



INTRODUCTION TO R: HOW TO PLAY WITH STATISTICAL OBJECTS

Elizabeth Page-Gould



WORKSHOP OVERVIEW

- Goal

- *Use R to analyze your data*

- Roadmap

1. Getting started with R

2. Specific statistical analyses
in R

3. Some fancy stuff in R



OUTLINE

.....

- R user interface
- R concepts
- Basic descriptive and inferential statistics
- Graphing
- Trouble shooting
- R wizardry

GRAND OVERVIEW OF R



WHAT IS R?

.....

- R is:
 - A computer program that can do statistics
 - Open-source
 - It's free!!
 - Widely used
 - Most cutting-edge
 - Syntax-based
 - Object-oriented



WHAT IS “OBJECT ORIENTATION?”



- *A programming approach where concepts are represented as “objects”*
- An object is a thing that has:
 - Attributes
 - *Features of the object that describe it*
 - Functions
 - *Actions that can be done with the object*
- Objects are created in R with the “assignment arrow”: <-

EXAMPLE: OBJECT ORIENTATION IN R

- Store variables in objects
 - `group.1 <- c(4, 6, 8, 7)`
 - `group.2 <- c(2, 1, 3, 2)`
- Store analyses in objects
 - `analysis <- t.test(group.1, group.2)`
- Extract attributes from the object
 - `analysis$statistic`
- Use the object(s) in a function
 - `boxplot(group.1, group.2)`

OBJECT-ORIENTED STATISTICS

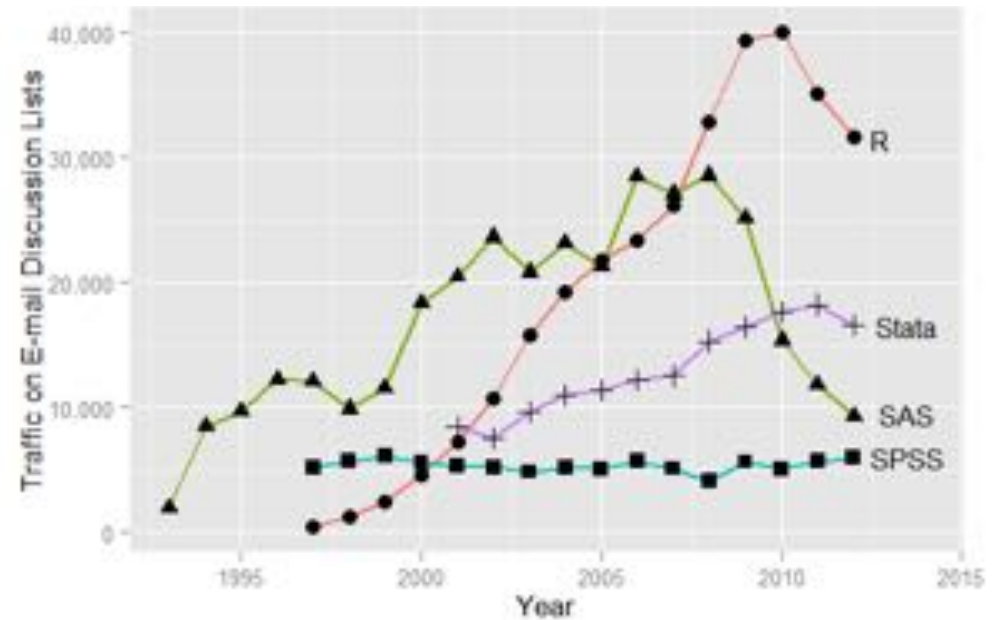
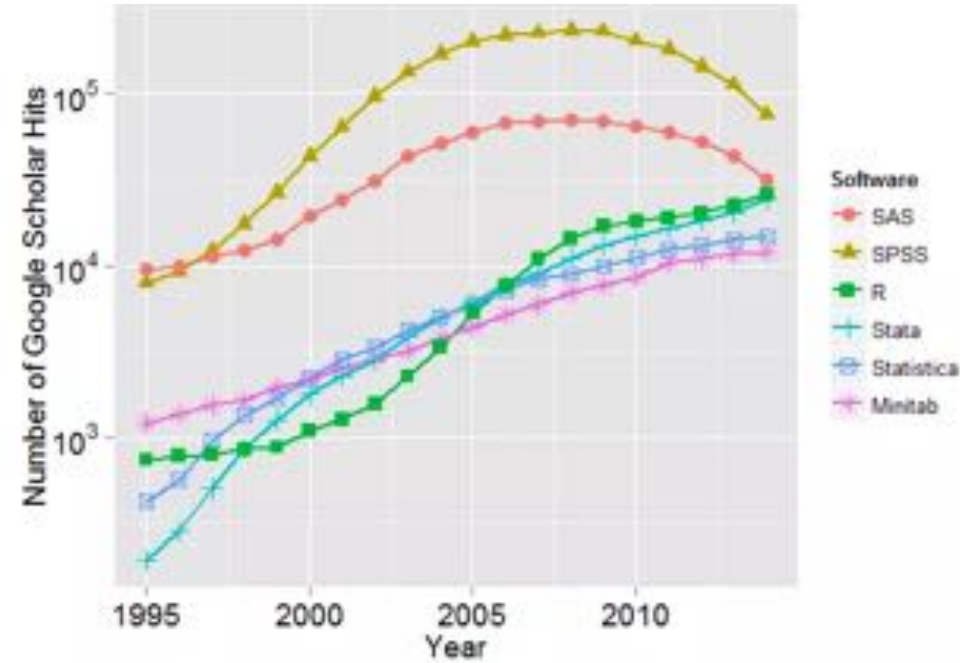
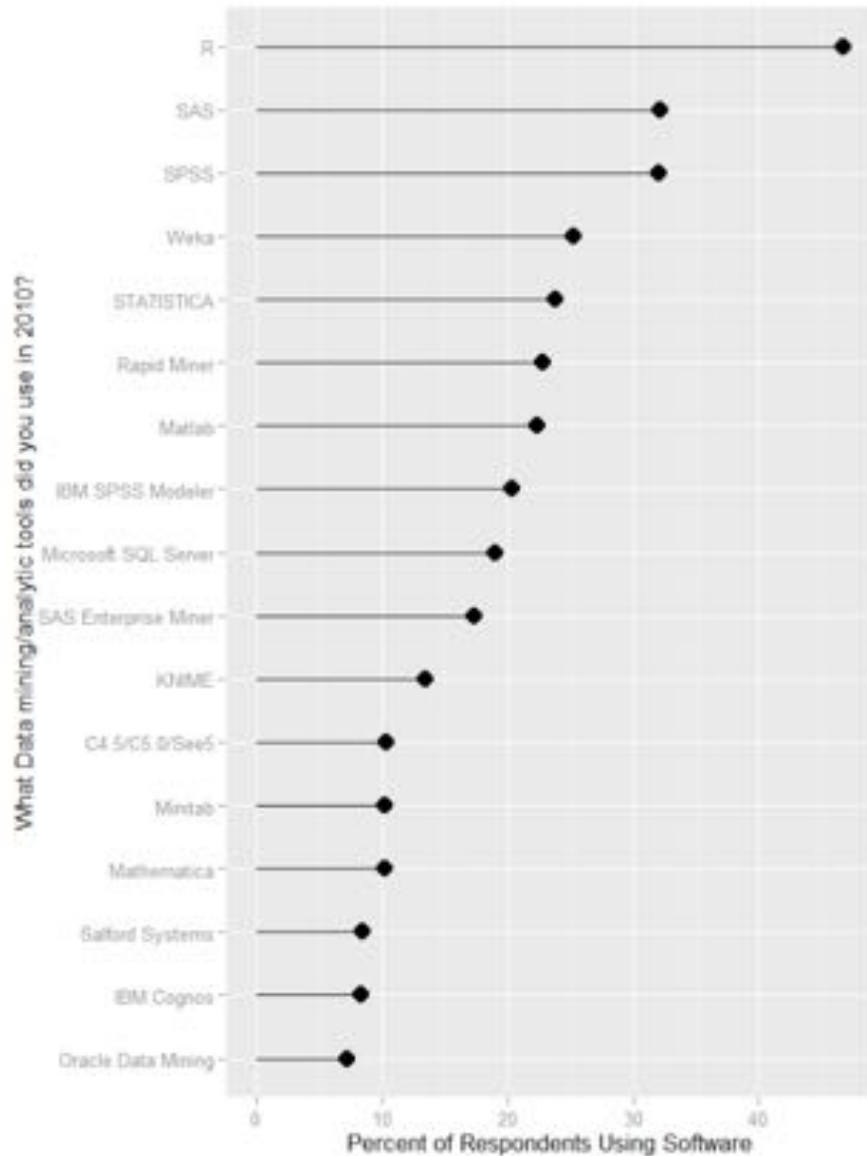


- Why do people love R so much?
 - *Once you start thinking about statistics in an object-oriented way ... it's a whole new world*
- Object orientation applied to statistics
 - Both data and statistical analyses are things that you want to know stuff about (i.e., attributes) and want to do stuff to (i.e., functions)
 - Example: What if I did a t-test and put it in an object?
 - An attribute: *its degrees of freedom*
 - A function: *print a nice summary of results*

HISTORY OF R

- In the beginning, there was S
 - John Chambers (Bell Labs)
 - S evolved to S-PLUS
- Then, there was R
 - *Reverse-engineered, open-source version of S*
 - Developed by Ross Ihaka and Robert Gentleman (University of Auckland, New Zealand)
 - Ihaka & Gentleman (1996), *Journal of Computational and Graphical Statistics*





Usage Statistics

Source: <http://r4stats.com/articles/popularity/>



INSTALLING R

- Find the link for your operating system under “Download and Install R”
- <http://cran.utstat.utoronto.ca/>

Download and Install R

Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:

- [Download R for Linux](#)
- [Download R for \(Mac\) OS X](#)
- [Download R for Windows](#)

R is part of many Linux distributions, you should check with your Linux package management system in addition to the link above.

WHAT YOUR COMPUTER KNOWS ABOUT R



- R is an installed software program
- Computer may not know which files to associate with R
 - Relevant file extensions to R:
 - *.R: *R scripts*
 - *.Rhistory: *A history of the commands you have run recently*
 - *.Rdata: *A workspace file that holds a bunch of saved objects*
 - .Rprofile: *A file in your working directory that is automatically run when you start your R session (if you start up in a certain way)*

WHAT R KNOWS ABOUT YOUR COMPUTER



1. R knows how to ask your computer's processor to compute things
2. R knows how to read and write files to your hard drive
 - Importing files:
 - R can retrieve files by name, if they are in a specific folder called your “working directory”
 - The working directory has an address called a “path”
 - You must tell R the path to your working directory
 - Writing files:
 - R can export data or your workspace to the working directory, but you must do this manually
 - Otherwise, nothing you do within an R session is saved

EXAMPLE: SET YOUR WORKING DIRECTORY IN R

1. On your computer, find the path of your working directory
 - Windows:
 - Press the SHIFT key while right-clicking on the folder
 - Select “Copy As Path”
 - Mac:
 - Right-click on folder that has your R files
 - Press the OPTION key
 - Select “Copy [folder] as Pathname”
2. Go back to R
 - At the command prompt, use the “setwd” command:
 - `setwd("path")`

EXAMPLE: WRITING FILES AND WORKSPACES

- Write a simple file to your workspace:
 - `write("This is some output", "Output File.txt")`
- Save whole workspace:
 - `save.image("Most Recent Workspace.RData")`

R AND YOUR COMPUTER DON'T TALK REGULARLY



- You are the mediator
- Negative Impact:
 - If you don't save your syntax as you go, you may not be able to get it back
 - Solution:
 - Write your commands in an *.R script and save that
- Positive Impact:
 - You won't mess up your raw data
 - Always start analyses with your raw data
 - **Do not edit data before reading it into R**
 - Clean data and build scales/metrics in R

RSTUDIO

- *RStudio is a “development environment” for R*
- It provides a nice user interface
- When you open RStudio, it automatically runs R in the background



R AND RSTUDIO



- I strongly recommend RStudio
- R is the statistical package that actually does the stats
 - R must be installed to run RStudio
 - RStudio is just a friendly layer that sits on top of R
- Remember: *R and RStudio are two different programs!*
 - You have to update and maintain them separately

INSTALLING RSTUDIO

- Find the link for your operating system under “Installers for Supported Platforms”
- <http://www.rstudio.com/products/rstudio/download/>

RStudio Desktop 0.99.489 — Release Notes

RStudio requires R 2.11.1 (or higher). If you don't already have R, you can download it [here](#).



