

MSAN 593

Exploratory Data Analysis

Paul Intrevado

Exploratory Data Analysis

Wednesday 10th August, 2016

23:08



UNIVERSITY OF
SAN FRANCISCO

Master of Science
in Analytics

Table of Contents

- 1 Course Introduction
- 2 Introduction to R
- 3 Advanced Techniques in R
- 4 Exploratory Data Analysis in R

Section 1

Course Introduction

Section Contents

1 Course Introduction

- Who Am I
- About this Class
- Reference Textbooks

Subsection 1

Who Am I

Who Am I?

- Ph.D. Operations Management (McGill, 2015)
 - Research focused on service operations
 - Model and solve optimization problems (e.g., MIPs)
- B.Sc., M.Sc. Industrial Engineering (Purdue, 2005, 2007)
- B. Commerce (McGill, 2004)
- Assistant Prof @ USF as of August 2014
- Taught 30+ classes @ McGill, SCU and USF
- I am 100% MSAN-affiliated
- I teach
 - MSAN 593 - Exploratory Data Analysis
 - MSAN 601 - Linear Regression Analysis
 - MSAN 605/625/627/632 - Practicum
- MSAN Practicum Supervisor (2014), MSAN Practicum Director (2015-), Associate Director of DI (2016-)

What I Do

$$\min_{\mathbf{y}, \mathbf{z}^+, \mathbf{z}^-} \sum_{j \in \mathbb{J}} \sum_{\nu \in \mathbb{V}} \sum_{t \in \mathbb{T}} r^{(t-1)} \left[\sum_{\omega \in \Omega} \left[f_{j\nu\omega}^+ z_{j\nu\omega t}^+ - f_{j\nu\omega}^- z_{j\nu\omega t}^- \right] + \gamma_\nu (\kappa_{j\nu t} - \rho_{j\nu t} + \sum_{i \in \mathbb{I}} y_{ij\nu t}) \right] \quad (4.1)$$

subject to

$$\kappa_{j\nu t} = \kappa_{j\nu(t-1)} + \sum_{\omega \in \Omega} C_{\nu\omega} (z_{j\nu\omega t}^+ - z_{j\nu\omega t}^-) \quad \forall j \in \mathbb{J}, \nu \in \mathbb{V}, t \in \mathbb{T} \quad (4.2)$$

$$\begin{aligned} \rho_{j\nu t} &= \rho_{j\nu(t-1)} + \sum_{\omega \in \Omega} C_{\nu\omega} (z_{j\nu\omega t}^+ - z_{j\nu\omega t}^-) - \sum_{i \in \mathbb{I}} y_{ij\nu(t-1)} + \alpha_{j\nu(t-1)} - \psi_{j(t-1)}^{(\nu+1 \rightarrow \nu)} + \psi_{j(t-1)}^{(\nu \rightarrow \nu-1)} \\ &\quad \forall j \in \mathbb{J}, \nu \in \mathbb{V}, t \in \mathbb{T} \end{aligned} \quad (4.3)$$

$$\alpha_{j\nu t} = \mu_{\nu t} \left(\kappa_{j\nu t} - \rho_{j\nu t} + \sum_{i \in \mathbb{I}} y_{ij\nu t} \right) \quad \forall j \in \mathbb{J}, \nu \in \mathbb{V}, t \in \mathbb{T} \setminus \{\mathbb{T}\} \quad (4.4)$$

$$\psi_{jt}^{(\nu \rightarrow \nu-1)} = \tau_t^{(\nu \rightarrow \nu-1)} \left(\kappa_{j\nu t} - \rho_{j\nu t} + \sum_{i \in \mathbb{I}} y_{ij\nu t} - \alpha_{j\nu t} \right) \quad \forall j \in \mathbb{J}, \nu \in \mathbb{V} \setminus \{1\}, t \in \mathbb{T} \setminus \{\mathbb{T}\} \quad (4.5)$$

Awards from Alumni

*Most Likely to have Been a
General in a Former Life*

Awarded to

Paul Intrevado



UNIVERSITY OF
SAN FRANCISCO

Awards from Alumni

*Most Likely to Call You an Idiot in the
Form of An Impeccably Worded Email*

Awarded to

Paul Intrevado



UNIVERSITY OF
SAN FRANCISCO

Awards from Alumni

MSAN Superlative Award 2016

Most likely to date his teacher

Paul Intrevado

Uni. Of San Francisco



MSAN Class of 2016

Software Programming Experience

- C / C++ (compiler)
- MATLAB (scientific interpreted language)
- CPLEX (optimization solver)
- R
- Python
- SQL
- CSS / HTML (web)
- MS Excel / MS Access / VBA

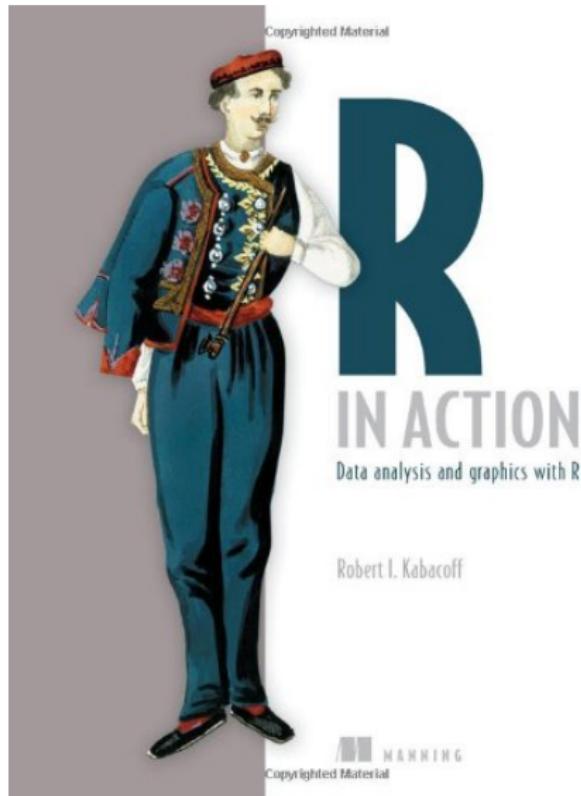
Subsection 2

About this Class

Syllabus

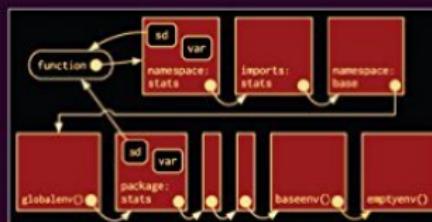
Subsection 3

Reference Textbooks



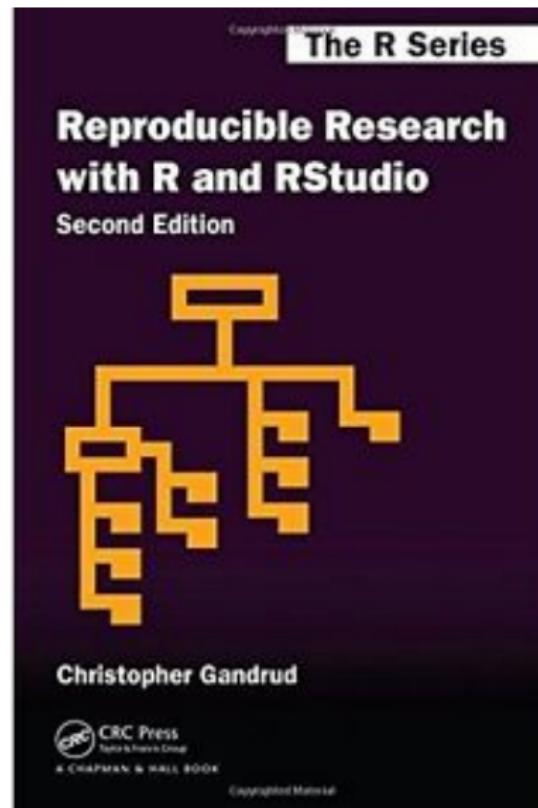
The R Series

Advanced R

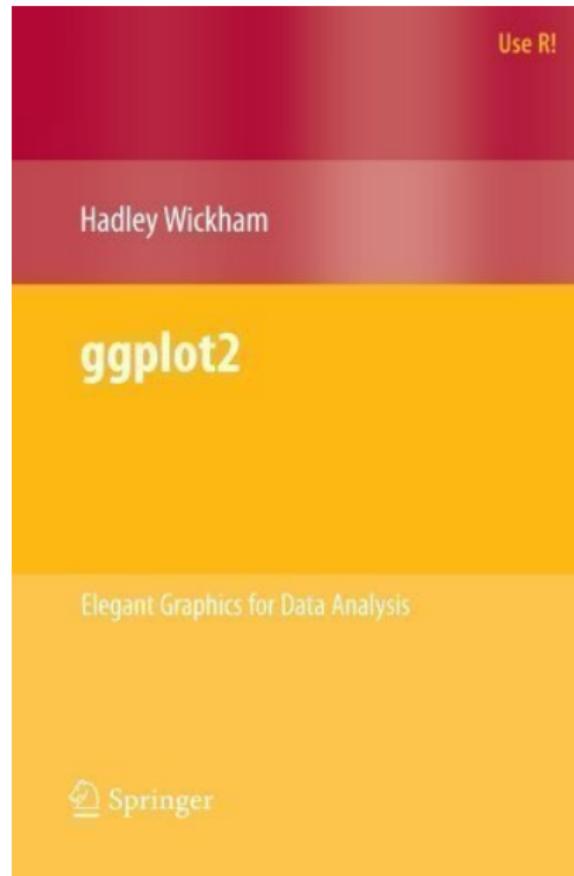


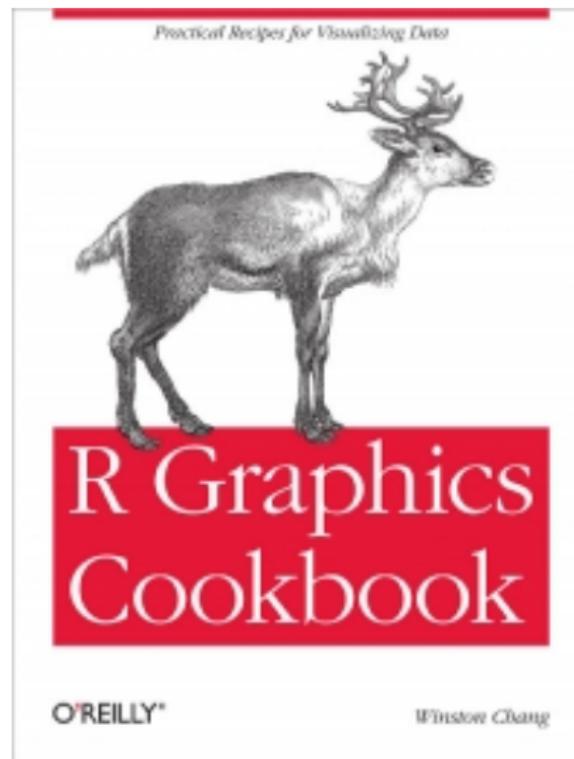
Hadley Wickham

 CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK



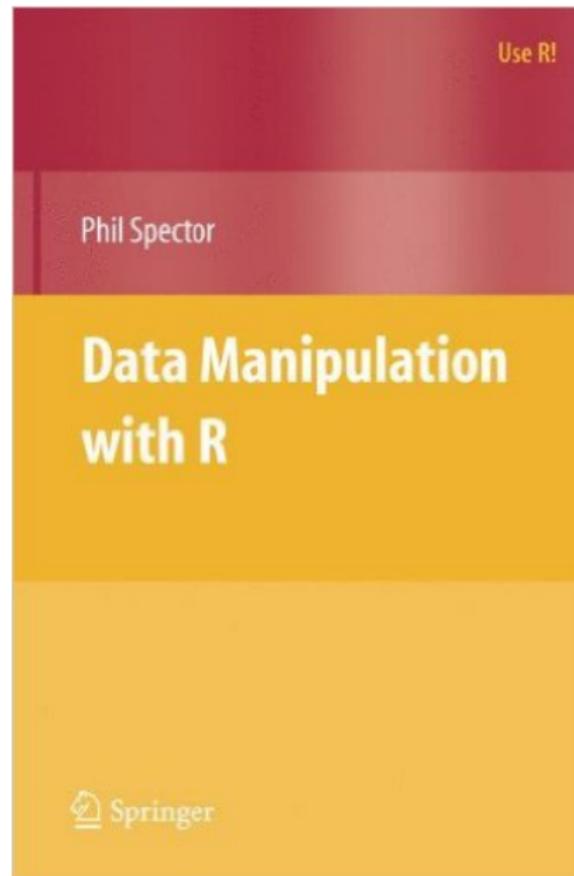






O'REILLY®

Winston Chang



Section 2

Introduction to R

Section Contents

2 Introduction to R

- Data & Data Structures in R
- RMarkdown
- Creating & Manipulating Data in Base R
- Control Flow in R
- Importing & Manipulating Data in R
- Subsetting
- Useful R Functions

What About Excel?



Excel is Great for Certain Things...

The screenshot shows a Microsoft Excel spreadsheet titled "grades". The table contains student data across 28 rows and 14 columns. The columns are labeled: A, B, C, D, E, F, G, H, I, J, K, L, M, and I. The data includes Midterm and Final scores, along with various assignment grades (Asn #1 through Asn #6) and final points. The "Letter Grade" column uses conditional formatting to color the text based on the value. Row 17 is highlighted with a blue selection bar, and the cell containing "Student 16" is highlighted with a green selection bar.

