

Introduction to R for Psychometrics

Matthew Sigal

Multi-Health Systems & York University

The R Project for Statistical Computing

What is R? Who is R? How can R help me?

Organization

This presentation will introduce the R computing environment. We will briefly address some history and fundamentals, but largely focus on applications: how R can be utilized for the tasks that we commonly deal with at MHS. This will include: data manipulation tasks, running basic and advanced statistical analyses, and key techniques for programming user defined functions.

What is R?

- R is an open source language and environment for statistical computing and graphics.
- Based upon S, which was developed by John Chambers and colleagues at Bell Laboratories in the late 1970s.
- R is a re-implementation of S that began development circa 1993 by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand.



The current version of R is 3.1.3, code-named "Smooth Sidewalk".

Who is R?

- Since mid-1997, the R Development Core Team has been maintained with write access to the R source code
- The R Foundation, a not for profit organization with a board of 5 board members and 2 auditors, controls the design and evolution of base R.

However, the R Project is actually a large scale effort and features the work thousands of contributors world-wide.

- UseRs are able to develop their own collections of functions which can be “packaged” and hosted for public distribution on the Comprehensive R Archive Network (<http://cran.r-project.org/>)
- As of March 2015, the CRAN package repository hosts **6,417** available packages and counting!

What can R do for me?

- Interactivity and expressiveness.
- No black boxes! All code is available and open.
- Primarily designed as a programming language for *statistical operations*, and almost every statistical method, especially those on the cutting edge, have R implementations.
- One of R's strengths is the ease with which well-designed publication-quality plots can be produced.
- Additional packages provide a framework for everything from running spatial analyses, to text mining, to the construction of interactive visualizations, and even to the creation of...

What can R do for me?

- Interactivity and expressiveness.
- No black boxes! All code is available and open.
- Primarily designed as a programming language for *statistical operations*, and almost every statistical method, especially those on the cutting edge, have R implementations.
- One of R's strengths is the ease with which well-designed publication-quality plots can be produced.
- Additional packages provide a framework for everything from running spatial analyses, to text mining, to the construction of interactive visualizations, and even to the creation of...
- **this presentation!**

Free?

Yes, free! R is an official part of the Free Software Foundation's GNU project with the standard General Public License. This means:

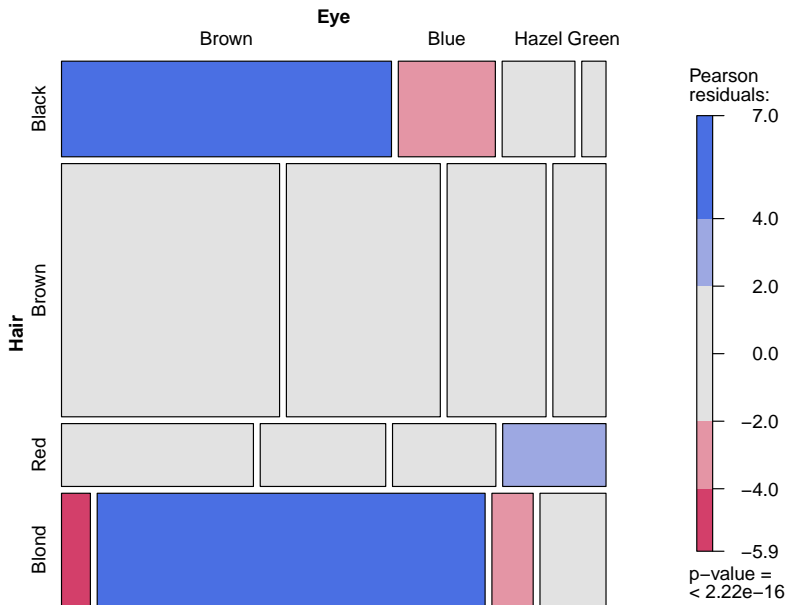
- you can obtain, copy, and re-use R without paying anything (freeware);
- you can access the source code and modify it (open source);
- you can redistribute your modifications back to the public for the improvement of the community at large (awesome).

As part of its mandate, R will always be free. However, a company called **Revolution Analytics** has proprietary rights for Revolution R.

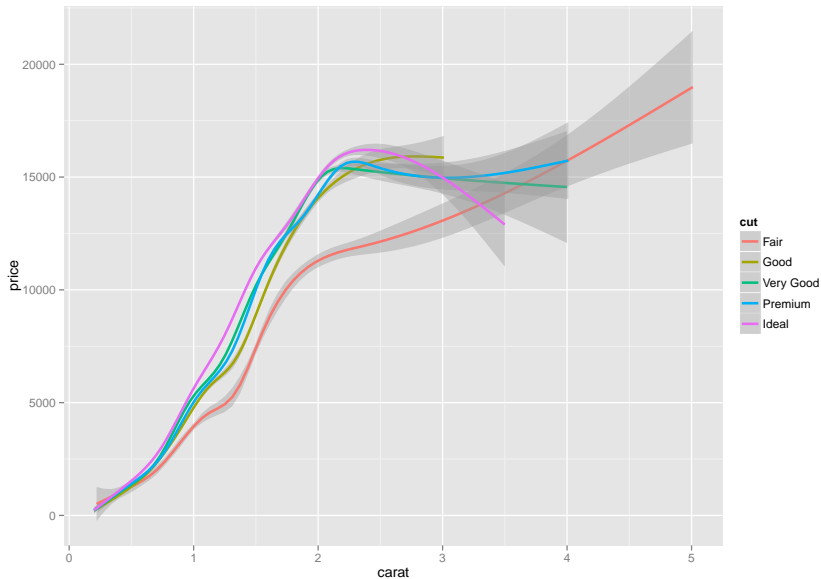
Revolution R is a parallel version that has been optimized for big data analytics. As testament to R's influence, Microsoft has recently purchased Revolution Analytics, and are rumoured to incorporate features that communicate with R into some of their applications.

... But why all of this interest?

Categorical Data Analysis



Non-linear Regression



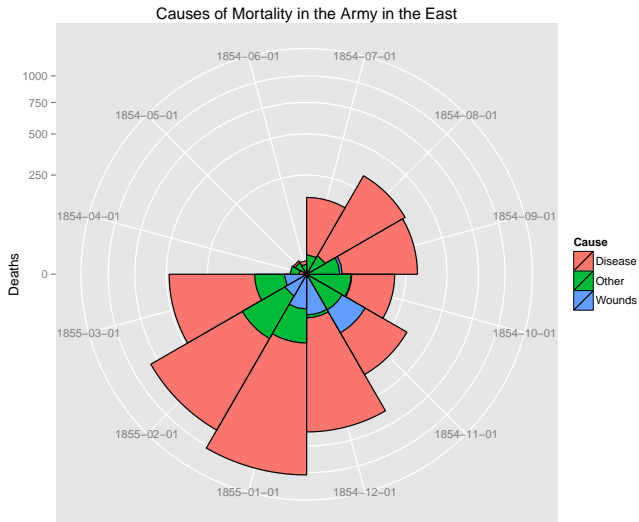
Spatial Mapping

London Cycle Hire Journeys

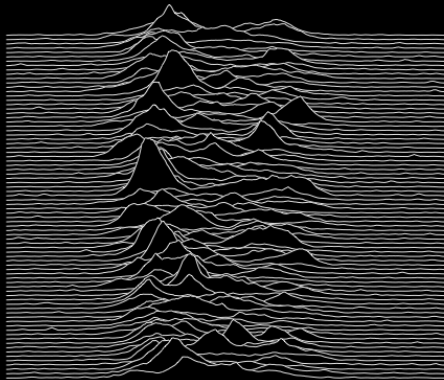
Thicker, yellower lines mean more journeys



Polar Coordinates



Unknown Pleasures



Coded by Markus Gesmann, magesblog.com

- A challenge and strength of R is that it is entirely syntax based.
- Even those not interested in programming new functions must become familiar with handling objects and calling functions by writing scripts.
- While R operates on spreadsheets, it is not “spreadsheet software”.
- Learning R will cause some frustration and require persistence, however it is well worth the effort!

“R is basically a big global family of fluffy altruistic people contributing to the goal of producing a versatile data analysis tool that is free for everyone to use.”

— Andy Field, *Discovering Statistics Using R*, p. 63

“R is basically a big global family of fluffy curmudgeonly altruistic people contributing to the goal of producing a versatile data analysis tool that is free for everyone to use.”

— Andy Field, *Discovering Statistics Using R*, p. 63

- Large and active community of users who post tutorials online and answer questions on message boards.
- It rarely takes more than a quick Google search to find the solution to most R problems.
- Most packages are well-documented and package authors are generally accessible via online forum or e-mail.

Useful Resources

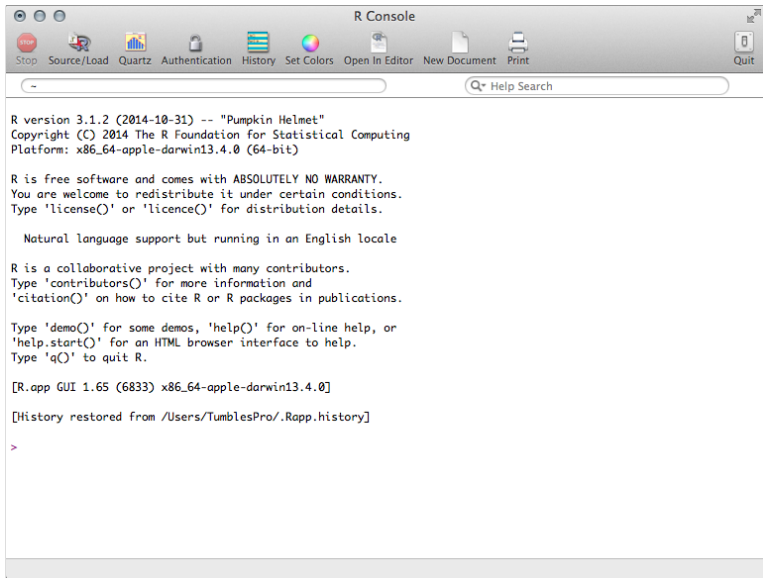
- Stack Exchange (<http://www.stackexchange.com>)
 - [Stack Overflow](#) (questions tagged [R])
 - [Cross Validated](#) (general statistics)
- R-Bloggers (www.r-bloggers.com)
- DataCamp (www.datacamp.com)
- Quick-R (www.statmethods.net) - examples and tips
- R-seek (rseek.org) - restricted search of R specific sites
- One Page R (onepager.togaware.com) - cheat sheets

Venting

- [Why 'R' is hard to learn](#)
- [The R Inferno](#)

- R is available for Windows, Mac OS X, and Linux.
- R is downloaded from CRAN (cran.r-project.org/), and installs:
 - **R.exe**: a text-based command-line version, run via command prompt
 - **RScript.exe**: an interface designed to be used in scripts and CMD batch files
 - **RGui.exe**: both a command-line interface and a barebones GUI.
- We highly recommend (and assume) you will install the RStudio IDE promptly afterwards, which is also freely available for Windows, Mac OS X, and Linux (from rstudio.com).

The R Environment: RGui.EXE



The RStudio Environment

Unfortunately the user environment that comes with R leaves a lot to be desired. Fortunately, some other fluffy altruistic people developed something better: RStudio.

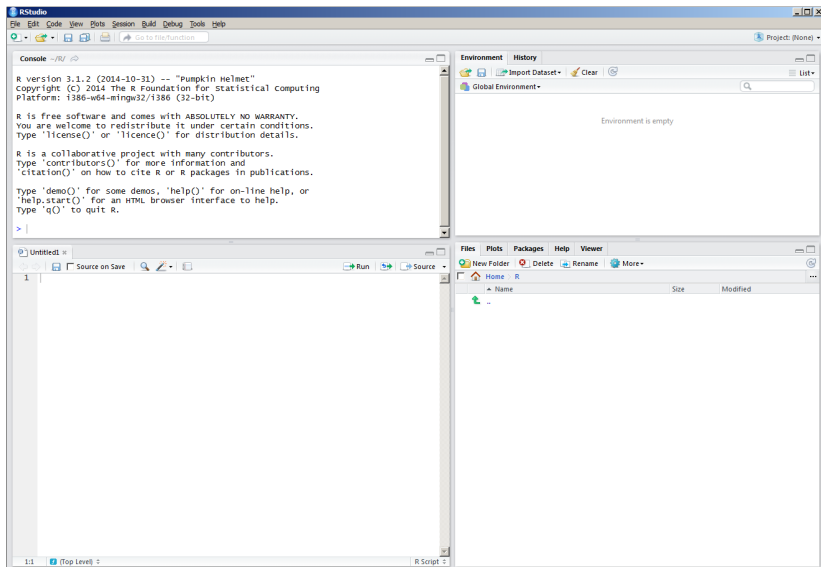
RStudio IDE is a powerful and productive user interface for R, currently on version 0.98.1103.

Tips for Installing R and RStudio

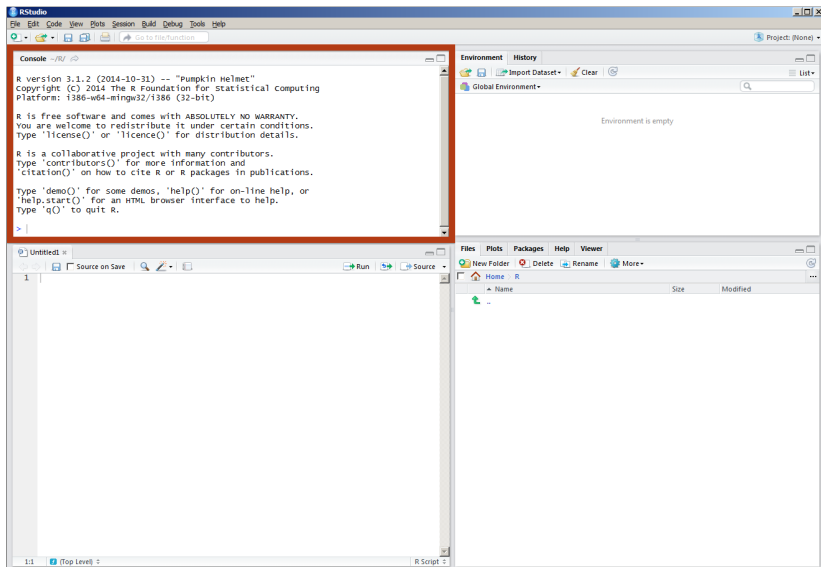
- When installing, the default options are acceptable.
- Upon launching RStudio, it can be useful to change some options (Tools -> Global Options)
- For example, set “Save Workspace to .RData on Exit” to Never
- Set Console to upper left corner, Source to lower left.