Installers for Supported Platforms

Installers	Size	Date	MD5
RStudio 0.99.489 - Windows Vista/7/8/10	73.9 MB	2015-11-05	7ef8c00311d3c03b6c9abe22826497d6
RStudio 0.99.489 - Mac OS X 10.6+ (64-bit)	56.2 MB	2015-11-05	05cf866b07df6552583f98314ed09d38
RStudio 0.99.489 - Ubuntu 12.04+/Debian 8+ (32-bit)	77.4 MB	2015-11-05	1bf2997d91b6eaf0b483fbc52cca29b5
RStudio 0.99.489 - Ubuntu 12.04+/Debian 8+ (64-bit)	83.9 MB	2015-11-05	ed089d88cc2e5901e311c66f7b1ada8b
RStudio 0.99.489 - Fedora 19+/RedHat 7+/openSUSE 13.1+ (32-bit)	76.8 MB	2015-11-05	642ede6193cc3ff24a55c3ffe20c31bc
RStudio 0.99.489 - Fedora 19+/RedHat 7+/openSUSE 13.1+ (64-bit)	77.7 MB	2015-11-05	1a71fbfd49730695515d4f5343885d6b

- When you want to use R, open RStudio instead!

RSTUDIO WORKSPACE



The screenshot displays the RStudio interface with three main panes:

- Source Editor:** Contains R code for loading data and specifying a structural equation model (SEM).
- Console:** Shows the output of the `summary()` function for the fitted model.
- Environment/History:** Lists the objects in the workspace, including `grad.data`, `grad.analysis`, and `grad.model`.
- Documentation:** Displays the help page for the `specifyModel` function.

Source Editor Code:

```
5- ##### PATH ANALYSIS #####
6- #Opening Graduate School Interest Data
7- grad.data <- read.csv("graduate.school.csv")
8- attach( grad.data )
9- names( grad.data )
10-
11- #Specify the path model
12- grad.model <- specifyModel()
13- perceived.value ~> intent.to.apply, path1
14- external.pressure ~> intent.to.apply, path2
15- perceived.control ~> intent.to.apply, path3
16- intent.to.apply ~> application.behaviour, path4
17- perceived.control ~> application.behaviour, path5
18-
```

Console Output:

```
> summary( grad.analysis )

Model Chi-square = 0.8472239  Df = 2  Pr(>ChiSq) = 0.6546779
AIC = 26.84722
BIC = -7.341465

Normalized Residuals
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
0.00000 0.00000 0.00000 0.05686 0.00000 0.43200

R-square for Endogenous Variables
      intent.to.apply application.behaviour
      0.5998                0.3425

Parameter Estimates
      Estimate Std. Error z value Pr(>|z|)
path1    0.44421348  0.06212966  7.149781 8.691620e-13 intent.to.apply <-- perceived.value
path2    0.02948266  0.03036854  0.970829 3.326335e-01 intent.to.apply <-- external.pressure
path3   -0.06358002  0.05794492 -1.097249 2.725324e-01 intent.to.apply <-- perceived.control
```

Environment/History:

- `grad.data` 60 obs. of 6 variables
- `example.object` in[12]
- `grad.analysis` objectiveML[25]
- `grad.model` semmod[39]

Documentation: `specifyModel` (sem) - R Documentation: Specify a Structural Equation Model

A TYPICAL R SESSION

- Open R Studio
- Set the working directory to the folder with your data files
- Load the R packages that you want to use
- Read in your data
- Do stats
- (Maybe) Save what you did

WORKING DIRECTORY: SETWD() AND GETWD()

- Working Directory
 - *A directory or folder where R will look for files*
- Functions for dealing with the working directory:
 - What is my working directory?
 - `getwd()`
 - How do I change my working directory?
 - `setwd("PathOfWorkingDirectory")`

PACKAGES

- *R packages contain functions that you can use*
- R is prepackaged with the following packages:
 - Base
 - Stats
 - Graphics
 - *These packages will do most of the stuff you want!*
- Add-on packages
 - *Do everything else that you want!*

ADD-ON PACKAGES

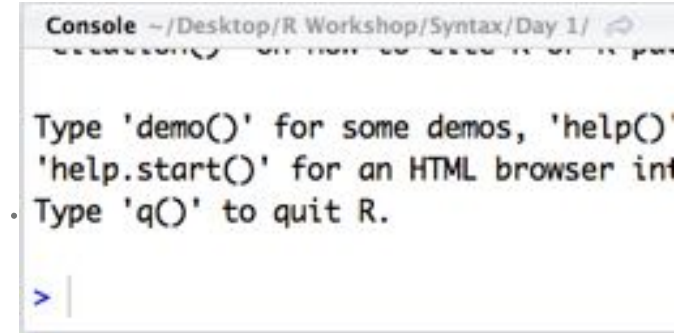
- *Add new functions to your R session that are not normally distributed with R*
- You must **INSTALL** a package **once** per R Version
 - `install.packages("package.name")`
 - The package name is case sensitive
 - Package name must be in quotes inside the function
- You must **LOAD** the package **every time** you start a new R session
 - `library(package.name)`

EXAMPLE: LOAD A PACKAGE

- “readxl” package
 - *Allows you to read excel files into R*
 - Functions include:
 - `read_excel(...)`: *Reads *.xls and *.xlsx files*
 - `excel_sheets(...)`: *Prints a list of the sheet names in an excel file*
- Install it!
 - `install.packages("readxl")`
- Load it!
 - `library(readxl)`

CONSOLE

- *The console is how you talk to R*
 - Type commands after the command prompt (“>”)
 - Hit the “Enter” or “Return” key to send a command to the R interpreter
 - R displays the result of your command
- Each line of syntax is interpreted one at a time



```
Console ~/Desktop/R Workshop/Syntax/Day 1/
Type 'demo()' for some demos, 'help()'
'help.start()' for an HTML browser interface
Type 'q()' to quit R.

> |
```

MATHEMATICAL OPERATORS

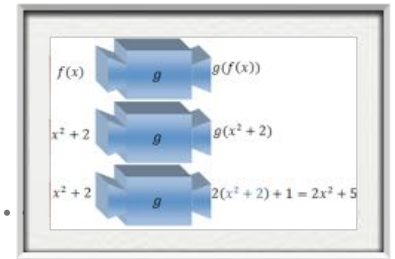
- You can type mathematical operators directly into the console

Operator	Description	Example
+	Addition	> 5 + 2 [1] 7
-	Subtraction	> 5 - 2 [1] 3
*	Multiplication	> 5 * 2 [1] 10
/	Division	> 5 / 2 [1] 2.5
^ or **	Exponent	> 5 ^ 2 > 5**2 [1] 25 [1] 25
%%	Modulus (remainder)	> 5%%2 [1] 1

ASSIGNMENT ARROW

- *To save the output of a command as an object, use the assignment arrow, “<-”*
 - `result <- 2 + 3`
 - `print(result)`

FUNCTIONS



- *Functions are the commands that you type at a command prompt in the Console*
- General form:
 - `function.name()`
- Functions always:
 - Perform an action
 - Return an object
- Functions usually have “arguments”
 - *Objects and settings for the function*

FUNCTION ARGUMENTS



- *Information you give to a function so it will do what you want it to do*
- General Form:
 - `function.name(
 first.argument.name=first.argument.value,
 second.argument.name=second.argument.value
)`
 - Some functions take no arguments (e.g., `setwd()`)
 - Most functions take at least one argument
- Arguments are passed to a function inside the parentheses that follow the function call
- Most function arguments have *default values*, so you don't have to specify an argument unless you don't want its default

EXAMPLE: T-TEST FUNCTION

- “t.test()” function
- At a minimum, it must take a data object
 - `t.test(group.1)`
- Can have other arguments, too
 - `t.test(group.1, mu=5)`
 - `t.test(group.1, mu=5, alternative="greater")`

R'S DEFAULT FUNCTION

- The R console uses the default function of “print()”
- If you only type an object with no function call, R will assume you want to print the object
 - e.g., these two statements are equivalent:
1. `print(object)`
2. `object`

HOW DO I KNOW WHAT ARGUMENTS A FUNCTION TAKES?



- Help files are your best friends!!
 - *Help files are mini-manuals for R that are specific to each function*
- In RStudio, they appear in the bottom right-hand panel
- For every function, you can access the help files with one of these commands:

`1. ?function.name`

- (Note there are no parentheses after the function name)

`2. help(function.name)`

HELP FILES

- Common Structure
 - `function.name{its.package}`
 - Description: *Brief, broad description of what the function does*
 - Usage: *Syntax reference*
 - Arguments: *Explains the purpose of each argument and the meanings of its possible values*
 - Details: *Provides detailed usage notes*
 - Value: *Describes the object that the function returns*
- Embracing help files is your only option



UNDERSTANDING USAGE SYNTAX



- *Usage syntax shows you the generic form for your command and all the possible options you can use*
 - `function.name(first.argument.name=first.argument.value, second.argument.name=second.argument.value)`
 - If an argument is named without a value, then you must supply it
 - If an argument is named with a value, then the value is a default
- There may be example syntax for more than one function

EXAMPLE: READ_EXCEL FUNCTION

1. Open the help file for the `read_excel()` function
2. Think about the syntax you would need to read in an excel file with the following constraints:
 - The name of the excel file is “Friendship_Data.xlsx”
 - The data is in the first sheet
 - The header row (variable names) are in Row 2, and the data follows after that row
3. Read in the excel data and save it in an object called “friendship.data”

SOURCE FILES

- *A text file that has R syntax in it*
 - Always ends in *.R
- Best practices:
 - Type your commands into the source file (because it will be permanent)
 - Send commands to the console directly from the source file
 - Run the entire line on which your cursor is placed by:
 - Windows: Pressing the “Ctrl” and “Enter” keys
 - Mac: Pressing the “⌘” and “return” keys
 - Highlight a command in the source file and click the “Run” button

WORKSPACE

- *The (virtual) environment in which your objects are stored, your packages are loaded, and your functions are accessible*
 - In RStudio, you can view the objects in your workspace in the upper right-hand panel
- What's in your workspace?
 - `ls()`
- You can save your current workspace and reload it:
 - Save the current workspace:
 - `save(list=ls(all=T), file="Saved_Workspace.Rdata")`
 - Load a saved workspace:
 - `load("Saved_Workspace.Rdata")`

GLOBAL NAMESPACE

- Your workspace is also your default, global namespace
- “namespace”
 - *A dictionary for R that maps names onto objects and functions*
- Objects in the global namespace can be directly referenced on the console line
- Some objects and functions are not in your global namespace
 - Some objects are stored in other objects
 - If an object is not in the global workspace, then you need to tell R where to find it

NAMESPACE ERRORS

- Issues pertaining to the global namespace are common problems with R
- If you request an object from the wrong namespace, you will get an error like:
 - `Error in [function(x)] : object '[x]' not found`
- If you create a new object with the same name as another object, the second object will overwrite the first one

HOW TO USE ANOTHER NAMESPACE

- Reference the namespace before the object name, separated by a dollar sign
 - `namespace$object`
- Use the `with()` function
 - `with(namespace, object)`
 - `with(namespace, function(object))`

NAMESPACE BEST PRACTICES FOR THE PSYCHOLOGICAL RESEARCHER

- Clean your data and build scales within your dataset
 - Use the `within()` function
 - *Keeps all data objects in the dataset's namespace*
- When conducting analyses, specify which dataset you want the analysis to use
 - Some analyses will have a *data=* argument where you can identify your namespace
 - You can always use the `with()` function
- Store analysis objects in the global namespace

HISTORY



- *History files are a list of the commands you've typed on the console within your work session*
- Two ways to access your History:
 - At Console command prompt:
 - Use the up or down arrows to scroll through your history
 - Press return/enter key to run the current command
 - RStudio has a GUI
 - Double-click on a command to send it to the Console
 - Press return/enter key to run the command
- Save the current history file:
 - `savehistory("Saved_History_File.Rhistory")`
- Load a saved history file:
 - `loadhistory("Saved_History_File.Rhistory")`



DOING STATS WITH R!

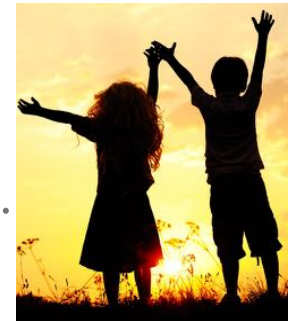


STATISTICS IN R

.....