	A	B	C	D	E	F	G	H	I	J	K	L	M	I
1	Student	Midterm	Final	Asn #1	Asn #2	Asn #3	Asn #4	Asn #5	Asn #6	V1	V2	Final Points	Letter Grade	
2	Student 1	95.5	91.78	100	100	100	100	100	100	94.54	93.42	94.54	A	
3	Student 2	93.2	89.04	100	100	100	100	100	100	92.48	91.23	92.48	A	
4	Student 3	95.5	86.3	100	100	100	100	100	100	91.80	89.04	91.80	A	
5	Student 4	94.3	86.3	100	100	100	100	100	100	91.44	89.04	91.44	A	
6	Student 5	95.5	82.88	100	100	75	100	100	100	89.26	85.47	89.26	A	
7	Student 6	79.5	86.3	100	100	100	100	100	100	87.00	89.04	89.04	A	
8	Student 7	84.1	85.6	100	100	100	100	100	100	88.03	88.48	88.48	A	
9	Student 8	94.3	80.14	100	100	100	100	100	100	88.36	84.11	88.36	A	
10	Student 9	94.3	80	100	100	100	100	100	100	88.29	84.00	88.29	A	
11	Student 10	89.8	82.19	100	100	100	100	100	100	88.04	85.75	88.04	A	
12	Student 11	90.9	81.51	100	100	100	100	100	100	88.03	85.21	88.03	A	
13	Student 12	93.2	78.77	100	100	100	100	100	100	87.35	83.02	87.35	A	
14	Student 13	89.8	81.5	100	75	100	100	100	100	86.86	84.37	86.86	A	
15	Student 14	86.4	81.5	100	100	100	100	100	100	86.67	85.20	86.67	A	
16	Student 15	93.2	76.71	100	100	100	100	100	100	86.32	81.37	86.32	A	
17	Student 16	97.7	71.23	100	100	100	100	100	100	84.93	76.98	84.93	A-	
18	Student 17	86.4	76.71	100	100	100	100	100	100	84.28	81.37	84.28	A-	
19	Student 18	87.5	77.4	100	100	100	100	75	100	84.12	81.09	84.12	A	
20	Student 19	90.9	75.34	100	100	100	75	100	100	84.11	79.44	84.11	A-	
21	Student 20	81.8	78.77	100	100	100	100	100	100	83.93	83.02	83.93	A-	
22	Student 21	76.1	78.77	100	100	100	100	100	100	82.22	83.02	83.02	B+	
23	Student 22	84.1	71.92	100	100	100	100	100	100	81.19	77.54	81.19	B+	
24	Student 23	85.2	71.23	100	100	100	100	100	100	81.18	76.98	81.18	B+	
25	Student 24	67	76.03	100	100	100	100	100	100	78.12	80.82	80.82	B+	
26	Student 25	85.2	69.18	100	100	100	100	100	100	80.15	75.34	80.15	B+	
27	Student 26	81.8	70.55	100	100	100	100	100	100	79.82	76.44	79.82	B+	
28	Student 27	81.8	70.55	100	100	100	100	100	100	79.82	76.44	79.82	B+	

...but Not Everything

Sample Data

- Six columns of data with ~ 1.05 million rows
- Column 5: `startDate`
- Column 6: `endDate`
- **Objective:** test to see if `endDate < startDate`

...but Not Everything

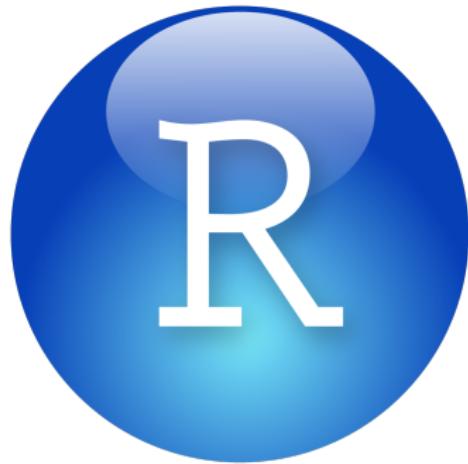
Sample Data

- Six columns of data with ~ 1.05 million rows
- Column 5: `startDate`
- Column 6: `endDate`
- **Objective:** test to see if `endDate < startDate`

RESULTS

- **Excel:** good luck...
- R: 33 min (poor coding technique)
- R: 58.5 sec (improved coding technique)

R or Python?



Vectorization in R

- Vectorized code saves time asking **type** questions
- There is an optimized engine—a basic linear algebra system (BLAS)—that is highly efficient at solving linear algebra problems
- A lot of R functions are written in C (or variants)
- MATLAB, Mathematica and the NumPy package for Python are also vectorized

<http://www.noamross.net/blog/2014/4/16/vectorization-in-r-why.html>

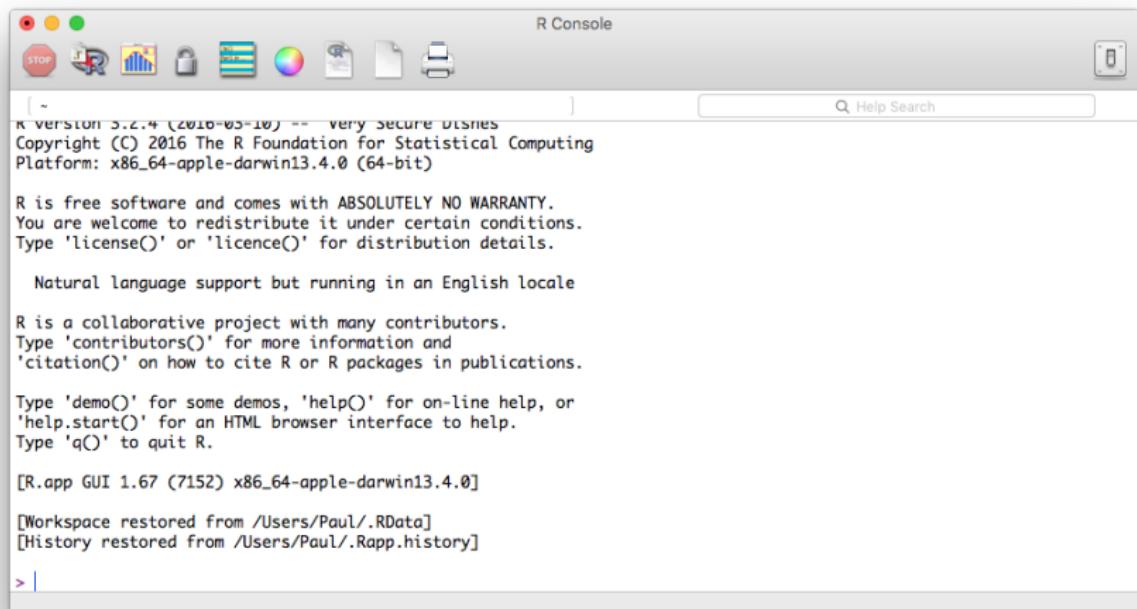
Why Use R?

- Open source (free)
- Runs on just about any platform
- Great visualization capabilities (ggplot2)
- Read/write from/to various data sources
- Scripting language (interpreted)
- Deep library of advanced data manipulation and statistical packages

Installing R

- RStudio is a nice, user-friendly integrated development environment (IDE), but can be quirky at times — still highly recommended and what I will use in class
- Required to install R regardless
- You can even run R from a terminal window if you wish
- If you are curious to see some basic R demos, type `demo()` to see the list of demonstration code available and then run one if you like, e.g., `demo(persp)`

This is what R Looks Like



This is what RStudio Looks Like

~/workbench - RStudio

userTrend.R* Q1Report.Rnw* userData*

Source on Save Run Source

```
1 # User Trend Analysis
2 # Breakdown of active and non-active users
3
4 library(plyr)
5 library(ggplot2)
6
7(userData <- read.csv("userDataTrends.csv"))
8(userData <- subset(userData, select = -c(id, group)))
9(userData$active <- as.factor(userData[,1]))
10
11 states <- levels(userData$state)
12
13 names(userData)
14 count(userData, "active == 1")
15 View(userData)
16
17 summary(subset(userData, active == 1)$state)
18 summary(subset(userData, active == 0)$state)
19
20 qplot(state, age, color = active, data = userData,
21       main = "Breakdown of Users by Age and State") +
22       opts(plot.title = theme_text(size = 19))
23 |
```

R Script

Console ~ /

```
active...1 freq
1 FALSE 310
2 TRUE 270
> View(userData)
> summary(subset(userData, active == 1)$state)
IA IL IN KS MI MN MO ND NE OH SD
19 26 21 21 27 49 22 26 19 16 24
> summary(subset(userData, active == 0)$state)
IA IL IN KS MI MN MO ND NE OH SD
26 27 18 31 27 49 22 32 19 33 26
> qplot(state, age, color = active, data = userData,
+       main = "Breakdown of Users by Age and State") +
+       opts(plot.title = theme_text(size = 19))
>
```

Workspace History

Load Save Import Dataset Clear All

Data

userData 580 obs. of 5 variables

Values

active integer[270]

states character[11]

Functions

split(group, location, ...)

Files Plots Packages Help

Zoom Export Clear All

Breakdown of Users by Age and State

active

0 1

RStudio

RStudio has Four Panels

- Console
- Scripting/Viewing
- Files/Packages/Help/Viewer
- Environment/History/Plots

Open up RStudio for a guided tour <class01intro.R>

Notes on R

- R is case-sensitive
- I require you to use the assignment operator ‘`<-`’ instead of the equality operator ‘`=`’ for all submitted code, even though both work, e.g.,

Syntax	Comments
<code>x <- 5</code>	standard syntax, required
<code>x = 5</code>	poor syntax, not permitted
<code>5 -> x</code>	awkward syntax, not permitted (but it works)

- Keyboard shortcut for assignment operator
 - Option (alt) + -

Basic R Help Functions

Function	Action
?foo	Help on the function <code>foo</code>
??foo	Search the help system for instances of the function <code>foo</code>
<code>RSiteSearch("foo")</code>	Search for the string <code>foo</code> in online help manuals and archived mailing lists
<code>data()</code>	List all available example datasets contained in currently loaded packages
<code>getwd()</code>	List the current working directory
<code>setwd("~/Desktop")</code>	Change working directory to <code>Desktop</code>
<code>rm()</code>	Remove (delete) one or more objects
<code>ls()</code>	List the objects in the current directory

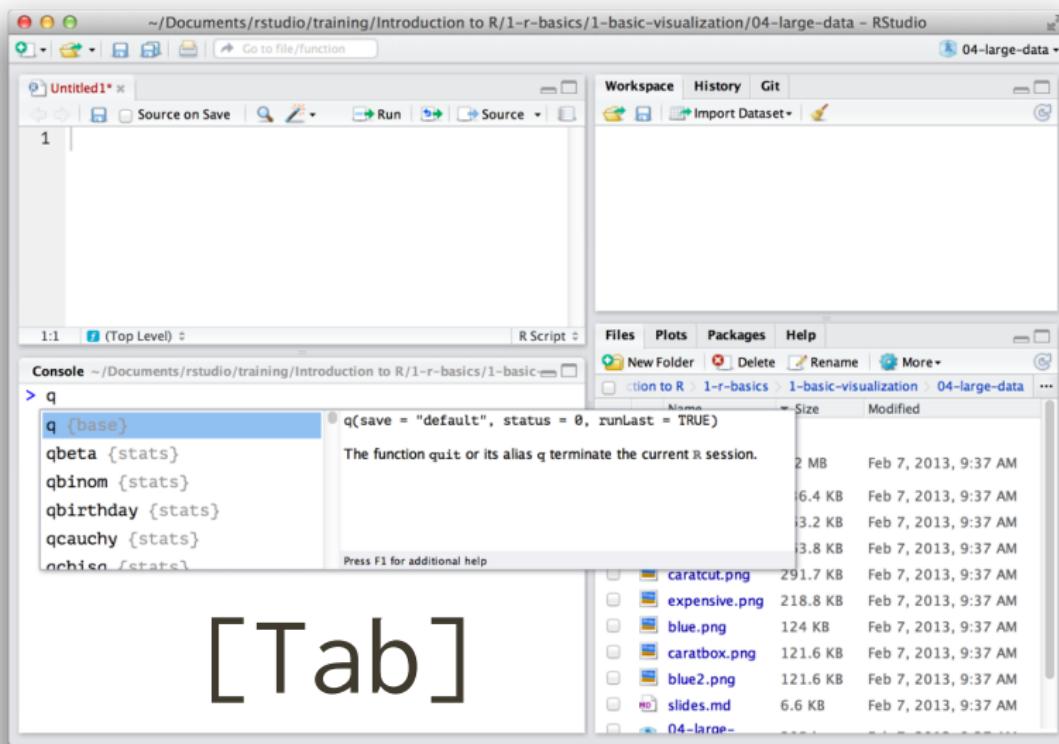
The Best Place for Answers to R Questions?



Basic R Workspace Functions

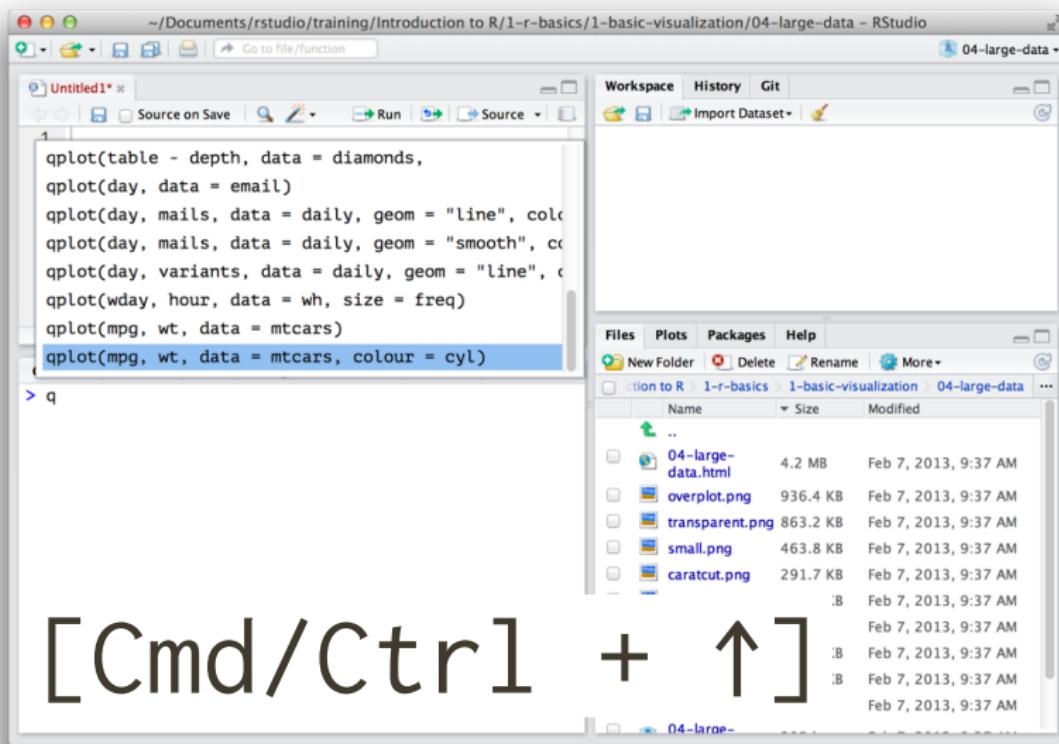
Function	Action
<code>getwd()</code>	List the current working directory.
<code>setwd("mydirectory")</code>	Change the current working directory to <i>mydirectory</i> .
<code>ls()</code>	List the objects in the current workspace.
<code>rm(<i>objectlist</i>)</code>	Remove (delete) one or more objects.
<code>help(options)</code>	Learn about available options.
<code>options()</code>	View or set current options.
<code>history(#)</code>	Display your last # commands (default = 25).
<code>savehistory("myfile")</code>	Save the commands history to <i>myfile</i> (default = .Rhistory).
<code>loadhistory("myfile")</code>	Reload a command's history (default = .Rhistory).
<code>save.image("myfile")</code>	Save the workspace to <i>myfile</i> (default = .RData).
<code>save(<i>objectlist</i>, file="myfile")</code>	Save specific objects to a file.
<code>load("myfile")</code>	Load a workspace into the current session (default = .RData).

