_boxplot( ) +  
3   geom_jitter(aes(size=pfatal, color=mean.weather),  
4               position=position_jitter(w=.3, h=.0))+  
5   ggtitle("Number of accident by day of the week")+  
6   xlab("Day of the week. 0=Sunday") +  
7   ylab("Number of accidents in a day")  
8 newplot
```

# Split - Apply - Combine - Exercise VII



## VERY COMMON PARADIGM IN R.

- Virtually unnecessary to use *for* loops in *R* if computations for each chunk are independent and do not depend on other chunks.
- Makes it easy to parallelize your work (routines are set up to use multiple machines)
- Most common usage is *ddply()*
- Next most common usage is *dlply()* followed by *ldply()* to extract.
- The *dplyr* package is specifically design for LARGE data frames and is much faster.

# Split - Apply - Combine - Summary (Simple)

Most simple usage is with *ddply()* and *summarize()*

```
1 sumstat <- ddply( dataframe, "chunking variable",
2                   summarize,
3                   v1=....,
4                   v2=....,
5                   v3=...., .... )
```

Sometimes it is more convenient write your own function

```
1 sumstat <- ddply( dataframe, "chunking variable",
2                   function(x){
3                       res1 <- function of x$...
4                       res2 <- function of x$...
5                       res <- data.frame(res1, res2, ...,
6                                         stringsAsFactors=FALSE)
7                       return(res)
8                   })
```

Passing additional arguments to *ddply()* functions.

Sometimes you need to pass additional variables other than the chunk to be processed.

Example: Refer back to the cereal dataset. For each shelf group (and for an “arbitrary” variable), compute

- Number of observations
- Number of observations with missing values
- Mean of the variable
- SD of the variable

# Split - Apply - Combine - Advanced

```
1 sumstat <- function(x, var){
2   # Compute some summary statistics for a data frame
3   values <- x[,var] # extract the variable values
4   n <- length(values)
5   nmiss <- sum(is.na(values))
6   mean <- mean(values, na.rm=TRUE)
7   sd <- sd(values, na.rm=TRUE)
8   res <- data.frame(n,nmiss,mean,sd,
9                     stringsAsFactors=FALSE)
10  return(res)
11 } # end of sumstat
12
13
14 sumstat(cereal, "calories")
15 sumstat(cereal, "weight")
```



# Split - Apply - Combine - Advanced

But how is the second (and additional arguments passed to *ddply()*)?

```
1 res <- ddply( dataframe, "chunking variable",
2               functionname,
3               y=xxx, z=xxxx)
4
5 res <- ddply( dataframe, "chunking variable",
6               function(x, y, z){
7                 res1 <- function of $x$, $y$, and $z
8                 res2 <- function of $x$, $y$, and $z
9                 res <- data.frame(res1, res2, ...,
10                                stringsAsFactors=FALSE)
11                 return(res)
12               }, y=xxx, z=xxxx)
13
14 ddply(cereal, "shelf", sumstat, var="calories")
15 ddply(cereal, "shelf", sumstat, var="weight")
```

Using *dply()* and *ldply()*

- It is convenient to do ALL computations for a chunk, return a list, and then extract from the list a needed rather than having several different function.
- Some output (like plots) cannot be stored in data frames.

Example: Refer back to the cereal dataset. For each shelf group (and for two “arbitrary” variable), compute

- Plot of  $Y$  vs.  $X$  (use *aes\_string()* in *ggplot()*)
- Regression of  $Y$  on  $Y$ . Use *lm( x[, Yvar] ~ x[, Xvar]*)
- Return both in a list

```
sumstat(cereal, "calories", "fat")
```

should return a list with 2 elements.

Using *dply()* and *ldply()*

```
1 sumstat <- function(x, Yvar, Xvar){
2   # Do the plot (use aes_string)
3   plot <- ggplot(data=x, aes_string(x=Xvar, y=Yvar))+
4     ggtitle(paste("Scatterplot of ", Yvar, " vs. ", Xvar, " ")) +
5     geom_point(position=position_jitter(h=.1, w=.1))+
6     geom_smooth(method="lm", se=FALSE)
7   fit <- lm( x[,Yvar] ~ x[, Xvar], data=x)
8   list(plot=plot, fit=fit)
9 }
10
11 res<- sumstat(cereal, "calories", "fat")
12 length(res)
```

Using *dlply()* and *ldply()*

- Now use *dlply()* to make a list of lists, once for each shelf

```
1 res <- plyr::dlply(cereal, "shelf", sumstat,  
2                   Yvar="calories", Xvar="fat")  
3 length(res)  
4 res[[1]]  
5 res[[2]]  
6 res[[3]]
```

Using *dply()* and *ldply()*

- Now use *ldply()* to make a data frame of slope and se

```
1 se.summary <- plyr::ldply(res, function(x){
2     # x is now the list of the flot and the fit
3     slope <- summary(x$fit)$coefficients[2,]
4     slope
5 })
6 se.summary
```

	shelf	Estimate	Std. Error	t value	Pr(> t )
1	1	0.3703704	3.581444	0.1034137	0.9187781230
2	2	10.8333333	2.626300	4.1249412	0.0005759948
3	3	11.7948718	3.698559	3.1890452	0.0030617682

Using *dply()* and *ldply()*

- Now use *ldply()* to extract the plots

```
1 plyr::l_ply(res, function(x){  
2   plot(x$plot) # needed because within a function  
3 })
```

- Notice use of *l\_ply()* because no output returned.
- Notice use of *plot()* WITHIN function to force display.
- All plots are sent to the plot window in *Rstudio*.
- It is possible to send the plots to a pdf file (see code).

## Split - Apply - Combine - Advanced

Row or Column operations on a MATRIX or ARRAY (less common)  
Note MATRIX differs from a data.frame because all values must have same type.

ARRAY is a 3+ dimensional object.

```
1 mat <- matrix(1:30, nrow=6)
2 mat
```

```
> mat
```

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	1	7	13	19	25
[2,]	2	8	14	20	26
[3,]	3	9	15	21	27
[4,]	4	10	16	22	28
[5,]	5	11	17	23	29
[6,]	6	12	18	24	30

# Split - Apply - Combine - Advanced

Row or Column operations on a MATRIX or ARRAY.

```
1 aapply(mat, 1, sum)
2 aapply(mat, 2, mean)
3 aapply(mat, 1, function(x){
4   res <- prod(sin(x))
5   return(res)
6 })
```

```
> aapply(mat, 1, sum)
```

```
  1  2  3  4  5  6
65 70 75 80 85 90
```

```
> aapply(mat, 2, mean)
```

```
  1    2    3    4    5
3.5  9.5 15.5 21.5 27.5
```

```
> aapply(mat, 1, function(x)....
```

```
                1                2                3                4
-0.0046076865  0.6204102446  0.0302615870  0.0002842306 -0.5
```



Refer back to the accident dataset.

- For each month, compute the number of days, weekdays and weekends. Hint: Use the *unique()* function on the dates within each month. Why do I want all three values?
- For each month, compute the total number of accidents with injury, those on weekends, and those on weekdays. Again, why do I want all three values?
- For each month, find the ratio of the number of accidents on weekday to weekends.
- Plot these over the year
- Add a suitable comparison line if accidents were uniformly spread over the days of the week. Note that the number of weekends and weekdays differs among months.

```
1 ... read accident data ....
2 ... convert dates to internal R format ...
3
4 # get the month for each accident date
5 accidents$month <- as.numeric(format(
6     accidents$mydate, "%m"))
```

## Split - Apply - Combine - Advanced - Exercise VIII

```
1  mysummary <- function(accidents){
2    # Compute the number of weekend and weekdays in the month (
3    # Compute the number of accidents on weekend/weekdays
4    # Report the two ratio.
5    DaysOfMonth <- unique(accidents$mydate)
6    DaysOfWeeks <- format(DaysOfMonth, "%w")
7    nDays      <- length(DaysOfMonth)
8    nWeekdays <- sum(DaysOfWeeks %in% 1:5)
9    nWeekends  <- sum(DaysOfWeeks %in% c(0,6))
10
11    AccDaysOfWeek <- format(accidents$mydate, "%w")
12    nAccTotal     <- length(accidents$mydate)
13    nAccWeekdays <- sum(AccDaysOfWeek %in% 1:5)
14    nAccWeekends  <- sum(AccDaysOfWeek %in% c(0,6))
15
16    rAccWdWe <- nAccWeekdays/nAccWeekends
17    rDaysWdWe <- nWeekdays/nWeekends
18
19    res <- data.frame(nDays,
20                      nWeekdays,
```

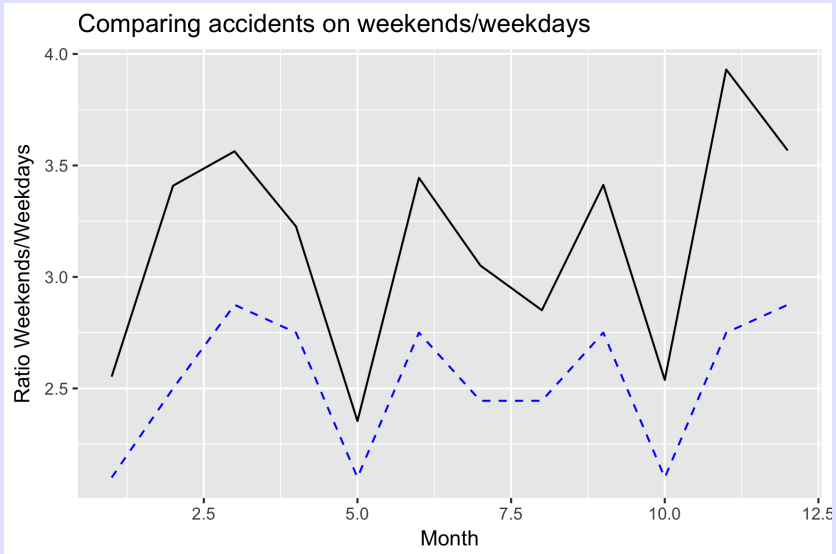
```
1 testdata <- subset(accidents, accidents$month == 1)
2 dim(testdata)
3
4 mysummary(testdata)
5
6 > mysummary(testdata)
7      nDays      nWeekdays    nWeekends      nAccTotal nAccWeek
8      31.000000      21.000000      10.000000 10637.000000  7643.00
```

# Split - Apply - Combine - Advanced - Exercise VIII

```
1 results <- ddply(accidents, "month", mysummary)
2 results
3
4      month nDays nWeekdays nWeekends nAccTotal nAccWeekdays n
5 1         1    31         21         10     10637         7643
6 2         2    28         20          8     11724         9065
7 3         3    31         23          8     13165        10280
8 4         4    30         22          8     12248         9350
9 5         5    31         21         10     13220         9278
10 6         6    30         22          8     13644        10574
11 7         7    31         22          9     13527        10188
12 8         8    31         22          9     13027         9644
13 9         9    30         22          8     13904        10753
14 10        10    31         21         10     14429        10351
15 11        11    30         22          8     14544        11594
16 12        12    31         23          8     10345         8080
```

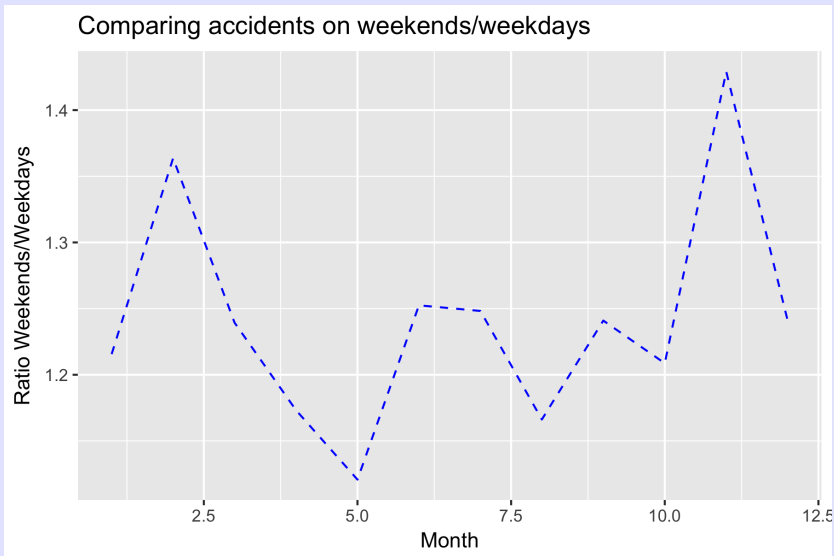
```
1 newplot <- ggplot(data=results, aes(x=month, y=rDaysWdWe))+  
2   ggtitle("Comparing accidens on weekends/weekdays")+  
3   xlab("Month")+ylab("Ratio Weekends/Weekdays")+  
4   geom_line(group=1, color="blue", linetype=2)+  
5   geom_line(aes(y=rAccWdWe,group=1))  
6 newplot
```

# Split - Apply - Combine - Advanced - Exercise VIII