BONUS: [RStudio keyboard shortcuts cheatsheet](#) for even faster coding.

The RStudio Environment



The Console



The Console

The main window in R is called the Console. This is where R syntax is executed and you will see the results of the submitted code. We can interact with R directly through the Console window by typing commands at the `>` prompt and pressing `<return>`.

```
5 + 16  
## [1] 21
```

Using the Console, we can also create variables. Variables will be discussed more extensively later, but for now we can type the following lines into the console and press `<return>`.

```
x <- c(3, 5, 5, 7, 9)  
x  
## [1] 3 5 5 7 9
```


The Console

The variable `x` is saved in the R Workspace, and it will still exist until it is deleted. You can check the contents of the Workspace by using the command `ls()`.

```
ls()  
## [1] "x"
```

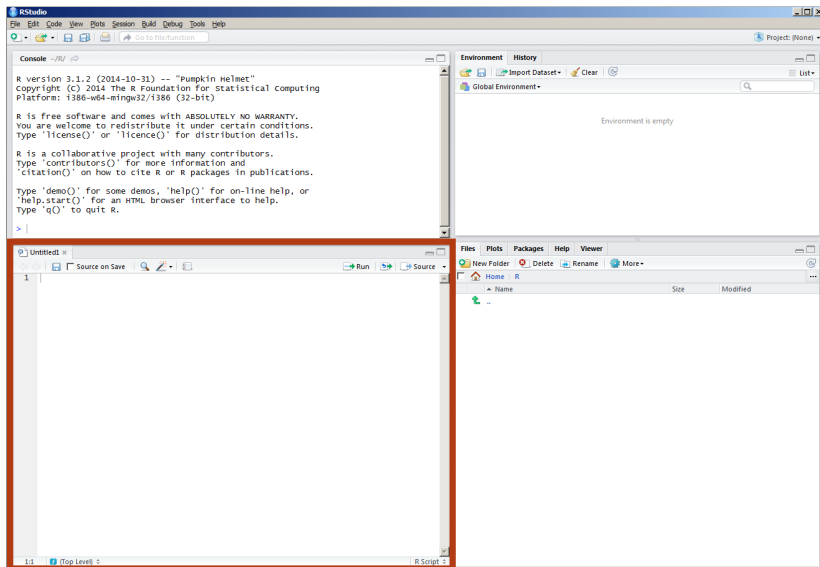
Assigning Values in R

Note: We typically use `<-` to assign variables rather than `=`. In most cases you can use `=`, but you will sometimes get some strange behaviour. As a rule it is safer to use `<-` when assigning values to variables.

R does not produce any output when you submit assignments. If you want to print the results you can add parentheses:

```
(x <- c(3, 5, 5, 7, 9))  
## [1] 3 5 5 7 9
```

The Source Window (or Editor)



The Source Window (or Editor)

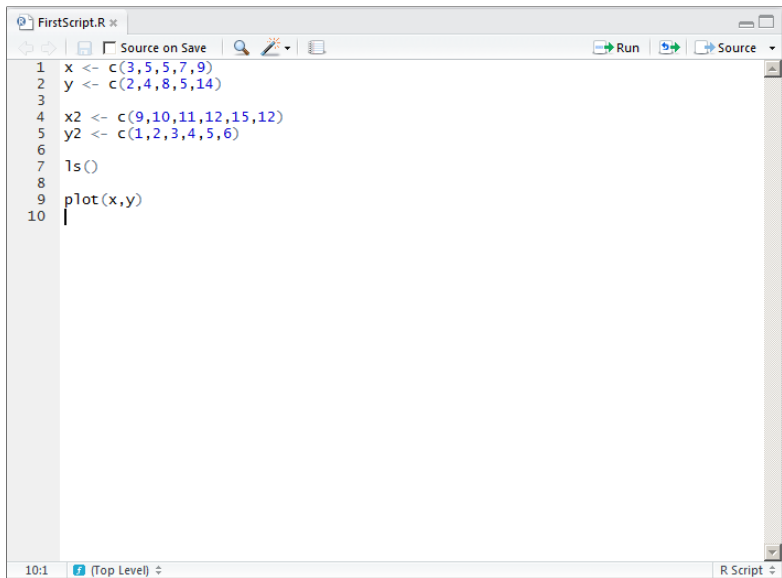
Upon launching RStudio, you may have to create a new script via `File -> New File -> R Script` or open a pre-existing script via `File -> Open File`.

.R files are plain text: can be modified with any text editor, cross-platform.

In the Editor, we can write a series of R commands that we would like to run. Once in the editor window, code can be run line by line by pressing `<ctrl> + R` or `<ctrl> + Enter`. Multiple lines can be run simultaneously by highlighting them and using the same shortcut.

We can also highlight the desired code and select 'Run' from the top right of the Editor.

The Source Window (or Editor)



The screenshot shows the R Source Window (Editor) with a script named "FirstScript.R". The script contains the following R code:

```
1 x <- c(3,5,5,7,9)
2 y <- c(2,4,8,5,14)
3
4 x2 <- c(9,10,11,12,15,12)
5 y2 <- c(1,2,3,4,5,6)
6
7 ls()
8
9 plot(x,y)
10 |
```

The window includes a toolbar with icons for navigation, saving, and running. The status bar at the bottom indicates the current position is 10:1 and the file is at the Top Level.

Coding Conventions

Basic structure of R syntax:

```
out <- t.test(y ~ x, data = dat, alternative = "less")
```

- the **function** we are calling is `t.test()`
- `y ~ x` is an example of R's **formula interface**, in which only variable names are provided
- `data` and `alternative` are **arguments** for `t.test()`
- `dat` and `"less"` are **objects** that are matched to their **arguments**
- output from `t.test()` is being returned and stored in another object, `out`, via `<-` (the **assignment operator**).

NOTES:

- No use of `"` to designate end of line (can use `;`, but is often unnecessary).
- No inclusion of anything like `EXECUTE`.

Comments in R

```
# The # symbol is used to demark comments in R
x <- c(3, 5, 5, 7, 9)
# y <- c(1, 2, 3, 4, 5)
x # print X
## [1] 3 5 5 7 9
y # print Y
## Error in eval(expr, envir, enclos): object 'y' not found
```

Commenting code inline is **highly encouraged**, and doubly so for collaborative projects.

Note: We will soon introduce a better way of managing your comments and code through the use of **Markdown**.

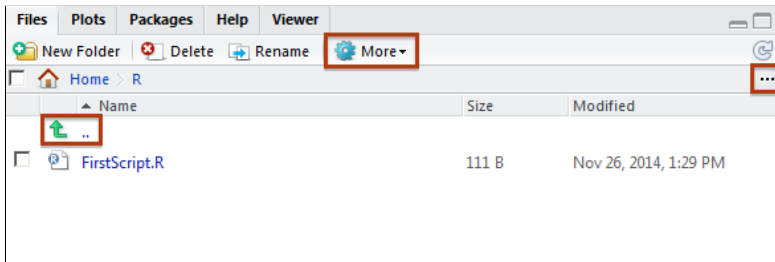
Spacing in R Syntax

R is almost always oblivious to spacing. There are a few notable exceptions, but they tend to come out in more advanced features (e.g., using `if` statements and embedded functions). The following will all produce the same results, so feel free to use whatever spacing structure you find most aesthetically appealing.

```
x <- c(3, 5, 5, 7, 9)
x <- c(3,5,5,7,9)
x <- c (3,5,5,7,9)
x <- c( 3, 5, 5, 7, 9 )
```

If you want to follow some leading sources on coding style, [Google](#) and [Hadley Wickham](#) (a leading R programmer) both have R-specific style guides you can follow.

File Manager



- Navigate by clicking on folders or the double dots/green arrow to go up a level.
- Use the ellipsis to open a file browser.
- Under **More**, lives the very useful Set As Working Directory command.

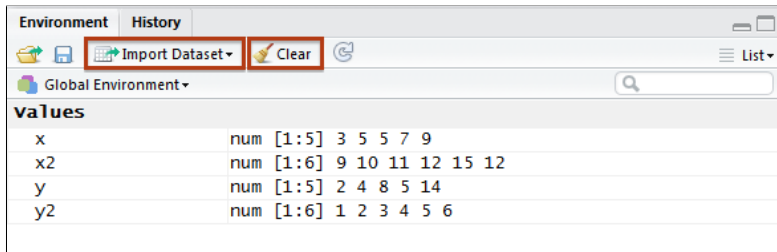
- What you see in the file pane shows R's **working directory**.
- This is very important to keep track of, as this is the directory R will look in when you specify a filename!

Tips and Tricks:

- If you open RStudio using the default shortcut, the working directory is set under **Tools -> Global Options**
- This usually is something like:
C:/Users/your.name/Documents/R/
- However, if you open RStudio by double clicking an .R or .Rmd file, the working directory will *automatically* be set to the same directory the file is in.
- The working directory can also be manually modified via `setwd()`.

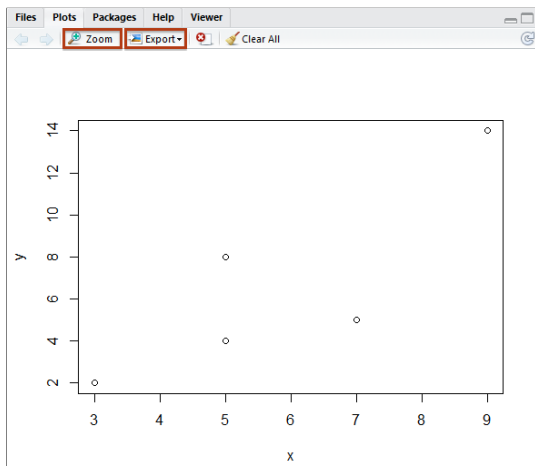
It is highly recommended to keep a project's datafiles and R scripts in the same directory, especially if you plan to knit a report.

Environment



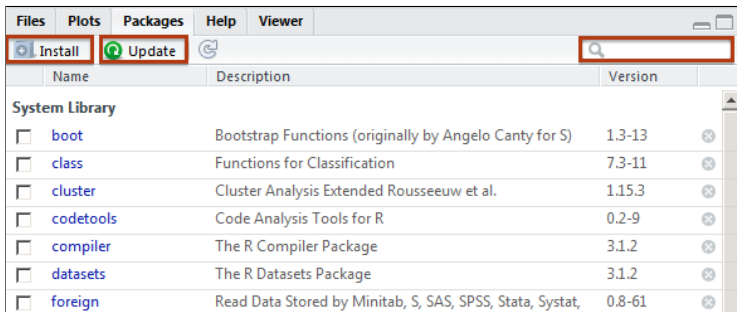
- Live updating, shows everything currently in the workspace.
- Headings for **Data**, **Values**, and **Functions**
- Same basic information as using `ls()`, but also denotes variable type, length, and the first 10 elements.
- Can click on an object to see what it contains.
- Two very handy commands: `Import Dataset` and `Clear` to clean the workspace.

Plot Panel



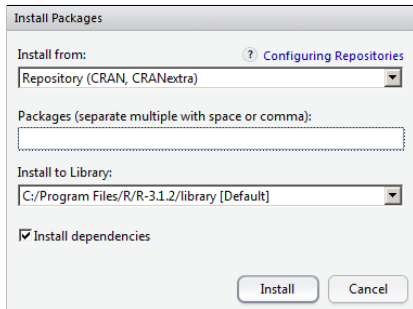
- Unlike base R, we can navigate through many plots via arrows.
- Use Zoom command to see plot in higher resolution.
- Use Export to save current plot in a variety of formats.

Packages



- Base functions and packages are shipped with R.
- Many packages exist and can be installed from:
 - CRAN or locally - via Install
 - GitHub (e.g., [Trending R Repositories](#))
 - Other Repositories (e.g., [Bioconductor.org](#), [rOpenSci.org](#))
- Use Update periodically to keep packages current.

Installing Packages




- Text input live updates with available CRAN packages
- Always leave `Install dependencies` checked
- Can also use: `install.packages("NAME", dep = TRUE)` in the console (my preferred method).

Installing Packages


`install.packages("polycor", dep = T)` installs: `polycor`, `mvtnorm`, and `sfsmisc`.

Note: Just because a package has been installed, doesn't mean that it is available for use! We need to load packages that we want to use.

<input type="checkbox"/>	<code>polycor</code>	Polychoric and Polyserial Correlations	0.7-8	
--------------------------	----------------------	--	-------	---

The package appears in our list, but is unchecked.

`library(polycor)` will make `polycor`'s functions available for use.