- Intro to Example Datasets
- Descriptive Statistics
- Recoding Data and Building Scales
- Inferential Statistics

BRIEF INTRODUCTION TO FRIENDSHIP DATA



➤ Design:

- Men and women (variable: *sex*) randomly assigned to describe a cross-sex friend or same-sex friend in detail (variable: *friendship.type*)
- *sex*: -1 = female, 1 = male
- *friendship.type*: -1 = same-sex friend, 1 = cross-sex friend

➤ Other measures of interest:

- Demographics — Age (*age*), ethnicity (*ethnicity*), perceived socioeconomic status (*subjective.SES*)
- Questions about the friend (e.g., closeness of friendship, comfort with friend) — (*friend.**)
- Big 5 Personality — *BFI1* - *BFI44*
- Self-esteem — *SE1* - *SE10*

BRIEF INTRODUCTION TO GENERAL SOCIAL SURVEY DATA

GSS General Social Survey

- Survey design
 - Massive national survey of U.S. adults conducted every 2 years
- Interesting questions
 - Ever paid for sex — *prostitution*: yes = 1, no = -1
 - Ever been married — *ever.married*: yes = 1, no = -1
 - Happiness — *happiness* (3-point Likert) higher values = more happy
 - Number of Children — *children*
 - Respondent sex — *sex*: -1 = female, 1 = male
 - Own a gun — *own.gun*: yes = 1, no = 0

BRIEF INTRODUCTION TO CORTISOL DATA



➤ Design:

- 65 participants randomly assigned to evaluation (1) or no-evaluation (-1) *condition* of Trier Social Stress Test (TSST)
- Cortisol (nMol/L) measured at three time points (“*time*”)
 - Baseline
 - Stress: 20 minutes after start of TSST (49 minutes after baseline sample)
 - Recovery: 24 minutes after the stress sample



DESCRIPTIVE STATISTICS

.....

- Sample descriptives
 - Frequency information
 - Centrality
 - Spread

FREQUENCIES WITH TABLE()

- *The table() function displays counts of identical observations for either a single data vector or a dataframe*
- General Usage:
 - `table(object)`
- Cross-tabulate:
 - `table(object.1, object.2, ...)`

EXAMPLE: FIND SAMPLE SEX DISTRIBUTION

- Number of males and females in sample
 - `table(sex)`
- Percentage of males and females in sample
 - `table(sex) / sum(table(sex)) * 100`
- Distribution of experimental condition across the sexes
 - `table(sex, condition)`

MEAN

- *Calculates the mean of a numeric object*
- General usage:
 - `mean(object, na.rm=T)`
 - Without the “na.rm=T” argument, `mean()` will return a null value if there is any missing data in the object
 - `mean(cbind(object.1, object.2), na.rm=T)`

EXAMPLE: FIND MEAN AGE OF SAMPLE

- Mean age of sample
 - `mean(age, na.rm=T)`

NOTES ON CALCULATING THE MEAN

- Common uses:
 - Calculate means for Methods sections
 - To check oneself while building scales and analyzing data
 - To centre variables around the mean
- NOTE:
 - `mean()` returns a single value, so it won't extract mean values for each individual

ROWMEANS()

- *R has built-in functionality for calculating per-row means*
 - This is especially handy for building scales!
 - See “Building scales with rowmeans()” slide

MEDIAN

- *The median() function finds the median (50th percentile) value*
- General usage:
 - `median(object, na.rm=T)`

EXAMPLE: FIND MEDIAN AGE OF SAMPLE

- Median age of sample
 - `median(age, na.rm=T)`

VARIANCE AND SD

- *Variance and standard deviation are easy to find in R*
- General Usage:
 - Variance:
 - `var(object, na.rm=T)`
 - `var(cbind(object.1, object.2, object.3, object.4), na.rm=T)`
 - Standard Deviation:
 - `sd(object, na.rm=T)`
 - `sd(cbind(object.1, object.2, object.3, object.4), na.rm=T)`

EXAMPLE: FIND SD OF AGE FOR SAMPLE

- SD of age for sample
 - `sd(age, na.rm=T)`

RANGE

- *Minimum and maximum values of a numeric object*
- General Usage:
 - `range(object, na.rm=T)`

EXAMPLE: FIND AGE RANGE FOR SAMPLE

- Age range of sample
 - `range(age)`

NICE SUMMARIES WITH DESCRIBE()

- `describe{Hmisc}` provides information about:
 - Variance, N, mean, SD, median, trimmed mean, median absolute deviation, minimum, maximum, range, skew, kurtosis, standard error
- General usage:
 - `library(Hmisc)`
 - `describe(object)`

EXAMPLE: GET FULL DESCRIPTIVES FOR AGE

- Full descriptives for age
 - `library(Hmisc)`
 - `describe(age)`



RECODING DATA AND BUILDING SCALES

-
- Recoding variables
 - Building scales
 - Finding scale reliability

RECODING DATA

- *You want to “recode” data when the raw data values are different from the values you need for your analysis*
- Continuous variables: reverse-coding
- Categorical variables: recoding

RECODING CONTINUOUS VARIABLES

- Example context:

- You measure mood with this scale (variable: *mood*):

Right now, how would you describe your mood?

Very Good	Good	Mildly Good	Neutral	Mildly Bad	Bad	Very Bad
1	2	3	4	5	6	7

- But, you want higher numbers to represent better mood ...

- Reverse-code mood so that $1 \rightarrow 7$, $2 \rightarrow 6$, ... $7 \rightarrow 1$

- Easy to reverse-code in R

- Add the min and max values together (e.g., $7 + 1 = 8$)

- Subtract all values from this sum

- `reversed.mood <- 8 - mood`

EXAMPLE: REVERSE-CODE SUBJECTIVE SES

- Values of possible responses: 1 - 10
 - High values of subjective.SES represent *low* SES
- Recode subjective.SES such that higher values represent higher SES
 - `ses <- 11 - subjective.SES`

RECODING CATEGORICAL VARIABLES

- *The ifelse() function can be used to recode data*
- General Form:
 - `new.coding <-
 ifelse(old.coding==test.value,
 "true.value", "false.value")`
- Nested ifelse() functions can be used to recode multiple category levels
 - `new.coding <-
 ifelse(old.coding==test.value.1,
 "true.value",
 ifelse(old.coding==test.value.2,
 "second.true.value", NA))`

EXAMPLE: NUMERICALLY CODE SEX

- Categorical variable `sex.label` has two levels: *male*, *female*
- Effect-code `sex.label` such that `male = 1` and `female = -1`
 - ```
sex <- ifelse(sex.label=="female", -1,
 ifelse(sex.label=="male", 1, NA))
```

# BUILDING SCALES WITH ROWMEANS()

---

- General usage:
  - `rowMeans( dataframe.object, na.rm=T )`
  - `rowMeans( cbind( object.1, object.2, object.3 ), na.rm=T )`
- Building scales (e.g., items: item.1 - item.5 ) from a 5-point likert ( 1 - 5) where some data could be missing and item.4 needs to be reverse-coded
  - `scale.values <- rowMeans( cbind( item.1, item.2, item.3, 6-item.4, item.5 ), na.rm=T )`

## EXAMPLE: BUILD SELF-ESTEEM SCALE

---

- Variables SE1 - SE10 contain answers to self-esteem items
  - Measured on 1 - 6 Likert scale
  - Items SE2, SE4, SE5, SE6, SE8, and SE9 need to be reverse-coded first
- Calculate the average self-esteem for each participant
  - `self.esteem <- rowMeans( cbind( SE1, 7-SE2, SE3, 7-SE4, 7-SE5, 7-SE6, SE7, 7-SE8, 7-SE9, SE10 ), na.rm=T )`

## Reliability analysis

Call: `alpha(x = friendship.data[55:`

| raw_alpha | std.alpha | G6(smc) | average |
|-----------|-----------|---------|---------|
| 0.64      | 0.66      | 0.61    |         |

Reliability if an item is dropped:

|      | raw_alpha | std.alpha | G6(smc) | average |
|------|-----------|-----------|---------|---------|
| SE1  | 0.31      | 0.33      | 0.20    |         |
| SE2- | 0.66      | 0.66      | 0.50    |         |
| SE3  | 0.64      | 0.66      | 0.49    |         |

## Item statistics

|      | n   | r    | r.cor | r.drop | mean | sd  |
|------|-----|------|-------|--------|------|-----|
| SE1  | 185 | 0.86 | 0.77  | 0.63   | 4.4  | 1.1 |
| SE2- | 185 | 0.73 | 0.52  | 0.40   | 3.2  | 1.5 |
| SE3  | 185 | 0.73 | 0.52  | 0.37   | 4.8  | 1.1 |

Non missing response frequency for

|     | 1    | 2    | 3    | 4    | 5    | 6    | m |
|-----|------|------|------|------|------|------|---|
| SE1 | 0.01 | 0.05 | 0.12 | 0.26 | 0.44 | 0.11 |   |
| SE2 | 0.15 | 0.21 | 0.21 | 0.20 | 0.15 | 0.09 |   |
| SE3 | 0.02 | 0.04 | 0.05 | 0.22 | 0.41 | 0.26 |   |

# SCALE RELIABILITY IN R

.....

- Two primary approaches:

- Cronbach's  $\alpha$ : `alpha{psych}`

1. `alpha( dataframe.object,  
keys=c("V1","V4", "V5")  
)`

2. `alpha( cbind( item.1,  
item.2, item.3 ),  
keys=c("item.2") )`

- McDonald's  $\omega$ :  
`omega{psych}`

- `omega( dataframe.object,  
nfactors=3 )`

# EXAMPLE: SCALE RELIABILITY OF SELF-ESTEEM

---

- Calculate Cronbach's alpha for self-esteem
  - `reliability.analysis <- alpha( cbind( SE1, SE2, SE3, SE4, SE5, SE6, SE7, SE8, SE9, SE10 ), keys=c( "SE2", "SE4", "SE5", "SE6", "SE8", "SE9" ) )`
  - `print( reliability.analysis )`



# INFERENCE STATISTICS

.....

- Comparing means
  - t-test
  - Regression
  - ANOVA
    - 1-way ANOVA
    - Factorial ANOVA
    - Repeated-measures ANOVA
- Evaluating covariance
  - Correlation
  - Regression
    - GLM
    - Moderated regression
    - Generalized linear models

# T-TEST

---

- *The `t.test()` function can perform 1-sample, independent-samples, or paired t-tests to compare the difference between two means*
- Usage:
  - 1-sample t.test:
    - `t.test( object, mu=[comparison value] )`
  - Independent samples t.test:
    - `t.test( object.1, object.2 )`
    - `t.test( object~group )`
  - Paired t.test:
    - `t.test( object.1, object.2, paired=TRUE )`

# EXAMPLE: T-TESTS OF FRIENDSHIP CLOSENESS

---

- 1-sample t-test
  - Are the mean closeness with friends (*friend.close*) significantly different from the midpoint of the scale (i.e., closeness > 4)?
  - What is the 95% confidence interval around the sample mean of closeness?
  - `t.test( friend.close, mu=4 )`
- Independent samples t-test
  - Is there a significant difference in how close participants feel to their same- and cross-sex friends?
  - `t.test( friend.close ~ friend.type )`
- Paired t-test
  - Do people rate their friends as being similarly close on the IOS closeness measure and the explicit closeness question?
  - `t.test( friend.close, friend.IOS, paired=T )`

# REGRESSION FOR DIFFERENCES BETWEEN TWO MEANS

---

- *The linear model is the most useful function in R*
- Save your results in an object and print them using the `summary()` function
- Two functions of great value:
  - `lm()`: general linear model (i.e., with a normally-distributed dependent variable)
    - `linear.model <- lm( formula, data=dataframe.object )`
    - `summary( linear.model )`
  - `glm()`: generalized linear model (i.e., dependent variable has any distribution from the exponential family)
    - `generalized.linear.model <- glm( formula, data=dataframe.object, family=family.object )`

# “FORMULA” SYNTAX

---

- Dependent variables are predicted by a tilde “ $\sim$ ”
  - So, the formula to regress “y” on “x” is:  $y \sim x$
- The intercept ...:
  - Is an additive term in the model, represented with the number 1
  - Is estimated by default, so you do not have to declare it
    - e.g.,  $y \sim 1 + x$  is equal to  $y \sim x$
- Linear main effects are added to the model with a “+” sign

# ANOVA

---

- *ANOVA is simply a different way of evaluating explained variance in linear modelling*
- R represents this by having the “`anova()`” function be a wrapper around `lm()`
- You must always wrap the `anova()` function around a `lm()` function:
  - CORRECT:
    - `anova( lm( y ~ x, data=dataframe.object ) )`
  - INCORRECT:
    - `anova( y ~ x, data=dataframe.object )`