```
1 newplot <- ggplot(data=results, aes(x=month, y=rAccWdWe/rDay
2   ggtitle("Comparing accidens on weekends/weekdays")+
3   xlab("Month")+ylab("Ratio Weekends/Weekdays")+
4   geom_line(group=1, color="blue", linetype=2)
5 newplot
```





General steps for the Split-Apply-Combine paradigm.

- What form is data in? Array, Dataframe, List?
- What form should results be in? Array, Dataframe, List, NULL (for plots)
- Create function to do the application; test with a subset of the data;

```
1 myfunction <- function() {} # define your function
2 testdata <- subset( mydata,
3                     mydata$variable == testvalue)
4 myfunction(testdata)
```

- Run the member of the *plyr* package.

The *lm()* and *glm()* functions.

## R standard models - *lm()* and *glm()*

Regression/ANOVA in *R* is done using the *lm()* function.

Logistic regression/ANOVA is done using the *glm()* function.

<i>Y</i> <i>Continuous</i>	Regular fegression	Regular ANOVA	Regular ANCOVA	<i>lm()</i>
<i>Y</i> <i>Categorical</i>	Logistic regression	Logistic ANOVA	Logistic ANCOVA	<i>glm()</i>
	<i>X</i> Continuous	<i>X</i> Categorical	<i>X</i> Mixed	

## R standard models - *lm()* regression

Fit a line to the relationship between Calories and Grams of Fat in the cereal dataset.

```
1 result <- lm( calories ~ fat, data=cereal)
2 str(result)  # Yikes!
3 names(result)
```

Use specialized functions (called METHODS) to extract information from model objects. Look at help pages

```
1 summary(result)
2 anova(result) # Caution Type I only
3 coef(result)
4 sqrt(diag(vcov(result))) # SE of coefficients
5 confint(result)
6 summary(result)$r.squared
7 summary(result)$sigma
8 predict(result, new data) # predictions at new $X$ values
9
10 methods(class=class(result))  # shows methods available
```

## R standard models - *lm()* regression - CAUTION!

The *anova()* gives Type I (incremental) tests, but you want Type III (marginal) tests in regression.

Need to look at output from *summary()* method in more complex cases for continuous *X* variables. (Do not use output in summary table for categorical *X* variables).

```
1  # Compare the following results
2  result <- lm(calories ~ fat, data=cereal)
3  anova(result)
4  summary(result)
5
6  result2 <- lm(calories ~ fat + protein, data=cereal)
7  anova(result2)
8  summary(result2)
9
10 result3 <- lm(calories ~ protein + fat, data=cereal)
11 anova(result3)
12 summary(result3)
```

Use the `car::Anova(fitobject, type=3)` to get the marginal tests that are invariant to term order.

```
1 car::Anova(cereal.fit2, type=3)
2 car::Anova(cereal.fit3, type=3)
```

## R standard models - *lm()* regression - CAUTION!

```
> car::Anova(cereal.fit2, type=3)
```

```
Anova Table (Type III tests)
```

	Sum Sq	Df	F value	Pr(>F)	
(Intercept)	110850	1	293.8425	< 2.2e-16	***
fat	7593	1	20.1285	2.604e-05	***
protein	206	1	0.5453	0.4626	
Residuals	27916	74			

```
> car::Anova(cereal.fit3, type=3)
```

```
Anova Table (Type III tests)
```

	Sum Sq	Df	F value	Pr(>F)	
(Intercept)	110850	1	293.8425	< 2.2e-16	***
protein	206	1	0.5453	0.4626	
fat	7593	1	20.1285	2.604e-05	***
Residuals	27916	74			



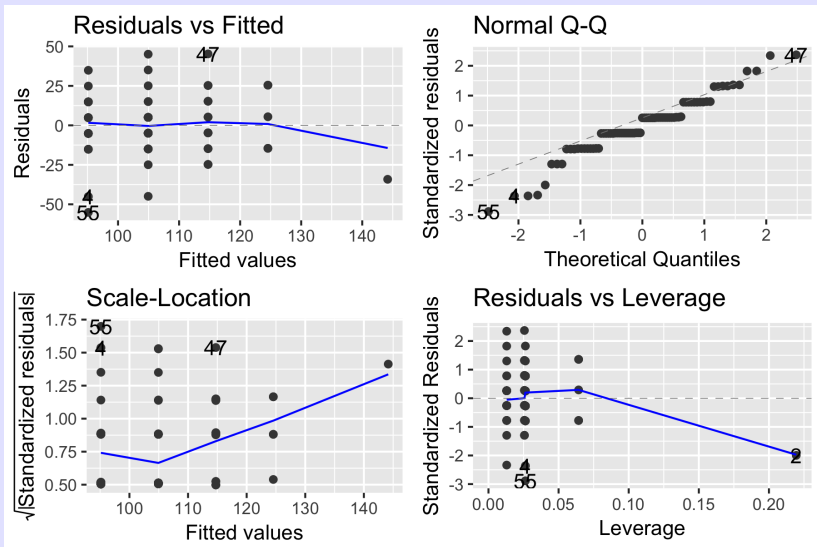
# R standard models - *lm()* regression

## Diagnostic plots

```
1 layout(matrix(1:4,2,2))
2 plot(cereal.fit)  # standard R graphics AVOID
3 layout(1)
4
5 library(ggfortify)
6 autoplot(cereal.fit)
7
8 # Residual plots from the car() package
9 library(car)
10 residualPlots(cereal.fit)
```

- Residual plots should have random scatter around 0 with constant spread
- Normal QQ plot should have points close to 45-degree line
- Scale-location should be flat over time
- Leverage - should be flat (not useful with 1 *X* variable)

# R standard models - $lm()$ regression



### Making predictions

- Predictions and confidence intervals for MEAN response at new  $X$ .
- Predictions and prediction intervals for INDIVIDUAL response at new  $X$ .

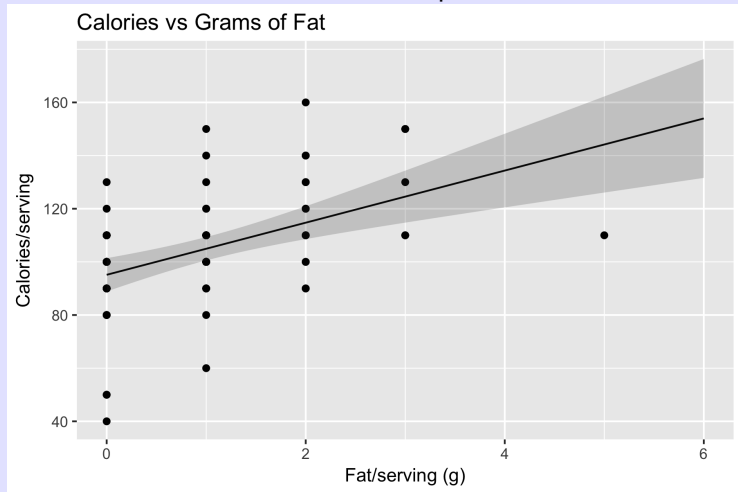
## R standard models - *lm()* regression

Confidence intervals for the MEAN response.

```
1  # First set up the points where you want predictions
2  newfat <- data.frame(fat=seq(0,6,.1))
3  newfat[1:5,]
4  str(newfat)
5
6  predict.avg <- predict(cereal.fit, newdata=newfat,
7                        se.fit=TRUE,interval="confidence")
8  # This creates a list that you need to restructure to make
9  predict.avg.df <- cbind(newfat, predict.avg$fit,
10                         se=predict.avg$se.fit)
11  tail(predict.avg.df)
12
13 # Add the line and confidence intervals to the plot
14 plotfit.avgci <- newplot +
15   geom_ribbon(data=predict.avg.df,
16             aes(x=fat,y=NULL, ymin=lwr, ymax=upr),alpha=0.2)
17 plotfit.avgci
```

# R standard models - $lm()$ regression

Confidence limits for the MEAN response.



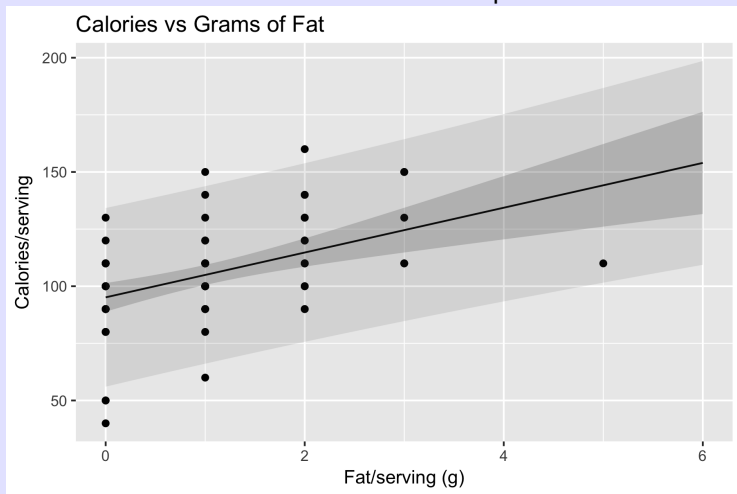
## R standard models - *lm()* regression

Prediction intervals for the INDIVIDUAL response.