Useful R Keyboard Shortcuts: Autocomplete

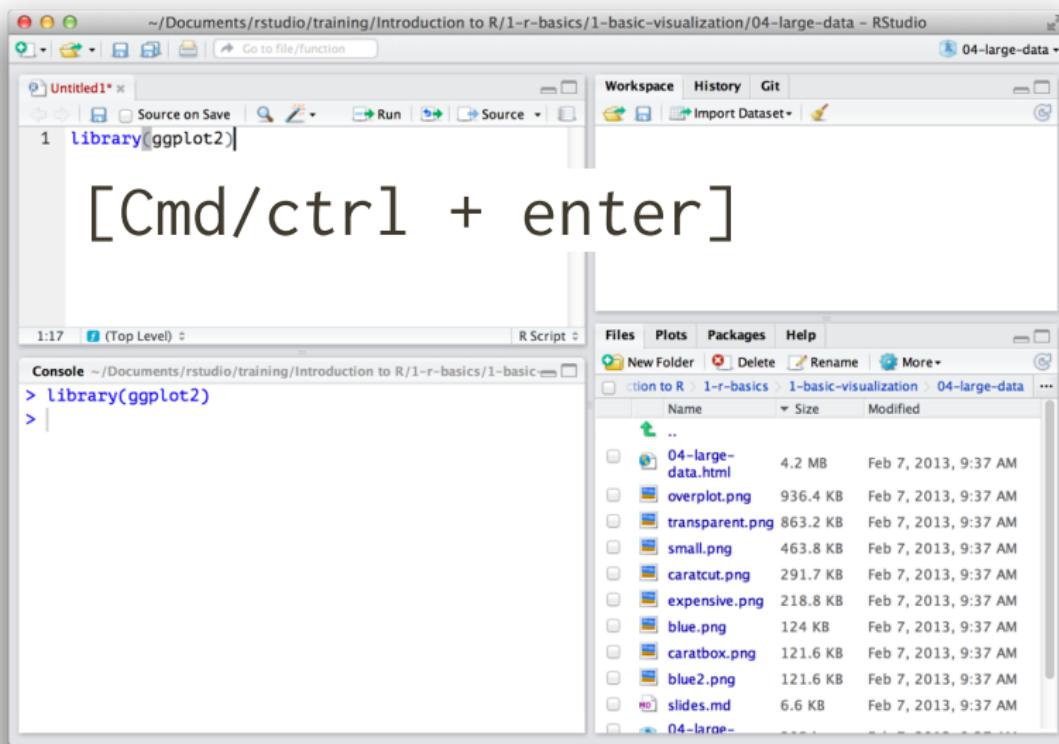


[Tab]

Useful R Keyboard Shortcuts: History [Console Only]



Useful R Keyboard Shortcuts: Execute Code [Console Only]

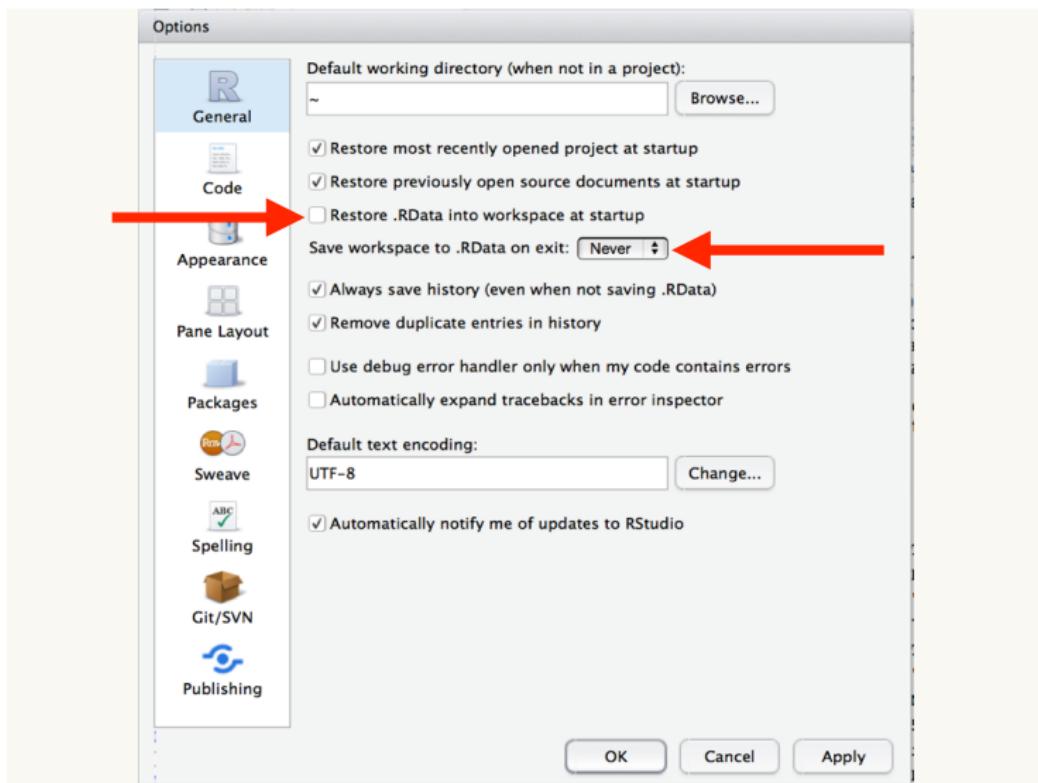


Useful R Keyboard Shortcuts: Restarting an R Session

The screenshot shows the RStudio interface. In the top-left corner, there's a window titled "Untitled1" containing the R code: `library(ggplot2)`. Below this, the "Console" tab shows the command being run: `> library(ggplot2)`. A message "Restarting R session..." appears in the console. To the right of the console is the "Files" panel, which lists several files in the current directory: `04-large-data.html`, `overplot.png`, `transparent.png`, `small.png`, `caratcut.png`, `expensive.png`, `blue.png`, `catarbox.png`, `blue2.png`, `slides.md`, and `04-large-`. The file `04-large-data.html` is highlighted. The status bar at the bottom indicates the time is 1:17.

[Cmd/ctrl + shift + F10]

IMPORTANT R Setting



A Brief Digression

- Whenever writing code, you want to be sure to clear your environment to ensure the fidelity of your results
- In each and every R script file I write, I always include the following two lines of code

```
rm(list=ls())
cat("\014")
```

- 1 `rm(list=ls())` removes all objects in the current environment
- 2 On a Mac, `cat("\014")` clear the console windows (same as `ctrl + l`)

Packages in R

- Vanilla R comes with extensive capabilities
- ...BUT...
- some of the most exciting features in R are available as optional modules called Packages that you can download and install
- Packages are—like R—free, user-contributed modules that you can download and install
- There are ~ 7,000 R packages [June 2016]

Packages in R

- Vanilla R comes with extensive capabilities
- ...BUT...
- some of the most exciting features in R are available as optional modules called Packages that you can download and install
- Packages are—like R—free, user-contributed modules that you can download and install
- There are ~ 7,000 R packages [June 2016]
- Given R packages are user-contributed, some contain errors (some of which we have found right here in MSAN), so feel free to use R for analysis, but maybe double-check your output before you buy or sell billions of dollars of stock based on R package calculations

What are R Packages?

- Packages are collections of functions, data and compiled code in a well-defined format
- The `library()` function shows you which packages are saved in your library (i.e., which packages have been downloaded)
n.b. `library()` **does not** tell you which packages are loaded, it only tells you which packages are downloaded
- `search()` tells you which packages are loaded and ready to use
- `install.packages("myPackageName")` is used to install packages, and `library("myPackageName")` loads the package into your current working session
- You only ever need to download a package once, but you always need to load packages when restarting an R session

Loading, Using & Maintaining R Packages [EXAMPLE]

```
> install.packages("microbenchmark")
    % Total  % Received % Xferd  Average Speed   Time   Time   Time   ...
                  Dload  Upload Total Spent Left  ...
0      0      0      0      0      0      0      0 --::-- --::-- --::-- ...
```

The downloaded binary packages are in
/var/folders/jm/3w7pqfms0nvg_ypvnvkkk83h0000gn/...

```
> microbenchmark(3^3)
Error: could not find function "microbenchmark"
```

```
# using the double colon operator allows you to access functions from
#   packages that are not loaded
```

```
> microbenchmark:::microbenchmark(3^3)
Unit: nanoseconds
expr min  lq   mean median     uq   max neval
 3^3 203 209 365.53    271 372.5 6112   100
```

```
# don't be confused: the above statement calls the microbenchmark function
#   (second) from the microbenchmark package (first)
```

Loading, Using & Maintaining R Packages [EXAMPLE] [CONT'D]

```
# this loads the package
> library(microbenchmark)

> microbenchmark(3^3)
Unit: nanoseconds
expr  min   lq   mean median   uq   max neval
 3^3 156 159 311.88    226 312 5372   100
```

- Packages are often updated by their authors, so be sure to keep your packages up to date
 - `update.packages()` will update all packages tab
 - `installed.packages()` will list all packages you have downloaded, along with their version numbers, dependencies and other information

Loading, Using & Maintaining R Packages [EXAMPLE] [CONT'D]

A far less cumbersome way to install, load and update packages in RStudio is using the appropriate icons in the “Packages” window

The screenshot shows the RStudio interface with the "Packages" tab selected in the top navigation bar. Below the navigation bar is a toolbar with "Install" and "Update" buttons, a search bar, and a refresh icon. The main area is a table listing packages:

	Name	Description	Version	Uninstall
<input type="checkbox"/>	geepack	Generalized Estimating Equation Package	1.2-0.1	
<input checked="" type="checkbox"/>	ggplot2	An Implementation of the Grammar of Graphics	2.1.0	
<input type="checkbox"/>	ggvis	Interactive Grammar of Graphics	0.4.2	
<input type="checkbox"/>	git2r	Provides Access to Git Repositories	0.15.0	
<input type="checkbox"/>	googleVis	R Interface to Google Charts	0.5.10	
<input checked="" type="checkbox"/>	graphics	The R Graphics Package	3.2.4	
<input checked="" type="checkbox"/>	grDevices	The R Graphics Devices and Support for Colours and Fonts	3.2.4	
<input type="checkbox"/>	grid	The Grid Graphics Package	3.2.4	

Final Notes on R Packages

Packages sometimes (often?) have dependencies on other packages, e.g.,

```
#load the package MatchIt  
> library("MatchIt", lib.loc="/Library/Frameworks/R.framework/...")  
Loading required package: MASS
```

- Without asking, when loading the MatchIt package, the MASS package is automatically loaded
 - This is helpful in one respect, since MatchIt leverages certain functionality from MASS; if MASS wasn't automatically loaded, then calling certain functions from MatchIt might throw an error
- n.b.** Be careful of function masking: because there are so many R packages available from so many different authors, it often happens that different packages have identically named functions

Final Notes on R Packages [CONT'D]

When calling a function, R searches the Global Environment first, then iterates through all packages for the function, beginning from the most recently added

```
> search()
[1] ".GlobalEnv"           "package:reshape2"   "package:plyr"
[4] "package:MatchIt"      "package:MASS"       "package:ggplot2"
[7] "tools:rstudio"        "package:stats"    "package:graphics"
[10] "package:grDevices"     "package:utils"     "package:datasets"
[13] "package:methods"       "Autoloads"       "package:base"

> (.packages())
[1] "reshape2"   "plyr"       "MatchIt"     "MASS"       "ggplot2"    "stats"
[7] "graphics"   "grDevices"  "utils"       "datasets"   "methods"    "base"
```

Final Notes on R Packages [CONT'D]

When calling a function, R searches the Global Environment first, then iterates through all packages for the function, beginning from the most recently added

```
> search()
[1] ".GlobalEnv"           "package:reshape2"   "package:plyr"
[4] "package:MatchIt"      "package:MASS"       "package:ggplot2"
[7] "tools:rstudio"        "package:stats"    "package:graphics"
[10] "package:grDevices"     "package:utils"     "package:datasets"
[13] "package:methods"       "Autoloads"       "package:base"

> (.packages())
[1] "reshape2"   "plyr"       "MatchIt"     "MASS"       "ggplot2"    "stats"
[7] "graphics"   "grDevices"  "utils"       "datasets"   "methods"    "base"
```

- If you are using a lot of packages and want to be certain you are calling a function from a specific package, use the double colon operator, e.g., `plyr::rename()`

Final Notes on R Packages [CONT'D]

- If you feel you have too many packages loaded and want to unload (detach) one, you can use the following command:
`detach("package:MatchIt", unload=TRUE)`
n.b. Be aware that when unloading MatchIt you do not automatically unload the dependency that was loaded when you loaded MatchIt, namely, MASS
- You may choose to use `require("myPackageName")` in lieu of `library("myPackageName")` to load a package, but be cautious when doing so
 - `require()` attempts to load a package and returns a logical to indicate whether or not the could not be loaded; if the package could not be loaded, a **warning** is thrown
 - `library()` attempts to load a package and throws an **error** if the package could not be loaded

How Big is Big Data in R?

- R holds data in memory, effectively limiting data to the amount of RAM a computer has access to
- It is not uncommon to work with a data set containing 100,000,000 elements (e.g., 100,000 observations of 1,000 variables or 1,000,000 observations of 100 variables) without difficulty
- Of course all of the above approximations depend on what type of data is contained in each variable, e.g., I am currently working on a data set with 2.2 million records and twenty variables, which takes approximately one minute to load into memory
- The other issue to consider is what techniques and/or functions will be applied to the data

How Big is Big Data in R? [CONT'D]

- The more complex and memory intensive the task, the smaller the data will be required to be
- Basic plotting would likely require far less computational exertion than a complex statistical learning model

Data Sets in R?

- R comes built in with multiple data sets you can play with
- Many (most?) packages also have data sets
- `data()` will bring up a list of all data sets available across all loaded packages
- `help(<nameOfDataSet>)` will provide you a detailed description of the data set in question

Subsection 1

Data & Data Structures in R

Data Structures & Dimensionality

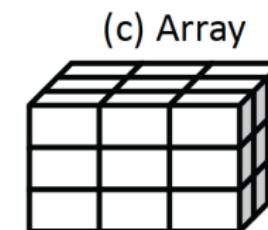
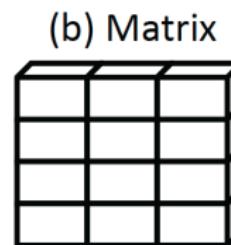
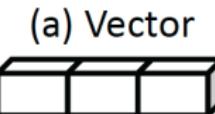
Dimension	Homogeneous	Heterogeneous
1	Atomic Vector	List
2	Matrix	Data Frame
n	Array	

Homogeneous All contents must be of the same type

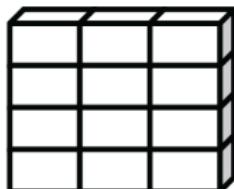
Heterogeneous Contents can be of different types

n.b. There are no 0-dimensional (scalar) types in R, only vectors of length one

Data Structures



(d) Data frame



(e) List

Columns can be different modes

Vectors
Arrays
Data frames
Lists

Vectors

- The basic data structure in R is the vector
- There are two types of vectors: atomic vectors and lists

Properties of Vectors

- Type (`typeof()`)
- Length (`length()`)
- Attributes (`attributes()`)

n.b. Use `is.atomic()` or `is.list()` to determine if an object is a vector, **not** `is.vector()`

Atomic Vectors

Four Common Types of Vectors

- Logical
- Integer
- Double (numeric)
- Character

```
> doubleAtomicVector <- c(1, 3.14, 99.999)

# use L prefix to get integers instead of doubles
> integerAtomicVector <- c(1L, 3L, 19L)

> logicalAtomicVector <- c(TRUE, FALSE, T, F)

> characterAtomicVector <- c("this", "is a", "string")
```

TRY THIS

- 1 Create the vector `myFavNum` of you favorite fractional number
- 2 Create the vector `myNums` of your seven favorite numbers
- 3 Create the vector `firstNames` of the first names of two people next to you
- 4 Create the vector `myVec` of the last name and age of someone you know

ANSWER THESE

- 1 Guess and then check what types your vectors are.
- 2 Check the length of each vector.
- 3 Did you write the code in the console window or the editor?
- 4 How do you execute a line of code in the editor?
- 5 How do you execute multiple lines of code simultaneously in the editor?
- 6 Did you leverage the TAB button for auto-completion?