## EXAMPLE: 1-WAY ANOVA IN R

---

- Does happiness differ between people who have and haven't paid for sex?
  1. Run the linear model that tests whether happiness is predicted by purchasing a prostitute.
  2. Run an ANOVA to find the F-statistics and  $df$  that correspond to this test

# FACTORIAL ANOVA

---

- *Factorial ANOVA involves multiple factors, typically interacting with each other*
- Best to use `Anova{car}`
  - Anova can give you “Type III Sums of Squares”
    - Output will match what you get in SAS and SPSS
- Usage:
  - `Anova( linear.model, type=3 )`

## EXAMPLE: FACTORIAL ANOVA IN R

---

- Do men and women report the same number of kids and does marital status affect reported number of kids?
  1. Run a linear model predicting number of children from sex and whether a person has ever been married
  2. Use Anova with Type III sums of squares to find  $F$ -statistics and  $df$

# REPEATED MEASURES ANOVA IN R

---

- *Repeated-measures and mixed-effects ANOVA involve at least one variable that is measured repeatedly*
- Use the `aov()` function, specifying the participant as an “error stratum”
- Usage:
  - `aov(y ~ repeated.x1*x2  
+ Error(subject.id/repeated.x1) )`

## EXAMPLE: MIXED-EFFECTS ANOVA IN R

---

- Does cortisol change over time differently based on condition of the TSST?
- Run a mixed-effects ANOVA predicting cortisol from time and condition

# ESTIMATING COVARIANCE RELATIONSHIPS

---

- Find the correlation of two items:
  - `cor( item.1, item.2, use="complete.obs" )`
- Testing for the significance of a correlation between two items:
  - `cor.test( item.1, item.2, use="complete.obs" )`
- Find the correlation of multiple items:
  - `cor( dataframe.object, use="complete.obs" )`
- Testing for the significance of a correlation between multiple items:
  - `library( psych )`
  - `corr.test( dataframe.object )`

## EXAMPLE: CORRELATION AND CORRELATION TEST

---

- Is there a relationship between a respondent's self-esteem and their feelings of closeness with their friends?
- Find the straight-up correlation (effect size)
- Run a *t*-test on this correlation to find its significance

# PARTIAL CORRELATION

---

- *Partial correlation estimates the correlation between two variables after their shared variance with a third variable has been “partialled out”*
- `partial.r{ psych }`
  - `library(psych)`
  - `partial.r( dataframe.object,  
column.numbers.to.correlate,  
column.numbers.to.partial.out )`
  - `corr.p( partial.r.object, n=sample.size )`

## EXAMPLE: CORRELATION AND CORRELATION TEST

---

- Is there a relationship between self-esteem and friendship closeness when comfort with a friend is taken into account?
- Find the partial correlation between self-esteem and friendship closeness, partialling out comfort with friend

# GENERAL LINEAR MODEL: SIMPLE REGRESSION

---

- You already know linear modelling in R!
  - Use `lm()` or `glm()` functions
- Usage:
  - `summary( lm( y ~ x ) )`
  - `summary( glm( y ~ x ) )`
- Get standardized slopes with `lm.beta{QuantPsyc}`
  - `lm.beta( model.object )`
- Get effect sizes with `getDeltaRsquare{rockchalk}`
  - `getDeltaRsquare( model.object )`

# EXAMPLE: GLM WITH ONE PREDICTOR

---

- Does self-esteem predict greater friendship closeness?
  - Centre self-esteem first
  - Run a glm and examine effects

# MODERATED REGRESSION

---

- Still use the `lm()` or `glm()` functions
  - e.g., `lm( y ~ x1 * x2, data=dataframe.object )`
- The difference between simple regression and moderated regression is simply in the symbols you use in the formula
  - Main effects are always added with a “+”
  - Interaction effects are added using the “\*” or “:” signs, with “\*” automatically including the lower-order effects for the interaction
    - $y \sim x1 * x2$  is equal to  $y \sim 1 + x1 + x2 + x1:x2$
    - $y \sim x1:x2$  is equal to  $y \sim 1 + x1:x2$

# EXAMPLE: MODERATED REGRESSION

---

- Is the relationship between self-esteem and friendship closeness moderated by how comfortable people feel with their friends?
- Centre the moderator, comfort with friend
  - *Note:* We already centered self-esteem

# GENERALIZED LINEAR MODELS

---

- *Generalized Linear Models predict non-normal dependent variables (DV)*
  - Simply add a “family=...” argument to the `glm()` function!
- Most common examples:
  - Logistic Regression
    - *Used when your DV is a binary variable coded with 0 or 1 (e.g., yes/no, correct/error present/absent)*
    - `summary( glm( y ~ x, family=binomial ) )`
    - *Note: You must ensure your DV is coded with zeros and ones*
  - Poisson Regression
    - *Used when your DV is an integer count variable*
    - `summary( glm( y ~ x, family=poisson ) )`
    - *Note: You must ensure your DV is a positive integer*

# EXAMPLE: LOGISTIC REGRESSION

---

- Does the amount of TV that Americans watch in a day predict whether they own a gun?
- Run a logistic regression predicting gun ownership from TV watching

## EXAMPLE: POISSON REGRESSION

---

- Does the amount of TV that Americans watch in a day predict the number of children they have?
- Run a poisson regression predicting number of children from TV watching

# CHI-SQUARE TEST

---

- *Frequently we want to know if observations were distributed equally across all possible outcomes, for which we would conduct a Chi-square test*
- In R,  $\chi^2$  tests are conducted by wrapping a `chisq.test()` function around a `table()` function
  - Goodness of Fit Test for 1 Factor:
    - `chisq.test( table( object.1 ) )`
  - Test of Independence for Multiple Factors:
    - `chisq.test( table( object.1, object.2, ... ) )`

# EXAMPLE: CHI-SQUARE TESTS

---

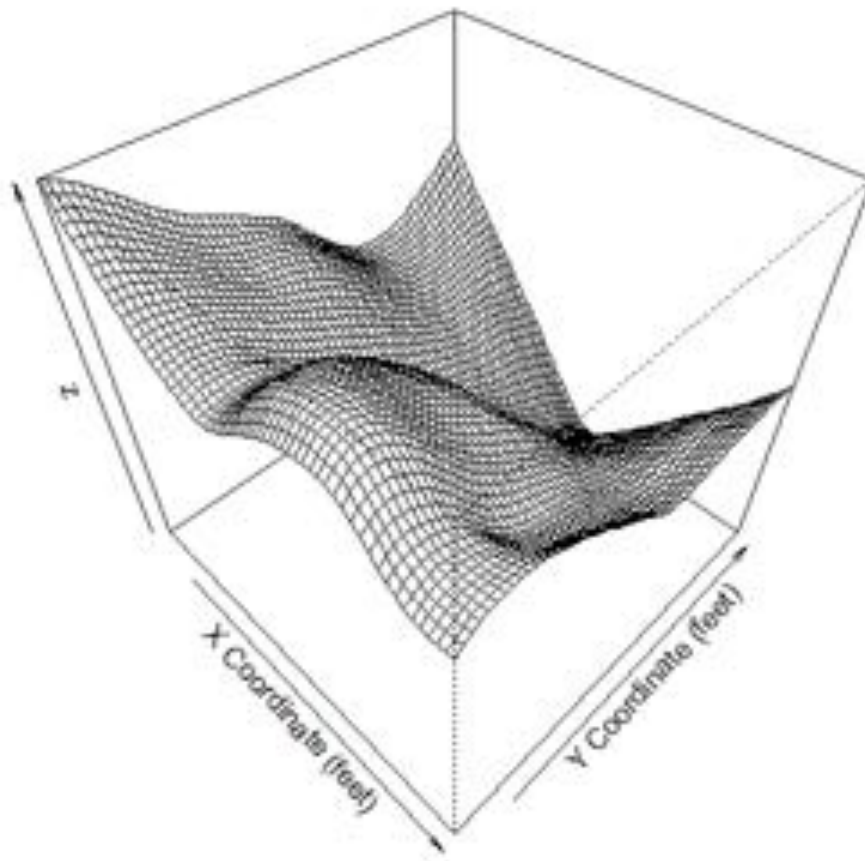
- *Goodness-of-Fit*

- Was experimental condition (“friend.type”) equally distributed across participants in the friendship.data?

- *Test of Independence*

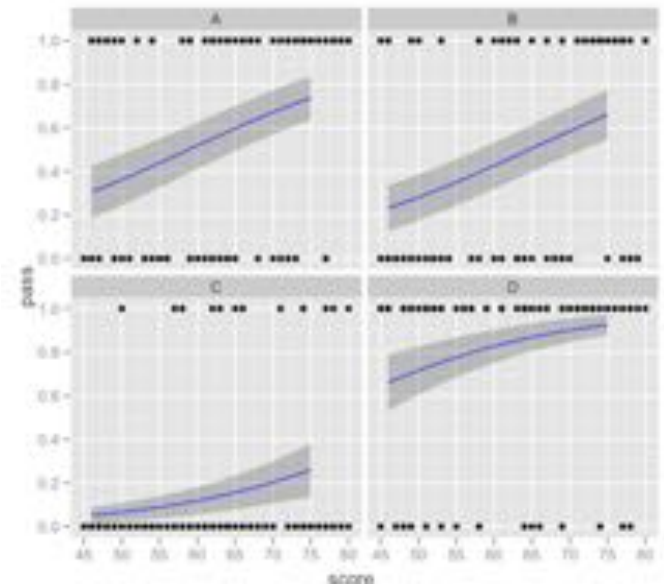
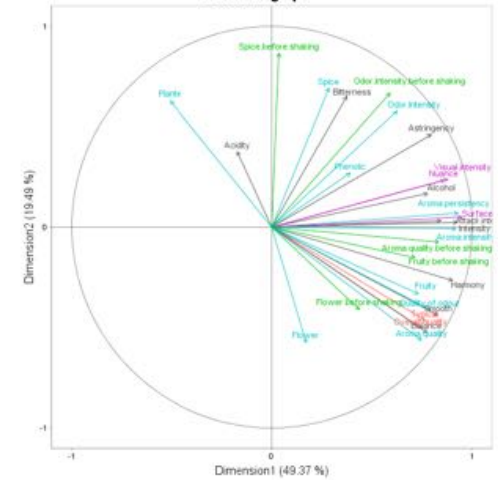
- Was experimental condition equally distributed across both sexes of participants?

Surface elevation data



- active group odor
- active group visual
- active group odor after shaking
- active group taste
- supplementary group overall

Variables graph



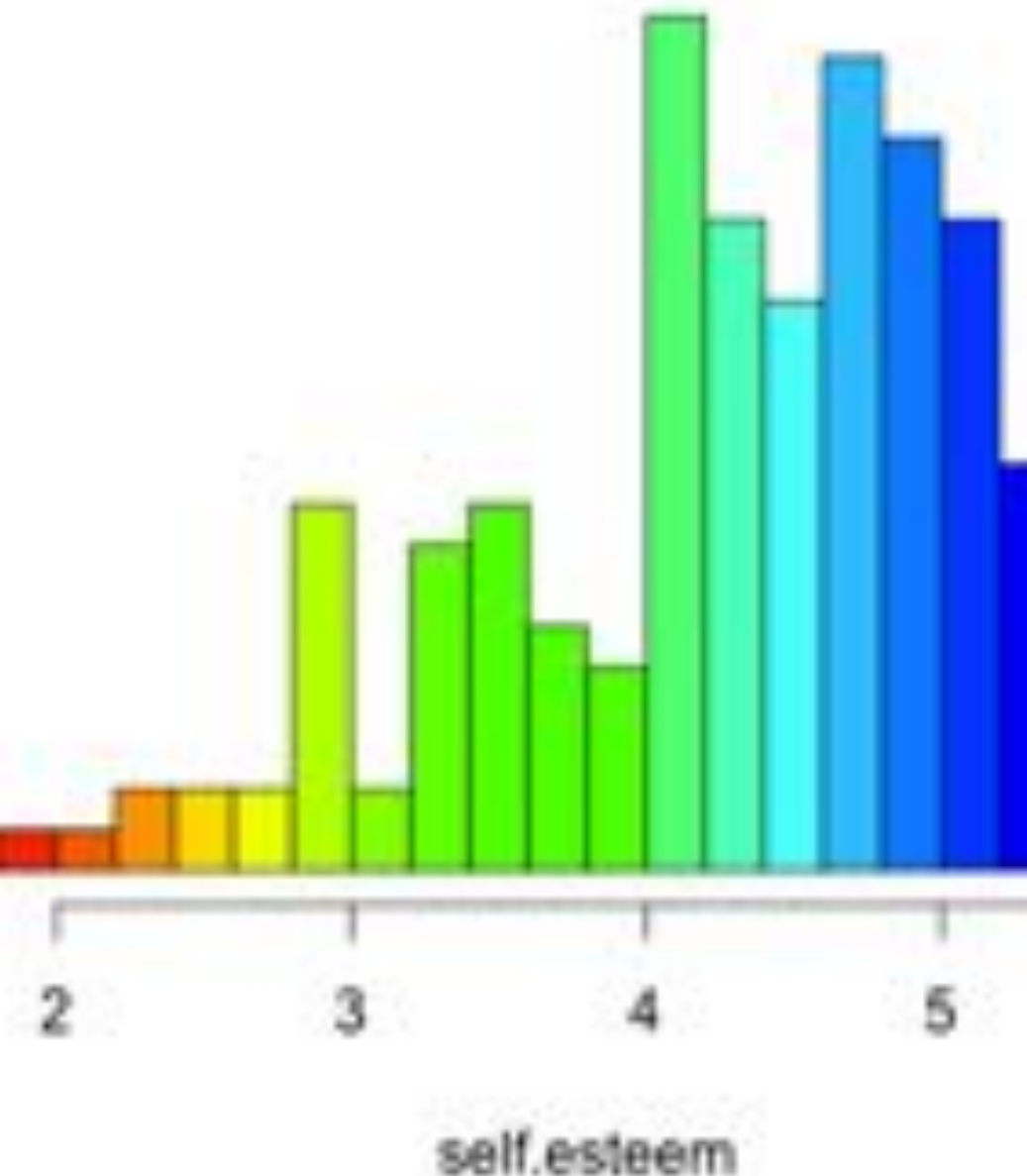
Graphing in R

# PLOTS

---

- *R* provides powerful graphing capabilities
- Commonly-used built-in plots:
  - Histogram: `hist()`
  - Scatterplot: `plot()`
  - Boxplot: `boxplot()`
- Most plots take similar arguments