```
1 predict.indiv <- predict(cereal.fit, newdata=newfat,  
2                           interval="prediction")  
3 # This creates a list that you need to restructure to make  
4 predict.indiv.df <- cbind(newfat, predict.indiv)  
5 tail(predict.indiv.df)  
6  
7 # Add the prediction intervals to the plot  
8 plotfit.indivci <- plotfit.avgci +  
9     geom_ribbon(data=predict.indiv.df,  
10                aes(x=fat,y=NULL, ymin=lwr, ymax=upr),  
11                alpha=0.1)  
12 plotfit.indivci
```

# R standard models - $lm()$ regression

Prediction intervals for INDIVIDUAL response.

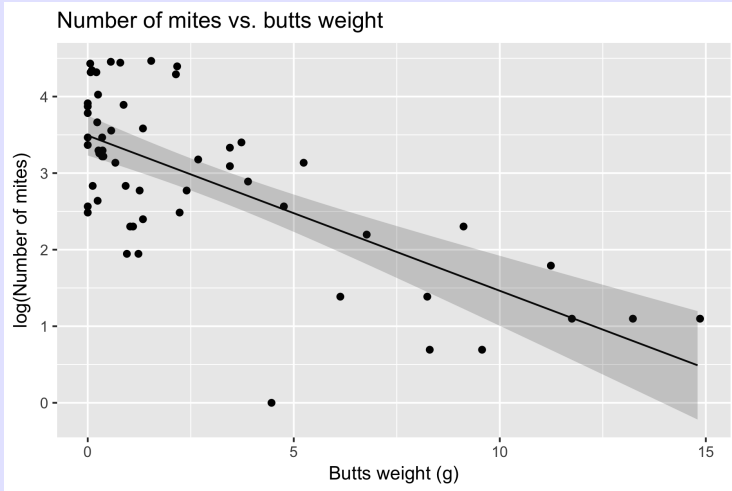


Return to Birds 'n Butts.

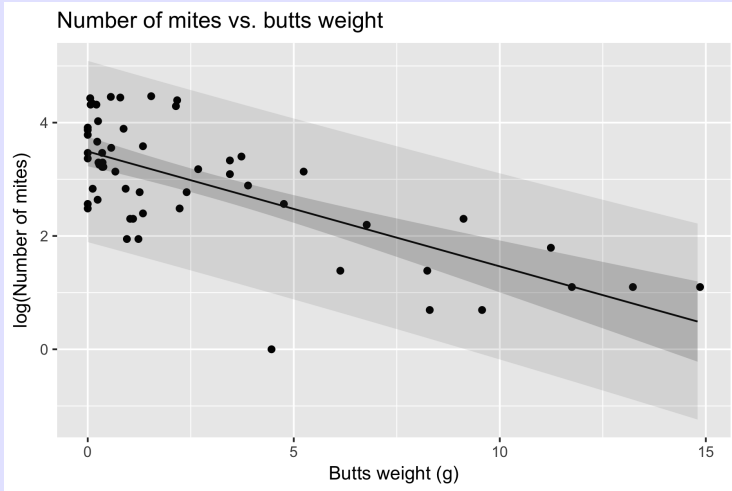
- Fit the relationship on the  $\log()$  scale (include all points).
- Predict the MEAN number of  $\log(\text{mites})$  and find ci
- Predict the INDIVIDUAL number of  $\log(\text{mites})$  and find pi
- Back transform and plot on the original scale



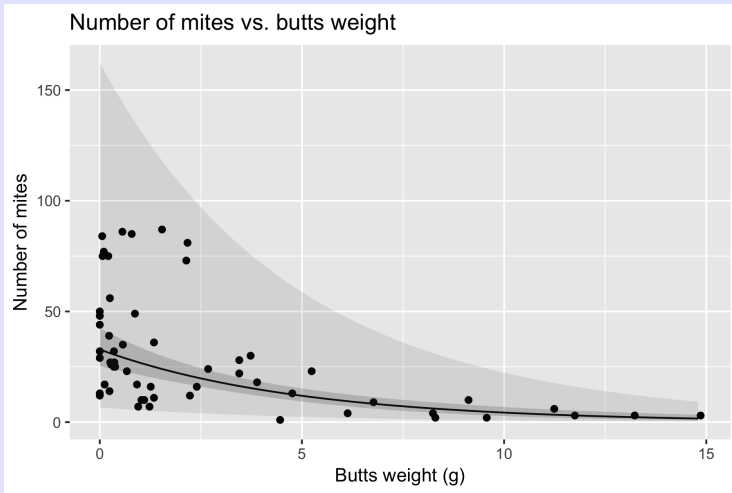
# R standard models - $\text{lm}()$ regression



# R standard models - $\text{lm}()$ regression



# R standard models - $\text{lm}()$ regression



# R Simple Regression - Summary

- Check that your design is suitable for regression
  - Linear relationship between  $Y$  and  $X$  (plot the data).
  - No outliers/influential points (plot the data).
  - Constant SD about line (plot the data).
  - Independent observations.
  - $X$  measured without/small error.
- $H : \beta_1 = 0$
- Plot the model diagnostics
- Make predictions.
  - Predict mean and CI for MEAN response
  - Predict individual and PI for INDIVIDUAL responses.

See <http://www.stat.sfu.ca/~cschwarz/CourseNotes> for much more details

# R standard models - *lm()* CRD ANOVA

Is there a relationship between MEAN amount of sugars/serving and display shelf?

CRD implies that

- Single factor with 2+ levels (e.g. Display shelf (1, 2, 3); Species (HOFI, HOSI))
- Complete randomization of treatments (levels) to experimental units; or random sample from relevant population of each level.
- Observational unit = Experimental unit. One measurement per e.u.
  - Avoid pseudo-replication - see my CourseNotes
- $H: \mu_1 = \mu_2 = \mu_3 \dots = \mu_k$
- Two standard analyses:
  - $k = 2$  Two-sample t-test - equal and unequal variance between two groups
  - $k > 2$  Single-factor CRD ANOVA - need to assume equal variance in all groups

CAUTION: NOT ALL EXPERIMENTS are CRD ANOVAs!

Seek advice before analyzing complex designs.

RECOMMEND that categorical variables be ALWAYS coded as alpha numeric values to avoid potential problems in modeling.

Categorical variables MUST be declared as FACTORS in *R*.

I often create a new variable for the factor representation, esp. if levels are ordered.

FACTORS are INDEXES into list of levels – look at `str(cereal)` for details.

Is there a relationship between MEAN sugar/serving and display shelf?

Categorical variables MUST be declared as FACTORS in *R*.

I often create a new variable for the factor representation, esp. if levels are ordered.

FACTORS are INDEXES into list of levels – look at `str(cereal)` for details.

```
1 cereal$shelfF <- factor(cereal$shelf)
2 cereal[1:3,]
3 str(cereal)
```

## R standard models - *lm()* CRD ANOVA

FACTORS are INDEXES into list of levels – look at `str(cereal)` for details.

```
> cereal[1:3,]
```

	name	mfr	type	calories	protein	fat	sodium	fib
1	100%_Bran	N	C	60	4	1	130	
2	100%_Natural_Bran	Q	C	110	3	5	15	
3	All-Bran	K	C	80	4	1	260	

```
> str(cereal)
```

```
'data.frame': 77 obs. of 16 variables:
```

```
$ name      : chr  "100%_Bran" "100%_Natural_Bran" "All-Bran"
```

```
$ shelf     : int   3 3 3 3 3 1 2 3 1 3 ...
```

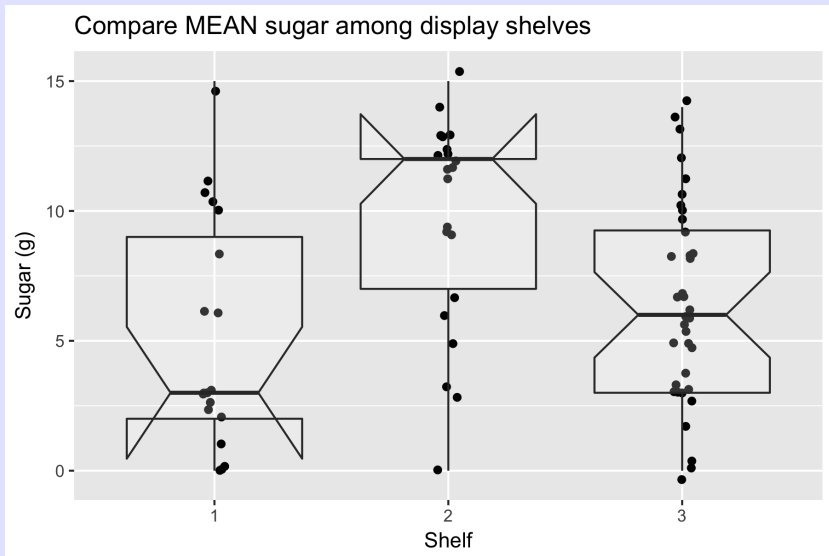
```
$ shelfF    : Factor w/ 3 levels "1","2","3": 3 3 3 3 3 1 2 3
```



Cereal data - difference in MEAN sugar/serving among different display shelves?

- Check data (box plots) for outliers; missing data?
- Preliminary table of means and sd's.
  - Check that SD's are approximately equal for each group
  - Look at sample sizes of each group
- Declare variable as a FACTOR
- Overall test using *lm()*; but doesn't tell you where differences lie
- Multiple-comparisons (Tukey procedure in the *lsmeans()* functions
- Display the results

# R standard models - $lm()$ CRD ANOVA



# R standard models - *lm()* CRD ANOVA

## Preliminary summary statistics

```
1 library(plyr)
2 sumstat <- ddply(cereal, "shelfF", sf.simple.summary,
3                 variable="sugarss", crd=TRUE)
4 sumstat
```

```
> sumstat
```

	shelfF	n	nmiss	mean	sd	se	lcl	ucl
1	1	20	1	5.105263	4.483237	1.0285251	2.944412	7.266114
2	2	21	0	9.619048	4.128876	0.9009947	7.739606	11.49849
3	3	36	0	6.527778	3.835817	0.6393028	5.229924	7.825632

## R standard models - *lm()* CRD ANOVA

Is there a relationship between MEAN sugar/serving and display shelf?

Again use the *lm()* function to fit the model.

```
1 shelf.fit <- lm(sugars ~ shelfF, data=cereal)
2 str(shelf.fit)
3 anova(shelf.fit)
4 summary(shelf.fit) # Not useful
```

CAUTION. *anova()* gives Type I (incremental) and not marginal tests. This is OK for a single factor CRD as then both are the same. DO NOT USE output from *summary()* for categorical variables.

- Estimates depend on internal contrast matrix used by *R*.
- Estimates depend on order of factor levels.

	Estimate	Std. Error	t value	Pr(> t )	
(Intercept)	5.105	0.937	5.449	6.55e-07	***
shelfF2	4.514	1.293	3.490	0.000822	***
shelfF3	1.423	1.158	1.228	0.223293	

Is there a relationship between MEAN sugar/serving and display shelf?

Use the *lsmeans()* function from the *lsmeans* package.

Note that the *lsmeans()* is applied to model object.

Because of naming conflicts, use *lsmeans::lsmeans()*

- Create least squares mean object (*lsmo*) first
- Use *cld()*, *pairs()*, *summary()* to investigate

```
1 # use the lsmeans() package to get the individual
2 # means and the pairwise comparisons and plot them
3 library(lsmeans)
4 shelf.fit.lsmo <- lsmeans::lsmeans(shelf.fit, ~shelfF)
```

## R standard models - *lm()* CRD ANOVA

Is there a relationship between MEAN sugar/serving and display shelf?

Obtain the compact-letter-display (cld)

```
1 # Where do difference in marginal means lie?  
2 shelf.fit.cld <- cld(shelf.fit.lsmo)  
3 shelf.fit.cld
```

```
> shelf.fit.cld
```

	shelfF	lsmean	SE	df	lower.CL	upper.CL	.group
1		5.105263	0.9369889	73	3.237847	6.972679	1
3		6.527778	0.6807066	73	5.171131	7.884424	1
2		9.619048	0.8912542	73	7.842781	11.395315	2

Confidence level used: 0.95

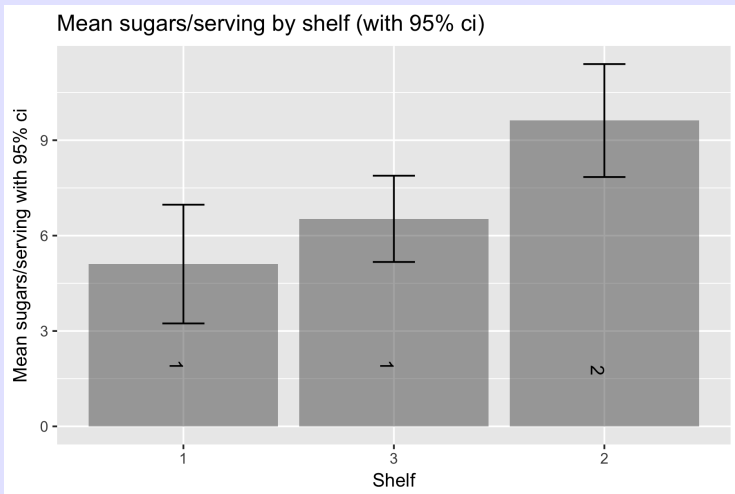
P value adjustment: Tukey method for a family of 3 means  
significance level used: alpha = 0.05

Is there a relationship between MEAN sugar/serving and display shelf?

Plot the results from the cld using ggplot() or my functions.

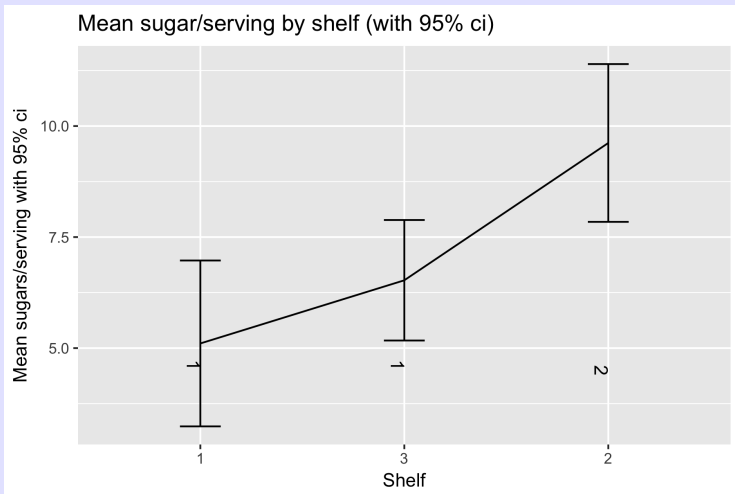
```
1 sf.cld.plot.bar(shelf.fit.cld, "shelfF")+  
2   ggtitle("Mean sugar/serving by display shelf (with 95% ci)"  
3   xlab("Shelf")+ylab("Mean sugarss/serving with 95% ci")  
4  
5 sf.cld.plot.line(shelf.fit.cld, "shelfF")+  
6   ggtitle("Mean sugar/serving by display shelf (with 95% ci)"  
7   xlab("Shelf")+ylab("Mean sugarss/serving with 95% ci")
```

# R standard models - $lm()$ CRD ANOVA





# R standard models - *lm()* CRD ANOVA



Is there a relationship between MEAN sugar/serving and display shelf?

Obtain the pairwise differences