Accessing Elements of a Vector

- To access the individual elements of a vector

```
> (myAtomicVector <- c(1, 2, 3, 4, -99, 5, NA, 4, 22.223))  
[1] 1.000 2.000 3.000 4.000 -99.000 5.000     NA  
[8] 4.000 22.223
```

```
> myAtomicVector[5]  
[1] -99
```

```
> myAtomicVector[c(1, 2, 5, 9)]  
[1] 1.000 2.000 -99.000 22.223
```

```
> myAtomicVector[10]  
[1] NA
```

```
> myAtomicVector[3:8]  
[1] 3 4 -99 5 NA 4
```

TRY THIS

- 1 Add `myFavNum` to the seventh entry of `myNums` and store the result in a variable named `myFirstAddition`
- 2 Add `myFavNum` to each of the seven entries of `myNums` and store the result in a variable named `mySecondAddition`
- 3 Add `myFavNum` to **all** of the values in `myNums` and store the result in a variable named `myFirstSum`
- 4 Add `myFavNum` to the smallest number in `myNums` and store the result in a variable named `thisIsGettingMoreComplex`
- 5 Add the second entry of `myNums` to the age of the person you select for `myVec` and store the result in a variable named `whatTypeOfVectorIsThis`
 - Does what we did make sense? Did it work? Why?

SOLUTION

```
# preamble
myFavNum <- 3.1415
myNums <- c(1, 3, 55, 33, 86, -sqrt(2), -110)
# also works myNums <- 1:7
firstNames <- c("Jeff", "Terence", "David")
myVec <- c("Parr", 99)

1 myFirstAddition <- myFavNum + myNums[7]
2 mySecondAddition <- myFavNum + myNums
3 myFirstSum <- myFavNum + sum(myNums)
4 thisIsGettingMoreComplex <- myFavNum + min(myNums)
5 whatTypeOfVectorIsThis <- sum(c(myNums[2], myVec[2]))
Error in sum(c(myNums[2], myVec[2])) :
  invalid 'type' (character) of argument
```

Missing Values

Missing values are specified with `NA`, which is a logical vector of length one.

- `NA` will always be *coerced* to the correct type if used inside `c()`
- You can create `NAs` of a specific type with
 - `NA_real_` (double)
 - `NA_integer_`
 - `NA_character_`

```
> c(1, 2, 3, NA)  
[1] 1 2 3 NA
```

```
> x[1]  
[1] 1
```

```
> x <- c(1, 2, 3, NA)  
  
> typeof(x)  
[1] "double"
```

```
> x[4]  
[1] NA  
  
> typeof(x[4])  
[1] "double"
```

na.rm = TRUE

- Certain functions will fail when applied to vectors with one or more NAs

```
> (myAtomicVector_01 <- c(99.1, 98.2, 97.3, 96.4, NA))  
[1] 99.1 98.2 97.3 96.4    NA
```

```
> sum(myAtomicVector_01)  
[1] NA
```

```
> mean(myAtomicVector_01)  
[1] NA
```

```
> sum(myAtomicVector_01, na.rm = TRUE)  
[1] 391
```

```
> mean(myAtomicVector_01, na.rm = TRUE)  
[1] 97.75
```

Types & Tests

To check the type of a vector, use `typeof()`, or more specifically

- `is.character()`
- `is.double()`
- `is.integer()`
- `is.logical()`
- `is.na()`

Coercion

Coercion is a great feature in R which can make coding easy, but may also have unintended consequences.

- All elements in an atomic vector must be the same type
- If you attempt to combine different types in an atomic vector they will be coerced to the most flexible type
- Most to least flexible types: character, double, integer, logical
- When a logical vector is coerced to numeric (double or integer), TRUE = 1 and FALSE = 0

```
> x <- c("abc", 123)
```

```
> typeof(x)
[1] "character"
```

You can explicitly coerce using `as.character()`, `as.double()`, `as.integer()`, and `as.logical()`

A Brief Digression: `str()`

- A quick way to figure out what data structure an object is composed of is to use `str()`, which is short for structure
- `str()` provides a concise description for any R data structure

Conditionally Subsetting Atomic Vectors

- The syntax is awkward and takes some time to get used to
- Once you understand the sequence of events in conditional subsetting, it will feel more natural

```
> (myAtomicVector_01 <- c(99.1, 98.2, 97.3, 96.4))  
[1] 99.1 98.2 97.3 96.4
```

```
> myAtomicVector_01[myAtomicVector_01 > 98]  
[1] 99.1 98.2
```

- What is actually happening

- 1 The `myAtomicVector_01 > 98` part of the statement tests each element of the vector to see whether it is > 98 and returns a **LOGICAL** value for each test which, in this case, returns the logical vector (`T T F F`)
- 2 The vector (`T T F F`) is passed to `myAtomicVector_01`, which returns the first two elements and omits the final two
 - An equivalent statement would be
`myAtomicVector_01[c(T, T, F, F)]`

Logical Operators

Operator	Description
<code><</code>	Less than
<code><=</code>	Less than or equal to
<code>></code>	Greater than
<code>>=</code>	Greater than or equal to
<code>==</code>	Exactly equal to
<code>!=</code>	Not equal to
<code>! x</code>	Not x
<code>x y</code>	x or y
<code>x & y</code>	x and y
<code>isTRUE(x)</code>	Test if x is TRUE

TRY THIS

```
> myAtomicVector <- c(1, 4, 3, 2, NA, 3.22, -44, 2, NA, 0, 22, 34)
```

- 1 How many positive numbers (> 0) are there in this vector?
- 2 How many negative numbers (< 0) are there in this vector?
- 3 How many 0's are there in this vector?
- 4 How many NAs are there in this vector?
- 5 How many numbers in the vector are non-zero **and** not NAs?
- 6 What is the sum of the positive numbers in this vector?
- 7 What is the sum of the negative numbers in this vector?

SOLUTION

```
1 sum(myAtomicVector > 0, na.rm = T)
2 sum(myAtomicVector < 0, na.rm = T)
3 sum(myAtomicVector == 0, na.rm = T)
4 sum(is.na(myAtomicVector))
5 sum(myAtomicVector != 0 & !is.na(myAtomicVector), na.rm = T)
# there is a simpler solution to this question
6 sum(myAtomicVector[myAtomicVector > 0], na.rm = T)
7 sum(myAtomicVector[myAtomicVector < 0], na.rm = T)
```

Lists

- Lists are different from atomic vectors as elements of a list can be of any type, including lists
- A list is constructed using `list()` instead of `c()`

```
> myList <- list(10:12, "abc", c(3.1415, 9), c(T, F, F, F))

> str(myList)
List of 4
 $ : int [1:3] 10 11 12
 $ : chr "abc"
 $ : num [1:2] 3.14 9
 $ : logi [1:4] TRUE FALSE FALSE FALSE
```

- Lists are recursive, i.e., a list can contain lists, making them fundamentally different from atomic vectors
- `is.list()` (test if list), `as.list()` (coerce to list), `unlist()` (convert to atomic vector + coercion)

JSON Data

- JSON stands for JavaScript Object Notation
- JSON data is a list and is
 - light-weight
 - language-independent
 - easy to read and write
 - text-based, human readable data exchange format
- Using `jsonlite` package

```
> mtcars[1:2, 1:2]
      mpg cyl
Mazda RX4     21    6
Mazda RX4 Wag 21    6

> (json_01 <- toJSON(mtcars[1:2, 1:2]))
[
  {"mpg":21, "cyl":6, "_row":"Mazda RX4"},  

  {"mpg":21, "cyl":6, "_row":"Mazda RX4 Wag"}]
```

Names

- A name is a vector **attribute**
- Names need not be unique
- Not all elements of a vector are required to have a name

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

```
> names(x)
```

```
NULL
```

```
> x <- c(1, 2, 3); names(x) <- c("a", "b", "c")
```

```
> names(x)
```

```
[1] "a" "b" "c"
```

```
> x <- c(a = 1, b = 2, c = 3)
```

```
> names(x)
```

```
[1] "a" "b" "c"
```

```
> x <- c(a = 1, b = 2, 3)
```

```
> names(x)
```

```
[1] "a" "b" ""
```

Factors

- An important use of attributes is to define factors
- A factor is a vector of elements from a discrete set, and is used to store categorical (ordinal or nominal) data
- Factors are built on top of **integer vectors** using two attributes
 - 1 The `class()` ‘factor’, which makes them behave differently from regular integer vectors
 - 2 The `levels()`, which defines the discrete set of permissible values

```
> x <- factor(c("M", "F", "F", "M"))
> levels(x)
[1] "F" "M"

> x
[1] M F F M
Levels: F M

> class(x)
[1] "factor"

> typeof(x)
[1] "integer"

> x[2] <- "c"
Warning message:
...
invalid factor level, NA generated

> x
[1] M      <NA> F      M
Levels: F M
```

Nominal Factors

- Although we (intelligent humans) have an inherent ability to understand the ordering of the ordinal categories below, R does not, and unless told, will treat them as nominal categorical variables
- Nominal (unordered) factors are sorted automatically by R, e.g., alphabetically, numerically, etc.

n.b. The terms *ordered* and *sorted* are **not** synonymous here

Nominal Factors [EXAMPLE]

```
> bodyType <- factor(c("healthy", "healthy", "healthy", "obese", "overweight",
+                         "overweight", "skinny"))
> bodyType
[1] healthy    healthy    healthy    obese      overweight overweight skinny
Levels: healthy obese overweight skinny

> levels(bodyType)
[1] "healthy"   "obese"     "overweight" "skinny"

> str(bodyType)
Factor w/ 4 levels "healthy","obese",...: 1 1 1 2 3 3 4

> bodyType <- "obese"
[1] NA NA NA NA NA NA NA
Warning message:
In Ops.factor(bodyType, "obese") : < not meaningful for factors
```

Nominal Factors [CONT'D]

- Even though nominal factors are ordered due to the underlying integer mapping, logical comparisons based on levels fail

```
> bodyType[bodyType < "obese"]
[1] <NA> <NA> <NA> <NA>
Levels: healthy obese overweight skinny
Warning message:
In Ops.factor(bodyWeight, "obese") : < not meaningful for factors
```

- Nominal factors *can* be filtered if we access the underlying integer mapping, but weird results may arise

```
> levels(bodyType)
[1] "healthy"      "obese"        "overweight"    "skinny"

> str(bodyType)
Factor w/ 4 levels "healthy","obese",...: 4 1 3 2

> bodyType[as.integer(bodyType) < 3] # 3 is mapped to "overweight"
[1] healthy obese
Levels: healthy obese overweight skinny
```

Ordinal Factors

- We can create ordinal factors by including the option `ordered = TRUE`
- By creating an ordinal set of factors, we are telling R to explicitly use the ordering we are providing
- Let's examine a messier version of `bodyType`, where instead of the body type being explicit (e.g., "obese"), the body types are coded for brevity

```
(bodyType <- factor(c("h", "h", "h", "ob", "ov", "ov", "s"),
                     levels = c("s", "h", "ov", "ob"),
                     labels = c("Skinny", "Healthy", "Overweight", "Obese"),
                     ordered = TRUE))
```

Ordinal Factors [CONT'D]

Let's examine exactly what is being executed

```
(bodyType <- factor(c("h", "h", "h", "ob", "ov", "ov", "s"),
                     levels = c("s", "h", "ov", "ob"),
                     labels = c("Skinny", "Healthy", "Overweight", "Obese"),
                     ordered = TRUE))
```

- 1 `c("h", "h", "h", "ob", "ov", "ov", "s")` is what we want to classify as a factor
- 2 `levels = c("s", "h", "ov", "ob")` provides the levels
 - Omitting this enables R to order the levels itself
- 3 `labels = c("Skinny", "Healthy", "Overweight", "Obese")` are the *nice* labels we want to see instead of the more obscurely-coded factors
 - n.b.** labels are mapped directly to levels
- 4 `ordered = TRUE` instructs R to order the factors according to levels

Ordinal Factors [EXAMPLE]

```
(bodyType <- factor(c("h", "h", "h", "ob", "ov", "ov", "s"),
                     levels = c("s", "h", "ov", "ob"),
                     labels = c("Skinny", "Healthy", "Overweight", "Obese"),
                     ordered = TRUE))
> levels(bodyType)
[1] "Skinny"      "Healthy"      "Overweight"   "Obese"

> str(bodyType)
Ord.factor w/ 4 levels "Skinny"<"Healthy"<...: 2 2 2 4 3 3 1

> bodyType < "Obese"
[1] TRUE  TRUE  TRUE FALSE TRUE  TRUE  TRUE

> bodyType[bodyType < "Obese"]
[1] Healthy  Healthy  Healthy  Overweight Overweight Skinny
Levels: Skinny < Healthy < Overweight < Obese
```

YOU TRY IT

Use the following code to create the variable `myCyl`

```
myCyl <- mtcars$cyl
```

- 1 Create an ordered factor from `myCyl`, mapping the levels to 'Small', 'Medium' and 'Large'
- 2 How many observations have cylinders \leq 'Medium'?