<input checked="" type="checkbox"/>	<code>polycor</code>	Polychoric and Polyserial Correlations	0.7-8	
-------------------------------------	----------------------	--	-------	---

In the console, use ? and ?? to search for a particular function name, and for pattern matching, respectively. For example:

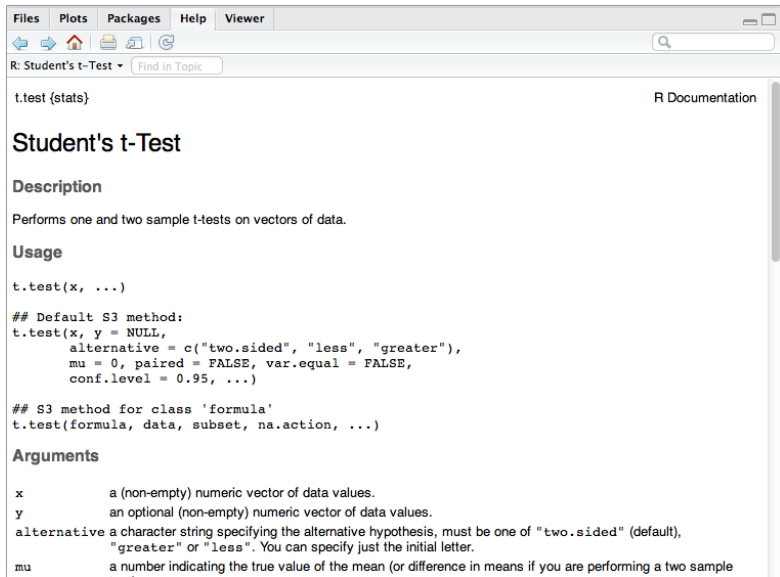
```
?t.test
```

```
??noise
```

Notes:

- The ? approach only searches loaded packages, and only for an exact match (?ttest gives an error)
- The ?? approach searches packages installed locally for your keyword, and they do not need to be loaded.

Getting Help



The screenshot shows the R Help Viewer window. The title bar includes tabs for Files, Plots, Packages, Help, and Viewer. Below the tabs is a search bar and a breadcrumb trail: R: Student's t-Test > Find in Topic. The main content area displays the documentation for the `t.test` function. It starts with the function signature `t.test(stats)` and the text "R Documentation". The title "Student's t-Test" is prominently displayed. Under the "Description" section, it states: "Performs one and two sample t-tests on vectors of data." The "Usage" section shows the function signature `t.test(x, ...)` followed by two sets of comments: one for the default S3 method and another for the S3 method for the 'formula' class. The "Arguments" section lists the parameters: `x` (a (non-empty) numeric vector of data values), `y` (an optional (non-empty) numeric vector of data values), `alternative` (a character string specifying the alternative hypothesis, must be one of "two.sided" (default), "greater" or "less". You can specify just the initial letter.), and `mu` (a number indicating the true value of the mean (or difference in means if you are performing a two sample

Files Plots Packages Help Viewer

R: Student's t-Test > Find in Topic

`t.test(stats)` R Documentation

Student's t-Test

Description

Performs one and two sample t-tests on vectors of data.

Usage

```
t.test(x, ...)
```

```
## Default S3 method:
t.test(x, y = NULL,
       alternative = c("two.sided", "less", "greater"),
       mu = 0, paired = FALSE, var.equal = FALSE,
       conf.level = 0.95, ...)
```

```
## S3 method for class 'formula'
t.test(formula, data, subset, na.action, ...)
```

Arguments

<code>x</code>	a (non-empty) numeric vector of data values.
<code>y</code>	an optional (non-empty) numeric vector of data values.
<code>alternative</code>	a character string specifying the alternative hypothesis, must be one of "two.sided" (default), "greater" or "less". You can specify just the initial letter.
<code>mu</code>	a number indicating the true value of the mean (or difference in means if you are performing a two sample

R help files generally follow a template:

- **Function Name {Package}**: e.g. - `t.test {stats}`
- **Description**: Brief idea of what the function does.
- **Usage**: Function call, with defaults indicated
 - `t.test(x, y = NULL, alternative = c("two.sided", "less", "greater"), mu = 0, paired = FALSE, var.equal = FALSE, conf.level = 0.95, ...)`
- **Arguments**: Detailed list of function's parameters.
- **Details**: Generally technical details about function's behaviour.
- **Value**: Description of the output generated from the function.
- **See Also**: A list of other functions that you may find useful.
- **Examples**: The best part of a help file!

- `demo()`: brings up a list of demonstrations for a set of tasks.
 - Base R only has 16, but many packages have additional demos.
 - All available: `demo(package=.packages(all.available=TRUE))`
 - Example: `demo(lm.glm, package="stats")`
- `browseVignettes()` brings up a list of vignettes.
 - These are short documents on a particular task, usually feature R syntax, output, and PDFs that include all output.
 - Browse from available demos from a particular package:
`browseVignettes(package="grid")`
 - Example: `vignette("rotated", package="grid")`

R + RStudio + Markdown = Dynamic Documents

.R files (like SPSS syntax files) only ever have the commands and comments that you decide to include in them.

- No output!
- No plots!
- No narrative!

A substantially more powerful method of working is to work with **R Markdown (.Rmd) files** instead of R scripts (.R). The benefits of working with Markdown are:

- Flexibility in text formatting (titles, subtitles, quotes)
- The ability to include lists, links, and images
- The ability to output the document to HTML, PDF, even DOCX!
- But, most of all: the ability to include raw R code (“R chunks”) and R output in the same document!

R + RStudio + Markdown = Dynamic Documents

.R files (like SPSS syntax files) only ever have the commands and comments that you decide to include in them.

- No output!
- No plots!
- No narrative!

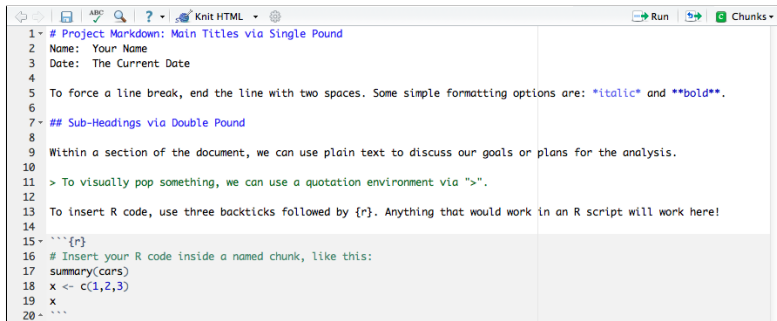
A substantially more powerful method of working is to work with **R Markdown (.Rmd) files** instead of R scripts (.R). The benefits of working with Markdown are:

- Flexibility in text formatting (titles, subtitles, quotes)
- The ability to include lists, links, and images
- The ability to output the document to HTML, PDF, even DOCX!
- But, most of all: the ability to include raw R code (“R chunks”) and R output in the same document!

Reproducible research is the best research!

Markdown Example: .Rmd Code

Create a new Markdown file via File -> New File -> R Markdown.

A screenshot of the RStudio editor window. The top toolbar shows icons for navigation, saving, and running. The main editor area displays an R Markdown file with the following content:

```
1 # Project Markdown: Main Titles via Single Pound
2 Name: Your Name
3 Date: The Current Date
4
5 To force a line break, end the line with two spaces. Some simple formatting options are: italic and bold.
6
7 ## Sub-Headings via Double Pound
8
9 Within a section of the document, we can use plain text to discuss our goals or plans for the analysis.
10
11 > To visually pop something, we can use a quotation environment via ">".
12
13 To insert R code, use three backticks followed by {r}. Anything that would work in an R script will work here!
14
15 ```{r}
16 # Insert your R code inside a named chunk, like this:
17 summary(cars)
18 x <- c(1,2,3)
19 x
20 ```
```

Markdown Example: Knit .html File

Project Markdown: Main Titles via Single Pound

Name: Your Name

Date: The Current Date

To force a line break, end the line with two spaces. Some simple formatting options are: *italic* and **bold**.

Sub-Headings via Double Pound

Within a section of the document, we can use plain text to discuss our goals or plans for the analysis.

To visually pop something, we can use a quotation environment via ">".

To insert R code, use three backticks followed by {r}. Anything that would work in an R script will work here!

```
# Insert your R code inside a named chunk, like this:
```

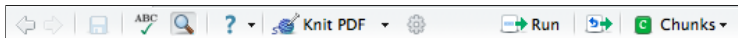
```
summary(cars)
```

```
##      speed      dist
##  Min.   : 4.0    Min.   : 2.00
##  1st Qu.:12.0    1st Qu.: 26.00
##  Median :15.0    Median : 36.00
##  Mean   :15.4    Mean   : 42.98
##  3rd Qu.:19.0    3rd Qu.: 56.00
##  Max.   :25.0    Max.   :120.00
```

```
x <- c(1,2,3)
x
```

```
## [1] 1 2 3
```

Markdown Tips and Tricks



- May need to run: `install.packages(knitr)` before you are able to knit successfully.
- Knit to PDF requires a TeX distribution installed.
- Look at the Markdown Quick Reference page to see more functionality options, such as:
 - Lists (unordered via asterisks, ordered via numbers)
 - Incorporate Links, Images, LaTeX Equations, Page Breaks, Tables.
- Give chunks descriptive names for easier navigation, e.g.:

```
``{r loaddata} vs. ``{r chunk1}
```

- Use the Chunks menu to run Previous/Current/Next/All Chunks.
- Incorporate Chunk options like: `eval`, `echo`, `cache`.

Basic R

Working with the Fundamental Object Types

Data Types

When we created `x` earlier, we saw that it was “numeric”. Objects in R have different types. It might seem cruel, but if you learn how to handle these types of objects you will be in a better position to correct problems with your code and manage data sets.

The most useful variable types to know about are:

- Numeric scalars/vectors
- Logical vectors
- Character vectors
- Matrices
- Lists
- Data frames

Numeric Scalars/Vectors

Basic unit of analysis. These objects are either a single element (a 'scalar'; e.g., `x <- 5` or `x <- "Rrr"`) or a series of elements collected together inside a single object (a vector; e.g., `x <- c(5, 10, 15)`).

We use the function `c()` which stands for 'combine' or 'concatenate' to create a vector.

```
x <- 5 # a scalar.  
x <- c(3, 5, 5, 7, 9) # combine numbers into a vector  
                      # via the function c().
```

We can access an entry in a vector using square brackets:

```
x[4] # Returns the 4th value of the vector x  
## [1] 7  
x[-4] # Returns all values of x but the 4th  
## [1] 3 5 5 9
```

Numeric Scalars/Vectors

Shortcuts: Using `:`, `seq()`, and `rep`.

```
(one2ten <- c(1:10))  
## [1] 1 2 3 4 5 6 7 8 9 10
```

```
(bytwos <- seq(from = 1, to = 10, by = 2))  
## [1] 1 3 5 7 9
```

```
(repeats <- rep(1:4, times = 2))  
## [1] 1 2 3 4 1 2 3 4  
(repeats <- rep(1:4, each = 2))  
## [1] 1 1 2 2 3 3 4 4
```

NOTE: Typically assignment is done *invisibly*. Surrounding an assignment with `(...)` is an R shorthand for `print(...)`, which evaluates the line and also prints whatever is returned.

Logical Vectors

Only contain **TRUE** and **FALSE** values.

Logical Vector Operations

- `==` exactly equals (NOTE: this is not an assignment!)
- `<` less than
- `>` greater than
- `<=` less than or equal to
- `>=` greater than or equal to
- `!=` not equal to

For example:

```
x < 7 # Identifies positions in x less than 7
## [1] TRUE TRUE TRUE FALSE FALSE
x != 5 # Identifies positions in x not equal to 5
## [1] TRUE FALSE FALSE TRUE TRUE
```

Logical Vectors

Filtering: A logical vector can be used to 'filter' or 'subset' a numeric vector based on some criteria. For example, if we wanted to select or filter all the cases from the vector `x` that are less than 7:

```
x # Print x ***
## [1] 3 5 5 7 9
x < 7 # Print positions less than 7 ***
## [1] TRUE TRUE TRUE FALSE FALSE
filter <- x < 7 # Save logical vector
x[filter] # Print values of x less than 7 ***
## [1] 3 5 5
x_less_7 <- x[filter]
x_less_7 # Print x_less_7 ***
## [1] 3 5 5
# Lines marked with `***` only print output to the console!
```

Logical Vectors

Filtering: A logical vector can be used to 'filter' or 'subset' a numeric vector based on some criteria. For example, if we wanted to select or filter all the cases from the vector `x` that are less than 7:

```
x # Print x ***  
## [1] 3 5 5 7 9  
x < 7 # Print positions less than 7 ***  
## [1] TRUE TRUE TRUE FALSE FALSE  
filter <- x < 7 # Save logical vector  
x[filter] # Print values of x less than 7 ***  
## [1] 3 5 5  
x_less_7 <- x[filter]  
x_less_7 # Print x_less_7 ***  
## [1] 3 5 5  
# Lines marked with `***` only print output to the console!
```

One-liner:

```
(x_less_7 <- x[x < 7])  
## [1] 3 5 5
```

Logical Vectors

We can also chain together such statements using:

- a vertical bar or pipe: | (not a capital i; found on backslash key) to signify OR
- an ampersand: & to signify AND
- an exclamation point: ! to signify NOT

This code will print (“return”) the values of the vector ‘x’ that are not exactly equal to 5 AND are greater than 3.

```
x[x != 5 & x > 3]  
## [1] 7 9
```

If we want the location of those values, we can use the function `which()`:

```
which(x != 5 & x > 3)  
## [1] 4 5
```

Character Vectors

Like numeric scalars/vectors but for non-numeric data (strings).

```
y <- "abc"  
y  
## [1] "abc"
```

```
class(y)  
## [1] "character"  
y[1] # the first element of y  
## [1] "abc"  
y[2] # the second element of y  
## [1] NA
```


Character Vectors

Like numeric vectors, y can be longer than a single element:

```
y <- c("abc", "de", "123", "")
y
## [1] "abc" "de"  "123" ""
y[1]
## [1] "abc"
y[2]
## [1] "de"
y[3]
## [1] "123"
class(y[3]) # even though this only has numbers it is not numeric!
## [1] "character"
```

Character Vectors

There are many ways we can manipulate character vectors in R. For example:

```
substring(y, first = 1, last = 2)
## [1] "ab" "de" "12" ""
# Returns the first two characters in each entry of 'y'

# Combine strings:
a <- 9
paste0("This is a number I like: ", a, ". It's a good one, huh?")
## [1] "This is a number I like: 9. It's a good one, huh?"
paste0(y, " is the ", 1:4, " entry in character vector y.")
## [1] "abc is the 1 entry in character vector y."
## [2] "de is the 2 entry in character vector y."
## [3] "123 is the 3 entry in character vector y."
## [4] " is the 4 entry in character vector y."
# this even works on vectors!
```

Advanced: Character vectors can be manipulated with regular expressions using base functions `grep()` and `sub()`, or the functions in the `stringr` package.