```
1 # Estimate all of the pairwise differences
2 pairs(shelf.fit.lsmo)
3 confint(pairs(shelf.fit.lsmo))
4
5 summary(pairs(shelf.fit.lsmo), infer=TRUE)
```

## R standard models - *lm()* CRD ANOVA

Is there a relationship between MEAN sugar/serving and display shelf?

Obtain the pairwise differences

```
> pairs(shelf.fit.lsmo)
```

contrast	estimate	SE	df	t.ratio	p.value
1 - 2	-4.513784	1.293167	73	-3.490	0.0023
1 - 3	-1.422515	1.158149	73	-1.228	0.4405
2 - 3	3.091270	1.121470	73	2.756	0.0199

```
> confint(pairs(shelf.fit.lsmo))
```

contrast	estimate	SE	df	lower.CL	upper.CL
1 - 2	-4.513784	1.293167	73	-7.6076039	-1.419965
1 - 3	-1.422515	1.158149	73	-4.1933116	1.348282

```
> summary(pairs(shelf.fit.lsmo), infer=TRUE)
```

contrast	estimate	SE	df	lower.CL	upper.CL	t.ratio
1 - 2	-4.513784	1.293167	73	-7.6076039	-1.419965	-3.490
1 - 3	-1.422515	1.158149	73	-4.1933116	1.348282	-1.228

Return to Birds n' Butts.

Is there a difference in MEAN butts weight between the 3 nest contents?

Is there a difference in MEAN butts weight between the 3 nest contents?

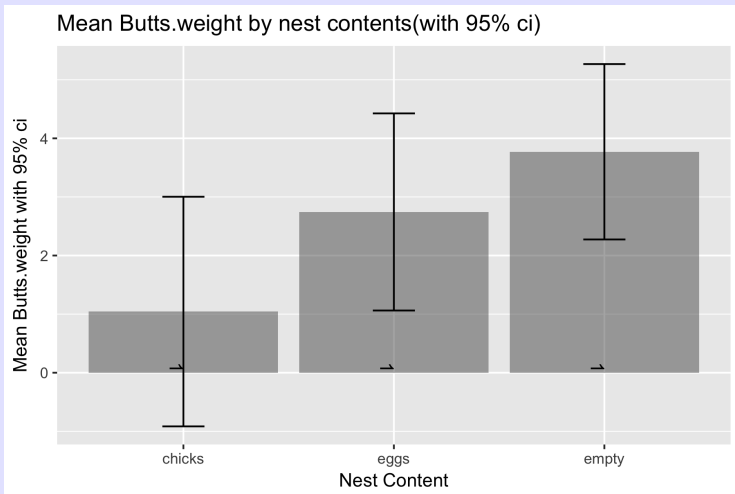
```
> nc.fit.cld
```

Nest.contentF	lsmean	SE	df	lower.CL	upper.CL	.group
chicks	1.043571	0.977	54	-0.9159789	3.003122	1
eggs	2.743684	0.838	54	1.0616159	4.425753	1
empty	3.770833	0.746	54	2.2742021	5.267465	1

Confidence level used: 0.95

P value adjustment: tukey method for a family of 3 means  
significance level used: alpha = 0.05

# R standard models - *lm()* CRD ANOVA



# R Single Factor CRD ANOVA - Summary

- Check that your design is a single-factor CRD
  - Single factor with at least 2 levels.
  - Complete randomization of treatments to experimental units.
  - Experimental Unit = Observational Unit (no pseudo-replication)
- $H : \mu_1 = \mu_2 = \dots = \mu_k$
- Preliminary plots - check for outliers
- Preliminary summary table - check SD's are about equal
- Declare a FACTOR variable
- Use `my.fit <- lm(Y ~ GroupF)` to fit model
- Use `anova(my.fit)` for overall comparison. CAUTION about Type I tests
- Use `my.lsmo <- lsmeans::lsmeans(my.fit, ~ GroupF)` to get marginal means etc
- Plot the results.

See <http://www.stat.sfu.ca/~cschwarz/CourseNotes> for much more details

## R standard models - *lm()* CRD ANCOVA

Is the relationship between calories and fat the same for all display shelves?

Still a CRD which implies

- Single factor with 2+ levels (e.g. Display shelf (1, 2, 3))
- Single continuous variable as a factor (e.g. fat).
- Complete randomization of treatments (levels) to experimental units; or random sample from relevant population of each level.
- Observational unit = Experimental unit. One measurement per e.u.
  - Avoid pseudo-replication - see my CourseNotes
- Three possible models:
  - Different slope and intercept in all groups (non-parallel slope model)
  - Parallel slope but different intercepts across groups (parallel slope model)
  - Same line for all groups (co-incident line model)

CAUTION: NOT ALL EXPERIMENTS are CRDs!

Seek advice before analyzing complex designs.



Analysis requires a grouping variable (*A*) and a continuous variable (*X*)

Three models are:

- Non-parallel slope model:  $Y \sim X + A + X : A$
- Parallel slope model  $Y \sim X + A$
- Co-incident line model  $Y \sim X$

Cautions in using *R*

- Declare grouping variable as factor.
- Adjust call to *lm()* to obtain MARGINAL tests
- Use *lsmeans* package for multiple comparisons

RECOMMEND that categorical variables be ALWAYS coded as alpha numeric values to avoid potential problems in modeling.

Categorical variables MUST be declared as FACTORS in R.

I often create a new variable for the factor representation, esp. if levels are ordered.

FACTORS are INDEXES into list of levels – look at `str(cereal)` for details.

Is the relationship between calories and fat the same for all display shelves?

Why might it be different?

Declare shelf as a factor in the usual way.

```
1 cereal$shelfF <- factor(cereal$shelf)
2 cereal[1:3,]
3 str(cereal)
```

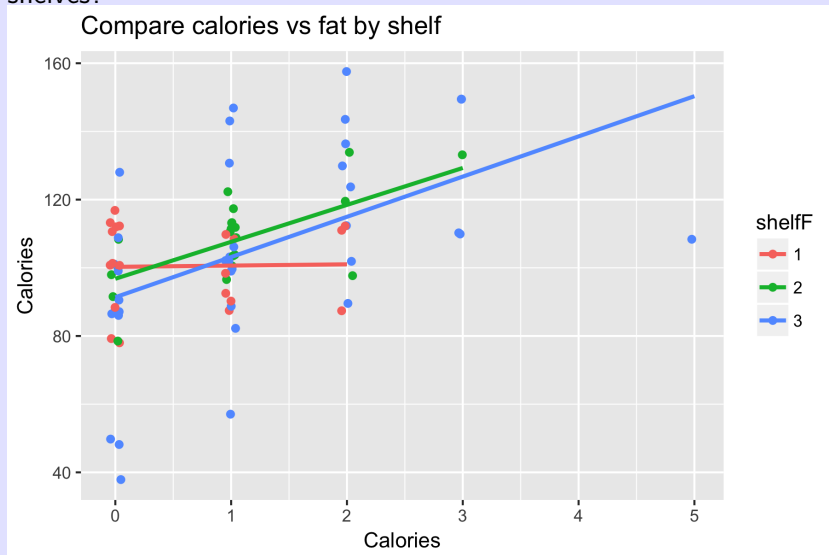
Is the relationship between calories and fat the same for all display shelves?

Preliminary plot

```
1 preimplot <- ggplot(data=cereal,  
2       aes(x=fat, y=calories, color=shelfF))+  
3       ggtitle("Compare calories vs fat by shelf")+  
4       xlab("Calories")+ylab("Calories")+  
5       geom_point(position=position_jitter(width=0.05))+  
6       geom_smooth(method="lm", se=FALSE)  
7 preimplot
```

# R standard models - $lm()$ CRD ANCOVA

Is the relationship between calories and fat the same for all display shelves?



## R standard models - *lm()* CRD ANCOVA

Is the relationship between calories and fat the same for all display shelves?

Non-parallel slope model:

```
1 shelf.fit <- lm(calories ~ shelfF + fat + shelfF:fat,  
2                 data=cereal,  
3                 contrasts=list(shelfF=contr.sum))  
4 anova(shelf.fit) # CAUTION - Type I (incremental)  
5 car::Anova(shelf.fit, type=3) # marginal tests
```

CAUTION. *anova()* gives Type I (incremental) and not marginal tests.

CAUTION: Use *car::Anova()* but set the contrast matrix in the call.

## R standard models - *lm()* CRD ANCOVA

Is the relationship between calories and fat the same for all display shelves?

Non-parallel slope model:

Anova Table (Type III tests)

Response: calories

	Sum Sq	Df	F value	Pr(>F)
(Intercept)	303495	1	805.8500	< 2.2e-16 ***
shelfF	572	2	0.7592	0.471782
fat	2693	1	7.1495	0.009298 **
shelfF:fat	1166	2	1.5484	0.219687
Residuals	26740	71		

No evidence that slopes are different.

## R standard models - *lm()* CRD ANCOVA

Is the relationship between calories and fat the same for all display shelves?

Non-parallel slope model - what are the separate slopes?

DO NOT USE output from *summary()* for categorical variables.

- Estimates depend on internal contrast matrix used by R.
- Estimates depend on order of factor levels.

Coefficients:

	Estimate	Std. Error	t value	Pr(> t )	
(Intercept)	96.144	3.387	28.387	<2e-16	***
shelfF1	4.134	4.683	0.883	0.3803	
shelfF2	0.642	5.283	0.122	0.9036	
fat	7.666	2.867	2.674	0.0093	**
shelfF1:fat	-7.296	4.455	-1.638	0.1059	
shelfF2:fat	3.167	4.322	0.733	0.4661	



Is the relationship between calories and fat the same for all display shelves?

Non-parallel slope model - what are the separate slopes?

Use the *lstrends()* function from the *lsmeans* package.

- Create least squares mean object (*lsmo*) first
- Use *cld()*, *pairs()*, *summary()* to investigate

```
1 library(lsmeans)
2 shelf.fit.lsmo <- lsmeans::lstrends(shelf.fit,
3   ~shelfF, var="fat")
4 shelf.fit.cld <- cld(shelf.fit.lsmo)
5 shelf.fit.cld
```

Is the relationship between calories and fat the same for all display shelves?

Obtain the compact-letter-display (cld)

shelfF	fat.trend	SE	df	lower.CL	upper.CL	.group
1	0.3703704	5.905229	71	-11.4043217	12.14506	1
2	10.8333333	5.602192	71	-0.3371204	22.00379	1
3	11.7948718	2.779466	71	6.2527747	17.33697	1

Confidence level used: 0.95

P value adjustment: Tukey method for a family of 3 means  
significance level used:  $\alpha = 0.05$

Is the relationship between calories and fat the same for all display shelves?

Parallel slope model:

```
1 shelf.fit2 <- lm(calories ~ shelfF + fat, data=cereal,  
2                 contrasts=list(shelfF=contr.sum))  
3 car::Anova(shelf.fit2, type=3) # marginal tests
```

CAUTION. `anova()` gives Type I (incremental) and not marginal tests.

CAUTION: Use `car::Anova()` but set the contrast matrix in the call.

## R standard models - *lm()* CRD ANCOVA

Is the relationship between calories and fat the same for all display shelves?

Parallel slope model:

Anova Table (Type III tests)

Response: calories

	Sum Sq	Df	F value	Pr(>F)
(Intercept)	342333	1	895.5195	< 2.2e-16 ***
shelfF	216	2	0.2823	0.7548
fat	7026	1	18.3785	5.457e-05 ***
Residuals	27906	73		

No evidence that intercepts are different.

Evidence that common slope is useful.

## R standard models - *lm()* CRD ANCOVA

Is the relationship between calories and fat the same for all display shelves?

Parallel slope model - what is common slope?

DO NOT USE output from *summary()* for categorical variables, but ok for continuous variable for common slope.

- Estimates for categorical variables depend on internal contrast matrix used by *R*.
- Estimates for categorical variables depend on order of factor levels.

Coefficients:

	Estimate	Std. Error	t value	Pr(> t )	
(Intercept)	95.3297	3.1856	29.925	< 2e-16	***
shelfF1	-0.7752	3.5145	-0.221	0.826	
shelfF2	2.3802	3.3775	0.705	0.483	
fat	9.9092	2.3114	4.287	5.46e-05	***

Is the relationship between calories and fat the same for all display shelves?

Parallel slope model - what are the separate intercepts?

Use the *lsmean()* function from the *lsmeans* package to investigate differences among the intercepts.

- Create least squares mean object (*lsmo*) first
- Use *cld()*, *pairs()*, *summary()* to investigate

```
1 library(lsmeans)
2 shelf.fit2.lsmo <- lsmeans::lsmeans(shelf.fit2, ~shelfF)
3 shelf.fit2.cld <- cld(shelf.fit2.lsmo)
4 shelf.fit2.cld
```

Is the relationship between calories and fat the same for all display shelves?

Obtain the compact-letter-display (cld)

shelfF	lsmean	SE	df	lower.CL	upper.CL	.group
3	103.7625	3.304364	73	97.17693	110.3481	1
1	104.5924	4.474918	73	95.67385	113.5109	1
2	107.7477	4.266658	73	99.24430	116.2512	1

Confidence level used: 0.95

P value adjustment: Tukey method for a family of 3 means  
significance level used:  $\alpha = 0.05$

You could continue then to fit the co-incident model that we have previous fit.

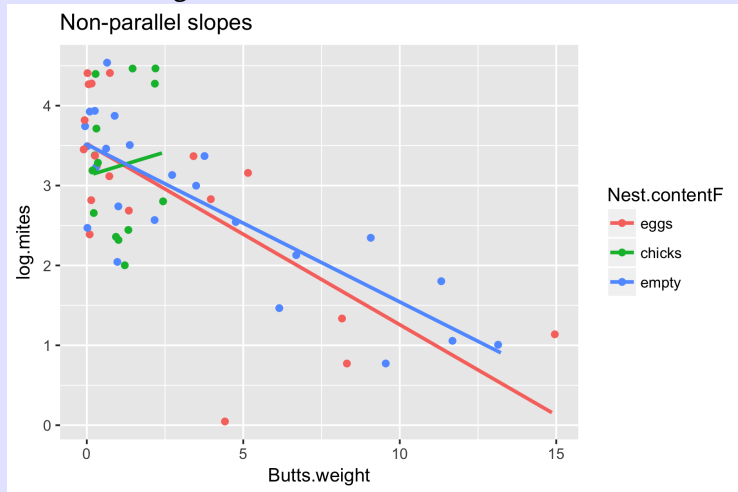


Return to Birds n' Butts.

Is there a difference in relationship between  $\log$  *Number of Mites* and *Butts Weight* the same for all 3 nest contents?

# R standard models - $lm()$ CRD ANCOVA

Is there a difference in relationship between  $\log$  *Number of Mites* and *ButtsWeight* the same for all 3 nest contents?



Are you worried about anything?

Is there a difference in relationship between log *Number of Mites* and *ButtsWeight* the same for all 3 nest contents? Non-parallel slope model:

```
1 np.fit <- lm(log.mites ~ Nest.contentF +  
2               Butts.weight +  
3               Butts.weight:Nest.contentF, data=butts,  
4               contrasts=list(Nest.contentF=contr.sum))  
5 car::Anova(np.fit, type=3)
```

CAUTION. *anova()* gives Type I (incremental) and not marginal tests.

CAUTION: Use *car::Anova()* but set the contrast matrix in the call.

Is there a difference in relationship between  $\log$  *Number of Mites* and *ButtsWeight* the same for all 3 nest contents? Non-parallel slope model:

Response: `log.mites`

	Sum Sq	Df	F value	Pr(>F)	
(Intercept)	288.693	1	446.7996	<2e-16	***
Nest.contentF	0.654	2	0.5063	0.6057	
Butts.weight	0.729	1	1.1281	0.2932	
Nest.contentF:Butts.weight	1.024	2	0.7925	0.4582	
Residuals	32.953	51			

No evidence that slopes are different.

Is there a difference in relationship between *log Number of Mites* and *ButtsWeight* the same for all 3 nest contents? Non-parallel slope model - what are the separate slopes?

Use the *lstrends()* function from the *lsmeans* package.

- Create least squares mean object (*lsmo*) first
- Use *cld()*, *pairs()*, *summary()* to investigate

```
1 library(lsmeans)
2 np.fit.lsmo <- lsmeans::lstrends(np.fit, ~Nest.contentF,
3                               var='Butts.weight')
4 np.fit.cld <- cld(np.fit.lsmo)
5 np.fit.cld
```

Is there a difference in relationship between log *Number of Mites* and *ButtsWeight* the same for all 3 nest contents? Obtain the compact-letter-display (cld)

Nest.c	Butts.weight.trend	SE	df	lower.CL	upper.CL	.gro
eggs	-0.2264520	0.04705214	51	-0.3209131	-0.1319909	1
empty	-0.1972181	0.03912420	51	-0.2757632	-0.1186730	1
chicks	0.1190728	0.28018026	51	-0.4434126	0.6815581	1

Confidence level used: 0.95

P value adjustment: Tukey method for a family of 3 means  
significance level used: alpha = 0.05

## R standard models - *lm()* CRD ANCOVA

Is there a difference in relationship between log *Number of Mites* and *ButtsWeight* the same for all 3 nest contents? Parallel slope model:

```
1 p.fit <- lm(log.mites ~ Nest.contentF + Butts.weight,  
2             data=butts,  
3             contrasts=list(Nest.contentF=contr.sum))  
4 car::Anova(p.fit, type=3)
```

CAUTION. *anova()* gives Type I (incremental) and not marginal tests.

CAUTION: Use *car::Anova()* but set the contrast matrix in the call.

Is there a difference in relationship between  $\log$  *Number of Mites* and *ButtsWeight* the same for all 3 nest contents? Parallel slope model:

Response: `log.mites`

	Sum Sq	Df	F value	Pr(>F)
(Intercept)	447.86	1	698.6061	< 2.2e-16 ***
Nest.contentF	0.09	2	0.0716	0.931
Butts.weight	30.48	1	47.5405	6.657e-09 ***
Residuals	33.98	53		

No evidence that intercepts are different.

Evidence that common slope is useful.



## R standard models - *lm()* CRD ANCOVA

Is there a difference in relationship between  $\log$  *Number of Mites* and *ButtsWeight* the same for all 3 nest contents? Parallel slope model - what is common slope?

DO NOT USE output from *summary()* for categorical variables, but ok for continuous variable for common slope.

- Estimates for categorical variables depend on internal contrast matrix used by *R*.
- Estimates for categorical variables depend on order of factor levels.

Coefficients:

	Estimate	Std. Error	t value	Pr(> t )	
(Intercept)	3.49042	0.13206	26.431	< 2e-16	***
Nest.contentF1	-0.02536	0.15198	-0.167	0.868	
Nest.contentF2	-0.03067	0.17030	-0.180	0.858	
Butts.weight	-0.20543	0.02979	-6.895	6.66e-09	***

Is there a difference in relationship between *log Number of Mites* and *ButtsWeight* the same for all 3 nest contents? Parallel slope model - what are the separate intercepts?

Use the *lsmean()* function from the *lsmeans* package to investigate differences among the intercepts.

- Create least squares mean object (*lsmo*) first
- Use *cld()*, *pairs()*, *summary()* to investigate

```
1 library(lsmeans)
2 p.fit.lsmo <- lsmeans::lsmeans(p.fit, ~Nest.contentF)
3 p.fit.cld <- cld(p.fit.lsmo)
4 p.fit.cld
```

Is there a difference in relationship between log *Number of Mites* and *ButtsWeight* the same for all 3 nest contents? Obtain the compact-letter-display (cld)

Nest.content	F	lsmean	SE	df	lower.CL	upper.CL	.group
chicks		2.893058	0.2200048	53	2.451785	3.334332	1
eggs		2.898371	0.1836875	53	2.529941	3.266802	1
empty		2.979761	0.1661959	53	2.646414	3.313107	1

Confidence level used: 0.95

P value adjustment: Tukey method for a family of 3 means  
significance level used: alpha = 0.05

You could continue then to fit the co-incident model that we have previous fit.

# R Single Factor CRD ANCOVA - Summary

- Check that your design is a single-factor CRD
  - One categorical and one continuous variable.
  - Complete randomization of treatments to experimental units.
  - Experimental Unit = Observational Unit (no pseudo-replication)
- Fitting three models.
- Preliminary plots - check for outliers
- Declare a FACTOR variable
- Use `my.fit <- lm(Y ~ X + GroupF + X:GroupF)` + contrast option.
- Use `car::Anova(my.fit)` for overall comparison.
- Use `my.lsmo <- lsmeans::lstrends(my.fit, ~ GroupF, var="X")` to get separate slopes

See <http://www.stat.sfu.ca/~cschwarz/CourseNotes> for much more details

# R standard models - *glm()* CRD logistic ANOVA

Is there a relationship between PROPORTION of fatalities and day of the week?

CRD implies that

- Single factor with 2+ levels (e.g. Day of Week (1...7));
- Complete randomization of treatments (levels) to experimental units; or random sample from relevant population of each level.
- Observational unit = Experimental unit. One measurement per e.u.
  - Y variable must be CATEGORICAL (e.g. yes/no; live/die; small/medium/large)
  - Avoid pseudo-replication - see my CourseNotes
- $H : \pi_1 = \pi_2 = \pi_3 \dots = \pi_k$
- Two standard analyses:
  - $\chi^2$  tests
  - Logistic ANOVA (esp. if only 2 levels of response)

CAUTION: NOT ALL EXPERIMENTS are CRDs!

Seek advice before analyzing complex designs.

Is there a relationship between PROPORTION of fatalities and day of the week?

Read in the dataframe:

- Use *read.csv()* in the usual fashion.
- Check the names of the variables.
- Check the dimensions of the data frame (!)

```
1 radf <- read.csv("../sampledata/road-accidents-2010.csv",
2                   header=TRUE,
3                   as.is=TRUE,
4                   strip.white=TRUE)
5 dim(radf)
6 names(radf)
```

## R standard models - *glm()* CRD logistic ANOVA

Is there a relationship between PROPORTION of fatalities and day of the week?

BOTH *X* and *Y* MUST be declared as FACTORS in *R* because they are categorical.

Put the category you wish to model (i.e. fatalities) last in the factor levels list.

```
1 # Declare day of week (X) as a factor because categorical
2 radf$Day_of_WeekF <- factor(radf$Day_of_Week)
3
4 # Declare response (Y) also as factor because categorical
5 radf$fatal <- recode(radf$Accident_Severity,
6                     ' 1="yes";2:3="no"  ')
7 radf$fatalF <- factor(radf$fatal, level=c("no","yes"),
8                     order=TRUE)
```



## R standard models - *glm()* CRD logistic ANOVA

Is there a relationship between PROPORTION of fatalities and day of the week?

Preliminary summary statistics

```
1 # Preliminary tabulation
2 xtabs(~fatalF+Day_of_WeekF, data=radf)
3 round(prop.table(xtabs(~fatalF+Day_of_WeekF, data=radf),2),3)
```

	Day_of_WeekF						
fatalF	1	2	3	4	5	6	7
no	16489	22231	22819	22808	22602	25214	20520
yes	305	221	226	210	208	261	300

	Day_of_WeekF						
fatalF	1	2	3	4	5	6	7
no	0.982	0.990	0.990	0.991	0.991	0.990	0.986
yes	0.018	0.010	0.010	0.009	0.009	0.010	0.014

## R standard models - *glm()* CRD logistic ANOVA

Is there a relationship between PROPORTION of fatalities and day of the week?