Matrices and Arrays

- By giving an atomic vector a dimension attribute, it can behave like a multi-dimensional array
- A special case of the array is a matrix, a two-dimensional array
- Matrices and arrays are created with `matrix()` and `array()`

```
> x <- matrix(1:10, ncol = 5, nrow = 2)          > y <- array(1:12, c(2, 3, 2))
# can drop ncol and nrow to shorten             > y
                                                , , 1
> x
 [,1] [,2] [,3] [,4] [,5]                      [,1] [,2] [,3]
[1,]    1    3    5    7    9                      [1,]    1    3    5
[2,]    2    4    6    8   10                      [2,]    2    4    6
, , 2
                                                [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12
```

Selected Functional Generalizations

1D

- 1 `length()`
- 2 `names()`
- 3 `c()`

nD

- 1 `nrow()`, `ncol()`, `dim()`
- 2 `rownames()`, `colnames()`,
`dimnames()`
- 3 `cbind()`, `rbind()`,
`abind()`

n.b. a matrix or array can also be one-dimensional, e.g., an object that is defined as a matrix is permitted to only have one column or one row; although they may look and behave alike, a vector and a one-dimensional matrix behave differently and may generate strange output when using certain functions, e.g., `tapply()`

Data Frames

This is why we use



Data Frames

- Most common way of storing data in R
- A data frame is a list with equal-length vectors
- Each vector must be of the same data type

Sample data frame:

Summary of Data

A data frame with 60 observations on 3 variables.

- [, 1] len numeric Tooth length
- [, 2] supp factor Supplement type (VC or OJ)
- [, 3] dose numeric Dose in milligrams/day

```
> str(ToothGrowth)
'data.frame': 60 obs. of 3 variables:
 $ len : num 4.2 11.5 7.3 5.8 6.4 10 11.2 11.2 5.2 7 ...
 $ supp: Factor w/ 2 levels "OJ","VC": 2 2 2 2 2 2 2 2 2 ...
 $ dose: num 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 ...
```



```
> ?ToothGrowth
```

Creating and Manipulating Data Frames

Create a data frame using `data.frame()`

```
# this is sloppy coding etiquette and is only for exposition

> (xyz <- data.frame(1:3, c("a", "b", "c")))
X1.3 c..a....b....c..
 1      1             a
 2      2             b
 3      3             c

> str(xyz)
'data.frame': 3 obs. of  2 variables:
 $ X1.3          : int  1 2 3
 $ c..a....b....c.: Factor w/ 3 levels "a","b","c": 1 2 3
```

- Surround code with `()` to automatically print the result to the console

Creating and Manipulating Data Frames [CONT'D]

Create a data frame using `data.frame()`

```
> (xyz <- data.frame(numberColumn = 1:3, letterColumn = c("a", "b", "c")))
  numberColumn letterColumn
1              1             a
2              2             b
3              3             c

> str(xyz)
'data.frame': 3 obs. of  2 variables:
 $ numberColumn: int  1 2 3
 $ letterColumn: Factor w/ 3 levels "a","b","c": 1 2 3
```

- After creating the data frame, the first column of untitled numbers are row numbers
- Observe that even though the entries in `letterColumn` are characters that an `str(letterColumn)` shows the column to be a Factor

Creating and Manipulating Data Frames [CONT'D]

- If you want to suppress R's default behavior of turning strings into factors, use the options `stringsAsFactors = FALSE`

```
> (xyz <- data.frame(numberColumn = 1:3, letterColumn = c("a", "b", "c"),
  stringsAsFactors = F))

  numberColumn letterColumn
1              1             a
2              2             b
3              3             c

> str(xyz)

'data.frame': 3 obs. of  2 variables:
 $ numberColumn: int  1 2 3
 $ letterColumn: chr  "a" "b" "c"
```

Creating and Manipulating Data Frames [CONT'D]

- Recall that a data frame is a list, which means that `typeof(myDataFrame)` will output a list
- Instead use `class()` or `is.data.frame()`
- An object can be coerced to a data frame using `as.data.frame()`

Combine/Append Data Frames

- When a data frame already exists, you can combine/append another data frame or a vector to the original data frame, but there are some caveats
 - 1 Use `cbind()` to column-bind two data frames
 - n.b. the number of columns in each data frame must be equal, and row names are ignored

```
> (myDataFrame_01 <- data.frame(x = 1:3, y = c("A", "B", "c")))  
  x y  
1 1 A  
2 2 B  
3 3 c  
  
> (myDataFrame_02 <- cbind(myDataFrame_01, data.frame(z = -1:-3)))  
  x y  z  
1 1 A -1  
2 2 B -2  
3 3 c -3
```

Combine/Append Data Frames [CONT'D]

- When a data frame already exists, you can combine/append another data frame or a vector to the original data frame, but there are some caveats
 - 2 Use `rbind()` to row-bind two data frames
 - n.b. the **number** and the **names** of columns must match

```
> myDataFrame_03 <- rbind(myDataFrame_01, data.frame(x = -99, y = "ZzZ"))
      x     y
1    1     A
2    2     B
3    3     c
4 -99   ZzZ

> (myDataFrame_04 <- rbind(myDataFrame_01, data.frame(x = -99, y = 1)))
      x     y
1    1     A
2    2     B
3    3     c
4 -99 <NA>
Warning message: In `<-factor`(`*tmp*`, ri, value = 1) :
  invalid factor level, NA generated
```

Combine/Append Data Frames [CONT'D]

- When a data frame already exists, you can combine/append another data frame or a vector to the original data frame, but there are some caveats
 - 3 Use `rbind()` to row-bind a data frame with a vector
 - n.b. behavior can become quirky if the length of your vector is not equal to the number of columns in the data frame

```
> (myDataFrame_05 <- data.frame(x = 1:3, y = 98:100, z = 1000:1002))  
   x   y   z  
1 1  98 1000  
2 2  99 1001  
3 3 100 1002  
  
> (myDataFrame_06 <- rbind(myDataFrame_05, qqq = -1:-3))  
   x   y   z  
1 1  98 1000  
2 2  99 1001  
3 3 100 1002  
qqq -1  -2  -3
```

ANSWER THESE: rbind()

```
> myDataFrame_05 <- data.frame(x = 1:3, y = 98:100, z = 1000:1002)  
  
> myDataFrame_06 <- rbind(myDataFrame_05, ???)
```

- Based on the `myDataFrame_06` code, what happens if we replace `???` with:
 - (a) `qqq = -1`
 - (b) `qqq = -1:-2`
 - (c) `qqq = -1:-99`
 - (d) `qqq = c(-1, -2)`
 - (e) `qqq = c("-1", -2)`
 - (f) `qqq = c("a", -2, -3))`

SOLUTION

```
> myDataFrame_05 <- data.frame(x = 1:3, y = 98:100, z = 1000:1002)  
  
> myDataFrame_06 <- rbind(myDataFrame_05, ???)
```

- (a) Entire additional row of -1's
- (b) Entire additional row of repeating -1's and -2's
- (c) Additional row: -1, -2, -3
- (d) Entire additional row of repeating -1's and -2's
- (e) Entire additional row of repeating -1's and -2's as **characters** (non numeric), thereby changing **all** all data frame column types to **characters**
- (f) Additional row: a, -2, -3 as **characters** (non numeric), thereby changing **all** all data frame columns types to **characters**

Combine/Append Data Frames [CONT'D]

- When a data frame already exists, you can combine/append another data frame or a vector to the original data frame, but there are some caveats
 - 4 Use `cbind()` to column-bind a data frame with a vector
 - n.b. behavior can become quirky if the length of your vector is not equal to the number of rows in the data frame

```
> (myDataFrame_07 <- data.frame(x = 1:3, y = 98:100, z = 1000:1002))  
   x   y   z  
1 1  98 1000  
2 2  99 1001  
3 3 100 1002  
  
> (myDataFrame_08 <- cbind(myDataFrame_05, qqq = -1:-3))  
   x   y   z qqq  
1 1  98 1000  -1  
2 2  99 1001  -2  
3 3 100 1002  -3
```

ANSWER THESE: cbind()

```
> myDataFrame_07 <- data.frame(x = 1:3, y = 98:100, z = 1000:1002)  
  
> myDataFrame_08 <- cbind(myDataFrame_07, ???)
```

- Based on the `myDataFrame_08` code, what happens if we replace `???` with:
 - (a) `qqq = -1`
 - (b) `qqq = -1:-2`
 - (c) `qqq = -1:-99`
 - (d) `qqq = c("-1", -2)`
 - (e) `qqq = c("a", -2, -3))`

SOLUTION

```
> myDataFrame_07 <- data.frame(x = 1:3, y = 98:100, z = 1000:1002)  
  
> myDataFrame_08 <- cbind(myDataFrame_05, ???)
```

- (a) Entire additional column of -1's
- (b) <arguments imply differing number of rows: 3, 2>
- (c) Extends the length of all other columns and repeats those values until -99
- (d) <arguments imply differing number of rows: 3, 2>
- (e) <arguments imply differing number of rows: 3, 2>
- (f) Additional column: a, -2, -3 as **factors** (non numeric)

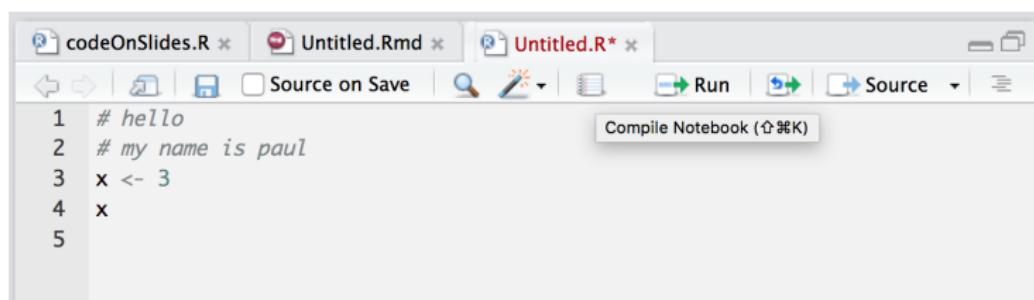
Subsection 2

RMarkdown

Technical Reporting & Presentation Tools

- There are many ways to generate pdf- and web-based technical reports and presentations
 - 1 \LaTeX
 - 2 Lyx
 - 3 Markdown (RMarkdown and other variations)
 - 4 ...
- The (arguably) default, most generic and most flexible technical report/presentation tool is \LaTeX
- When dealing with exclusively with R code and R output, RMarkdown is a versatile and easy way to embed code and graphical output in a report or presentation
- Alternate R-based packages exist for report generation

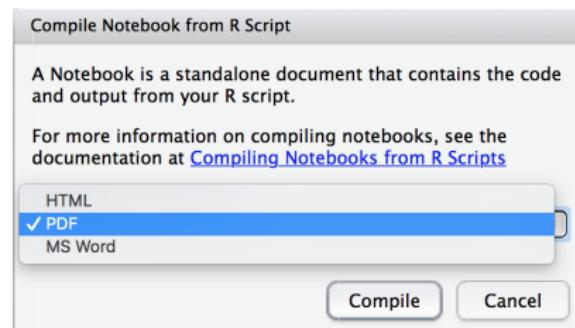
The Simplest RMarkdown Execution



A screenshot of the RStudio interface. The top menu bar shows three tabs: "codeOnSlides.R", "Untitled.Rmd", and "Untitled.R*". Below the tabs is a toolbar with various icons. The main workspace contains the following R code:

```
1 # hello
2 # my name is paul
3 x <- 3
4 x
5
```

In the top right corner of the workspace, there is a button labeled "Compile Notebook (⌃⌘K)".



The Simplest RMarkdown Execution

- Rmd source file
- pdf output

Untitled.R

Paul

Tue Jul 5 10:51:43 2016

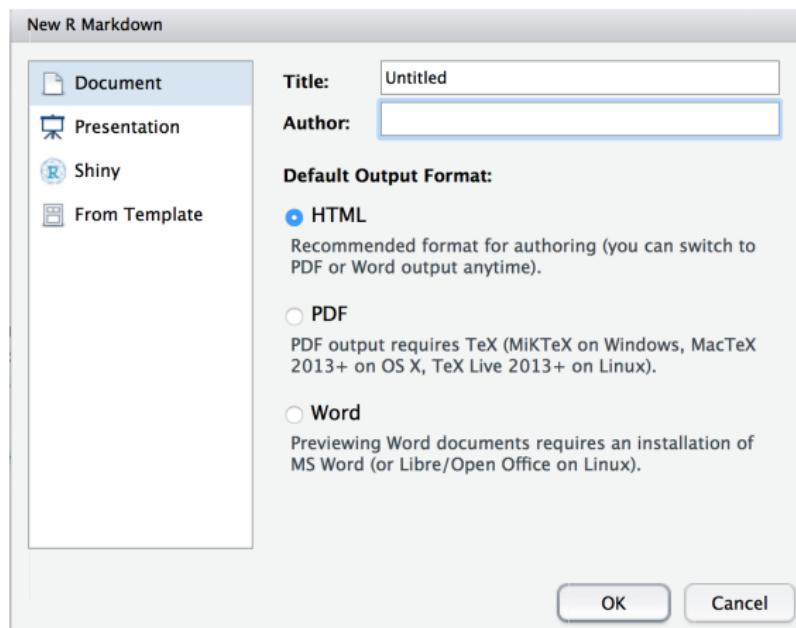
```
# hello
# my name is paul
x <- 3
x
```

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

A Smarter RMarkdown Document

Create a new RMarkdown file

File ⇒ New File ⇒ RMarkdown...



MacTex is Required for RMarkdown pdf Output



RMarkdown: The YAML Header

YAML a recursive acronym that means “YAML Ain’t Markup Language”

- The header **begins** and **ends** with three dashes ---
- There are many header options, we will examine a few basic options

```
---
```

```
title: "Untitled"
```

```
author: "Paul Intrevado"
```

```
date: "July 5, 2016"
```

```
output: pdf_document
```

```
---
```

```
---
```

RMarkdown: Body Text

- How do you write plain text?

Just like this

- How do you comment out a line of text?