Basic R Exercises, Pt. 1

We are now ready to attempt the Exercise questions pertaining to "Numeric and Logical Vectors".

Factors are a special type of character vector that are extremely useful in psychological research, and are used to code categorical data.

We could use a numeric variable for this purpose:

```
group <- c(1,1,1,2,2,2,3,3,3, sample(c(1:3), 11, replace=TRUE))
class(group)
## [1] "numeric"
str(group)
##  num [1:20] 1 1 1 2 2 2 3 3 3 3 ...
```

... But it can lead to mistakes:

```
(group+2)
##  [1] 3 3 3 4 4 4 5 5 5 5 4 5 5 4 5 5 4 3 5 3
```

Also, the labels are not useful (1, 2, and 3, rather than “Group1”, “Group2”, and “Control”).

```
xtabs(~group)
## group
## 1 2 3
## 5 6 9
```

We can convert a numeric vector into a factor using the `factor()` command.

Note the `levels` argument for telling R which levels you want to keep (in most cases, you will keep all of them and this can be omitted), and the `labels` argument for specifying their names.

```
group1 <- factor(group)
class(group1)
## [1] "factor"
str(group1)
## Factor w/ 3 levels "1","2","3": 1 1 1 2 2 2 3 3 3 3 ...
xtabs(~group1)
## group1
## 1 2 3
## 5 6 9
```

Factors

When using levels and labels, order matters!

```
group2 <- factor(group,
                  levels = c(1, 2, 3),
                  labels = c("Group1", "Group2", "Control"))
xtabs(~group2)
## group2
## Group1 Group2 Control
##      5      6      9

group3 <- factor(group,
                  levels = c(3, 2, 1),
                  labels = c("Group1", "Group2", "Control"))
xtabs(~group3)
## group3
## Group1 Group2 Control
##      9      6      5
```

But besides making output easier to read and interpret, converting our variables to factors also protects us from silly mistakes, like:

```
(group1+2)
## Warning in Ops.factor(group1, 2): '+' not meaningful for factors
## [1] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
```


Factors

If we already have a factor and want to rename its levels, it is pretty easy:

```
levels(group1)
## [1] "1" "2" "3"
levels(group1) <- c("Group1", "Group2", "Control")
group1
## [1] Group1 Group1 Group1 Group2 Group2 Group2 Control Control
## [9] Control Control Group2 Control Control Group2 Control Control
## [17] Group2 Group1 Control Group1
## Levels: Group1 Group2 Control
```

```
xtabs(~group1)
## group1
## Group1 Group2 Control
##      5      6      9
```

Factors typically are sorted alphabetically by functions (with the first group often used as the reference group) since factor levels are considered nominal.

One way of circumventing that behaviour is to make the factored “ordered”. This can be done either by setting `ordered = TRUE` when calling `factor`, or calling `ordered()` on the factor variable directly.

Matrices

Two-dimensional vectors: all elements must of the same type.

Matrices can be constructed directly via `matrix()`, which has the following defaults: `matrix(data, nrow, ncol, byrow=FALSE)` - data is filled in column-wise!

```
mat <- matrix(data = c(1,2,3,4,5,6,7,8,9,10,11,12), nrow = 3)
mat
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

Matrices

We can also stack vectors via `rbind()` (which binds vectors together by row) or `cbind()` (bind by column).

```
mat1 <- rbind(c(1, 2, 3, 4), c(3, 4, 5, 6), c(5, 6, 7, 8))
mat1
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    3    4    5    6
## [3,]    5    6    7    8
mat2 <- cbind(c(1, 2, 3, 4), c(3, 4, 5, 6), c(5, 6, 7, 8))
mat2
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
## [3,]    3    5    7
## [4,]    4    6    8
```

Matrices

Use **double subscripts** to navigate matrices.

```
mat1[1, 3] # first row, third column  
## [1] 3  
mat1[2, ] # second row, all columns  
## [1] 3 4 5 6  
mat1[ , 3] # all rows, third column  
## [1] 3 5 7
```

Matrices

Use **double subscripts** to navigate matrices.

```
mat1[1, 3] # first row, third column  
## [1] 3  
mat1[2, ] # second row, all columns  
## [1] 3 4 5 6  
mat1[ , 3] # all rows, third column  
## [1] 3 5 7
```

This can also be used to change values:

```
mat1[2, ] <- mat1[2, ] + 4
```

```
mat1  
##      [,1] [,2] [,3] [,4]  
## [1,]    1    2    3    4  
## [2,]    7    8    9   10  
## [3,]    5    6    7    8
```

Labeling a Matrix: For aesthetic purposes, we can label our rows and columns via `rownames()` and `colnames()`. This brings a matrix closer to a data frame where each row might pertain to a particular person, and each column a variable we have measured.

```
rownames(mat1) <- c("Row1", "Row2", "Row3")  
colnames(mat1) <- c("Col1", "Col2", "Col3", "Col4")
```

```
mat1
```

##		Col1	Col2	Col3	Col4
##	Row1	1	2	3	4
##	Row2	7	8	9	10
##	Row3	5	6	7	8

Basic R Exercises, Pt. 2

We are now ready to attempt the Exercise questions pertaining to "Matrices".

Lists

Lists are like a vector but each of its contents can be items of different data types. Many R functions return lists, so familiarity with them is important!

```
a <- 5
y <- c("abc", "de", "123", "")
iamalist <- list(a = a, yvar = y, mat = matrix(1:6, 2, 3))
iamalist
## $a
## [1] 5
##
## $yvar
## [1] "abc" "de"  "123" ""
##
## $mat
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

We can look at their structure to get a sense of what they contain:

```
str(iamalist)
## List of 3
## $ a : num 5
## $ yvar: chr [1:4] "abc" "de" "123" ""
## $ mat : int [1:2, 1:3] 1 2 3 4 5 6
```

Navigate lists either with square brackets and an index location, or using a dollar sign and the vector's name:

```
iamalist[[1]]  
## [1] 5  
  
iamalist$yvar  
## [1] "abc" "de"  "123" ""  
  
iamalist[[3]][1, 2]  
## [1] 3
```

NOTE: When we navigate lists with square brackets, we use the double square bracket notation: `[[...]]`. This is because using `[...]` returns a list object, while `[[...]]` returns the vector in its original type.

List output from `t.test()`:

```
t.test(iamalist$mat[1, ], iamalist$mat[2, ])
##
##  Welch Two Sample t-test
##
## data:  iamalist$mat[1, ] and iamalist$mat[2, ]
## t = -0.6124, df = 4, p-value = 0.5734
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -5.533916  3.533916
## sample estimates:
## mean of x mean of y
##          3          4
```

Saved as an object:

```
out <- t.test(iamalist$mat[1, ], iamalist$mat[2, ])
str(out)
## List of 9
## $ statistic : Named num -0.612
## ..- attr(*, "names")= chr "t"
## $ parameter : Named num 4
## ..- attr(*, "names")= chr "df"
## $ p.value : num 0.573
## $ conf.int : atomic [1:2] -5.53 3.53
## ..- attr(*, "conf.level")= num 0.95
## $ estimate : Named num [1:2] 3 4
## ..- attr(*, "names")= chr [1:2] "mean of x" "mean of y"
## $ null.value : Named num 0
## ..- attr(*, "names")= chr "difference in means"
## $ alternative: chr "two.sided"
## $ method : chr "Welch Two Sample t-test"
## $ data.name : chr "iamalist$mat[1, ] and iamalist$mat[2, ]"
## - attr(*, "class")= chr "htest"
```

If our output is saved as a list object, we can access elements of it just like we did before:

```
out$statistic # retrieve output by name
##           t
## -0.6123724
out[[3]]      # or via double square brackets
## [1] 0.5733923
```

Data Frames are the most common object type for data analysis in R. They are similar to matrices in many ways except for one large difference: each column can hold a different object type.

For example, our first column may pertain to an identification code (character), followed by age (numeric), followed by gender (character, factor), and so on.

To combine vectors of the **same length** into a dataframe use `data.frame()`, e.g.: `data.frame(var1, var2, var3)`

Data Frames

More typically, we obtain data directly by it in from an external file. Data can be read in various ways, but the most reliable is to convert to CSV and then use `read.csv()`. For example:

```
dat <- read.csv("exerciseDat.csv")
head(dat)
```

##	id	gender	mathPre	readPre	group	score1	score2	pass
## 1	1	0	55.15	56.31	mnemonic	48.33	68.74	1
## 2	2	1	53.30	30.45	mnemonic	45.91	43.89	0
## 3	3	1	56.96	69.53	cram	55.98	84.43	1
## 4	4	1	55.64	38.25	mnemonic	52.81	38.24	0
## 5	5	1	48.14	60.25	mnemonic	53.70	54.46	1
## 6	6	0	49.07	61.91	cram	46.30	61.31	1

Data Frames

SPSS, SAS, and STATA datafiles are best loaded via the haven library:

```
#install.packages("haven")  
library("haven")  
dat1 <- read_dta("filename.dta")  
dat2 <- read_sas("filename.sas7bdat")  
dat3 <- read_spss("filename.sav")
```

Excel files can be read via read.xlsx() from the xlsx library:

```
dat4 <- read.xlsx("myfile.xlsx", sheetName = "Sheet1")
```

R objects can be saved as .RData files using save(), and are brought back using load(). These do not require assignment!

```
load("myfile.RData")
```

However, base R and many packages also come with datasets built-in to use as examples.

```
data() # lists all datasets from base R  
data(package=.packages(all.available = TRUE))  
data(package="ggplot2") # only datasets from ggplot2
```

The following example uses data from Cowles and Davis' survey on Volunteering for Psychological Research, published in the *British Journal of Social Psychology* in 1987, and included in John Fox's car package.

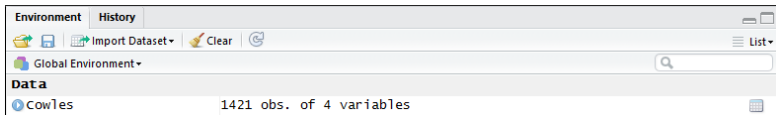
NOTE: If the car package is not installed, you must install it!

```
#install.packages("car", dep=T)
library(car)
```


Data Frames

```
data(Cowles) # loads Cowles
names(Cowles)
## [1] "neuroticism" "extraversion" "sex" "volunteer"
```

- Neuroticism: scale from Eysenck personality inventory
- Extroversion: scale from Eysenck personality inventory
- Sex: factor with 2 levels (female, male)
- Volunteer: factor with 2 levels (yes, no)



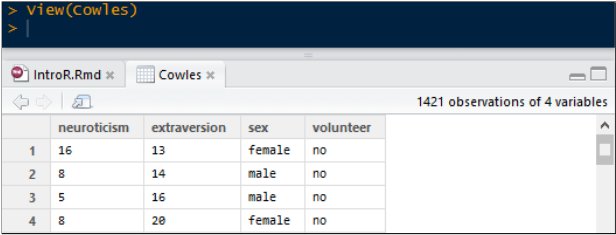
Data Frames

Data	
• Cowles	1421 obs. of 4 variables 
neuroticism :	int 16 8 5 8 9 6 8 12 15 18 ...
extraversion:	int 13 14 16 20 19 15 10 11 16 7 ...
sex :	Factor w/ 2 levels "female","male": 1 2 2 1 2 2 1 2 2 2 ...
volunteer :	Factor w/ 2 levels "no","yes": 1 1 1 1 1 1 1 1 1 1 ...

Variable list in RStudio's Environment Window

Data Frames

```
> View(Cowles)
> |
```



1421 observations of 4 variables

	neuroticism	extraversion	sex	volunteer
1	16	13	female	no
2	8	14	male	no
3	5	16	male	no
4	8	20	female	no

Using `View(Cowles)` opens a new tab to see data

After importing a dataset, it is good to make sure it was imported correctly:

- Use `head()`, `tail()`, and `str()` to understand dataset's structure
- Use `summary()` to get an overview of the descriptive statistics across all variables
- The `describe()` function from the `psych` library provides a more detailed summary
- Use `complete.cases()` if you want to filter out cases with missingness

Data Frames

```
head(Cowles)
```

##	neuroticism	extraversion	sex	volunteer
## 1	16	13	female	no
## 2	8	14	male	no
## 3	5	16	male	no
## 4	8	20	female	no
## 5	9	19	male	no
## 6	6	15	male	no

```
tail(Cowles, n = 8)
```

##	neuroticism	extraversion	sex	volunteer
## 1414	4	20	male	yes
## 1415	7	7	male	yes
## 1416	4	10	female	yes
## 1417	5	10	male	yes
## 1418	8	4	female	yes
## 1419	8	8	male	yes
## 1420	19	20	female	yes
## 1421	15	20	male	yes

Data Frames

```
str(Cowles)
## 'data.frame':    1421 obs. of  4 variables:
## $ neuroticism : int  16 8 5 8 9 6 8 12 15 18 ...
## $ extraversion: int  13 14 16 20 19 15 10 11 16 7 ...
## $ sex         : Factor w/ 2 levels "female","male": 1 2 2 1 2 2 1 2 2 2 ...
## $ volunteer   : Factor w/ 2 levels "no","yes": 1 1 1 1 1 1 1 1 1 1 ...
mean(Cowles$extraversion)
## [1] 12.37298
var(Cowles$extraversion)
## [1] 15.16079
median(Cowles$extraversion)
## [1] 13
summary(Cowles)
##   neuroticism   extraversion      sex      volunteer
##   Min.   : 0.00   Min.   : 2.00  female:780   no :824
##   1st Qu.: 8.00   1st Qu.:10.00  male  :641   yes:597
##   Median :11.00   Median :13.00
##   Mean   :11.47   Mean   :12.37
##   3rd Qu.:15.00   3rd Qu.:15.00
##   Max.   :24.00   Max.   :23.00
```

Data Frames

```
table(Cowles$sex)
##
## female    male
##      780    641
table(Cowles$sex, Cowles$volunteer)
##
##                no yes
##   female 431 349
##   male   393 248
```

How does R think about data?

```
dim(Cowles)
## [1] 1421    4
Cowles[40, 1] # 40th Case from "neuroticism"
## [1] 5
Cowles$neuroticism[40]
## [1] 5
Cowles[1:8, 2] # First 8 cases from "extraversion"
## [1] 13 14 16 20 19 15 10 11
Cowles$extraversion[1:8]
## [1] 13 14 16 20 19 15 10 11
```

Data Frames

Manipulating Variables: Data frames are basically lists with an additional restriction (each variable has the same length), and so we can access elements of the dataframe using \$, as we did with lists.