```
1 fatal.fit <- glm(fatalF ~ Day_of_WeekF,  
2                 family=binomial, data=radf)  
3  
4 anova(fatal.fit, test='Chi') # CAUTION Type I tests  
5 summary(fatal.fit) # not useful
```

	Df	Deviance	Resid. Df	Resid. Dev	Pr(>Chi)
NULL			154413	18990	
Day_of_WeekF	6	108.98	154407	18881	< 2.2e-16 ***

CAUTION. *anova()* gives Type I (incremental) and not marginal tests. This is OK for a single factor CRD as then both are the same.

## R standard models - *glm()* CRD logistic ANOVA

Is there a relationship between PROPORTION of fatalities and day of the week?

```
1 fatal.fit <- glm(fatalF ~ Day_of_WeekF,  
2                 family=binomial, data=radf)  
3  
4 anova(fatal.fit, test='Chi') # CAUTION Type I tests  
5 summary(fatal.fit) # not useful
```

DO NOT USE output from *summary()* for categorical variables.

- Estimates depend on internal contrast matrix used by *R*.
- Estimates depend on order of factor levels.

	Estimate	Std. Error	z value	Pr(> z )
(Intercept)	-3.99014	0.05779	-69.049	< 2e-16 ***
Day_of_WeekF2	-0.62094	0.08893	-6.982	2.91e-12 ***
Day_of_WeekF3	-0.62468	0.08836	-7.070	1.55e-12 ***
.....				

Is there a relationship between PROPORTION of fatalities and day of the week?

Use the *lsmeans()* function from the *lsmeans* package.

Note that the *lsmeans()* is applied to model object.

CAUTION - results are NOT means – they are logit's = log odds =  $\log \frac{p_{fatal}}{1-p_{fatal}}$ .

```
1 # We get the lsmeans (eventhough these are logits of propor
2 fatal.fit.lsmo <- lsmeans::lsmeans(fatal.fit,
3                                     ~Day_of_WeekF)
```

## R standard models - *glm()* CRD logistic ANOVA

Is there a relationship between PROPORTION of fatalities and day of the week?

Obtain the compact-letter-display (cld)

```
1 # where do the p(fatal) differ?  
2 fatal.fit.cld <- cld(fatal.fit.lsmo)  
3 fatal.fit.cld
```

Day_of_WeekF	lsmean	SE	df	asympt.LCL	asympt.UCL	.group
5	-4.688256	0.06965465	NA	-4.824793	-4.551718	1
4	-4.687759	0.06932233	NA	-4.823645	-4.551873	1
3	-4.614814	0.06684697	NA	-4.745847	-4.483780	1
2	-4.611080	0.06760018	NA	-4.743590	-4.478570	1
6	-4.570634	0.06221758	NA	-4.692593	-4.448675	1
7	-4.225373	0.05815552	NA	-4.339369	-4.111376	2
1	-3.990137	0.05778698	NA	-4.103411	-3.876863	2

Results are given on the logit (not the response) scale.

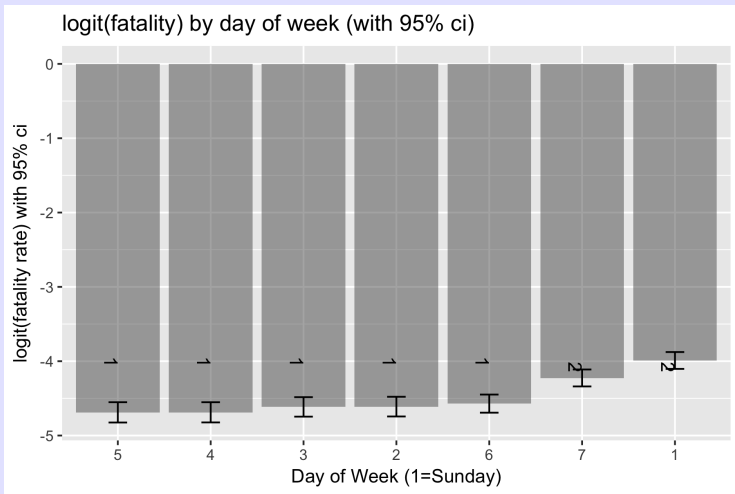
## R standard models - *glm()* CRD logistic ANOVA

Is there a relationship between PROPORTION of fatalities and day of the week?

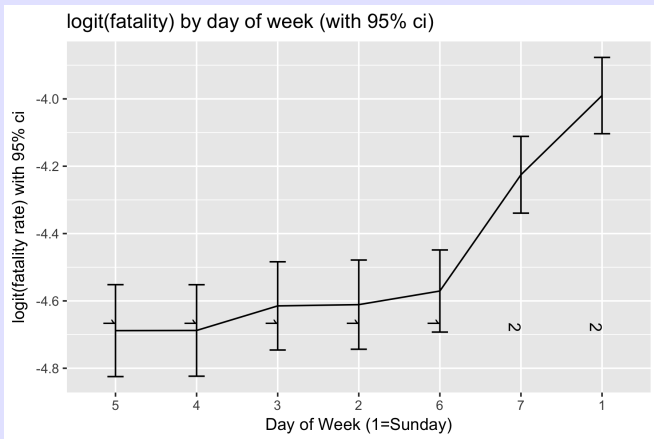
Plot the results from the `cld` using `ggplot()` or my functions (after suitable mixups)

```
1 fatal.fit.cld$upper.CL <- fatal.fit.cld$asympt.UCL
2 fatal.fit.cld$lower.CL <- fatal.fit.cld$asympt.LCL
3
4 sf.cld.plot.bar(fatal.fit.cld, "Day_of_WeekF")+
5   ggtitle("logit(fatality) by day of week (with 95% ci)")+
6   xlab("Day of Week (1=Sunday)")+
7   ylab("logit(fatality rate) with 95% ci")
8
9 sf.cld.plot.line(fatal.fit.cld, "Day_of_WeekF")+
10  ggtitle("logit(fatality) by day of week (with 95% ci)")+
11  xlab("Day of Week (1=Sunday)")+
12  ylab("logit(fatality rate) with 95% ci")
```

# R standard models - $glm()$ CRD logistic ANOVA



# R standard models - $glm()$ CRD logistic ANOVA





## R standard models - *glm()* CRD logistic ANOVA

Is there a relationship between PROPORTION of fatalities and day of the week?

Convert back to proportion scale

```
1 summary(fatal.fit.lsmo, type="response")
```

Day_of_WeekF	prob	SE	df	asympt.LCL	asympt.UCL
1	0.018161248	0.0010304237	NA	0.016247887	0.020074609
2	0.009843221	0.0006588538	NA	0.008632165	0.011054277
3	0.009806900	0.0006491324	NA	0.008612872	0.011000928
4	0.009123295	0.0006266780	NA	0.007973353	0.010273237
5	0.009118808	0.0006293754	NA	0.007964279	0.010281337
6	0.010245339	0.0006309094	NA	0.009079697	0.011411081
7	0.014409222	0.0008259012	NA	0.012876778	0.016141666

Make a nice plot of these results.

Is there a relationship between PROPORTION of fatalities and day of the week?

Obtain the pairwise differences

```
1 # Estimate all of the pairwise differences and get the odds
2 pairs(fatal.fit.lsmo)
3 summary(pairs(fatal.fit.lsmo), infer=TRUE, type="response")
```

contrast	odds.ratio	SE	df	asympt.LCL	asympt.UCL	z
1 - 2	1.8606824	0.16547650	NA	1.2840968	2.6961666	6.98
1 - 3	1.8676423	0.16502870	NA	1.2919739	2.6998128	7.06
.....						

Continue with the Accidents Dataset Is there a difference in PROPORTION of fatalities by weather conditions?

Only use *weather* codes 1-6. Delete records from other *weather conditions*.

Hint: use the `%in%` operator.

Do you see anything odd about the results from the *cld()* function?  
Do you see anything odd about the relationship with weather?

Is there a relationship between PROPORTION of fatalities and weather conditions?

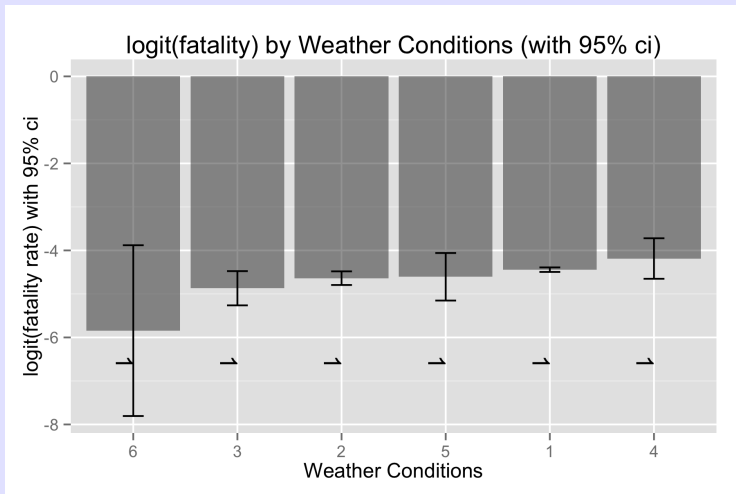
	Df	Deviance	Resid. Df	Resid. Dev	Pr(>Chi)
NULL			146023	18075	
WeatherF	5	14.934	146018	18060	0.01065 *

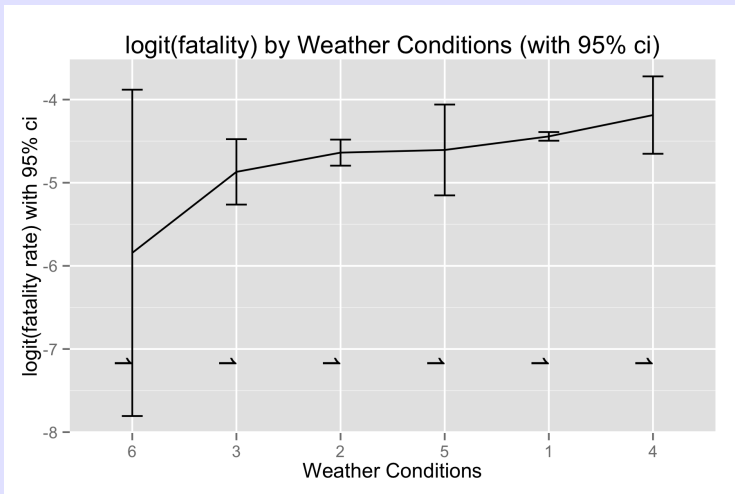
Is there a relationship between PROPORTION of fatalities and weather conditions?