[//]: (comment goes here)

<!-- (comment goes here) -->

- You can create sections / section headers using the “#” symbol

```
# Header 1  
## Header 2  
### Header 3  
#### Header 4  
##### Header 5  
##### Header 6
```

Header 1

Header 2

Header 3

Header 4

Header 5

Header 6

RMarkdown: Body Text [CONT'D]

- Inline equations are similar (identical?) to \LaTeX syntax
 - $e^{i\pi} - 1 = 0$ is written `$e^{i \ \backslash pi} - 1 = 0$`

RMarkdown: Body Text [CONT'D]

- Inline equations are similar (identical?) to \LaTeX syntax
 - $e^{i\pi} - 1 = 0$ is written `$e^{i \ \backslash pi} - 1 = 0$`
- Bold and italicized statements are written using
`**myBoldText**` and `*myItalicizedText*`

RMarkdown: Body Text [CONT'D]

- Inline equations are similar (identical?) to \LaTeX syntax
 - $e^{i\pi} - 1 = 0$ is written `$e^{i \ \backslash pi} - 1 = 0$`
- Bold and italicized statements are written using
`**myBoldText**` and `*myItalicizedText*`
- In-line code that is **not** executed can be included in backticks
(over tilde)

CODE To assign a value to a variable: ``myVar <- 1``

OUTPUT To assign a value to a variable: `myVar <- 1`

RMarkdown: Body Text [CONT'D]

- Inline equations are similar (identical?) to \LaTeX syntax
 - $e^{i\pi} - 1 = 0$ is written `$e^{i \ \backslash \pi} - 1 = 0$`
- Bold and italicized statements are written using
`**myBoldText**` and `*myItalicizedText*`
- In-line code that is **not** executed can be included in backticks
(over tilde)

CODE To assign a value to a variable: ``myVar <- 1``

OUTPUT To assign a value to a variable: `myVar <- 1`

- In-line code that **is** executed can be included as
``r <insert code here>``

CODE The product of 2 + 3 is ``r sum(c(2, 3))``

OUTPUT The product of 2 + 3 is 5

RMarkdown: Code Chunks

- At the heart of RMarkdown are the (poorly named) code chunks, which allow for great flexibility when including raw code as well as results, from simple computations to complex graphs and analyses

```
```{r <sectionTitle>, <options>}
<include code here>
```
```

- Use **ctrl + option + I** as a shortcut to include code chunk
 - **<sectionTitle>** is the *unique* name of the code chunk
 - **<options>** are a sequence of options separated by commas
- n.b.** all labels and code chunk options must be on the same line

RMarkdown: Selected Code Chunk Options

- `eval`: whether or not to run the code chunk
- `echo`: whether or not to include code in output document
- `include`: when FALSE, the code chunk is evaluated, but the results are not included in the output document
- `tidy`: whether or not to have *knitr* format printed code chunks

RMarkdown: Selected Global Chunk Options

- To set global options for all code chunks, include the following code chunk after the YAML header

```
```{r <sectionTitle>, include = FALSE}
knitr::opts_chunk$set(<options>)
```
```

- include = FALSE** is included so that the code chunk is not printed to the output document

E.g. To have all code chunks in an RMarkdown document be suppressed, include all code in the output document

```
```{r preamble, include = FALSE}
knitr::opts_chunk$set(echo = FALSE)
```
```

Including Non-R Code in Code Chunks

- RMarkdown is not limited to R code
- *knitr* can run code from a variety of other languages including but not limited to Python, Ruby and Bash
- To include non-R code in a code chunk, set the `engine` code chunk to tell *knitr* which language you are using

E.g. To include Python code

```
```{r engine = 'python'}
print "Hello World"
```
```

- Additional non-R programming language interpretation is available using the *highlighter* package

RMarkdown Resources

- Reproducible Research with R and RStudio by Christopher Gandrud
 - This is a less technical, more pragmatic approach to RMarkdown
- Dynamic Documents with R and *knitr* by Yihui Xie
 - A more technical, detailed and rigorous treatment of RMarkdown and *knitr*

RMarkdown Resources [July 2016]

- RMarkdown Cheat Sheet

<http://www.rstudio.com/wp-content/uploads/2015/02/rmarkdown-cheatsheet.pdf>

- RMarkdown Reference Guide

<http://www.rstudio.com/wp-content/uploads/2015/03/rmarkdown-reference.pdf>

- RMarkdown PDF Documents: Overview

http://rmarkdown.rstudio.com/pdf_document_format.html

LAB

My First RMarkdown Document

Paul Intrevado

July 14, 2016

R Markdown

This is my first R Markdown document. I am required to submit all MSAN 593 homework in RMarkdown. I am going to import a large dataset from the Housing Affordability Data System (HADS) from Data.gov using the `read.csv` function. The Housing Affordability Data System (HADS) is a set of housing-unit level datasets that measures the affordability of housing units and the housing cost burdens of households, relative to area median incomes, poverty level incomes, and Fair Market Rents.

```
read.csv("~/Desktop/hadsData.txt")
```

This fails for a few reasons, namely, I read in the file and stored it no where. So I wasted my time waiting for R to read in the file, and then when it finally did, it printed some rows to the Console window, and then the following message was printed to the console [`reached getOption("max.print") -- omitted 64434 rows`] and voila, the data disappeared faster than it loaded. Now I know better.

```
hadsData <- read.csv("~/Desktop/hadsData.txt")
```

The HADS dataset has 64535 rows and 90 columns. It would be imprudent to run `str()` on all 90 variables

IMPORTANT: Style Guide

Now that Introduction to RMarkdown is complete, be sure to **thoroughly read** the Style Guide (Chapter 5), in Hadley Wickham's Advanced R. You will be held to that standard in your coding style moving forward.

<http://adv-r.had.co.nz/Style.html>

Subsection 3

Creating & Manipulating Data in Base R

LAB

- Let's create some data for an experiment
- There are five variables of interest
 - 1 First Name
 - 2 Last Name
 - 3 Total months of work experience
 - 4 Whether or not you are married
 - 5 Area code of your phone number

LAB [CONT'D]

- How do we input this into R?

LAB [CONT'D]

- How do we input this into R?

```
# single quotes also work if you prefer
f_name <- c("paul", "john", "aasif", "saurin", "alex")
l_name <- c("intrevado", "smith", "ragna", "patel", "ozsen")

mosWorked <- c(22, 32, 7, 6, 87)
married <- c(TRUE, FALSE, FALSE, TRUE, FALSE)
areaCode <- c(415, 707, 415, 510, 510)
```

LAB [CONT'D]

- Is the data in the format we want?

LAB [CONT'D]

- Is the data in the format we want?
- You can examine the **Environment** pane in RStudio (easier)...

LAB [CONT'D]

- Is the data in the format we want?

LAB [CONT'D]

- Is the data in the format we want?
- You can examine the **Environment** pane in RStudio (easier)...

LAB [CONT'D]

- Is the data in the format we want?
- You can examine the **Environment** pane in RStudio (easier)...
- ... or use code (cumbersome)

```
> str(f_name)
  chr [1:5] "paul" "john" "aasif" "saurin" "alex"

> str(l_name)
  chr [1:5] "intrevado" "smith" "ragna" "patel" "ozsen"

> str(mosWorked)
  num [1:5] 22 32 7 6 87

> str(married)
  logi [1:5] TRUE FALSE FALSE TRUE FALSE

> str(areaCode)
  num [1:5] 415 707 415 510 510
```

LAB [CONT'D]

- areaCode should probably be changed to a nominal **factor** as it isn't truly meant to be a **numeric** value, and no summary statistics need be computed on areaCode

LAB [CONT'D]

- areaCode should probably be changed to a nominal **factor** as it isn't truly meant to be a **numeric** value, and no summary statistics need be computed on areaCode

```
> areaCode <- as.factor(areaCode)

> str(areaCode)
Factor w/ 3 levels "415","510","707": 1 3 1 2 2

> areaCode
[1] 415 707 415 510 510
Levels: 415 510 707
```

LAB [CONT'D]

■ What is the

- 1 mean
- 2 median
- 3 standard deviation
- 4 variance
- 5 maximum
- 6 minimum
- 7 75th percentile

of the number of months worked?

LAB [CONT'D]

```
> mean(mosWorked)
[1] 30.8

> median(mosWorked)
[1] 22

> sd(mosWorked)
[1] 33.23703

> var(mosWorked)
[1] 1104.7

> max(mosWorked)
[1] 87

> min(mosWorked)
[1] 6

> quantile(mosWorked)
 0%  25%  50%  75% 100%
   6    7   22   32   87
```

LAB [CONT'D]

- Take a shortcut by using the `summary()` function

```
> summary(mosWorked)
   Min. 1st Qu. Median     Mean 3rd Qu.     Max.
   6.0      7.0    22.0    30.8    32.0    87.0
```

which is a very similar output to the `quantile()` function except for the inclusion of the mean when calling `summary()`

LAB [CONT'D]

- How many people are married?
- What is the average number of people married?

LAB [CONT'D]

- How many people are married?

```
> sum(married)  
[1] 2
```

- What is the average number of people married?

```
> mean(married)  
[1] 0.4
```

- n.b.** Numerical summary statistics can be computed on a non-numeric (logical) vector

LAB [CONT'D]

- What is the mean age of married people who participated in this survey?
- This is a more complex question than any of the questions previously asked and requires us to create a link between the previously independent vectors

LAB [CONT'D]

- What is the mean age of married people who participated in this survey?
- This is a more complex question than any of the questions previously asked and requires us to create a link between the previously independent vectors
- We can collect and link all these vectors into a single data frame

```
> myExperimentalData <- data.frame(l_name, f_name, married,  
                                     areaCode, mosWorked)
```

```
> myExperimentalData  
    l_name   f_name  married areaCode  mosWorked  
1 intrevado paul     TRUE      415      22  
2     smith  john    FALSE      707      32  
3     ragna aasif    FALSE      415       7  
4     patel saurin   TRUE      510       6  
5     ozsen alex    FALSE      510      87
```

LAB [CONT'D]

- A more convenient, accessible, and aesthetically pleasing way is to observe the data frame in the **Environment** pane
- By clicking on the name of the **data frame**, `myExperimentalData`, a spreadsheet-like display will pop up in the **Console** pane

LAB [CONT'D]

- Let's explore this new data frame

```
> str(myExperimentalData)
'data.frame': 5 obs. of 5 variables:
 $ l_name   : Factor w/ 5 levels "intrevado","ozsen",...: 1 5 4 3 2
 $ f_name   : Factor w/ 5 levels "aasif","alex",...: 4 3 1 5 2
 $ married  : logi TRUE FALSE FALSE TRUE FALSE
 $ areaCode : Factor w/ 3 levels "415","510","707": 1 3 1 2 2
 $ mosWorked: num 22 32 7 6 87
```

- What has changed in the process of joining the individual vectors into a single data frame?

LAB [CONT'D]

- Let's explore this new data frame

```
> str(myExperimentalData)
'data.frame': 5 obs. of 5 variables:
 $ l_name    : Factor w/ 5 levels "intrevado","ozsen",...: 1 5 4 3 2
 $ f_name    : Factor w/ 5 levels "aasif","alex",...: 4 3 1 5 2
 $ married   : logi TRUE FALSE FALSE TRUE FALSE
 $ areaCode  : Factor w/ 3 levels "415","510","707": 1 3 1 2 2
 $ mosWorked : num  22 32 7 6 87
```

- What has changed in the process of joining the individual vectors into a single data frame?
- Both `l_name` and `f_name` have been converted from character vectors into factors while being merged into the data frame (and we weren't even asked)

LAB [CONT'D]

- To avoid this automatic character to factor coercion, include the additional argument `stringsAsFactors = FALSE` in the `data.frame` function

```
> myExperimentalData <- data.frame(l_name, f_name, married,
                                     areaCode, mosWorked,
                                     stringsAsFactors = FALSE)

> str(myExperimentalData)
'data.frame': 5 obs. of 5 variables:
 $ l_name    : chr  "intrevado" "smith" "ragna" "patel" ...
 $ f_name    : chr  "paul" "john" "aasif" "saurin" ...
 $ married   : logi  TRUE FALSE FALSE TRUE FALSE
 $ areaCode  : Factor w/ 3 levels "415","510","707": 1 3 1 2 2
 $ mosWorked: num  22 32 7 6 87
```

LAB [CONT'D]

- With the experimental data combined into a data frame, subsetting data is now far easier

```
> myExperimentalData <- data.frame(l_name, f_name, married,
                                     areaCode, mosWorked,
                                     stringsAsFactors = FALSE)

> str(myExperimentalData)
'data.frame': 5 obs. of 5 variables:
 $ l_name    : chr "intrevado" "smith" "ragna" "patel" ...
 $ f_name    : chr "paul" "john" "aasif" "saurin" ...
 $ married   : logi TRUE FALSE FALSE TRUE FALSE
 $ areaCode  : Factor w/ 3 levels "415","510","707": 1 3 1 2 2
 $ mosWorked: num 22 32 7 6 87
```

ANSWER THIS

Question

What is the mean number of months worked by married persons who participated in this survey?

ANSWER THIS

Question

What is the mean number of months worked by married persons who participated in this survey?

Solution

```
> mean(myExperimentalData$mosWorked[myExperimentalData$married])  
[1] 14
```

Subsection 4

Control Flow in R

Control Flow

- The ability to execute some statement repetitively, while only executing other statements if certain conditions are met
- R has the basic control structures one expects in a modern programming language
 - Repetition and Looping
 - `for()` loops
 - `while()` loops
 - `repeat()` loops
 - Conditional Execution
 - `if()`
 - `if() {} else if() {} else {}`
 - `ifelse()`
 - `switch()`

for()

- Executes a statement repetitively until a variable's value is no longer contained in the sequence seq

- The generic in-line syntax is

```
for (var in seq) expression
```

- A simple example which prints "MSAN" 5 times

```
for (i in 1:5) print('MSAN')
```

- It is possible to iterate over more complex sequences

```
> myVector <- factor(c("A", "A", "B", "C", "C", "C", "Zzz"))  
  
> for (k in levels(myVector)) print(k)  
[1] "A"  
[1] "B"  
[1] "C"  
[1] "Zzz"
```

for() [NOTES]

- The `seq` in a `for()` loop is evaluated at the start of the loop; changing it subsequently does not affect the loop
- You can make an assignment to the looping variable (e.g., `i`) within the body of the loop, but this will not affect the next iteration
- When the loop terminates, the looping variable contains its latest value
- The multi-line form of the `for()` loop

```
for (i in mySequence) {  
  < expression 1 >  
  ...  
  < expression n >  
}
```

while()

- Executes a statement repetitively until a condition is no longer true
- The generic in-line syntax is

```
while (condition) expression
```

- A simple example which prints “MSAN” 3 times

```
> n <- 3
```

```
> while (n > 0) {print('MSAN'); n <- n - 1}
```

n.b.