To overwrite a variable, simply use its name on the lefthand side of an assignment operator:

```
Cowles$extraversion <- NULL  
# this deletes extraversion from our dataset!
```

More often, we want to add a variable to our data frame. So long as we use a name not in use, it will be appended appropriately:

```
Cowles$sqrt.neuro <- sqrt(Cowles$neuroticism)  
names(Cowles)  
## [1] "neuroticism" "extraversion" "sex"          "volunteer"  
## [5] "sqrt.neuro"
```

Subsetting: Can use conditional expressions, like we did with vectors.

```
the.females <- Cowles[Cowles$sex == "female", ]  
dim(the.females)  
## [1] 780 5  
  
the.extraverted <- Cowles[Cowles$extraversion > 18, ]  
dim(the.extraverted)  
## [1] 73 5
```

Data Frames

In real applications, it is better done via `subset()` because the basic logical operators treat missing values in a somewhat surprising way (they are included in the subset!).

```
Cowles[1,3] <- NA # set first value of Gender column to missing
the.females1 <- Cowles[Cowles$sex == "female", ]
the.females2 <- subset(Cowles, sex == "female")
```

```
dim(the.females1) # with conditional expression
## [1] 780 5
dim(the.females2) # using subset()
## [1] 779 5
identical(the.females1, the.females2)
## [1] FALSE
```

Data Frames

Converting a character variable into a factor:

```
test.factor <- sample(c("Cat","Dog","Alligator"), size = 30, replace = TRUE)
```

```
test.factor
```

```
## [1] "Dog"      "Dog"      "Dog"      "Dog"      "Cat"
## [6] "Alligator" "Cat"      "Dog"      "Cat"      "Alligator"
## [11] "Dog"      "Cat"      "Alligator" "Dog"      "Cat"
## [16] "Cat"      "Dog"      "Dog"      "Alligator" "Alligator"
## [21] "Alligator" "Alligator" "Alligator" "Cat"      "Dog"
## [26] "Cat"      "Dog"      "Alligator" "Alligator" "Dog"
```

```
str(test.factor)
```

```
## chr [1:30] "Dog" "Dog" "Dog" "Dog" "Cat" "Alligator" ...
```

```
xtabs(~test.factor)
```

```
## test.factor
```

```
## Alligator      Cat      Dog
##           10      8      12
```

Data Frames

```
test.factor1 <- factor(test.factor,
                        levels = c("Alligator", "Cat", "Dog"),
                        labels = c("Alligator", "Cat", "Dog"))
str(test.factor1)
## Factor w/ 3 levels "Alligator","Cat",...: 3 3 3 3 2 1 2 3 2 1 ...
xtabs(~test.factor1)
## test.factor1
## Alligator      Cat      Dog
##          10         8      12
```

```
test.factor2 <- factor(test.factor)
str(test.factor2)
## Factor w/ 3 levels "Alligator","Cat",...: 3 3 3 3 2 1 2 3 2 1 ...
xtabs(~test.factor2)
## test.factor2
## Alligator      Cat      Dog
##          10         8      12
```

Converting a continuous variable into a categorical factor:

```
Cowles$ext.factor <- cut(Cowles$extraversion, 3)
str(Cowles$ext.factor)
## Factor w/ 3 levels "(1.98,9]","(9,16]",...: 2 2 2 3 3 2 2 2 2 1 ...
levels(Cowles$ext.factor)
## [1] "(1.98,9]" "(9,16]" "(16,23]"
levels(Cowles$ext.factor) <- c("Low", "Medium", "High")
str(Cowles$ext.factor)
## Factor w/ 3 levels "Low","Medium",...: 2 2 2 3 3 2 2 2 2 1 ...
```


Tips and Tricks:

- When importing with `read.csv()`, the following flags may be useful:
 - `header = TRUE` (variable names included in first row)
 - `sep = ""` (useful if dataset uses odd delimiter)
 - `na.strings = c("", " ", ".", "NA")`
 - `stringsAsFactors = FALSE`
 - `fileEncoding = "UTF-8-BOM"`
- If you have transformed variables or created new ones, you might want to save the final datafile.
 - Use: `write.csv(dataframe, "dataframe.csv", row.names = FALSE)`
 - Use `getwd()` to see where R is saving output to.

Basic R Exercises, Pt. 3

We are now ready to attempt the Exercise questions pertaining to "Factors and Data Frames".

Intermediate R

Programming Basics

Control Structures

IF statements evaluate a conditional statement, and return an expression if the value is true.

```
if (cond) expr

if (cond) expr1 else expr2

if (cond) {
  expr1
} else {
  expr2
}
```

Example:

```
x <- 5
if (x == 5) print("X is equal to 5")
## [1] "X is equal to 5"
if (x == 6) print("X is 6") else print("X is not 6")
## [1] "X is not 6"
```

Control Structures

FOR statements: Loop over a vector and perform an expression for each value.

```
for (var in seq) expr
```

Example:

```
for (i in 1:5) print(i)
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

Control Structures

WHILE statements: Loop over elements so long as the condition is TRUE.

```
while (cond) expr
```

Example:

```
x <- 1
while(x < 8) {
  print(paste("The value of X is:", x))
  x <- x + 1
}
## [1] "The value of X is: 1"
## [1] "The value of X is: 2"
## [1] "The value of X is: 3"
## [1] "The value of X is: 4"
## [1] "The value of X is: 5"
## [1] "The value of X is: 6"
## [1] "The value of X is: 7"
```

Control Structures

SWITCH statements: Return different values depending on input.

```
switch(expr, ...)
```

Example:

```
centre <- function(x, type) {  
  switch(type,  
    mean = mean(x),  
    median = median(x),  
    trimmed = mean(x, trim = .1))  
}
```

```
centre(Cowles$extraversion, "mean")  
## [1] 12.37298  
centre(Cowles$extraversion, "median")  
## [1] 13  
centre(Cowles$extraversion, "trimmed")  
## [1] 12.43448
```

Control Structures

IFELSE statements: Similar to basic `if()` but vectorized. These are very useful for recoding data, and can be nested.

```
ifelse  
ifelse(test, yes, no)
```

Example:

```
x <- 1:10  
ifelse(x %% 2 == 0, c("Value is even."), c("Value is odd."))  
## [1] "Value is odd." "Value is even." "Value is odd." "Value is even."  
## [5] "Value is odd." "Value is even." "Value is odd." "Value is even."  
## [9] "Value is odd." "Value is even."
```


Intermediate R

Examples of Applications

Splitting Samples

Creating Exploratory vs. Confirmatory Samples

```
round(prop.table(table(Cowles$sex, Cowles$volunteer))*100, 1)
##
##           no  yes
##  female 30.3 24.6
##   male  27.7 17.5

library('devtools')
source_gist("https://gist.github.com/mrdwab/6424112")

Cowles.EFA <- stratified(df = Cowles, group = c("sex", "volunteer"), size = .5)
Cowles.CFA <- Cowles[!rownames(Cowles) %in% rownames(Cowles.EFA), ]

round(prop.table(table(Cowles.EFA$sex, Cowles.EFA$volunteer))*100, 1)
##
##           no  yes
##  female 30.3 24.5
##   male  27.6 17.5
round(prop.table(table(Cowles.CFA$sex, Cowles.CFA$volunteer))*100, 1)
##
##           no  yes
##  female 30.2 24.6
##   male  27.7 17.4
```

Basic Statistical Analyses

The base **stats** package has many useful functions for basic statistical analyses, such as:

- `prop.test()`, `xtabs()` (cross-tabs and χ^2), and `t.test()`
- `cor()`, and `cor.test()`
- `anova()`, and `manova()`
- `lm()`, and `glm()` (linear and generalized linear models)
- `influence()` (calculate DFBETAS, DFFITS, covariance ratios, Cook's Distance, hat-values)
- `cov2cor()` (convert covariance matrix to correlation matrix)
- `p.adjust()` (adjust p -values for multiple comparisons)
- `residuals()` (extract model residuals)

Basic Statistical Analyses

Most statistical functions utilize the “formula interface”, which refers to a structured string with the following properties:

- the response variable(s) appear on the left-hand side of the equation
- the predictors or indicator variable(s) appear on the right-hand side of the equation, separated by +
- the two sides are separated by a tilde (~)

All functions that use the formula interface have a `data =` argument, which is very useful. For example, compare:

```
response ~ var1 + var2, data = data.name
```

```
data.name$response ~ data.name$var1 + data.name$var2
```

Basic Statistical Analyses

Additionally, these characters have specific behaviours:

- ***** denotes **all possible interactions** (e.g., `sex * volunteer` includes main effects for sex, volunteer, as well as their crossing, while adding extraversion would include the three-way interaction, and all second order interactions)
- **^** is used to incorporate variable crossings to the **specified degree** (e.g. `extraversion^2` includes extraversion and extraversion^2)
- **:** is used to denote **particular crossings**. For example, `sex:volunteer` will add their interaction.
- **-** is used to **remove a particular effect** from the model (e.g., `sex * volunteer * extraversion - sex:extraversion`)

Advanced Tip: Adding Many Variables

```
xnam <- paste0("x", 1:25) # vector contains x1, ..., x25
(fmla <- as.formula(paste("y ~ ", paste(xnam, collapse= "+"))))
## y ~ x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9 + x10 + x11 +
##      x12 + x13 + x14 + x15 + x16 + x17 + x18 + x19 + x20 + x21 +
##      x22 + x23 + x24 + x25

xnam <- paste0(names(Cowles[1:3]))
(fmla <- as.formula(paste("volunteer ~ ", paste(xnam, collapse= "+"))))
## volunteer ~ neuroticism + extraversion + sex
```

- `paste0()` combines element-wise, but with no spaces between
- `paste()` combines element-wise, but with whitespace between elements
- `collapse` argument is used to define what separates pasted elements

Basic Statistical Analyses

Linear Regression:

```
out <- lm(neuroticism ~ sex * volunteer + extraversion, data = Cowles)
summary(out)
##
## Call:
## lm(formula = neuroticism ~ sex * volunteer + extraversion, data = Cowles)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -13.0378  -3.5584  -0.0378   3.7029  11.8950
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)    13.84484    0.45817   30.217 < 2e-16 ***
## sexmale        -1.72595    0.33474   -5.156 2.88e-07 ***
## volunteeryes    0.22142    0.34693    0.638  0.523
## extraversion   -0.13450    0.03295   -4.082 4.71e-05 ***
## sexmale:volunteeryes -0.15936    0.52032   -0.306  0.759
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 4.796 on 1415 degrees of freedom
## (1 observation deleted due to missingness)
## Multiple R-squared:  0.04429,    Adjusted R-squared:  0.04159
## F-statistic: 16.39 on 4 and 1415 DF,  p-value: 3.888e-13
```

Basic Statistical Analyses

The output from most analyses can be passed to other functions, and will return different things. For example, try:

```
print(out)      # brief summary
summary(out)    # expanded summary
anova(out)      # ANOVA table
coef(out)       # model coefficients
head(resid(out)) # model residuals
plot(out)       # regression diagnostic plots
```


Basic Statistical Analyses

Logistic Regression:

```
out <- glm(volunteer ~ sex + extraversion, data = Cowles, family = binomial)
summary(out)
##
## Call:
## glm(formula = volunteer ~ sex + extraversion, family = binomial,
##      data = Cowles)
##
## Deviance Residuals:
##      Min       1Q   Median       3Q      Max
## -1.3873  -1.0509  -0.9034   1.2522   1.6855
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept)  -1.02592    0.19181  -5.349 8.87e-08 ***
## sexmale      -0.24902    0.10932  -2.278  0.0227 *
## extraversion  0.06552    0.01417   4.622 3.79e-06 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##      Null deviance: 1932.4  on 1419  degrees of freedom
## Residual deviance: 1905.2  on 1417  degrees of freedom
## (1 observation deleted due to missingness)
## AIC: 1911.2
##
## Number of Fisher Scoring iterations: 4
```

Graphical Analyses

Let's use the exercise dataset, and make sure it is read in nicely:

```
dat <- read.csv("exerciseDat.csv", header=TRUE, stringsAsFactors=FALSE)
head(dat)
##   id gender mathPre readPre   group score1 score2 pass
## 1  1      0  55.15  56.31 mnemonic  48.33  68.74    1
## 2  2      1  53.30  30.45 mnemonic  45.91  43.89    0
## 3  3      1  56.96  69.53    cram  55.98  84.43    1
## 4  4      1  55.64  38.25 mnemonic  52.81  38.24    0
## 5  5      1  48.14  60.25 mnemonic  53.70  54.46    1
## 6  6      0  49.07  61.91    cram  46.30  61.31    1
str(dat)
## 'data.frame':    300 obs. of  8 variables:
##  $ id      : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ gender  : int  0 1 1 1 1 0 1 0 0 0 ...
##  $ mathPre : num  55.1 53.3 57 55.6 48.1 ...
##  $ readPre : num  56.3 30.4 69.5 38.2 60.2 ...
##  $ group   : chr   "mnemonic" "mnemonic" "cram" "mnemonic" ...
##  $ score1  : num  48.3 45.9 56 52.8 53.7 ...
##  $ score2  : num  68.7 43.9 84.4 38.2 54.5 ...
##  $ pass    : int  1 0 1 0 1 1 1 1 0 0 ...
```

Graphical Analyses

```
dat$gender <- factor(dat$gender, levels = c(0,1), labels = c("Female", "Male"))
dat$group <- factor(dat$group, levels = c("cram", "mnemonic", "none"),
                    labels = c("cram", "mnemonic", "none"))
dat$pass <- factor(dat$pass, levels = c(0,1), labels = c("Fail", "Pass"))
str(dat)
## 'data.frame':    300 obs. of  8 variables:
## $ id      : int  1 2 3 4 5 6 7 8 9 10 ...
## $ gender  : Factor w/ 2 levels "Female","Male": 1 2 2 2 2 1 2 1 1 1 ...
## $ mathPre : num  55.1 53.3 57 55.6 48.1 ...
## $ readPre : num  56.3 30.4 69.5 38.2 60.2 ...
## $ group   : Factor w/ 3 levels "cram","mnemonic",...: 2 2 1 2 2 1 1 2 2 1 ...
## $ score1  : num  48.3 45.9 56 52.8 53.7 ...
## $ score2  : num  68.7 43.9 84.4 38.2 54.5 ...
## $ pass    : Factor w/ 2 levels "Fail","Pass": 2 1 2 1 2 2 2 2 1 1 ...
xtabs(~ gender + group, data = dat)
##           group
## gender   cram mnemonic none
## Female    36         32    36
## Male     69         72    55
```

Graphical Analyses

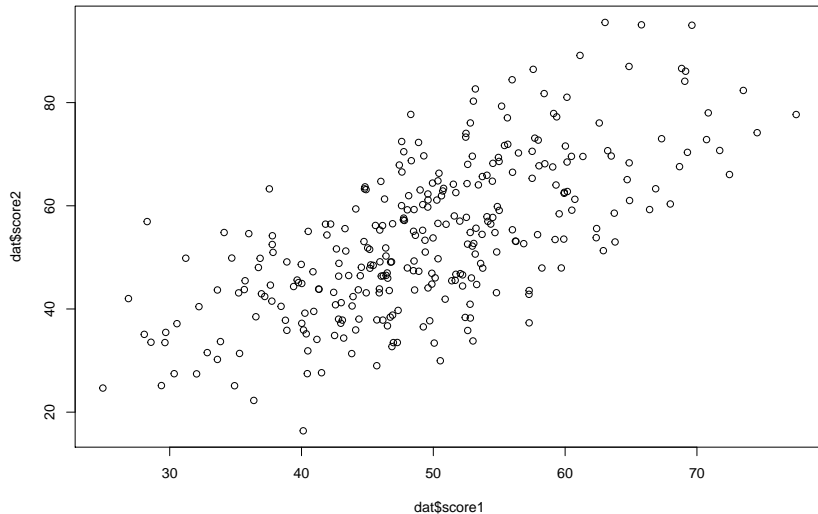
In R there are multiple ways to produce graphics. Two major approaches are using built-in functions (base), or using additional packages.

Advantages of base graphics:

- No need to include `library()` in your code
- Very quick to produce a plot (e.g., `hist(dat$mathPre)`)
- `plot()` is a **generic function**, meaning it will produce different plots depending on what it was passed.
 - For example: `plot(dat$score1, dat$score2)`
 - However, often this is not useful: `plot(dat$score1)`

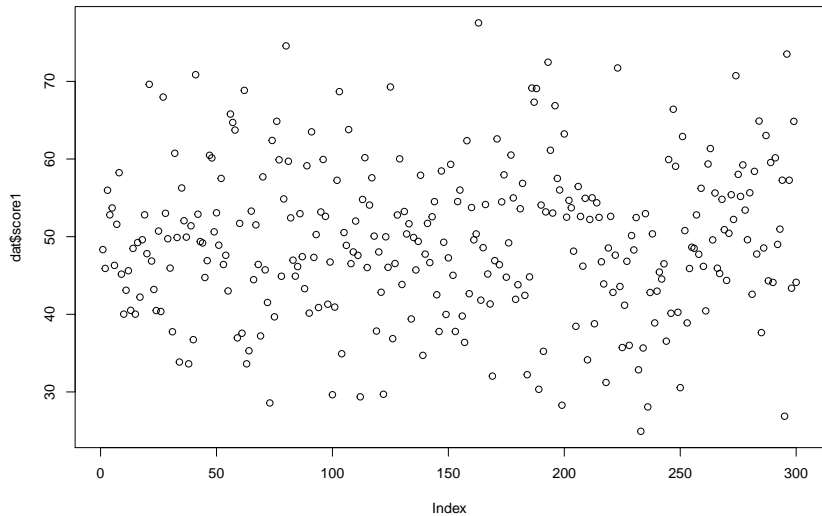
Base Graphics

```
plot(dat$score1, dat$score2)
```



Base Graphics

```
plot(dat$score1)
```



ggplot2 Fundamentals

A substantially more powerful package for plotting is Hadley Wickham's `ggplot2`.

General principals for using `ggplot2`:

- 1 Define the data you want to plot and create a plot template object with `ggplot()`
- 2 Specify the aesthetics of the shapes that will be used to represent the data with `aes()`
- 3 Specify the graphical shapes (geoms) that will be used to view the data
 - Add them with the appropriate function
 - e.g., `geom_point()` or `geom_line()`
- 4 Call the object to render and view it

ggplot2: Q-Plots

```
#install.packages("ggplot2", dep=T)
library(ggplot2)
```

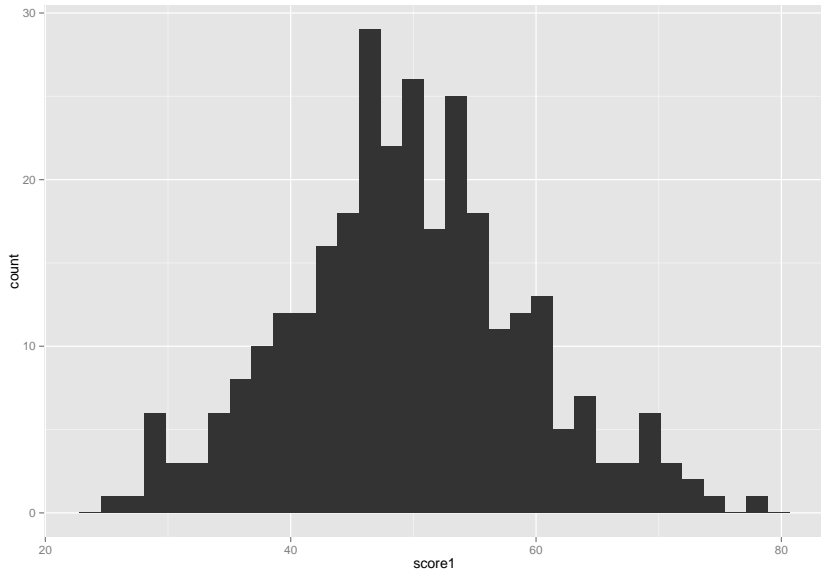
For starting out with ggplot2, we will use `qplot()`, which is a convenience function that stands for *quick* plot.

The primary arguments for `qplot()` are:

- `x` - values we want on the x-axis
- `y` - values we want on the y-axis (not necessary, depending on plot)
- `data` - dataset
- `geom` - geometric object to use ("point" if `x` and `y` are given, "histogram" if not)

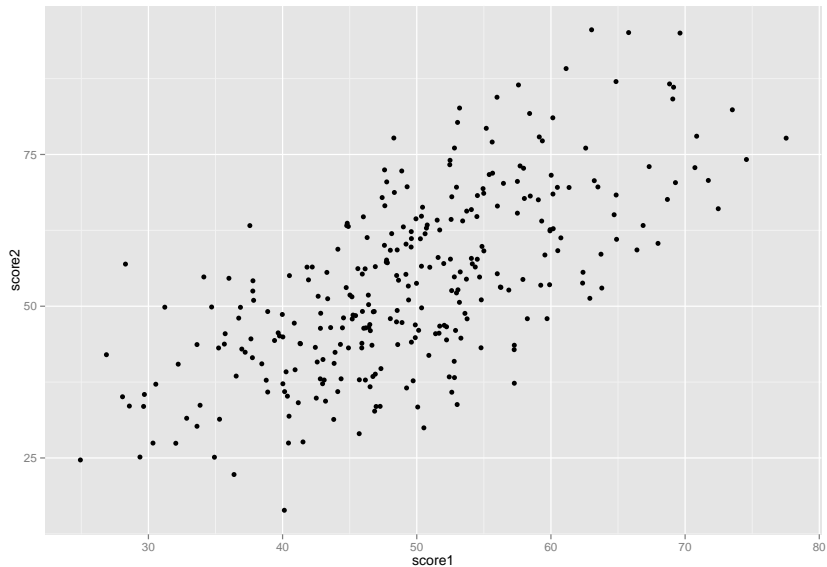
ggplot2: Q-Plots

```
qplot(score1, data = dat, geom = "histogram")
```



ggplot2: Q-Plots

```
qplot(score1, score2, data = dat, geom = "point")
```



Graphic Manipulation:

- `xlim` and `ylim`: define limits for x and y axis
 - e.g., `xlim = c(0, 100)`
- `binwidth` when creating histograms
- `log`: log transform any of the variables?
 - Can be x, y, or xy for both
 - However, I usually prefer to transform my variables myself
 - e.g., `dat$score1.log <- log(dat$score1)`

Graphic Manipulation:

- `main`, `xlab`, and `ylab`: title sections of graphic
- `colour`, `size`: Differentiate points by grouping variable
- `facets`: Create separate plots by grouping variable
 - e.g., one variable: `facets = ~ gender`
 - e.g., two variables: `factets = group ~ gender`
- `theme`: allows other customizations
 - try adding `+ theme_bw()` for black and white graphics
 - see [this link](#) for additional details

ggplot2: Q-Plots

```
qplot(score1, score2, data = dat, colour = gender,  
      facets = ~ group, main = "Scores by Group and Gender",  
      xlab = "Score on Test 1", ylab = "Score on Test 2")
```



ggplot2: Q-Plots

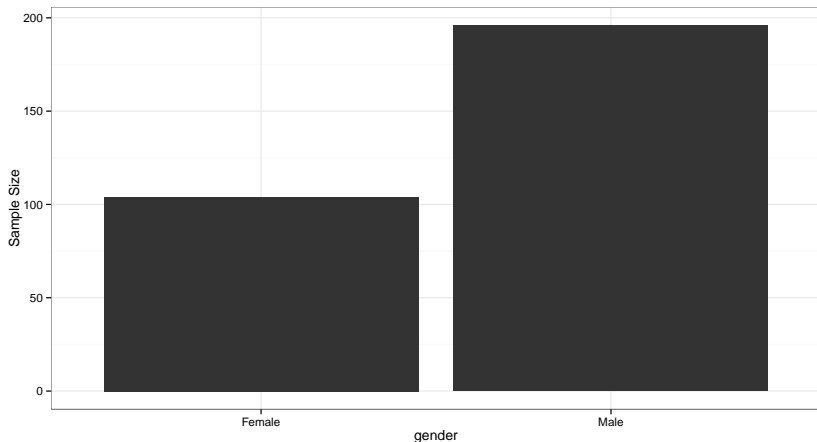
```
ggplot(score1, score2, data = dat, colour = gender,  
       facets = ~ group, main = "Scores by Group and Gender",  
       xlab = "Score on Test 1", ylab = "Score on Test 2")
```



Of course, there are additional geoms we can use!

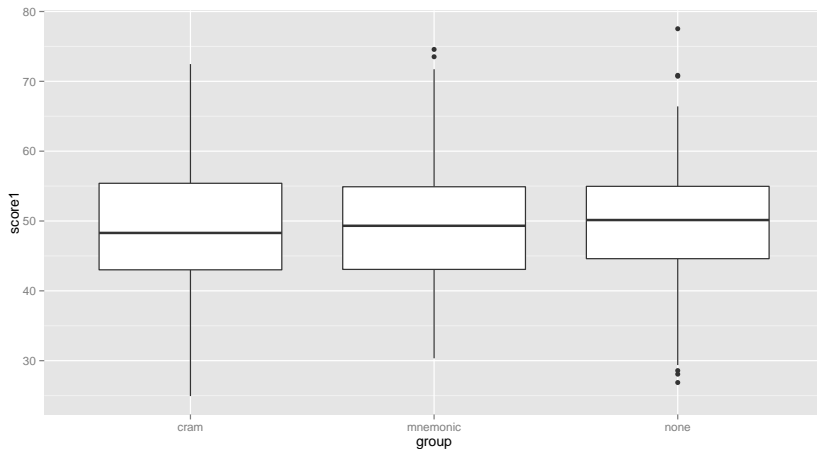
Bargraphs:

```
qplot(gender, data = dat, geom = "bar", ylab = "Sample Size") + theme_bw()
```



Boxplots:

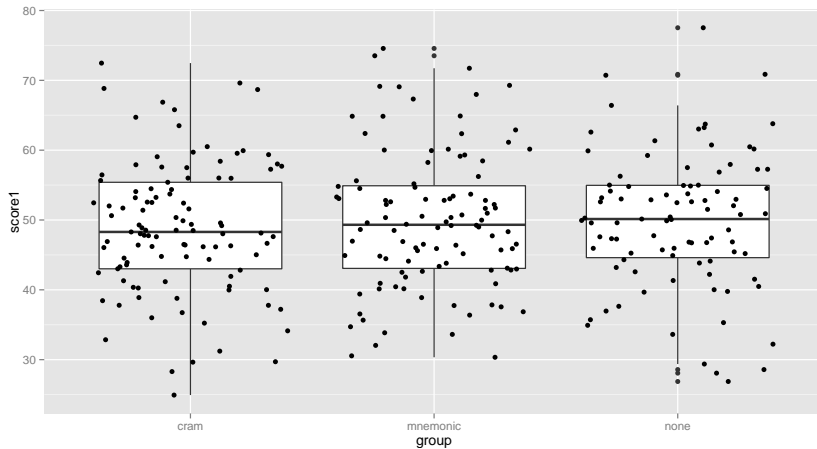
```
qplot(group, score1, data = dat, geom="boxplot")
```



ggplot2: Q-Plots

Geoms can be overlapped:

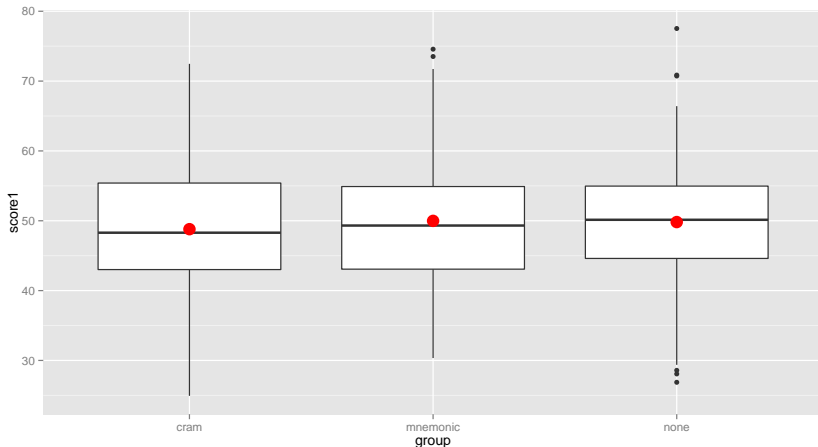
```
qplot(group, score1, data = dat, geom=c("boxplot", "jitter"))
```



ggplot2: Q-Plots

Geoms can be overlapped:

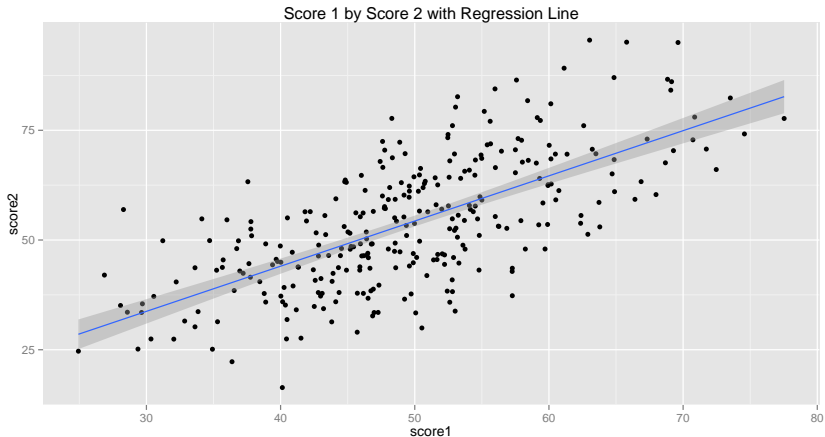
```
qplot(group, score1, data=dat, geom = 'boxplot') +  
  stat_summary(fun.y=mean, colour="red", size=5, geom="point")
```



ggplot2: Q-Plots

Geoms can be overlapped:

```
qplot(score1, score2, data = dat, geom = c("point", "smooth"), method = "lm",  
      main = "Score 1 by Score 2 with Regression Line")
```

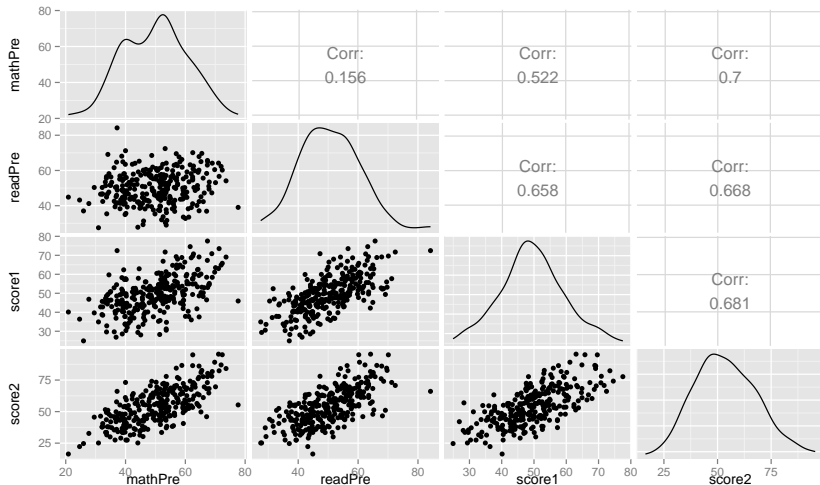


ggplot2: Scatterplot Matrix

Again, with many numeric variables, we are often interested in seeing all combinations at once:

```
#install.packages("GGally")  
library(GGally)  
ggpairs(na.omit(dat[c(3,4,6,7)]))
```

ggplot2: Scatterplot Matrix



ggplot2: Scatterplot Matrix

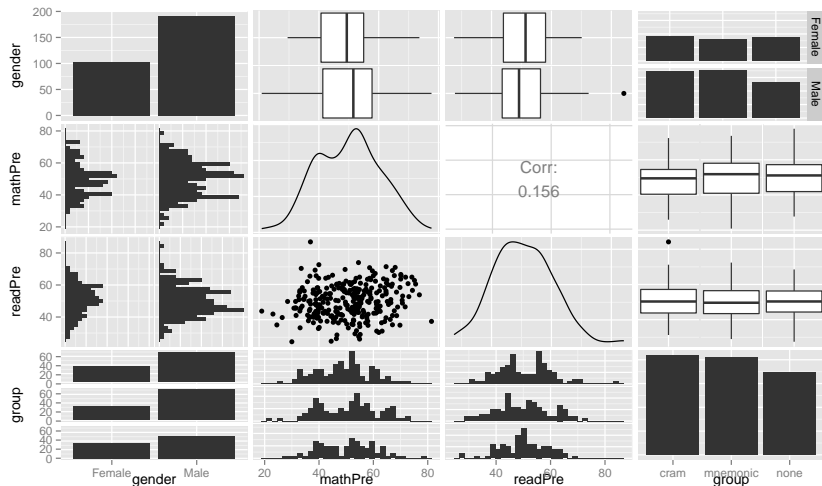
This actually works on practically all variable types!

```
library(GGally)
ggpairs(na.omit(dat[2:5]))
# Ignore id, but use first four variables in
# the dataset as a demonstration
```

- gender (factor)
- mathPre (numeric)
- readPre (numeric)
- group (factor)

ggplot2: Scatterplot Matrix

This actually works on practically all variable types!



Again, `qplot()` is a convenience function for learning the `ggplot2` basics. It should suffice for the purposes of this workshop, but if you want more control over your graphics (or to use more esoteric geoms), using the primary `ggplot()` function allows for that.

Basic R Exercises, Pt. 4

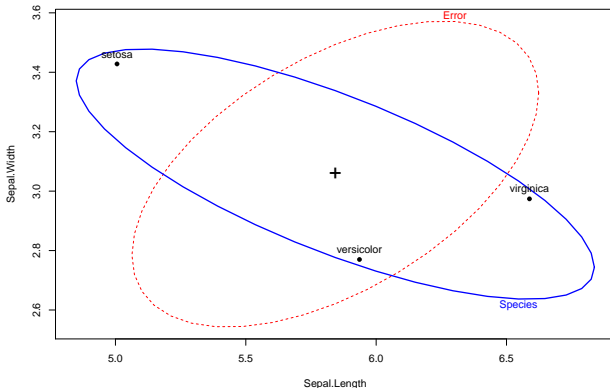
We are now ready to attempt the Exercise questions pertaining to "Graphing and Data Frames".

Alternate Packages for Graphical Output

Categorical data: `vcd` (`mosaic()`, `agreementplot()`, `fourfold()`).

Multivariate models (e.g., MANOVA/multivariate regression): **heplots**, uses ellipses to represent hypothesis and error SSCP matrices.

```
library(heplots)
imod <- lm(cbind(Sepal.Length, Sepal.Width, Petal.Length, Petal.Width) ~ Species,
           data = iris)
heplot(imod, size="effect")
```



Polychoric Correlations

Variety of packages:

- John Fox's `polycor::polychor()`
 - Only calculates pairwise correlations one at a time
 - Can use `polycor::hetcor()` (all categorical variables must be declared as factors, or else will be treated as continuous)
- William Revelle's `psych::polychoric()`
 - If not all variables are categorical, can also use `psych::mixed.cor()`

Exploratory Factor Analysis

Two choices for packages:

- Base R's `stats::factanal()` (maximum-likelihood only)
- William Revelle's `psych::fa()` (use `fa.poly()` for dictotomous/polytomous items)
 - The `psych` functions have additional support for multiple rotation, scoring, and factoring methods.

```
fa(r, nfactors=1, n.obs = NA, n.iter=1, rotate="oblimin",  
scores="regression", residuals=FALSE, SMC=TRUE, covar=FALSE,  
missing=FALSE, impute="median", min.err = 0.001, max.iter = 50,  
symmetric=TRUE, warnings=TRUE, fm="minres", alpha=.1, p=.05,  
oblique.scores=FALSE, np.obs, use="pairwise", ...)
```

- Rotation: 8 orthogonal, 7 oblique rotation options
- Factoring Method: `minres` (minimum residual, OLS), `wls` (weighted least squares), `gls` (generalized weighted least squares), `pa` (principal factors), `ml` (maximum likelihood), `minchi` (minimize sample size weight chi square).
- Factor Scores: `regression`, `correlation preserving` (tenBerge), `Anderson`, and `Bartlett`.

Exploratory Factor Analysis

Supplementary:

- `faoutlier` is a great package for running influential case diagnostics for factor analysis and SEM.
 - `gCD()`: Generalized Cook's Distance.
 - `LD()`: Likelihood Distance.
 - `robustMD()`: Robust Mahalanobis Distances.
 - `obs.resid()`: Model predicted residual outliers.
 - Works with `fa`, `sem`, and `lavaan` models.
- `nFactors` is useful for determining appropriate number of factors.
 - Implements parallel analysis, as well as 11 other criteria.

Confirmatory Factor Analysis and SEM

Recommended package: **lavaan**, whose primary functions are:

- `lavaan()` - general framework for all latent variable models
- `sem()` - defaults changed to accommodate SEM models
- `cfa()` - defaults changed to accommodate CFA models

... and are supplemented by:

- `fitMeasures()` - return a variety of fit measures
- `coef()` - return coefficients from model
- `residuals()` - return covariance/mean residuals from model
- `parameterEstimates()` - view parameter estimates
- `parTable()` - view model parameters as a dataframe
- `modificationIndices()` - calculate modification indices
- `inspect()` - useful for extracting particular output elements
- `anova()` - likelihood ratio test for nested models

Models in lavaan are plain text strings, but follow particular conventions.

- Use `=~` to define latent variables
- Use `~` to define regressions
- Use `~~` to define (co)variances
- Use `~ 1` to define intercepts
- Use `X*` to fix a parameter to X
- Use `equal()` to constrain parameters to be equal

lavaan Model Syntax

Basic Template for **lavaan model syntax**:

```
myModel <- '  
  # latent variable definitions  
  f1 =~ y1 + y2 + y3  
  f2 =~ y4 + y5 + y6  
  f3 =~ y7 + y8 +  
        y9 + y10  
  # regressions  
  f1 ~ f3 + f4  
  f2 ~ f4  
  y1 + y2 ~ x1 + x2 + x3  
  # (co)variances  
  y1 ~~ y1  
  y2 ~~ y4 + y5  
  f1 ~~ f2  
  # intercepts  
  f1 ~ 1; y5 ~ 1 '
```


Confirmatory Factor Analysis and SEM

The **Holzinger and Swineford (1939)** Example:

```
library(lavaan)
## This is lavaan 0.5-17
## lavaan is BETA software! Please report any bugs.
HS.model <- ' visual  =~ x1 + x2 + x3
              textual =~ x4 + x5 + x6
              speed   =~ x7 + x8 + x9 '

fit <- lavaan(HS.model, data=HolzingerSwineford1939,
              auto.var=TRUE, auto.fix.first=TRUE,
              auto.cov.lv.x=TRUE)
```

Confirmatory Factor Analysis and SEM