## Histogram of self.esteem



## HISTOGRAM

- *Displays the frequency or probability of values for a numeric object*
- General usage:
  - `hist( numeric.object )`
  - Change the number of bars:
    - `hist( numeric.object, breaks=20 )`
  - Plot probabilities instead of densities:
    - `hist( numeric.object, freq=F )`

# SCATTERPLOT

.....

➤ *A plot of points that intersect in multiple dimensions*

➤ `plot( x, y )`

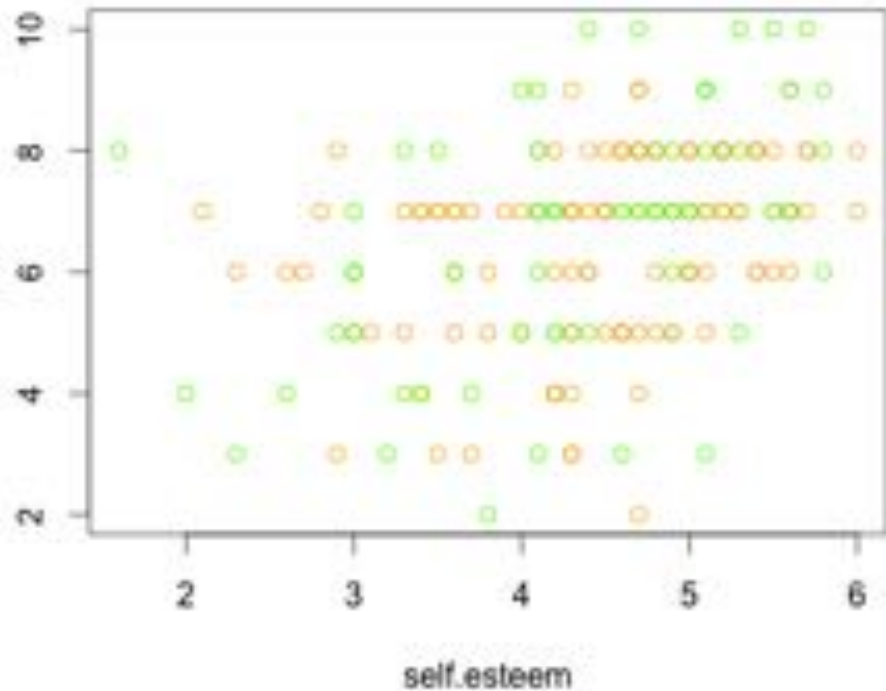
➤ Add a little noise around integer responses:

➤ `plot( x, jitter(y) )`

➤ Add dimensions through plot features:

➤ `z.colours <-  
 ifelse( z==1,  
 "green", "orange" )`

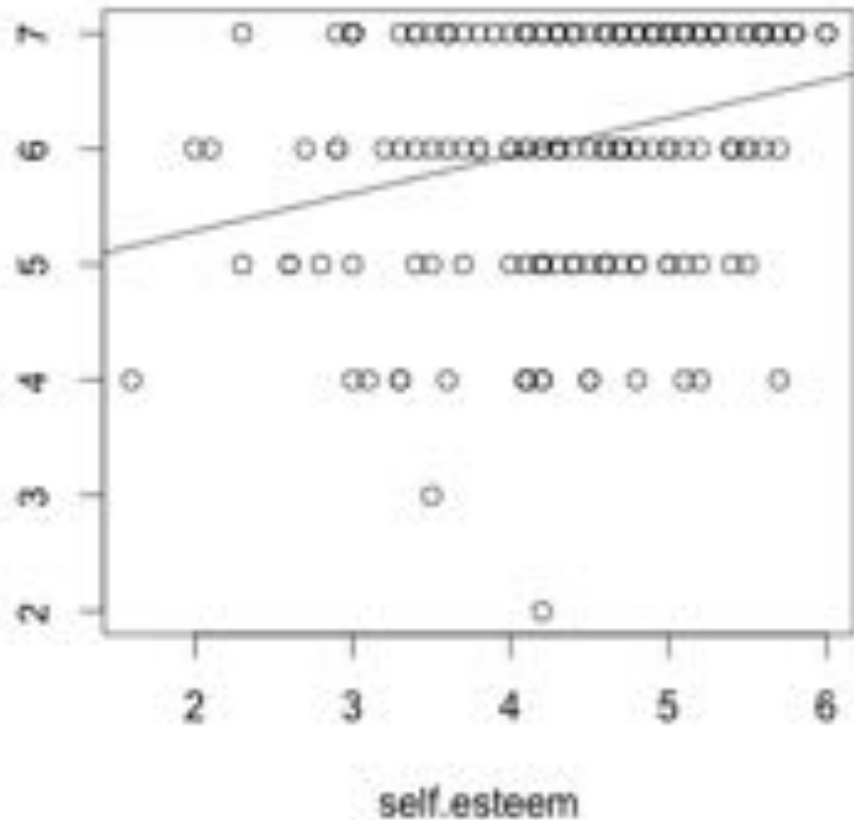
➤ `plot( x, y,  
 col=z.colours)`

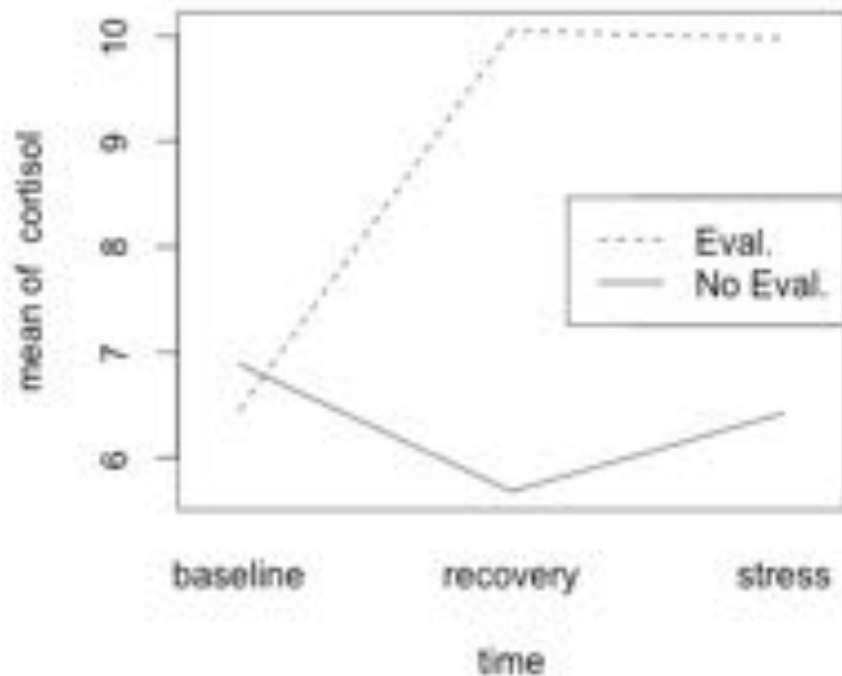


# ADDING LINES TO SCATTERPLOTS

.....

- Two-step process:
  - Create your scatter plot with `plot()`
    - `plot( x, y )`
  - Add lines one at a time with the `abline()` and `lm()` functions
    - `abline(lm( y ~ x ))`





# EASY INTERACTION PLOTS

.....

➤ For categorical predictors:

➤ `anova.model <- lm( y ~ x1 * x2 )`

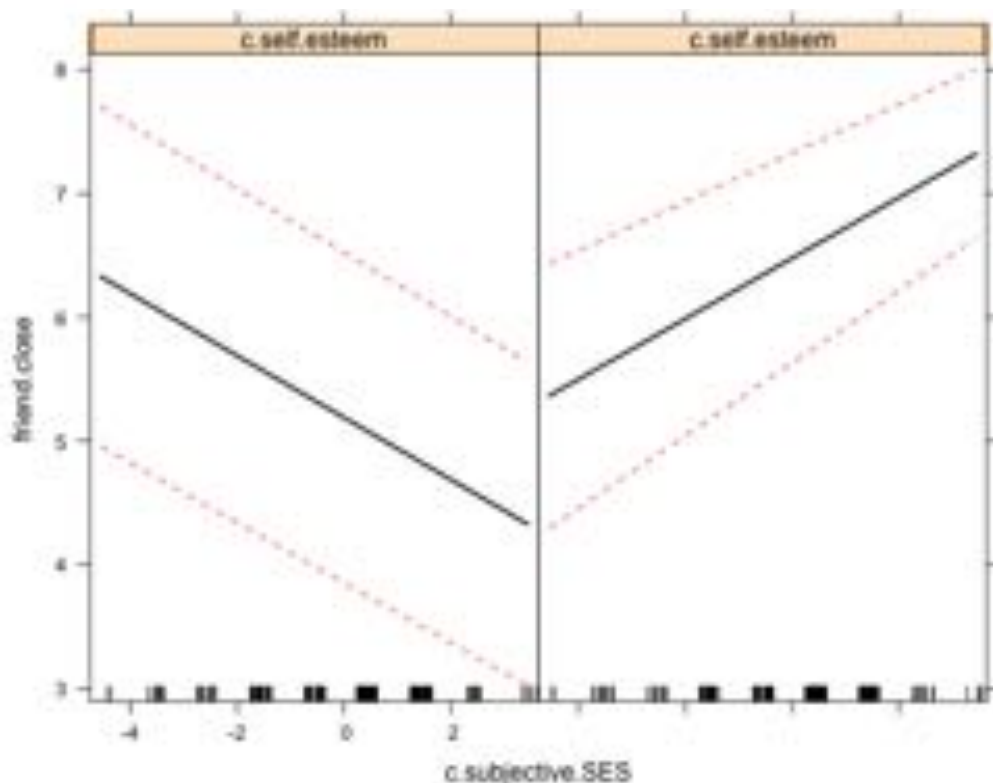
➤ `interaction.plot( x1, x2, y )`

➤ For continuous predictors:

➤ `effect{effects}`

➤ `moderated.regression <- lm( y ~ x1 * x2 )`

➤ `plot( effect( "x1 * x2", moderated.regression, default.levels=2 ) )`



# BOXPLOTS

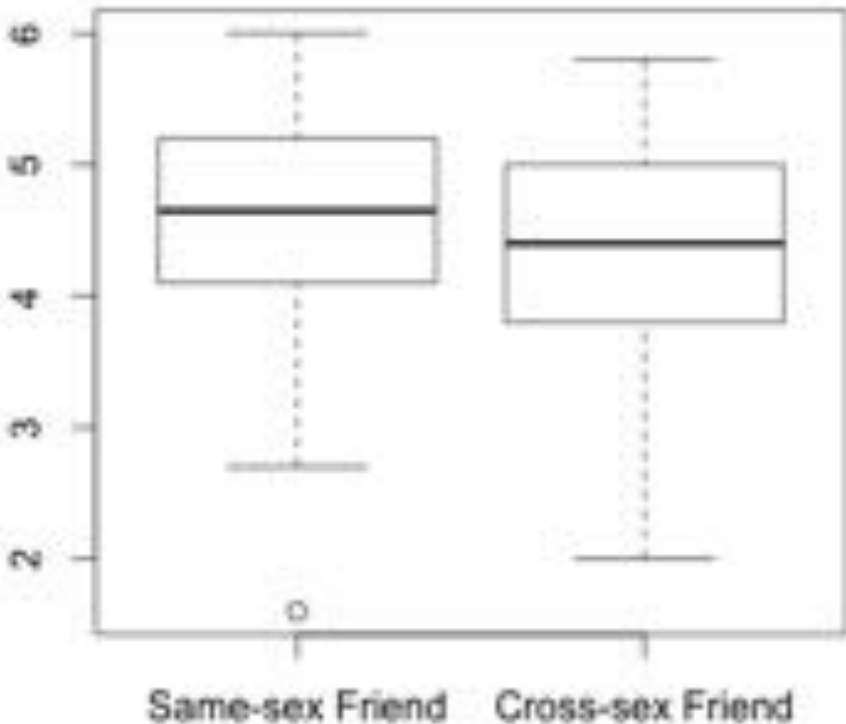
.....