- Even though the `while()` loop above is written in-line, curly braces were employed as there is more than one expression
- The use of a semi-colon is only required when writing more than one in-line expression; when used in the multi-line format, the semi-colon can be omitted, but the curly braces may not be omitted

repeat()

- The `repeat()` loop can be used when the terminal condition does not apply at the top of the loop
- There is no condition to end the `repeat()` loop; a `repeat()` loop must be terminated with a `break` command placed somewhere inside `repeat()` loop
- The `break` command immediately exists the innermost active `for()`, `while()` or `repeat()` loop
- The `next` command forces the next iteration of a loop to begin immediately, returning control to the top of the loop

```
> x <- 7
```

```
> repeat{  
+   print(x)  
+   x <- x + 2  
+   if (x > 10 ) break  
+ }  
[1] 7  
[1] 9
```

if()

- The `if()` control structure executes a statement if a given condition is true

- The generic in-line syntax is

```
if (condition) expression
```

- A simple example

```
> x <- 3
```

```
> if (x > 0) print(paste("x is: ", x, sep = ""))
[1] "x is: 3"
```

- The multi-line form for `if()` is

```
if (condition) {
  < expression 1 >
  ...
  < expression n >
}
```

if() {} else {}

- The `if()` control structure executes a statement if a given condition is true

- The generic in-line syntax is

```
if (condition) expression_01 else expression_02
```

- A simple example

```
> x <- -3
```

```
> if (x > 0) print("x positive") else print("x is negative")
[1] "x is negative"
```

- The multi-line form of `if() {} else {}` is

```
if (condition) {
  < expressions >
} else {
  < alternate expressions >
}
```

if() {} else {}: Formatting Pitfalls

- This code snippet will run without error

```
x <- -3
```

```
if (x > 0) {  
  print(paste("x is: ", x, sep = ""))  
} else {  
  print("x is negative")  
}  
[1] "x is negative"
```

- The code snippet will throw an error

```
x <- -3
```

```
if (x > 0) {  
  print(paste("x is: ", x, sep = ""))  
}  
else {  
  print("x is negative")  
}
```

```
Error: unexpected '}' in "  }"
```

```
if() {} else if() {} else {}
```

- The multi-line form of `if() {} else if() {} else {}` is

```
if (condition_01) {  
    < expressions 01 >  
} else if (condition_02) {  
    < expressions 02 >  
} else {  
    < expressions 03 >
```

- As many `else if () {}` clauses may be chained (sequenced) together as desired

ifelse() versus if() {} else {}

- If a vector $x : |x| > 1$ is passed to an `if()` statement, only the first element of the vector will be evaluated for conditional execution; moreover, R will throw a warning
- The `ifelse()` construct is a vectorized version of `if() {} else {}` which tests each element of a vector passed to it

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

> if (x > 2) {print("first element in vector > 2")}
[1] "first element in vector > 2"
Warning message:
In if (x > 2) { :
  the condition has length > 1 and only the first element will be used

> ifelse(x > 2, ">2", "<=2")
[1] ">2"  "<=2" "<=2"
```

switch()

- `switch()` chooses statements based on the value on an expression
- The multi-line form of `switch()` is

```
switch(expression,  
       condition_01 = command_01,  
       condition_02 = command_02,  
       ...  
       condition_n = command_n,  
)
```

- n.b.**
- If the expression passed to `switch()` is not a character, it is coerced to `integer`
 - If the expression passed to `switch()` is a character string, then the string is matched exactly (with some small edge cases, see documentation)

switch() [EXAMPLE]

```
grades <- c("A", "D", "F")

for (i in grades) {
  print(
    switch(i,
      A = "Well Done",
      B = "Alright",
      C = "C's get Degrees!",
      D = "Meh",
      F = "Uh-Oh"
    )
  )
}
```

```
[1] "Well Done"
[1] "Meh"
[1] "Uh-Oh"
```

LAB

titanic.csv

- 1 Using a `for()` loop, recode the entries in the Survived variable with "Survived" and "Perished"
- 2 Using an `if()` loop, create a new variable of type ordered factor in the data frame called ageClass, and map Age to: "Minor" if less than 18 yrs; $18 \text{ yrs} \leq \text{Adult} \leq 65 \text{ yrs}$; and "Senior" if older than 65 yrs
- 3 Ordering the passengers in descending order by last name, use a `while()` loop to identify the name of the 100th surviving passenger
- 4 Using a `switch()` statement, identify each passenger class, Pclass, as either "First Class", "Business Class" or "Economy", and print the results to the console

LAB: BONUS QUESTION

titanic.csv

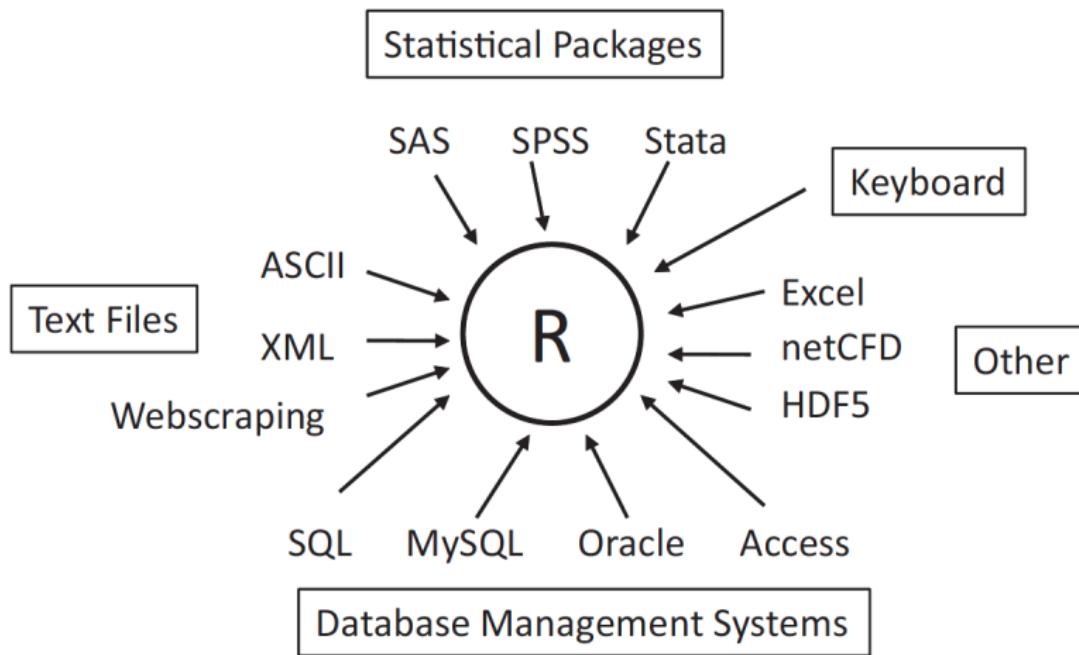
- Iterate through the data frame, and for variables that are numeric, create a histogram, for categorical variables create a bar chart, and skip over all others

- n.b.**
- 1 Be sure to correct and clean the variable types before you run code (e.g., there are only two truly numeric variables)
 - 2 After creating each graph, be sure to include a pop-up message that says "Press Enter for next Graph" to add a pause in the sequential execution

Subsection 5

Importing & Manipulating Data in R

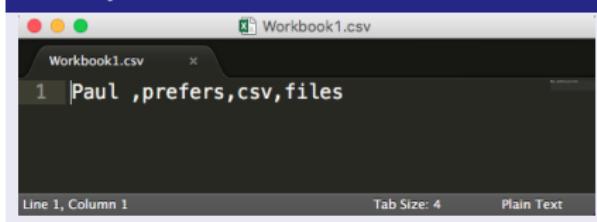
R can Import Data from a Multitude of Sources



Why csv Files?

How large is a file that contains the four separate words “*Paul prefers csv files*” ?

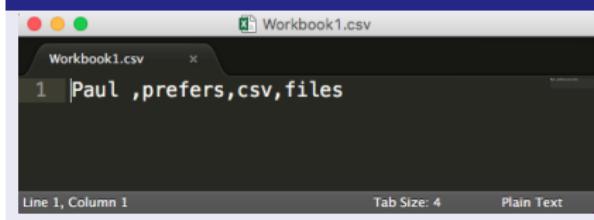
csv File (.csv)
23 bytes



Why csv Files?

How large is a file that contains the four separate words “*Paul prefers csv files*” ?

csv File (.csv)
23 bytes



Excel File v15 macOS (.xlsx)
22,687 bytes

A screenshot of Microsoft Excel for macOS. The ribbon menu shows "Home" selected. The worksheet area displays a 4x4 grid of cells. Cell G3 contains the value "1". Cells A1, B1, C1, and D1 contain the values "Paul", "prefers", "csv", and "files" respectively. The status bar at the bottom indicates "G3".

Importing Data

- You can import data directly from an Excel file if you like using `read.xlsx()` from the `xlsx` package, or from SQL using the `RODBC` package
- The cleanest (easiest?) and most universally-portable way to move data from one system/technology to another is using text-delimited files
- A comma-separated value (`csv`) file is one type of text-delimited file, where the delimiter is a comma
- **n.b.** Don't be fooled: even though the file has a `.csv` extension, this *is* a text file that can be opened and edited with any text editor
- Other common text-delimited file types include tab- and space-delimited files
 - typically these files have a `.txt` file extension

Common R Functions for Working with Data

| Function | Purpose |
|---|--|
| <code>length(object)</code> | Number of elements/components. |
| <code>dim(object)</code> | Dimensions of an object. |
| <code>str(object)</code> | Structure of an object. |
| <code>class(object)</code> | Class or type of an object. |
| <code>mode(object)</code> | How an object is stored. |
| <code>names(object)</code> | Names of components in an object. |
| <code>c(object, object, ...)</code> | Combines objects into a vector. |
| <code>cbind(object, object, ...)</code> | Combines objects as columns. |
| <code>rbind(object, object, ...)</code> | Combines objects as rows. |
| <code>object</code> | Prints the object. |
| <code>head(object)</code> | Lists the first part of the object. |
| <code>tail(object)</code> | Lists the last part of the object. |
| <code>ls()</code> | Lists current objects. |
| <code>rm(object, object, ...)</code> | Deletes one or more objects. The statement <code>rm(list = ls())</code> will remove most objects from the working environment. |

read.table()

- Perhaps the most flexible way to read a text-based file into R
- The default separator for `read.table()` is white space, i.e.,
`sep = ""` looks for one or more spaces, tabs, newlines or carriage returns to identify a new entry
- One has the flexibility to set `sep` to whatever separator the file uses, e.g., `sep = ","` for commas
- `read.table()` also accepts url addresses
- the `header` option, `FALSE` by default, indicates whether the file has a header
- `colClasses` can be used to create a vector of classes to be assumed for the columns
- There are **many** more options for `read.table()` that you should explore by reading the documentation

Stylized Variants of `read.table()`

- `read.csv()` is the equivalent of `read.table()`, but the default separator is `sep = ", "`, which saves you having to type that extra bit of code
- `read.csv2()` is the equivalent of `read.table()`, but the default separator is `sep = ";"` and the default character assumed for decimal points is `dec = ","`
- `read.delim()` is the equivalent of `read.table()`, but the default separator is `sep = "\t"`
- `read.delim2()` is the equivalent of `read.table()`, but the default separator is `sep = "\t"` and the default character assumed for decimal points is `dec = ","`

Subsection 6

Subsetting

Subsetting

- R's subsetting operators are powerful and fast
- Mastery of subsetting allows you to succinctly express complex operations in a way that few other languages can match

Subsetting Atomic Vectors

Let's create a default atomic vector

```
> (myAtomicVector_01 <- c(99.1, 98.2, 97.3, 96.4))
[1] 99.1 98.2 97.3 96.4

> str(myAtomicVector_01)
num [1:4] 99.1 98.2 97.3 96.4

> class(myAtomicVector_01)
[1] "numeric"

> is.atomic(myAtomicVector_01)
[1] TRUE

> typeof(myAtomicVector_01)
[1] "double"
```

n.b. note that the number after the decimal point gives the original position in the vector

How to Subset an Atomic Vector

Positive integers return elements at the specified positions

```
> (myAtomicVector_01 <- c(99.1, 98.2, 97.3, 96.4))  
[1] 99.1 98.2 97.3 96.4
```

```
> myAtomicVector_01[c(1, 3)]  
[1] 99.1 97.3
```

```
# real numbers are automatically truncated  
> myAtomicVector_01[c(1.999, 3.001)]  
[1] 99.1 97.3
```

```
> order(myAtomicVector_01)  
[1] 4 3 2 1
```

```
> myAtomicVector_01[order(myAtomicVector_01)]  
[1] 96.4 97.3 98.2 99.1
```

How to Subset an Atomic Vector

Negative integers omit elements at specified positions

```
> (myAtomicVector_01 <- c(99.1, 98.2, 97.3, 96.4))
```

```
[1] 99.1 98.2 97.3 96.4
```

```
> myAtomicVector_01[-c(1, 3)]
```

```
[1] 98.2 96.4
```

```
> myAtomicVector_01[c(-1, -3)]
```

```
[1] 98.2 96.4
```

```
> myAtomicVector_01[-c(-1, -3)]
```

```
[1] 99.1 97.3
```

```
> myAtomicVector_01[c(-1, - 3.99)]
```

```
[1] 98.2 96.4
```

How to Subset an Atomic Vector

Logical vectors select elements where the logical value is TRUE

```
> (myAtomicVector_01 <- c(99.1, 98.2, 97.3, 96.4))  
[1] 99.1 98.2 97.3 96.4
```

```
> myAtomicVector_01[c(TRUE, TRUE, FALSE, FALSE)]  
[1] 99.1 98.2
```

```
> myAtomicVector_01[c(T, TRUE, F, FALSE)]  
[1] 99.1 98.2
```

```
> myAtomicVector_01[c(T, T, F, F)]  
[1] 99.1 98.2
```

```
> myAtomicVector_01 > 98  
[1] TRUE TRUE FALSE FALSE
```

```
> myAtomicVector_01[myAtomicVector_01 > 98]  
[1] 99.1 98.2
```

How to Subset an Atomic Vector

Logical vectors can be quirky at times

- If the logical vector is shorter than the vector being subsetted, it will be recycled to be the same length

```
# equivalent to myAtomicVector_01[c(T, F, T, F)]
```

```
> myAtomicVector_01[c(T,F)]
```

```
[1] 99.1 97.3
```

```
> myAtomicVector_01[c(T, F, F)]
```

```
[1] 99.1 96.4
```

- A missing value in the index always yields a missing value in the output

```
> myAtomicVector_01[c(T, F, NA, T)]
```

```
[1] 99.1 NA 96.4
```

How to Subset an Atomic Vector

Character vectors can be employed to return elements with matching names **if the vector is named**

```
> (myAtomicVector_01 <- c(99.1, 98.2, 97.3, 96.4))  
[1] 99.1 98.2 97.3 96.4
```

```
> names(myAtomicVector_01) <- letters[1:4]
```

```
> myAtomicVector_01  
    a      b      c      d  
99.1 98.2 97.3 96.4
```

```
> myAtomicVector_01[c("a", "d")]  
    a      d  
99.1 96.4
```

- n.b.** partial character string matches will not work, e.g., if element name is abc, then `myAtomicVector_01["ab"]` will return NA

Subsetting Matrices and Arrays

- The most common way to subset matrices and arrays is to supply a one-dimensional index for each dimension, separated by a comma
- A blank index returns all entries in that dimension

```
> (myMatrix <- matrix(1:9, nrow = 3, byrow = TRUE))
   [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9

> colnames(myMatrix) <- letters[1:3]

> myMatrix
     a b c
[1,] 1 2 3
[2,] 4 5 6
[3,] 7 8 9
```

Subsetting Matrices and Arrays [CONT'D]

```
> rownames(myMatrix) <- letters[4:6]      > myMatrix[6]
                                             [1] 8
> myMatrix
  a b c
d 1 2 3
e 4 5 6
f 7 8 9
# recycling logical row entries
> myMatrix[c(T, F), ]
  a b c
d 1 2 3
f 7 8 9
> myMatrix[1, 1]
[1] 1
> myMatrix[1:2, "b"]
d e
2 5
# -----
# "[" will simplify results to the
# lowest possible dimensionality
# (more on this soon)
# -----
```

```
> myMatrix[1:2, ]
  a b c
d 1 2 3
e 4 5 6
```

Subsetting Data Frames

As data frames possess the characteristics of both lists and matrices, you can subset using two methods

```
> (myDataFrame <- data.frame(x = 1:3, y = -4:-6, z = LETTERS[1:3]))  
   x   y z  
1 1 -4 A  
2 2 -5 B  
3 3 -6 C  
  