```
summary(fit)
## lavaan (0.5-17) converged normally after 35 iterations
##
##   Number of observations                    301
##
##   Estimator                                ML
##   Minimum Function Test Statistic          85.306
##   Degrees of freedom                       24
##   P-value (Chi-square)                     0.000
##
## Parameter estimates:
##
##   Information                                Expected
##   Standard Errors                          Standard
##
##           Estimate  Std.err  Z-value  P(>|z|)
## Latent variables:
##   visual =~
##     x1           1.000
##     x2           0.554    0.100    5.554    0.000
##     x3           0.729    0.109    6.685    0.000
##   textual =~
##     x4           1.000
##     x5           1.113    0.065   17.014    0.000
##     x6           0.926    0.055   16.703    0.000
##   speed =~
##     x7           1.000
##     x8           1.180    0.165    7.152    0.000
##     x9           1.082    0.151    7.155    0.000
##
## Covariances:
```

Confirmatory Factor Analysis and SEM

Some general starting parameters:

- **Correlated Factors:** use `cfa()`, with: `std.lv = TRUE`, `std.ov = TRUE`.
- **Hierarchical Model:** use `cfa()`, with: `'std.lv = TRUE`, `std.ov = TRUE`, and `orthogonal = TRUE`.
- **Bifactor Model:** use `cfa()`, with: `'std.lv = TRUE`, `std.ov = TRUE`, and `orthogonal = TRUE`.

Confirmatory Factor Analysis and SEM

Reference Guide:

Beaujean, A.A. (2014). *Latent Variable Modeling using R: A Step-By-Step Guide*. Routledge Publishing.

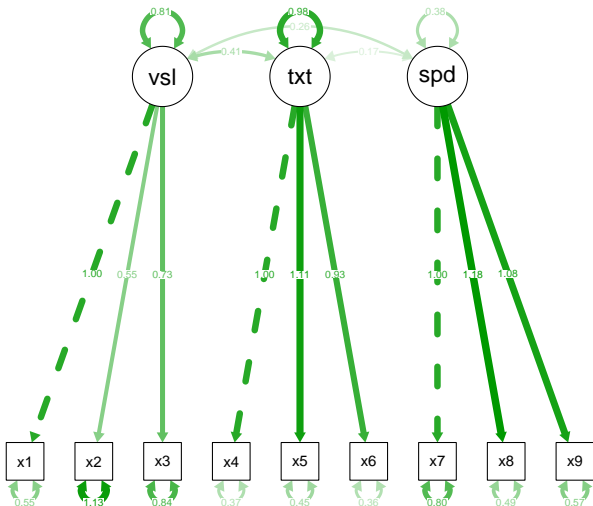
Supplementary:

- **lavaan.survey** has additional functions for dealing with complex sampling design.
- John Fox wrote the **sem** package, which is computationally faster although less user friendly

Plotting CFA/SEM Path Diagrams

Recommendation: **semPlot**, using function `semPaths()` to plot model.

```
library(semPlot)
semPaths(fit, what = "est")
```



(Multidimensional) Item Response Theory

Recommended package: **mirt**

- `mirt()` - primary function
- `bfactor()` - shorthand for setting up bifactor models
- `multipleGroup()` - shorthand for MG models

Supplemented by:

- `mirt.model()` - create mirt model object
- `testinfo()` - calculate test information/plot
- `itemplot()` - obtain item characteristic curves
- `iteminfo()` - calculate item information
- `itemfit()` - calculate item fit statistics
- `fscores()` - calculate factor scores (MAP, EAP, ML, or WLE)
- `wald()` - calculate Wald log-likelihood tests across models
- `DIF()` - investigate differential item functioning for MG models
- `itemplot(shiny=TRUE)` - interactive visualization of MIRT models

(Multidimensional) Item Response Theory

MIRT Model Syntax

- Simply entering a number will estimate an exploratory IRT model with that many factors.

- `mod1 <- mirt(data, 1)`

- Otherwise, model is specified by item number or variable name:

```
s <- 'F1 = 1,2,3,4-10
      F2 = 10-20
      COV = F1*F2'
model <- mirt.model(s)
fit <- mirt(data, model, itemtype = '2PL')
```

MIRT Parameters

- `itemtype` = Rasch, 2-4PL, graded, grsm, gpcm/nominal, ideal, partcomp, 2-4PLNRM
- `method` = for unidimensional: EM, for MIRT: QMCEM, MHRM
- see `?mirt` for more details

The psych library

- William Revelle's psych library has a many useful functions, primarily for scale construction using factor analysis, cluster analysis and reliability analysis, although others provide basic descriptive statistics.
- Some support for Item Response Theory is done using factor analysis of tetrachoric and polychoric correlations.
- Functions for simulating particular item and test structures are included.
- More details at the [Personality Project website](#)

CRAN's psychometric Task View

- Maintained by Patrick Mair (Harvard), last updated February 25th, 2015.
- Subheadings for: Item Response Theory, Correspondence Analysis, SEM/FA/PCA, Multidimensional Scaling, Classical Test Theory, Knowledge Structure Analysis, and Other Related Packages
- Link: [Psychometric Models and Methods](#)

Life on the Edge

Most R packages have two lives: the extensively tested “stable” build that lives on CRAN, and the cutting edge “developmental” build that may live outside of CRAN. Why go developmental?

- Cutting edge features
- Recent bug fixes

`mirt` in particular uses the developmental branch of their GitHub repository frequently. To install a package from source you need:

- On Windows, you must install [RTools.exe](#); on Mac OS X you may need to install [XCode's Command Line Tools](#).
- `install.packages("devtools")`
- `load.packages("devtools")`
- `install_github('philchalmers/mirt')`

Some packages **only** live on the fringes (insert grumble about draconian CRAN policies), so knowing how to install them can be very important!

Write It Yourself!

One of the best parts about R is how easy it is to create your own functions and packages. These can become community-wide staples (for example, the package `Hmisc` is literally a collection of Frank Harrell's miscellaneous functions.).

For instance: Manolo wrote a script for **calculating empirical percentiles**.

Empirical Percentiles

SET-UP

- 1 The Hmisc package needs to be loaded
- 2 Variables should be imported as INTEGERS
- 3 Variables must be defined as FACTORS
- 4 Must source() in Manolo's functions
 - `source("G:/R&D/_PRIVATE SHARED/R/EmPeRnd.R")`

Documentation

The documentation for `EmPeRnd()` and its helper functions is in the .R file: `G:/R&D/_PRIVATE SHARED/R/EmPeRnd.R`

Empirical Percentiles Example

Source script and datafile:

```
library(Hmisc)
source("G:/R&D/_PRIVATE SHARED/R/EmPeRnd.R")
mp <- spss.get("G:/R&D/_PRIVATE SHARED/R/MASC2.sav")
```

Round variables to integers:

```
mp$ROUND.TOT <- round(mp$MASC.TOTAL.50)
mp$ROUND.SEP <- round(mp$Raw.Sep)
mp$ROUND.GAD <- round(mp$GAD.INDEX.SEP07)
mp$ROUND.SANX <- round(mp$Raw.Sanx)
mp$ROUND.HR <- round(mp$Raw.HR)
mp$ROUND.PF <- round(mp$Raw.PP)
mp$ROUND.OCD <- round(mp$OCD.AUG15)
mp$ROUND.PHYS <- round(mp$Raw.Phys)
mp$ROUND.PANIC <- round(mp$PANIC)
mp$ROUND.TR <- round(mp$TR.5)
mp$ROUND.HA <- round(mp$HA.8ITEM)
```

Empirical Percentiles Example

Convert to Factors:

```
mp$ROUND.TOT <- factor(mp$ROUND.TOT, levels=150:0)
mp$ROUND.SEP <- factor(mp$ROUND.SEP, levels=27:0)
mp$ROUND.GAD <- factor(mp$ROUND.GAD, levels=30:0)
mp$ROUND.SANX <- factor(mp$ROUND.SANX, levels=27:0)
mp$ROUND.HR <- factor(mp$ROUND.HR, levels=15:0)
mp$ROUND.PF <- factor(mp$ROUND.PF, levels=12:0)
mp$ROUND.OCD <- factor(mp$ROUND.OCD, levels=30:0)
mp$ROUND.PHYS <- factor(mp$ROUND.PHYS, levels=36:0)
mp$ROUND.PANIC <- factor(mp$ROUND.PANIC, levels=21:0)
mp$ROUND.TR <- factor(mp$ROUND.TR, levels=15:0)
mp$ROUND.HA <- factor(mp$ROUND.HA, levels=24:0)
```

Empirical Percentiles Example

Create loop to calculate the percentiles (with `EmPeRnd()`):

```
for (i in 80:90) {  
  xta <- EmPeRnd(mp$WT.ALL.FINAL, mp[,i])  
  xta$scl <- substring(names(mp)[i],7)  
  if (i == 80) {  
    emptile.p <- xta[,c("scl", "l", "per", "per.txt")]  
    QA.emptile.p <- xta  
  } else {  
    emptile.p <- rbind(emptile.p,  
                      xta[,c("scl", "l", "per", "per.txt")])  
    QA.emptile.p <- rbind(QA.emptile.p, xta)  
  }  
}
```

In this example, the scores we want to create percentiles for are in columns 80:90, and the weighting values are in the variable `WT.ALL.FINAL`.

Empirical Percentiles Example

What does our output look like?

```
head(QA.emptile.p) # Full output, inc. weighted frequency & raw percentiles.
```

```
##   l      f      rawper per per.txt scl
## 0 0 0.0000000 0.0000000   1     1st TOT
## 1 1 0.0000000 0.0000000   1     1st TOT
## 2 2 0.0000000 0.0000000   1     1st TOT
## 3 3 2.8880424 0.0712042   1     1st TOT
## 4 4 0.4615546 0.1537880   1     1st TOT
## 5 5 0.3653974 0.1741763   1     1st TOT
```

```
head(emptile.p) # Truncated output.
```

```
##   scl l per per.txt
## 0 TOT 0   1     1st
## 1 TOT 1   1     1st
## 2 TOT 2   1     1st
## 3 TOT 3   1     1st
## 4 TOT 4   1     1st
## 5 TOT 5   1     1st
```

Write output to file:

```
write.csv(emptile.p, file="emptile.p.csv", row.names=F)
```

```
write.csv(QA.emptile.p, file="QA.emptile.p.csv", row.names=F)
```


Empirical Percentiles Example

What does a custom function look like?

```
EmPeRnd <- function(wth, fact) {  
  #EmPeRnd : calculates rounded empirical percentiles  
  ##Requires* functions WtdFreq and stndrdth  
  #'wth' is the weight variable  
  #'fact' is the score or factor for the rows  
  # Returns a dataframe with values:  
  # 'l' : the "levels" of 'fact' (it includes all levels including with 0 freq)  
  # 'f' : the weighted frequencies of the levels/values of 'fact'  
  # 'rawper' the empirical percentile  
  # 'per' the truncated empirical percentile (with the plus.one)  
  # 'per.txt' the truncated empirical percentile in ordinal text format  
  wf <- WtdFreq(wth, fact)  
  tot <- sum(wf$f)  
  ln <- dim(wf)[1]  
  for (i in 1:ln) {  
    if(i==1) {  
      wf[i,"rawper"] <- wf[i,"f"]/2/tot*100  
    } else {  
      wf[i,"rawper"] <- ((wf[i,"f"]/2) + sum(wf[1:i-1,"f"]))/tot*100  
    }  
  }  
  wf$per <- round(wf$rawper)  
  wf$per <- ifelse(wf$per < 1, 1, ifelse(wf$per > 99, 99, wf$per))  
  wf$per.txt <- stndrdth(wf$per)  
  return(wf)  
}
```

Magic Pipes

A recent addition to R is the `magrittr` package, whose main contribution is the addition of the pipe operator: `%>%`.

In basic R code, functions are evaluated from the inside out. For example:

```
library(dplyr)
head(arrange(group_by(select(mutate(iris,
Sepal.Size=Sepal.Length*Sepal.Width,
Petal.Size=Petal.Length*Petal.Width),
Sepal.Size, Petal.Size, Species), Species),
desc(Sepal.Size+Petal.Size)), n = 3)
```

What is this doing?

Magic Pipes

With some indenting, it gets a bit better:

```
head(
  arrange(
    group_by(
      select(
        mutate(iris,
          Sepal.Size=Sepal.Length*Sepal.Width,
          Petal.Size=Petal.Length*Petal.Width),
        Sepal.Size, Petal.Size, Species),
      Species),
    desc(Sepal.Size+Petal.Size)),
  n = 3)
```

```
## Source: local data frame [3 x 3]
## Groups: Species
##
##   Sepal.Size Petal.Size Species
## 1      25.08      0.60  setosa
## 2      23.20      0.24  setosa
## 3      23.10      0.28  setosa
```

Magic Pipes

The pipe, on the other hand, takes what is on the left and “pipes” its output to the next function. This can greatly enhance the readability of your code.

```
library(magrittr)
iris %>%
  mutate(Sepal.Size=Sepal.Length*Sepal.Width,
         Petal.Size=Petal.Length*Petal.Width) %>%
  select(Sepal.Size, Petal.Size, Species) %>%
  group_by(Species) %>%
  arrange(., desc(Sepal.Size+Petal.Size)) %>%
  head(3)
## Source: local data frame [3 x 3]
## Groups: Species
##
##   Sepal.Size Petal.Size Species
## 1      25.08      0.60  setosa
## 2      23.20      0.24  setosa
## 3      23.10      0.28  setosa
```

Summary!

- Base R is extremely powerful (and free)
- RStudio improves the usability of R immensely (and is also free)
- Additional packages can extend the software in many directions, with techniques appearing in R in advance of other software programs.
- The R language has some fundamental learning blocks, but will tend to “click” with perseverance.
- Once you have a solid repository of scripts, it is easy to adapt code from one project to another.
- R help is fairly accessible, and there are many (free) resources available.
- Knit reports provide a succinct summary of what you coded, and how to interpret it.

- Fox, J., & Weisberg, S. (2010). *An R Companion to Applied Regression* (2nd Ed.). Sage Publishing.
- Wickham, H. (2015). *Advanced R*. Chapman & Hall. ([online!](#))
- Matloff, N. (2011). *The Art of R Programming*. No Starch Press.
- Beaujean, A.A. (2014). *Latent Variable Modeling using R: A Step-By-Step Guide*. Routledge Publishing.
- Field, A., Miles, J., & Field, Z. (2012). *Discovering Statistics using R*. Sage Publishing.