➤ *Boxplots provide plots of centrality and spread for different levels of a category*

➤ General Usage:

➤ `boxplot( formula,  
data=dataframe.object  
)`

➤ e.g., `boxplot( y ~  
factor)`



# PLOT META-DATA

---

- Most plots in R have plot “parameters” as arguments
- Common parameters:
  - Title of Plot: `main=“Title String”`
  - X-axis Label: `xlab=“Axis Title String”`
  - Y-axis Label: `ylab=“Axis Title String”`
  - X-axis Limits: `xlim=c([lower limit], [upper limit])`
  - Y-axis Limits: `ylim=c([lower limit], [upper limit])`
  - Colours: `col=“[colour name]”`
  - Point Styles: `pch=[style number]`
  - Line Styles: `lty=[style number]`

# MATRIX OF PLOTS

---

- Using the parameter `par()` function, you can tell R how to display plots:
  - Grid of plots: `par( mfrow=c( [rows],[columns] ) )`
  - Default is: `par( mfrow=c(1,1) )`

# SAVING A PLOT WITH IMAGE FUNCTIONS

---

➤ Saving a plot has three steps:

1. Identify the format and file name in which the plot will be written

➤ General form: `format( "filename" )`

➤ Available formats: `pdf()`, `png()`, `jpeg()`, `bmp()`, `postscript()`

2. Run your plot command (e.g., `plot( x, y )`)

3. Restore the graphical output window:

➤ `dev.off()`

4. Example:

➤ `png( "file.png" )`

➤ `plot( x, y )`

➤ `dev.off()`

## EXAMPLE: PLOTS IN R

---

1. Create histograms for each of these 3 variables from friendship data in a 3-row matrix:
  - `friend.close`, `self.esteem`, `friend.comfort`
2. Use the `effects` package to create an interaction plot for the moderated regression we ran earlier
3. Save the interaction plot on your harddrive as “Figure 1.png”



# INTERACTING WITH DATA IN R

---

# CLASSES

---



- All objects have a “class”
  - *Way of categorizing objects in R*
  - Classes will most directly affect you through data classes
- You can always learn an object’s class if you don’t know it:
  - `class( object )`

# DATA CLASSES

---

- *Types of data (stored in 1-dimensional “vectors”)*
  - Numeric
  - Character
  - Factor
  - Boolean
- *Ways to store data*
  - Data Frames
  - Lists
  - Matrices
- Find the class of an object with `class(): class( object )`

# NUMERIC

.....

- *Any real number*
  - Floating point precision ranging from  $|2 \times 10^{-308}|$  to  $|2 \times 10^{308}|$
- Many analyses will require data to be numeric
  - E.g., t-test, correlation, regression



# CHARACTER

.....

- *Alphanumeric strings, stored as elements in a 1-dimensional array*



# FACTORS

.....

- *Alphanumeric strings, stored as ranked elements in a 1-dimensional array*
- *Factors have meta-data that contains the values of all category levels*
- Important note:
  - Factors are case-sensitive
    - e.g., “Hello!” ≠ “hello!”



# FACTOR VALUES VS. FACTOR LABELS

---



- Factors can be used to attach alphanumeric labels to numeric values

- `object <- c( -1, 1, -1, 0, 1 )`

- `as.factor( object )`

```
[1] -1 1 -1 0 1
```

```
Levels: -1 0 1
```

- `factor( object,  
 levels=c(-1, 0, 1),  
 labels=c("low", "medium", "high")  
 )`

```
[1] low high low medium high
```

```
Levels: low medium high
```

# FACTOR() AND ORDERED()



- There are two types of categorical variables represented by R, and you can set their meta-data how you want
  - Unordered (a.k.a, “nominal”) = `factor()`
  - Ordered (a.k.a., “ordinal”) = `ordered()`
    - `as.ordered( object )`  

```
[1] low medium high low high
```

```
Levels: high < low < medium
```
    - `ordered( object, levels=c(“low”, “medium”, “high”) )`  

```
[1] low medium high low high
```

      - `Levels: low < medium < high`
- For the most part, you will store categorical variables as factors

# BOOLEAN

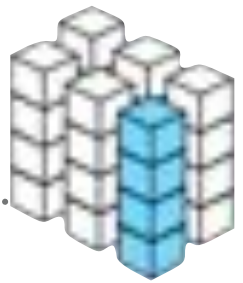
.....

- Boolean vectors contain the results of logical tests
- Booleans can take one of two values:
  - TRUE (also “T”)
  - FALSE (also “F”)
- Booleans will be returned as the result of a test, or can be used as data themselves



# VECTORS

---



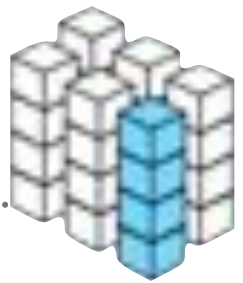
- *1-dimensional arrays of which all elements share the same class*
- The length of a vector equals its number of elements
  - You can find the length of a 1-dimensional object with `length()`
  - Example, where `vector.object` has 3 elements: accountant, mechanic, architect
    - `print( vector.object )`  

```
[1] "accountant" "auto mechanic"
 "architect"
```
    - `length( vector.object )`  

```
[1] 3
```

# CREATE A NEW VECTOR

---



- Vectors are created with the “combine” function, `c()`
  - `new.vector <- c( element.1, element.2 )`
- R automatically assigns vectors to a class, if a class is not specified
  - Combining all numbers?
    - `new.vector <- c( 1, 2, 9, 2, 3 )`
    - `class( new.vector )`  
[1] "numeric"
  - Combining any non-numeric elements
    - `new.vector <- c( 1, 2, 9, "puppy", 3 )`
    - `class( new.vector )`  
[1] "character"

# MANIPULATING DATA CLASSES

---

- If you think you know an object's class, you can ask R if you are right:

- `is.numeric( object )`

- `[1] FALSE`

- If you want to change an object's class, you can tell R to do so:

- `new.object <- as.numeric( object )`

- `is.numeric( new.object )`

- `[1] TRUE`

# IMPORTANT NOTE ABOUT MANIPULATING DATA

---



- NEVER FORGET: R does not understand you nor your data ...
- Always take a moment to think when you do something
  - Using a source/syntax file helps you take a moment
- Common way to go wrong with `as.numeric()`
  - If `is.factor( object ) == TRUE`
    - `as.numeric()` will rank all factor levels, sorting numbers in order first and then strings in alphabetical order second
  - If `is.character( object ) == TRUE`
    - `as.numeric()` will return numbers as numbers but non-numbers will become NA

# DATAFRAME

---

| Index | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9   |
|-------|----|----|----|----|----|----|----|----|----|-----|
| 0     | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10  |
| 1     | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20  |
| 2     | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30  |
| 3     | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40  |
| 4     | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50  |
| 5     | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60  |
| 6     | 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70  |
| 7     | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 80  |
| 8     | 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90  |
| 9     | 91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99 | 100 |

- A 2-dimensional (row X column) array that contains data objects arranged in columns
- Dataframes have meta-data
  - Names: *Variable names, names of columns*
  - Dimensions: *Two dimensions: (1) number of rows and (2) the number of columns*
  - Row names:
    - *Row names are sequential row numbers by default, but they can be set to have semantic labels, too*
- All values stored in the same column of a data frame have the same class (e.g., numeric), but data classes can vary across columns in a dataframe

# VIEW()

---

- *A method for looking at your dataframe in a spreadsheet-like format*
  - `View( dataframe.object )`
- Note that:
  - R will truncate the number of columns you can see
  - You cannot edit the dataframe that you view

# FINDING OUT STUFF ABOUT DATAFRAMES

---

- What are the names of the variables stored in the dataframe?
  - `names( dataframe.object )`
- What are the dimensions of the dataframe (Row, Column)?
  - `dim( dataframe.object )`
- How many rows does the dataframe have?
  - `nrow( dataframe.object )`
- How many columns does the dataframe have?
  - `ncol( dataframe.object )`
- What are the row names of the dataframe?
  - `row.names( dataframe.object )`

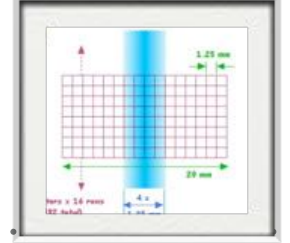
## EXAMPLE: EXPLORING DATA FRAME PROPERTIES

---

1. Print the variables names of `friendship.data`
2. Find out how many observations are in `friendship.data`

# ACCESSING OBJECTS IN A DATAFRAME

---



- Access a specific variable (“variable.name”) in a data.frame in one of two ways:
  1. Use with(): with(dataframe.object, variable.name)
  2. Use “\$”: dataframe.object\$variable.name
  3. Use “[]”: dataframe.object[,“variable.name”]
- Access specific rows:
  - `dataframe.object[1, ]` # first row
  - `dataframe.object[c(1, 4:5), ]` # 1st, 4th, and 5th rows
- Access specific columns:
  - `dataframe.object[,1]` # first column
  - `dataframe.object[,c(1, 4:5)]` # 1st, 4th, and 5th columns

# CREATING A DATAFRAME

---

- New data frames can be created with the `data.frame()` command
- Use cases:
  - Creating a dataframe, keeping the original object names:
    - `new.dataframe.object <- data.frame( column.1, column.2, column.3 )`
  - Creating a dataframe with new names for the first two objects:
    - `new.dataframe.object <- data.frame( New_Name_1=column.1, New_Name_2=column.2, column.3 )`
  - Creating a dataframe from new data:
    - `new.dataframe.object <- data.frame( New_Name_1=c( 1, 2, 3, 4, 5 ), Name_2=c( "a", "b", "c", "d", "e" ) )`

# GOOD THINGS TO KNOW ABOUT DATAFRAMES

---

- All column names must be unique
  - When there is a conflict, R adds the suffixes of “.x” and “.y” to the first and second instances of identical variable names
- Every object stored in a column must have the same length (i.e., number of rows) as the other columns

# LISTS

---

- *A multidimensional array that contains objects of varying lengths*
- Objects stored in the same list can have different dimensions
  - Data frames are a special type of list, where the objects all have the same lengths
- Lists are created with `list()`
  - ```
new.list = list(  
  object.1 = c(1, 3, 0, 8, 4, 3),  
  object.2.2 = c("orange",  
    "banana"),  
  object.2.3 = c(-1, NA, 3, 6)  
)
```

HIERARCHICAL LISTS

➤ *Lists can be stored within lists*

```
➤ hierarchical.list <- list(  
  list.1 = (  
    object.1.1,  
    object.1.2),  
  list.2 = (  
    object.2.1,  
    object.2.2,  
    object.2.3)  
)
```

➤ ... this knowledge will probably come in handy if you get into advanced data management and other wizardry with R

MATRIX



- *A 2-dimensional array (row X column) that has no meta-data*
- A matrix is identical to a dataframe, except that it has no row and column names
- Create a new matrix:
 - `new.matrix <- matrix(NA, nrow=1, ncol=42)`
 - `new.matrix <- cbind(object.1, object.2, object.3)`

COMMON ELEMENTS OF DATAFRAMES AND MATRICES

- Elements in a matrix or dataframe can be accessed by their row and column numbers
 - `cell.at.third.row.second.column <- matrix.object[3, 2]`
 - `entire.third.row <- matrix.object[3,]`