```
> fatal.fit.cld # interesting what happens
```

WeatherF	lsmean	SE	df	asympt.LCL	asympt.UCL	.group
6	-5.843544	1.00139007	NA	-7.806470	-3.880618	1
3	-4.869379	0.20076634	NA	-5.262921	-4.475836	1
2	-4.637980	0.07993974	NA	-4.794678	-4.481282	1
5	-4.605939	0.27873234	NA	-5.152311	-4.059568	1
1	-4.442536	0.02655303	NA	-4.494586	-4.390487	1
4	-4.186282	0.23748716	NA	-4.651805	-3.720759	1

# R standard models - *glm()* CRD logistic ANOVA - Exercise





Continue with the Accidents Dataset Is there a difference in PROPORTION of fatalities by speeds of the vehicles?  
Drop the very low speed limits because of a lack of data.



# R Single Factor CRD logistic ANOVA - Summary

- Check that your design is a single-factor logistic CRD
  - Single factor with at least 2 levels.
  - Complete randomization of treatments to experimental units.
  - Experimental Unit = Observational Unit (no pseudo-replication)
  - Response Variable ( $Y$ ) - categorical with 2 levels
- $H : \pi_1 = \pi_2 = \dots = \pi_k$
- Preliminary cross tabulations
- Declare BOTH  $X$  and  $Y$  as FACTOR variables
- Use `my.fit <- glm(Y ~ GroupF, family=binomial)` to fit model
- Use `anova(my.fit)` for overall comparison CAUTION about Type I tests
- Use `my.lsmo <- lsmeans::lsmeans(my.fit, ~ GroupF)` to get marginal logits and proportions etc
- Plot the results.

See <http://www.stat.sfu.ca/~cschwarz/CourseNotes> for much more details

Simulation Studies  
How many years sampling do I need to  
detect a trend?

# Simulation Study - Detecting a trend?

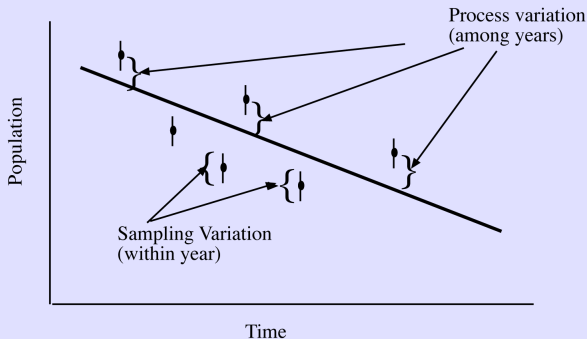
How many years sampling do I need to detect a trend?

- What size of trend needs to be detected?
  - Steeper trends are easier to detect than shallower trends
- How much years, what pattern of years, how many replicates in a year will be sampled?
  - More sampling makes it easier to detect trends.
  - Often length of time is more important than number of sampled years.
  - Number of sampling within a year is usually not that important because process error often dominates.  
We will assume 1 sample/year.
- What is the noise in the data?
  - Larger noise makes it harder to detect trends
  - Beware of difference between process and sampling error  
We will be ignoring process error in this exercise – DANGER!
- What is  $\alpha$  level and target power?
  - Common values are  $\alpha = 0.05$  and target power = 0.80.

Sometime difficult to get estimates of the variance components - seek help.

# Simulation Study - Detecting a trend?

## Process vs Sampling Variation



Sampling variation refers to the uncertainty of each estimate within each year, i.e. the standard error.

This can be reduced by increasing the sampling effort in each year. Process variation occurs because even if the yearly estimates had a standard error of 0, the points would not lie on the straight line. Process variation is unaffected by the sampling effort in each year.

It is DANGEROUS to assume that no process error exists in trend analyses.

# Simulation Study - Detecting a trend?

How many years sampling do I need to detect a trend?

- What size of trend needs to be detected?  
**2%/year starting from 100 in year 1**
- How much years, what pattern of years, how many replicates in a year will be sampled?  
**10 years; every year; 1 sample/year**
- What is the noise in the data?
  - Process standard deviation will be ignored (DANGEROUS)
  - Sampling standard deviation  
**20% of mean + process error (latter assumed to be zero)**
- What is  $\alpha$  level and target power?  
 **$\alpha = 0.05$  and target power of 80%**

# Simulation Study - Detecting a trend?

## General steps

- Create a function to generate data
- Fit a straight line to the data.
- Estimate slope,  $se(\text{slope})$  and p-value of testing if slope = 0.
- If p-value  $< \alpha$ , then detect trend = 1; else detect trend = 0.
- Repeat 1000 times and see what fraction of simulations detect trend.

# Simulation Study - Detecting a trend?

Arguments for function to generate data.

- Initial value (*Ivalue*)
- Trend (*Trend*) - fractional change per year
- Time points to measure *SampTimes* - vector
- Sampling standard deviation (*SampStdDev*) - fraction of mean response

```
GenData <- function(Ivalue, Trend, SampTimes, Nrepsyear,  
ProcessStdDev, SampStdDev)
```

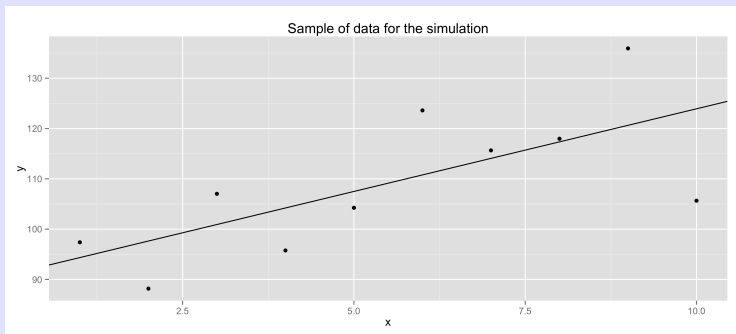
# Simulation Study - Detecting a trend - GenData

```
1 GenData <- function(Ivalue, Trend, SampTimes, Nrepsyear, Pro
2     mu <- Ivalue * (1+Trend)**(SampTimes-1)
3     Serror <- rnorm(length(mu), mean=0, sd=SampStdDev*mu)
4     basey <- mu + Serror
5     return(cbind(x=SampTimes, y=basey))
6 }
```



```
1 set.seed(234234)
2 test <- GenData(Ivalue=100,
3                 Trend=.02,
4                 SampTimes=1:10,
5                 SampStdDev=.20)
6 plot(test[, "x"], test[, "y"])
7 abline(coef(lm(y ~ x, data=as.data.frame(test))))
```

# Simulation Study - Detecting a trend - GenData - Testing



All error about line is assumed to be sampling error.

# Simulation Study - Detecting a trend - wrapping GenData

```
1  do.one.sim <- function(Ivalue, Trend,
2      SampTimes, Nrepsyear,
3      ProcessStdDev, SampStdDev,
4      alpha=0.05) # default value
5  {
6      # generate the test data, fit the line, extract
7      # the slope, se, and p-value
8      # and see if we detected anything?
9      mydata <- GenData(Ivalue=Ivalue,
10         Trend=Trend,
11         SampTimes=SampTimes,
12         Nrepsyear=Nrepsyear,
13         ProcessStdDev=ProcessStdDev,
14         SampStdDev=SampStdDev)
15      myfit <- lm(y ~ x, data=as.data.frame(mydata))
16
17      ... see next slide ..
```

# Simulation Study - Detecting a trend - wrapping GenData

```
1
2     ... see previous slide ..
3
4     mycoef <- coef(myfit)
5     my.fit.summary <- summary(myfit)
6     myslope.se <- my.fit.summary$coefficients[2,2]
7     myslope.pvalue <- my.fit.summary$coefficients[2,4]
8     detect <- myslope.pvalue < alpha
9     myres <- c(mycoef, myslope.se,
10               myslope.pvalue,
11               detect)
12     names(myres) <- c("intercept", "slope", "se",
13                      "pvalue", "detect")
14     return(myres)
15 }
```

# Simulation Study - Detecting a trend - wrapping GenData

```
1 test <- do.one.sim(Ivalue=100,  
2                     Trend=.02,  
3                     SampTimes=1:10,  
4                     SampStdDev=.20)  
5 test  
6  
7  
8  
9 > test
```

intercept	slope	se	pvalue	detect
87.52802531	3.81677419	1.80816001	0.06778245	0.00000000

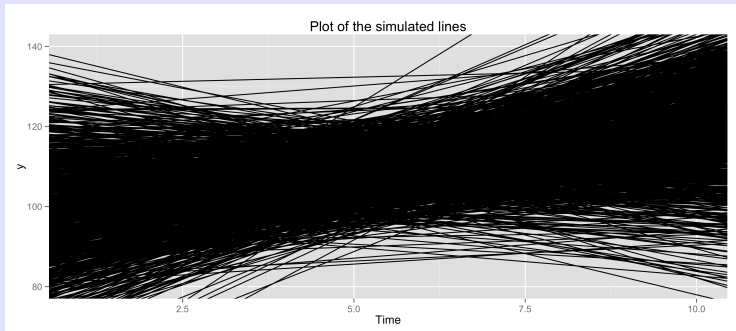
# Simulation Study - Detecting a trend - Replicating results 1000 times

```
1 myres <- rdply(1000, do.one.sim(Ivalue=100,  
2                               Trend=.02,  
3                               SampTimes=1:10,  
4                               SampStdDev=.20))  
5 myres[1:5,]  
6  
7 > myres[1:5,]  
8   .n intercept      slope      se      pvalue detect  
9 1  1  57.76776 7.3625620 2.531806 0.01964839      1  
10 2  2  88.64678 4.4472367 2.712953 0.13979077      0  
11 3  3  98.56670 2.0576494 1.796553 0.28517674      0  
12 4  4 100.25568 0.5551427 1.649608 0.74512886      0  
13 5  5  94.02433 3.0618447 1.814522 0.13000490      0
```

# Simulation Study - Detecting a trend - Replicating results 1000 times

```
1 cat("the power is ", mean(myres$detect),"\n")
2
3 plot(NULL, NULL, type="n",
4       xlim=c(1,10), ylim=c(80,140),
5       main='Estimated lines',
6       xlab="Time", ylab="Y")
7
8 the power is 0.138
```

# Simulation Study - Detecting a trend - Fitted lines



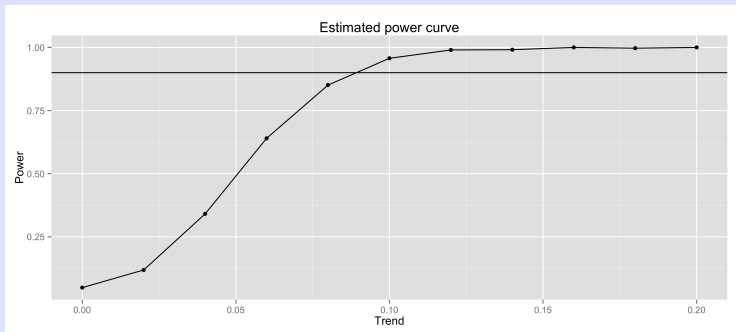


# Simulation Study - Detecting a trend - Exercise

Estimate power when trend varies from 0 to .20 by .02.

Plot the results.

# Simulation Study - Detecting a trend - Exercise



# Simulation Studies in *R* - Summary

- Complete set of random number generators available.
- Use *set.seed()* to initialize generators and to make results reproducible.
- Start at lowest level and test every step!
- Make extensive use of the *apply* family or the *plyr* package.
- Pass as many parameters as possible rather than hard coding. The ... feature is VERY useful (you will have to do some reading on this).

How do you get output from *R* into other documents?

- Separate files for text fragments and graphs
  - *ggsave()* - many different formats
  - *png()* + *dev.off()* - Base *R* avoid
  - *sink('filename.txt', split=TRUE)* ... *R* code ... *sink()*. Need to be careful about matching sinks; just plain ascii text
  - *write.csv()* - send tables to csv files for embedding in other documents.
- Simple Notebooks
  - HTML/ Word/ PDF notebooks from *Rstudio*.
- Markdown documents that get 90% of the way there
  - Rmarkdown - creates a generic document with some post-editing required
- Markdown using LaTeX that give final product with no intervention

## Extracting output from *R* - Graphical output

```
ggsave(plot=..., file=..., h=xx, y=xx, units="in", dpi=300)
```

```
1 myplot <- ggplot(data=cereal, aes(x=fat, y=calories))+  
2   geom_point()  
3 ggsave(myplot, file='myggplot.png',  
4   h=4, w=6, units="in", dpi=300))
```

- Able to save graphics in a variety of ways (look at suffix, e.g. \*.png format).
- Script runs and generates the plot with no user intervention.

# Extracting output from *R* - Text output

`sink('filename.txt', split=TRUE) ... R code ... sink()`

```
1 sink('Images/sample-textoutput.txt', split=TRUE)
2   fit <- lm(Calories ~ Fat, data=cereal)
3   anova(fit)
4   summary(fit)
5   confint(fit)
6 sink()
```

- Simple text output in raw ascii text.
- Script runs and generates the file with no user intervention.
- You need to do extensive formatting of textual output after the fact (groan).
- Creating an HTML notebook does not send info to file (groan).
- Careful of mismatched sinks (groan).

## Extracting output from *R* - Text output as \*.csv file