READING IN DATA



- There are many core and add-on functions for reading data files
- The main function is `read.table()`, with the more specific:
 - `read.csv()`: comma-delimited files
 - `read.delim()`: tab-delimited files

READ.TABLE()

- *read.table()* is the generic function for reading data into R as dataframes
- General usage:
 - `tabular.data <-
 read.table("[filename_on_your_computer]",
 sep="[delimiter]", header=TRUE)`
- TIP: Remember to assign the returned data to an object
 - This is usually useless:
`read.table("comma.delimited.infile.csv", sep=",",
header=TRUE)`
 - This is ready for analysis:
`infile.data <- read.table("comma_delimited_infile.csv",
sep=",", header=TRUE)`
- `Read.table()` has many options, so `read.csv()` and `read.delim()` were written as a `read.table()` with a specific set of defaults

READ.CSV()

- *Reads a delimited file into R as a data frame using a comma as the default delimiter*
- General usage:
 - `comma.delimited.data <-
 read.csv("[filename_on_your_computer].csv"
)`

READ.DELIM()

- *Reads a delimited file into R as a data frame using a tab as the default delimiter*
- General usage:
 - `tab.delimited.data <-
 read.delim("[filename_on_your_computer].txt"
)`

ADVANCED DATA MANAGEMENT

THE PERILS OF ATTACH()

- *attach()* is a function that places all the variable names of a *data.frame* in your global namespace
 - The corresponding *detach()* function will remove a dataset
- Benefits:
 - You can reference variables from a dataset directly, without specifying the dataset
- Cons:
 - Operations conducted on these variables are stored in the global namespace, not reflected in the original dataset
 - Increased chance for namespace conflicts, especially if *detach()* is not used rigorously

WITHIN()

- *The function `within()` provides a method for performing operations within a particular dataset*
- Any action performed inside a `within()` statement is conducted with a particular dataset and makes changes to that dataset
- Note:
 - You must always save the resulting object from a `within()` statement
- General Form:
 - `dataframe.object <- within(dataframe.object, {...}),` where `{...}` = a series of syntax commands

WITH()

- *The function `with()` is used to declare a dataset to use as the namespace for almost any operation*
- The actual function called with `with()` dictates the type of object that is returned
- General Form:
 - `with(dataframe.object, ...)`, where `...` is a syntax command

MERGE()

- *The merge() function can be used to add variables to a dataset or merge two datasets together*
- The default is to merge based on all columns with matching names
- General form:
 - `merged.dataset <- merge(data.1, data.2)`
 - `merged.dataset <- merge(data.1, data.2, by="id")`
 - *Note:* You must save the merged dataset as a new object

NOTES ON MERGE()

- *Want to keep rows from one of the datasets that are not represented in the other dataset?*
 - `merged.dataset <- merge(data.1, data.2, all.x=T)` #data.1 favoured
 - `merged.dataset <- merge(data.1, data.2, all.y=T)` #data.2 favoured
 - `merged.dataset <- merge(data.1, data.2, all.x=T, all.y=T)` #Both sets

EXAMPLE: MERGING DATASETS

1. Read in the datafile called “working_memory.csv”

- It is comma-delimited
- Variables names are on the first row, data starts in the next row

2. Merge friendship.data with the working memory data

- The subject variable match the subject numbers in friendship.data

3. Look at the newly merged data with View()

RBIND()

- *The `rbind()` function can be used to add cases to an existing dataset (i.e., merge additional rows)*
- Both dataframe objects must have the same number of columns and column names
- General form:
 - `full.data <- rbind(first.data,
second.data)`

SUBSET()

- *The subset() function allows you to take a subset of a dataframe based on a logical operation*
- General form:
 - `subset(dataframe.object,
logical.statement)`
 - For example:
 - `male.data <- subset(example.data,
sex=="male")`

HEAD() AND TAIL()

- *The head() and tail() functions print the first 5 rows and last 5 rows of a data frame object, respectively*
- General Form:
 - `head(dataframe.object)`
 - `tail(dataframe.object)`

DIM() AND LENGTH()

- *Sometimes it is useful to know an objects dimensions, for which `length()` and `dim()` are used for vectors and multidimensional data objects, respectively*
- `length()` is for vectors, and it returns the number of elements in the vector
 - `length(x)`
- `dim()` is for matrices or dataframes, and it returns the number of rows and the number of columns, in that order
 - `dim(dataframe.object)`

MANIPULATING CHARACTERS

- Strings are an integral part of data processing
- R contains many powerful functions for manipulating strings

SUBSTR()

- *Subset a string*
- General Usage:
 - `substr(string.object, start, stop)`

STRSPLIT()

- *Split a string along some value*
- General usage:
 - `returned.list <- strsplit(string.object,
"delimiter")`

TOUPPER() AND TOLOWER()

- *These two functions change the case of strings*
 - `tolower()` converts an entire string to lowercase
 - `toupper()` converts an entire string to uppercase

SUB()

- *Equivalent of find-and-replace, sub() substitutes one part of a string with another*
- General Usage:
 - `sub("replacement pattern", "replacement",
string.object)`

PASTE()

- *The paste() function can be used to combine strings intelligently*
- General Form:
 - `paste(first.value, second.value, sep="")`
 - If one of the values is a constant and the other is a series, the constant is attached to each value of the series
 - Example
 - `paste("item", 1:5, sep="")`
`[1] item1 item2 item3 item4 item5`

DIFFERENT FILE FORMATS

- *R has many libraries for importing data in other formats*
 - foreign: imports data from other stats packages
 - readxl: imports and exports data from and to Excel

EXPORTING DATA

- Use the `write.table()` function to export data as a normal delimited file
- Use `write.xlsx{xlsx}` to export data as an excel spreadsheet
- Use `write.foreign{foreign}` to export data in another stats package format



DEALING WITH MISSING DATA IN R

MISSING VALUES IN R



- R represents missing values as “NA”
- You need to specifically tell R what to do with NA values

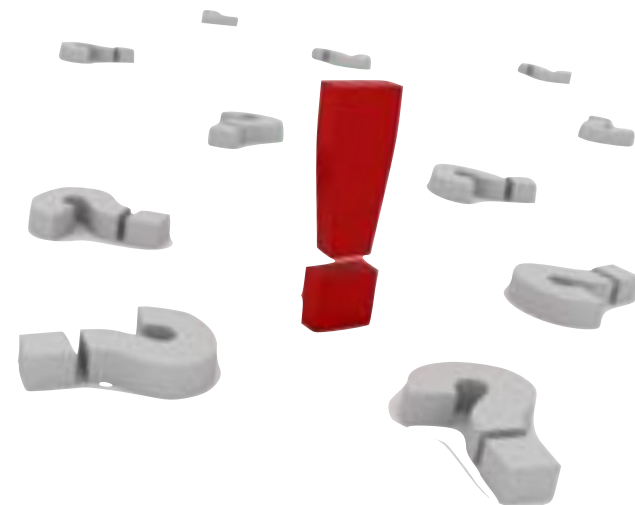
TESTING FOR MISSING VALUES WITH IS.NA()



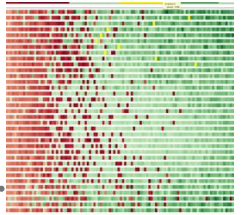
- Want to know if any values of an object are classified as missing?
 - `is.na(object)`
- Want to know how many missing values there are?
 - `sum(is.na(object))`

RECODE A VALUE AS MISSING

- You can code multiple values as missing when you read them into R:
 - `read.table("file.csv", sep="," , header=TRUE, na.strings=c("", "n/a", "N/A", "NA", "na", 999))`
- You can manually set a value to missing by using the unquoted string, NA
 - `print(x)`
[1] 2 3 2 1 5 1 2
 - `x[3] <- NA`
 - `print(x)`
[1] 2 3 NA 1 5 1 2

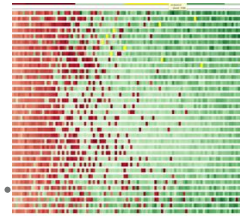


PATTERN OF MISSING DATA



- Is my data Missing Completely At Random (MCAR)?
 - The pattern of missing data is completely randomly distributed across all variables collected
- Is my data Missing At Random?
 - The pattern of missing data is not completely random, but it can be explained with other variables collected by the researcher
 - *If so, be sure to include the variables that are related to missingness as covariates in your analyses*
- Is my data Missing Not At Random?
 - The pattern of missing data is clearly not random, but none of the variables collected by the researcher can explain the pattern of missingness
 - *Inferential hypothesis testing is not valid*

TEST THE PATTERN OF MISSING DATA



- The `LittleMCAR{BaylorEdPsych}` function can test whether data is MCAR
- General Usage:
 - `mcAR.test <- LittleMCAR(dataframe.object)`
 - `mcAR.test$chi.square; mcAR.test$p.value`
- If it is not MCAR, then you should conduct a series of logistic regressions to try to predict missing values in the data
 - `y.missing <- ifelse(is.na(y), 1, 0)`
 - `glm(y.missing ~ x1 + x2, family=binomial)`

PAIRWISE DELETION



- *Pairwise deletion is a method of dealing with missing values by excluding any cases that have missing values on variables used in a current analysis*
- Pairwise deletion is conducted in R at the level of the analysis
 - E.g., `mean(x, na.rm=TRUE)`
 - E.g., `cor(x, y, use="pairwise.complete.obs")`
 - E.g., `lm(x, na.action="na.exclude")`
- Only valid when data is MCAR, and may still not be valid for your overall analysis

LISTWISE DELETION



- *Listwise deletion is a method of dealing with missing data by only including cases that have complete data*
- Valid only when data is MCAR
- The `na.omit()` function removes all rows from a dataset that contain at least one missing value
- ```
all.complete.cases.data <-
na.omit(dataframe.object)
```

# MULTIPLE IMPUTATION

---



- *Multiple imputation is a method for dealing with missing data when data is MCAR or MAR*
- Missing data points are “imputed” from the observed data points
  - This imputation is done multiple times to ensure that the imputation method is stable
- Your analysis is run once in each imputation
- Parameters are pooled across the imputations

# MULTIPLE IMPUTATION IN R

---



➤ *The mice package provides a number of helpful, simple functions for multiple imputation*

➤ 3 steps:

1. Impute some datasets

➤ `imputations <- mice( dataframe.object )`

2. Run an analysis on each imputed dataset

➤ `imputed.analysis <- with( imputations, lm( y  
~ x ) )`

3. Pool together the results of the analysis across the imputed datasets

➤ `summary( pool( imputed.analysis ) )`

# EXAMPLE: MULTIPLE IMPUTATION

---

1. Install and load the “mice” package
2. Centre self-esteem and SES in friendship data
3. Create 5 imputed datasets for friendship data
4. With the imputed datasets, run a moderated regression predicting working memory from self-esteem and SES
5. Pool the results to determine the regression results

INPUT  $x$

FUNCTION  $f$ :

# ADVANCED FUNCTIONS

---

*Useful Functions and Custom Function*

OUTPUT  $f(x)$



# GOOD FUNCTIONS TO KNOW

.....