```
write.csv(dataframe, "filename.csv", ....)
```

```
write.xls(dataframe, "filename.xls", ....)
```

```
1 write.csv(cereal, 'new file name.csv')
```

- Simple tables that can be then formatted using Excel (e.g. decimal points) etc.

*Rstudio* allows you to create notebooks in a variety of formats that combine textual and graphical output. Try it on a simple script.

- Scripts must be perfect with NO errors.
- Scripts must load all libraries
- You may need LaTeX to generate PDF.

Try it with *SampleScript.r* in *SampleData* directory.



# Extracting output from *R* - R Markdown

- A rudimentary script with *R* code and basic document formatting
- Creates a combined document (default in HTML), but there are converters to other word processors
- Suitable for a preliminary report for review but not production

*Rstudio* → File → NewFile → New Markdown.

Examine the *SampleRMarkdown.Rmd* file in the *sampledata* directory.

# Extracting output from *R* - R Markdown

Parts of the Rmarkdown document.

- YAML header
- Text with Markdown
- Code chunks begin with “`{r}`” and end with “`”`”.
- Inline code chunks have format ‘`r code`’.

Get the cheat sheet from *Rstudio* help.

Look at *Rstudio* options on debugging the report.

Try generating different document types

- HTML is easiest.
- Word documents can use a style file.
- PDF documents require LaTeX to be installed.

Get the cheat sheet from *Rstudio* help.

Create a report from the analysis of accidents. It should include the following elements.

- Summary of the number of accidents by month with proportion of fatal accidents.
- Figure of number of accidents per day.
- Analysis of proportion of fatalities by day of the week and by month.
- Graph of the results of the above.

Ultimate in turn-key reports.

- Single document with *R* code, document code, etc
- Integration by  $\text{\LaTeX}$  (steep learning curve)
- Simple changes to data are automatically propagated throughout the report. Suitable for high production environments where “same” output is regularly produced and no user intervention is needed (e.g. monthly reports at hospitals).

Look at *SampleSweave.Rnw* in the *sampledata* directory.

Create a report from the analysis of accidents. It should include the following elements.

- Summary of the number of accidents by month with proportion of fatal accidents.
- Figure of number of accidents per day.
- Analysis of proportion of fatalities by day of the week and by month.
- Graph of the results of the above.

Wide range of documents can be created

- Tradeoff between time and reproducibility and repeatedness.
- Sweave has a considerable learning curve to get very polished documents with tables and figures properly placed.

Many other options available - look at *Rstudio* File → New menu.

## *R* Summary



*R* is powerful and versatile.  
*R* is free, but not cheap.

- *R* has a steep learning curve.
- *R* is not consistent in usage and syntax.
- *R* creates nice graphics, but poor at textual output (e.g. nicely formatted tables are tedious to construct)
- Packages have NO quality control.
- Requires a fair degree of statistical sophistication.
  - E.g. *anova()* function gives Type I rather than Type III SS, F-tests without warning!

Work flow using *Rstudio*.

- Launch *Rstudio*.
- Navigate to directory with scripts and data.
- SET WORKING DIRECTORY!  
Check the console pane to see if done properly.
- Open the Script
- Highlight and Run.
- Create HTML notebook at end to ensure that script works properly.

*R* has a rich set of data structures (special case of objects):

- **vectors** - an ordered collection of the same data type (e.g. numbers, characters, logicals, etc.)
- **matrix** - a two-dimensional collection of the same data type.
- **array** - a 2+ dimensional collection of the same data type.
- **dataframe** - collection of vectors (of same length) but vectors can be different data types.
- **list** - an arbitrary collection of objects (including lists).

Other objects:

- **function** - contains a list of instructions
- **expressions** - fragments of *R* code or formulae

# R Summary - Object Types

R has several object types

- numbers (integer, real, or complex)
- characters ("abc")
- logical (TRUE or FALSE)
- Date, DateTime
- factor (CAUTION) of any type - index to a set of values
  - use *stringsAsFactors=FALSE* on all *read.csv()* operations.
  - use *stringsAsFactors=FALSE* on all *data.frame()* operations.
  - explicitly create factors for all categorical variables using *df\$varF = factor(df\$var)*.
  - distinguish between *size=var* or *size=varF* in *ggplot()*.
  - may need to order factors to sort levels on a graph.

Missing values *NA* are different from *Inf*, " ", 0, *NaN* etc.

The *str()* function is YOUR FRIEND!

# R Summary - Accessing values

R has several ways to access values (see reference card)

- vectors
  - `v[k]` - selects the  $k^{th}$  item
  - `v[-k]` - all but the  $k^{th}$  item
  - `v[1:4]`, `v[c(1,3,5)]`, `v[-(1:k)]` - sets of values
  - `v["name"]`, `v[c("name1","name2")]` - select by named component
  - `v[c(TRUE, FALSE, TRUE)]`, `v[v>3]`, `v[v == 24]` - use LOGICAL vector to select items
- data frames
  - `dafr[k,]`, `dafr[,k]` - all of  $k^{th}$  column/row
  - `dafr$name1`, `dafr["name1"]`, `dafr[,c("name1","name2")]` - select columns by names
  - `dafr[ dafr$v>10, c("v2","v3")]` - subsets of the data frame
- lists
  - `mylist$name` - select a named component of the list
  - `mylist[["name"]]` - element with certain name
  - `mylist[1]` vs. `mylist[[1]]` - CAREFUL

R is a bit clumsy.

- *read.table()*, *read.csv()*, *read.delim()*, *read.xlsx()*
  - *header=TRUE* - assumes variable names in first row
  - *as.is=TRUE* or *stringsAsFactors=FALSE* - don't convert character string to factors - RECOMMENDED
  - *strip.white=TRUE* - remove extra white space when ever possible - CAUTION of leading blanks in character strings
- *scan()* - a more general way to read files with odd organizations
- Several packages for access to database systems - see reference card.
- Define categorical variables as factor by making new variable (*df\$varF <- factor(df\$var)*) or replacing variable (*df\$var <- factor(df\$var)*) after creating data frame.

# R Summary - Programming

Try and vectorize.

Most of time you can avoid *if()* and *for()* control structures.

R will often cycle through shorter arguments

```
1 v <- 15:18
2 v + 3 # adds 3 to every value
3 v + c(2,3) # cycles through the (2,3) pair
4
5 v[ v==10] <- 15 # replaces for/if structure
6
7 Fatal <- c("no","yes")[1+ (Severity==1)]
```

*recode()* in *car* package is useful

*R* is a bit clumsy.

- `sink("filename", split=TRUE)` ... `sink()` - text output.
- `write.table()`, `write.csv()`, `write.xlsx()`- create output files.
- `ggsave` - graphical output. Don't forget `h=`, `w=`, `units=`, and `dpi=` arguments.
- *Rstudio* with the html notebook.
- *Rmarkdown* - combined documents with text, programming, and output..
- *Sweave* -  $\text{\LaTeX}$  and *R* integrated together to create complete document that is publication quality.



Many user-written extensions to *R*; but no quality control.

- Create a personal library for packages on your computer as this makes it easier to update on a regular basis.
- Load library prior to first use using the *library()* function.
- Very difficult to detach a package once it is loaded
- Beware of name conflicts, i.e. several packages with the same name for different objects. Use *package::function()* to ensure that correct function is used.

# R Summary - Functions

Make a collection of analyses for reuse.

Scripts and `source()` can serve a similar functionality.

```
1 myfunction <- function(arg1, arg2, arg3=defaultvalue) {  
2   # Comment describing the arguments and purpose of func  
3   arg1[3] <- new value # ok see below  
4   ...  
5   myresults <- ....  
6   return(myresults) # don't forget  
7 }
```

- Arguments can be any data structure or type
- Return can be any data structure or type (most commonly a list or a vector)
- Arguments are call-by-name, i.e. copies passed.
- New variables are local only
- AVOID SIDE EFFECTS IN A FUNCTION
- `browser()`, `trace()` - useful for debugging

Scripts and *source()* can serve a similar functionality.

```
1 source("MyDocuments/MyStuff/myRfunction.r")
2
3 # now you can use the functions you defined
4 # in myRfunction.r
```

# R Summary - Useful Builtin Functions - I

Base R has many useful functions; packages provide more

- `c(...)` - combine arguments into a vector
- `seq()`, `a:b` - generate sequences
- `is.na()` - tests for missing values – see other `is.xx` functions
- `str()` - show structure of an object
- `nrow()`, `ncol()` - number of rows and columns
- `match(x,y)`, `x %in% y` - which values of `x` are in `y`
- `unique()` - return unique values of object
- `xtabs()` - cross tabulations - useful for check recodes, etc – include the NAs
- `reshape()` - interchange between wide and long formats - documentation sucks

## R Summary - Useful Builtin Functions - II

Base *R* has many useful functions; packages provide more

- *cbind()*, *rbind()* - paste together columns and rows
- *split()*, *stack()* - split and stack dataframes
- Split-analyze-combine paradigm - RECOMMEND *plyr* package rather than Base *R* functions, esp. the *summarize* usage.

```
1 ddply(cereal, "shelf", summarize,  
2       mean.cal = mean(calories))  
3  
4 ddply(cereal, "shelf", function(x){  
5       ncereals <- nrow(x)  
6       fit <- lm( Calories ~ Fat, data=x)  
7       mycoef <- coef(fit)  
8       res <- data.frame(ncereals, mycoef, stringsAsFactors=FALSE)  
9       return(res)  
10    })
```

Base R has many useful functions; packages provide more

- Usual math functions.
  - CAUTION between *min()* and *pmin()*
- Usual statistical functions
  - NAs propagate, so many functions have `na.rm=TRUE`
  - *lm()* - basic linear models (e.g. simple ANOVA and regression)
  - *glm()* - generalized linear models (e.g. logistic regression)
  - *lmer()* - linear models with mixed effects (e.g. split-plot designs)
  - Use methods (specialized functions) to extract information from output
  - See <http://www.stat.sfu.ca/~cschwarz/CourseNotes>

*R* is a bit clumsy with dates and times

- *as.Date()* to convert to internal format (# of days since origin)
- *as.POSIXct()* to convert to constant date-time (avoid *POSIXlt()* unless really needed)
- *format='%m/%d/%Y'* to convert from external to internal and out to external formats
  - CAUTION - when converting from dates/datetime, the "%xx" gives CHARACTERS, not numbers

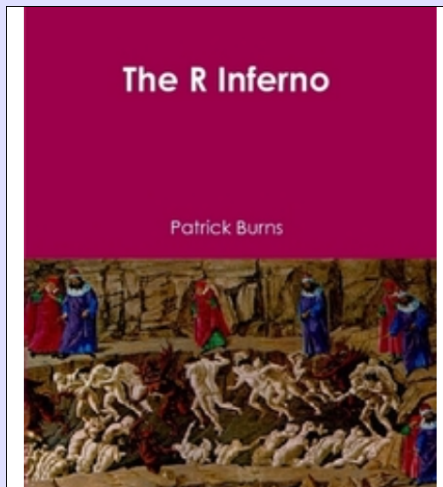
### Dealing with character strings

- *paste()* - combines strings, numbers, etc into a single string
- *substr()* - extracts substrings - CAUTION of syntax
- *grep()* - matching of patterns - CAUTION - complex syntax
- *stringr* package is easier to use in many cases



R has extensive facilities for plotting

- Base R - pen-on-paper paradigm AVOID
  - *plot()*, *hist()*, *boxplot()*, *stripchart*
  - *plot(lm.objectfit)* - residual plots and other diagnostic plots
  - many low level command to embellish plots - see reference card
  - many adjustments to plot attributes - see reference card
- Lattice graphics - plot objects - AVOID
  - Additional plot types - see reference card
  - CAUTION: Don't forget to print final object created
- *ggplot2* package - grammar of graphs - RECOMMENDED
  - Build a graph using various layers
  - Adjust final graph when done with axes etc
  - CAUTION: Don't forget to print final object created



If you are using *R* and you think you're in hell, this is a map for you. A book about trouble spots, oddities, traps, glitches in *R*. Even if it doesn't help you with your problem, it might amuse you (and hence distract you from your sorrow).

<http://www.burns-stat.com/documents/books/the-r-inferno/>

To err is human,  
but it really takes a computer to screw things up!

*R* is free, but not cheap.

`cschwarz@stat.sfu.ca`