- Mathematical Operators
- Logical Operators
- Setting R preferences with `options()`
- Hodgepodge of useful functions

# MATHEMATICAL FUNCTIONS

---

- Some math operations can be done with functions:

| Function (on object x)  | Description          | Example                                    |
|-------------------------|----------------------|--------------------------------------------|
| <code>abs( x )</code>   | Absolute value       | <pre>&gt; abs( - 7 )<br/>[1] 7</pre>       |
| <code>sqrt( x )</code>  | Square root          | <pre>&gt; sqrt( 9 )<br/>[1] 3</pre>        |
| <code>log( x )</code>   | Natural Logarithm    | <pre>&gt; log( 100 )<br/>[1] 4.60517</pre> |
| <code>log10( x )</code> | Normal Log10         | <pre>&gt; log10( 100 )<br/>[1] 2</pre>     |
| <code>exp( x )</code>   | Power of $e$ (“exp”) | <pre>&gt; exp( 5 )<br/>[1] 148.4132</pre>  |

*Source: statmethods.net (Quick-R)*

# LOGICAL OPERATORS IN R

---

- *Logical operators are used to compare values in R*
- When a logical operator is used, R will return a boolean TRUE/FALSE value
- Logical operators:
  - “==” is “equal to”
  - “!” is “not”
    - “!=” is “not equal to”
  - “>” is “greater than”
  - “>=” is “greater than or equal to”
  - “<” is “less than”
  - “<=” is “less than or equal to”

# OPTIONS()

---

- *The options() function sets R preferences*
- There are numerous options for your R work session
  - e.g., options( prompt= “(: ” )  
(:
  - For a list of available options:
    - ?options
- Curious about what options you have set?
  - getOption( “[option.name]” )

# UPDATE()

---

- *The `update()` function can be used to update linear model objects*
  - This can be very useful for model comparison
- General Form:
  - `original.model <- lm( formula.object )`
  - `reduced.model <- update( original.model,  
new.formula.object )`
- Notes:
  - In formula, “.” indicates something that already existed
  - “-” and “+” can be used to subtract and add terms to the models

# SCALE()

---

- *The scale() function standardizes (i.e., Z-scores) a variable or dataframe*
- General Form:
  - `zx <- scale( x )`
- The scale() function can be used to centre a variable if the “scale” argument is set to “F”:
  - `centered.x <- scale( x, scale=FALSE )`

# LEVELS()

---

- *The levels() function tells you how many levels a factor variable has*
- General form:
  - `levels( factor.object )`

# SIGNIF(), ROUND(), FLOOR(), CEILING()

---

- *R has a number of functions for specifying the floating point precision*
  - signif() prints a number with a particular significant digits
    - Usage: `signif( x, significant.digits )`
  - round() rounds to the digits specified
    - Usage: `round( x, round.digits )`
  - floor() creates an integer by rounding down to the nearest integer
    - Usage: `floor( x )`
  - ceiling() creates an integer by rounding up to the nearest integer
    - Usage: `ceiling( x )`

# APPLY()

---

- The `apply()` function is used to apply a function across the rows or columns of a dataset
- General Form:
  - `apply( function, data.object, 1.for.rows. 2.for.cols )`
  - Calculate row means:
    - `apply( mean, dataframe.object, 1 )`
  - Calculate column means:
    - `apply( mean, dataframe.object, 2 )`

# TAPPLY()

---

- The `tapply()` function is used to apply a function within each level of another variable
- General Form:
  - `tapply( object.to.manipulate, group.object, function )`
- Common uses:
  - To find group means



# CUSTOM FUNCTIONS

---

- Nesting Functions
- Writing Your Own Functions
- Loops

# NESTING FUNCTIONS

---

- Functions can be nested in one another
  - For example, the following two sets of commands are equivalent:
    - `sum( is.na( object ) )`
    - `missing.boolean <- is.na( object )`  
`sum( missing.boolean )`
- Nested functions are evaluated at the most inner parentheses, moving outward in order

# CUSTOM FUNCTIONS

---

- *R has core methods for creating your own custom functions*
  - Use the `function()` command
  - Save the function in the global workspace

# FUNCTION()

---

- *The function() function in R is a function that creates functions*
- General Usage:
  - ```
function.name <- function( argument ) {  
    x <- argument * 100  
    return( x )  
}
```
 - ```
y <- function.name(z)
```

## EXAMPLE: CREATE A FUNCTION TO SCORE A MULTI-DIMENSIONAL LIKERT SCALE

---

1. Create your own functions called “score.bfi()” that takes a 44-item Big 5 inventory and returns a list of per-participant scores for the Big 5 personality dimensions
  - 1.1. Create a dataframe that has the 44-items in the Big 5, called “big.five.items”
  - 1.2. Create a function called “score.bfi” that takes a 44-item dataframe as an argument
    - 1.2.1. The score.bfi() function calculates five new variables, one for each personality dimension
      - Some values will need to be reverse scored
    - 1.2.2. The function returns the five new variables in one dataframe
2. Call the function, saving the returned object in a new dataframe called “bfi.scores”

# INFO NEEDED TO WRITE EXAMPLE FUNCTION

---

## ➤ *Scoring the 44-item Big 5:*

### ➤ *Openness:*

➤ *Regular items:* 25, 5, 30, 20,  
40, 44, 15, 10

➤ *Reversed items:* 35, 41

### ➤ *Conscientiousness*

➤ *Regular items:* 3, 33, 38, 13,  
28

➤ *Reversed items:* 43, 8, 23, 18

## ➤ *Extraversion:*

➤ *Regular items:* 36, 1, 26, 16,  
11

➤ *Reversed items:* 6, 31, 21

## ➤ *Agreeableness*

➤ *Regular items:* 32, 42, 7, 17,  
22

➤ *Reversed items:* 2, 12, 27, 37

## ➤ *Neuroticism*

➤ *Regular items:* 19, 14, 39, 4,  
29

➤ *Reversed items:* 34, 24, 9

# LOOPS

---

- *Loops are operations that should be performed iteratively, such as once per element of a dataset*
- Each iteration
- General Form:
  - `for( iteration.id in loop.start:loop.end) {  
 ...commands to run on each iteration ...  
}`
- Example:
  - `for( i in 1:10 ) {  
 print( paste( "item", i, sep="_" ) )  
}`

## EXAMPLE: CORRELATE SELF-ESTEEM WITH EACH PERSONALITY DIMENSION

---

1. Create a loop that prints a correlation test of the variable “self.esteem” with each of the Big 5 personality dimensions that were returned by your `score.bfi()` function

```
➤ for(i in 1:dim(bfi.scores)[2]) {
 print(names(bfi.scores)[i])
 print(cor.test(bfi.scores[,i],
 friendship.data$self.esteem))
}
```



# TROUBLE SHOOTING PROBLEMS IN R

---

# BEST PRACTICES WITH HELP FILES

---

1. Unless you REALLY know a function well, always begin your use of that function with:
  - `?function.name`
2. Then, try out your syntax on the console without saving it to an object (to see what it returns)
3. When you are satisfied with the result, save the good syntax to your script syntax file

# WARNING MESSAGES VERSUS ERROR MESSAGES

---

- R will give you two types of messages:
  - Error messages occur when R terminates a step while executing a function
    - If an error message occurs, you must fix the problem
  - Warning message occur when R had to do something that may render its actions invalid
    - If a warning message occurs, you do not have to fix the problem
    - Warning messages should be taken as a tip or a sign that something may be wrong
    - You may ignore warnings, if your decision is based on understanding the root cause of the warning

# EMBRACING ERROR MESSAGES

---

- R speaks like a person with perspective-taking problems
  - But humans have the ability to empathize with R!
  - R is usually describing its problem completely, but you have to learn to understand its peculiarities of speech
  - The more error messages you read, the more understandable R will become

# PROCESSING MULTIPLE ERROR MESSAGES

---

- Error messages have cascading influence
  - If something that is needed by later operations breaks, those later operations will also break
- If you get a bunch of error messages:
  - Scroll up to your last command prompt
  - Scroll down through the error messages, trying to solve them in sequential order
  - If you solve an early error message, run the command again, because the later error messages may disappear

# QUICK-R

---

- <http://www.statmethods.net>
- *An \*awesome\* online blog about R*
  - Provides concise examples of how to do many analyses and graphs in R



# IDEAL GOOGLE SEARCHES

---

- Start broad, then add qualifiers
  - Typically start by type “R”, a space, and then an analysis you’d like to do
- If you want examples of a function, search for “R” and the function name, then add on arguments and special cases
- When you are unable to solve an error message:
  - The error message contains some things that the error output of the function and others that are unique to your setup
  - Use the “\*” operator that represents a “wildcard” (as in, anything) in Google to search for any instance of the error output

# GOOD FORM FOR R SCRIPTS

---

- Have one “Build.R” file that reads in everything
- When you’ve gotten that right, save your workspace
- Have an “Analysis.R” file that loads the saved workspace and does all analyses
  - You can also “source” your Build.R script in Analysis.R



# GRAND SUMMARY

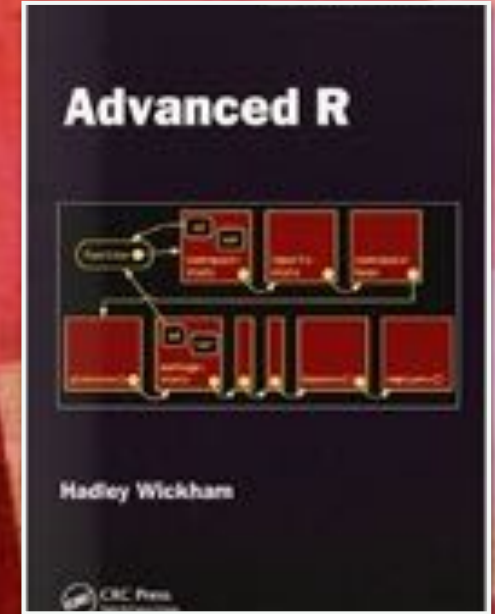
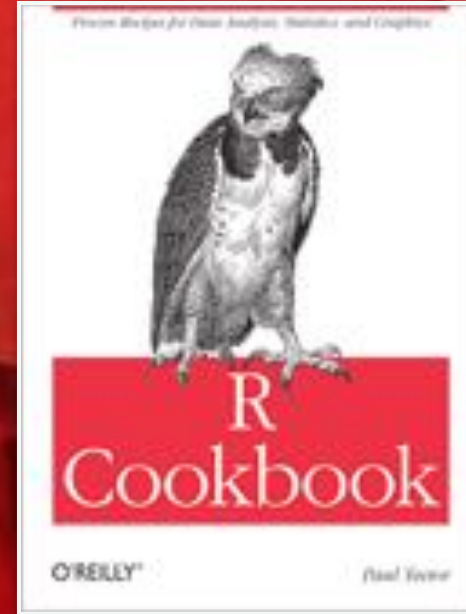
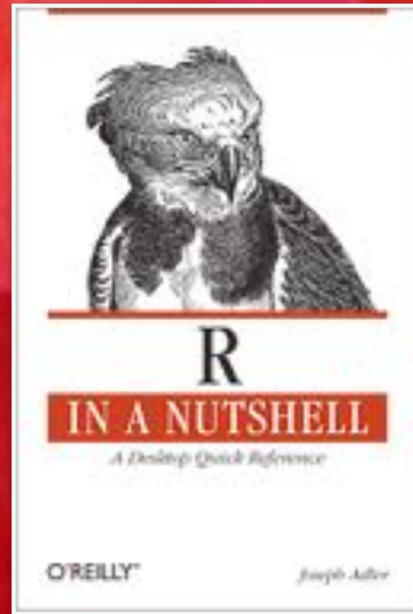
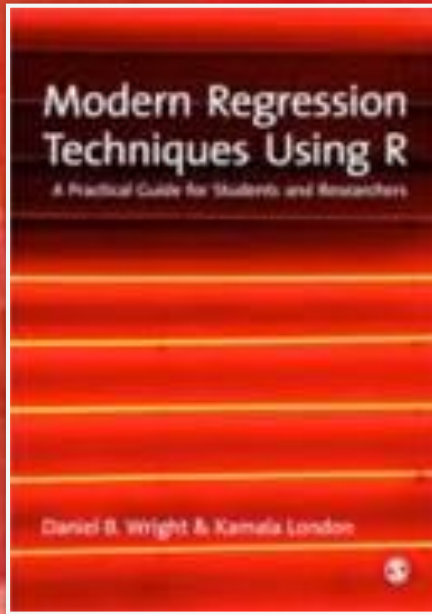
---

*Take-home Messages*

# GRAND SUMMARY

---

- What is R?
  - *A powerful statistical package*
- What can it do?
  - *Basically any analysis*
- What is so great about it?
  - *Open-source*
  - *Object-orientated approach to statistics*
  - *Increases reproducibility*
  - *Once you get used to it, it is the EASIEST package to use!*



# WANT TO KNOW MORE?

---

*Book Recommendations*



**THAT'S R IN A NUTSHELL!**

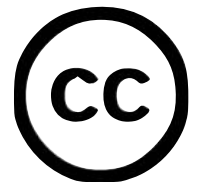
.....




# THANK YOU!!

---

- Workshop Sponsor
  - Psychology Graduate Department, University of Toronto
- Questions? Comments? Feedback?
  - [liz@psych.utoronto.ca](mailto:liz@psych.utoronto.ca)
- Workshop Materials:
  - <http://page-gould.com/r/uoft/>

# NOTE ABOUT DISTRIBUTION OF WORKSHOP MATERIALS



- The slides and syntax for this workshop are the original work of Elizabeth Page-Gould, distributed to you with Creative Commons 3.0 International License
- This means that YOU MAY:
  - Freely share, distribute, and even “remix” the slides and syntax
- BUT ONLY UNDER THE FOLLOWING CONDITIONS:
  -  ➤ With attribution: You provide attribution to Elizabeth Page-Gould with a link to the original workshop materials: <http://page-gould.com/r/uoft>
  -  ➤ Share alike: If you alter or remix this work in any way, you must also share your final product with a license that has similar conditions to this one (e.g., to be distributed freely)
  -  ➤ Non-commercial: You may not use these materials for commercial purposes without explicit permission from Elizabeth Page-Gould