> myDataFrame[1:2]  
   x   y  
1 1 -4  
2 2 -5  
3 3 -6  
  
> myDataFrame[1,3]  
[1] A  
Levels: A B C
```

Subsetting Data Frames [CONT'D]

- If you subset using a single vector, the result behaves as a list
- If you subset using two vectors, the result behaves as a matrix

```
> myDataFrame[1]          > myDataFrame[, c(1, 3)]  
  x                         x z  
1 1                         1 1 A  
2 2                         2 2 B  
3 3                         3 3 C  
  
> myDataFrame[1,]          > myDataFrame["x"]  
  x  y  z                  x  
1 1 -4 A                 1 1  
2 2  
3 3  
  
> myDataFrame[c(1, 3)]    2 2  
  x z  
1 1 A  
2 2 B  
3 3 C  
3 3  
  
> str(myDataFrame["x"]) # index as a list  
'data.frame': 3 obs. of 1 variable:  
$ x: int 1 2 3  
  
> myDataFrame[2, 2]        > str(myDataFrame[, "x"]) # index as a matrix  
[1] -5                      int [1:3] 1 2 3
```

Subsetting Operators

- We have seen the use of `[` to subset, but we can also use `[[` and `$` to subset
- Use `[[` when working with lists, because when `[` is used to subset a list, it always returns a list, whereas using `[[` to subset a list returns the **contents** of said list
- As data frames are lists of columns, you can use `[[` to extract a column from a data frame

```
> myDataFrame <- data.frame(x = 1:3, y = -4:-6, z = LETTERS[1:3])

> class(myDataFrame["x"])
[1] "data.frame"
> class(myDataFrame[1])
[1] "data.frame"
> class(myDataFrame[["x"]])
[1] "integer"
> class(myDataFrame[[1]])
[1] "integer"

> typeof(myDataFrame["x"])
[1] "list"
> typeof(myDataFrame[1])
[1] "list"
> typeof(myDataFrame[["x"]])
[1] "integer"
> typeof(myDataFrame[[1]])
[1] "integer"
```

Simplifying vs. Preserving Subsetting

- **Simplifying** subsets returns the simplest possible data structure that can represent the output
- **Preserving** subsetting keep the structure of the output the same as the input and is generally better programming etiquette because the result will always be the same
- The `drop` option when subsetting is one of the most common sources of programming error

`drop` logical. If TRUE the result is coerced to the lowest possible dimension. The default is to drop if only one column is left, but not to drop if only one row is left.

- Preserving is the same for all data types, but simplifying behavior varies slightly across data types

Simplifying vs. Preserving Subsetting (EXAMPLES)

Atomic Vectors

```
> myAtomicVector_01 <- c(99.1, 98.2, 97.3, 96.4)
> names(myAtomicVector_01) <- LETTERS[1:4]

> myAtomicVector_01
    A      B      C      D
99.1 98.2 97.3 96.4

> (x_01 <- myAtomicVector_01[1])
    A
99.1

> x_01["A"]
    A
99.1

> (x_02 <- myAtomicVector_01[[1]]) # strips away the names
[1] 99.1

> x_02["A"]
[1] NA
```

Simplifying vs. Preserving Subsetting (EXAMPLES)

Lists

```
> myList <- list(A = 1, B = 2)
```

```
> str(myList[1])
```

```
List of 1
```

```
 $ A: num 1
```

```
> typeof(myList[1])
```

```
[1] "list"
```

```
> class(myList[1])
```

```
[1] "list"
```

```
> str(myList[[1]])
```

```
num 1
```

```
> typeof(myList[[1]])
```

```
[1] "double"
```

```
> class(myList[[1]])
```

```
[1] "numeric"
```

Simplifying vs. Preserving Subsetting (EXAMPLES)

Factors

```
> (myFactor <- factor(LETTERS[1:3]))      # drops unused levels
[1] A B C
Levels: A B C

> myFactor[1]
[1] A
Levels: A B C

> myFactor[[1]]
[1] A
Levels: A B C

> typeof(myFactor[1])
[1] "integer"

> class(myFactor[1, drop = TRUE])
[1] "factor"

> (myFactor <- factor(LETTERS[1:3]))
[1] A B C
Levels: A B C

> myFactor[1]
[1] A
Levels: A

> myFactor[1, drop = TRUE]
[1] A
Levels: A

> typeof(myFactor[1, drop = TRUE])
[1] "integer"

> class(myFactor[1, drop = TRUE])
[1] "factor"
```

Simplifying vs. Preserving Subsetting (EXAMPLES)

Matrices / Arrays

```
> (myMatrix <- matrix(1:9, nrow = 3)) # if any dimension has length 1,  
    [,1] [,2] [,3]                      #   dimension is dropped  
[1,]    1    2    3  
[2,]    4    5    6  
[3,]    7    8    9  
  
> myMatrix[1, , drop = F]                > str(myMatrix[1, ])  
    [,1] [,2] [,3]                      int [1:3] 1 2 3  
[1,]    1    2    3  
  
> str(myMatrix[1, , drop = F])          > class(myMatrix[1, ])  
int [1, 1:3] 1 2 3                     [1] "integer"  
  
> class(myMatrix[1, , drop = F])        > typeof(myMatrix[1, ])  
[1] "matrix"                            [1] "integer"  
  
> typeof(myMatrix[1, , drop = F])       # why are  
[1] "integer"                           #   typeof(myMatrix[1, , drop = F])  
                                         #   and typeof(myMatrix[1, ])  
                                         #   both "integer" ?
```

Simplifying vs. Preserving Subsetting (EXAMPLES)

Data Frames

```
> (myDataFrame <- data.frame(x = 1:3, y = -4:-6))
   x   y
1 1 -4
2 2 -5
3 3 -6
> str(myDataFrame[[1]])
'data.frame': 3 obs. of 1 variable:
 $ x: int 1 2 3
> typeof(myDataFrame[[1]])
[1] "integer"
> class(myDataFrame[[1]])
[1] "integer"
# if the output is a single column,
# returns a vector instead of
# a data frame
> str(myDataFrame[[1]])
int [1:3] 1 2 3
> typeof(myDataFrame[[1]])
[1] "integer"
> class(myDataFrame[[1]])
[1] "integer"
```

Simplifying vs. Preserving Subsetting (EXAMPLES)

Data Frames [CONT'D]

```
> (myDataFrame <- data.frame(x = 1:3, y = -4:-6))
   x   y
1 1 -4
2 2 -5
3 3 -6
> str(myDataFrame[, "x"])
  int [1:3] 1 2 3
> str(myDataFrame[, "x", drop = F])
'data.frame': 3 obs. of 1 variable:
 $ x: int 1 2 3
> typeof(myDataFrame[, "x"])
[1] "integer"
> class(myDataFrame[, "x"])
[1] "integer"
> typeof(myDataFrame[, "x", drop = F])
[1] "list"
> class(myDataFrame[, "x", drop = F])
[1] "data.frame"
# if the output is a single column,
# returns a vector instead of
# a data frame
```

\$

- \$ is a shorthand operator often used to access variables within a data frame
- `x$y ≡ x[["y", exact = FALSE]]`

`exact` Controls possible partial matching of `[[` when extracting by a character vector (for most objects, but see under Environments). The default is no partial matching. Value NA allows partial matching but issues a warning when it occurs. Value FALSE allows partial matching without any warning.

Partial Matching with \$

```
> (myDataFrame <- data.frame(abc = 1:3, abd = -4:-6,
  xyz = LETTERS[1:3], stringsAsFactors = F))
  abc abd xyz
  1   1  -4    A                         # exact match to column name
  2   2  -5    B
  3   3  -6    C                         > myDataFrame[["abd", exact = F]]
                                         [1] -4 -5 -6

# when the call returns more than one
#   column of a data frame, a value
#   of NULL is returned
> myDataFrame[["a", exact = F]]          # partial matching b/c exact = F
                                         # call returns single column
                                         # therefore non-NULL return
                                         > myDataFrame[["x", exact = F]]
                                         [1] "A" "B" "C"

# ibid
> myDataFrame[["ab", exact = F]]          > myDataFrame$a
                                         NULL

                                         > myDataFrame$x
                                         [1] "A" "B" "C"
```

Dynamic Column Referencing with \$

If you want to dynamically access a column of a data frame, you might choose to store the name of that column in variable.
When doing so, be sure not access the column incorrectly.

```
> (myDataFrame <- data.frame(abc = 1:3, xyz = -4:-6))
  abc xyz
1   1  -4
2   2  -5
3   3  -6
> (dynColName <- "xyz")
[1] "xyz"

> myDataFrame$dynColName
NULL

> myDataFrame[dynColName]
xyz
1  -4
2  -5
3  -6

> myDataFrame[[dynColName]]
[1] -4 -5 -6
```

\$ versus [[

- \$ does partial matching
- [[does **not** do partial matching

```
> (myDataFrame <- data.frame(abc = 1:3, xyz = -4:-6))
   abc xyz
1    1   -4
2    2   -5
3    3   -6

> myDataFrame$a
[1] 1 2 3

> myDataFrame["a"]
Error in ` [.data.frame` (myDataFrame, "a") : undefined columns selected

> myDataFrame[["a"]]
NULL
```

LAB

- 1 Using the simple linear regression model
`myMod <- lm(mpg ~ wt, data = mtcars)`
 - 1 Extract the residual degrees of freedom
 - 2 Extract R^2 and R_A^2
- 2 Write code that would randomly permute the columns of a data frame.
- 3 Write code that would randomly permute the rows of a data frame.
- 4 Write code that would randomly permute both the rows and columns of a data frame.

Subsection 7

Useful R Functions

table()

table() is quick and dirty way to build a contingency table of the counts at each combination of factor levels

- Using the iris data set, how many of each type of Species are there?

```
> table(iris$Species)
```

| | setosa | versicolor | virginica |
|--|--------|------------|-----------|
| | 50 | 50 | 50 |

- Using the ToothGrowth data set, what is the count of dose by supp?

```
> table(ToothGrowth$dose, ToothGrowth$supp)
```

| | OJ | VC |
|-----|----|----|
| 0.5 | 10 | 10 |
| 1 | 10 | 10 |
| 2 | 10 | 10 |

with()

- Tired of having to retype the name of the data frame over and over and over?
- By enclosing a statement or multiple statements in `with()`, you can operate at the column (variable) level of a data frame without having to retype the name of the data frame

Using ToothGrowth data set, what is the count of dose by supp?

```
> with(ToothGrowth,  
      table(dose, supp)  
    )
```

| | OJ | VC |
|-----|----|----|
| 0.5 | 10 | 10 |
| 1 | 10 | 10 |
| 2 | 10 | 10 |

attach() & detach()

- Don't use these!
- `attach()` allows you to semi-permanently attach a data frame, thereby not having to enclose statement in a `with()`
- Can you guess what issue might arise when attaching multiple data frames?
- Again...DON'T USE THESE!

unique()

- Identifies unique values in data, with duplicate elements removed

Cars have how many unique cylinder sizes in mtcars?

```
> with(mtcars,  
      unique(cyl)  
    )
```

```
[1] 6 4 8
```

which()

- Returns the row number (index) of a subset of data

Which observations in mtcars have cars with 6 cylinders and 100 horsepower?

```
with(mtcars,  
      which(cyl == 6 & hp == 110)  
    )  
  
[1] 1 2 4 # rows (observations) 1, 2 and 4
```

subset()

- Confused about the complex manner in which vectors and data frames are subsetted?
- `subset()` return subsets of vectors or data frames which meet certain conditions

```
> subset(mtcars, hp > 250)
      mpg cyl disp  hp drat   wt qsec vs am gear carb
Ford Pantera L 15.8   8 351 264 4.22 3.17 14.5  0  1    5    4
Maserati Bora   15.0   8 301 335 3.54 3.57 14.6  0  1    5    8

> subset(mtcars, mpg > 32, select = c("hp", "disp"))
      hp disp
Fiat 128       66 78.7
Toyota Corolla 65 71.1
```

paste()

- Converts arguments to characters and then concatenates them

```
with(subset(iris, Sepal.Length >= 7.7),
     paste("The ", Species, " has a petal length of ",
           Petal.Length, sep = ""))
[1] "The virginica has a petal length of 6.7"
[2] "The virginica has a petal length of 6.9"
[3] "The virginica has a petal length of 6.7"
[4] "The virginica has a petal length of 6.4"
[5] "The virginica has a petal length of 6.1"
```

order()

- `order()` returns a permutation which rearranges its first argument into ascending or descending order
- the default ordering is `decreasing = TRUE`
- use the `na.last` option to decide whether to put all `NAs` at the end of an order list or not (default is `FALSE`)

```
> order(mtcars$disp)
[1] 20 19 18 26 28  3 21 27 32  9 30  8  1  2 10 11  6  4 12 13 14 31 23
[24] 22 24 29  5  7 25 17 16 15

> head(mtcars[order(mtcars$disp), ], 3)
      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Toyota Corolla 33.9   4 71.1 65 4.22 1.835 19.90  1  1     4     1
Honda Civic    30.4   4 75.7 52 4.93 1.615 18.52  1  1     4     2
Fiat 128       32.4   4 78.7 66 4.08 2.200 19.47  1  1     4     1
```

length()

- `length()` gets or sets the length of vectors (including lists) and factors
- Be cautious when using `length()` on anything other than an atomic vector, as the results may be confusing or unexpected

```
> length(LETTERS[1:10])
[1] 10

> length(matrix(1:12, ncol = 3))
[1] 12

> length(data.frame(x = 1:3, y = letters[1:3]))
[1] 2
```

seq()

- `seq()` and its variants are a quick way to generate sequences
- Common arguments include `from`, `to`, and `by`

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

> seq(1:10)
[1] 1 2 3 4 5 6 7 8 9 10

> seq(1, 10, 2) # the last argument is equivalent to by = 2
[1] 1 3 5 7 9

> seq(1, 10, length.out = 10)
[1] 1 2 3 4 5 6 7 8 9 10
```

seq_len() & seq_along()

- two stylized variants of `seq()` which are abbreviated versions of `seq()` and very fast results
- `seq_len()` only take one argument, `length.out`, and generates a sequence of integers beginning from one to `length.out`
- `seq_along()` is (ideally) passed a vector and generates a sequence of integers from 1 to `length(myVector)`

```
> seq_len(15)
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

> (myVec <- c(LETTERS[4:8]))
[1] "D" "E" "F" "G" "H"

> seq_along(myVec)
[1] 1 2 3 4 5
```

rep()

- `rep()` replicates the value(s) passed to it n times
- Setting the `each` equal to a positive integer repeats each entry sequentially that integer number of times

```
> rep(LETTERS[2:4], times = 3)
[1] "B" "C" "D" "B" "C" "D" "B" "C" "D"

> rep(letters[2:4], each = 2)
[1] "b" "b" "c" "c" "d" "d"

> rep(letters[2:4], times = 2, each = 2)
[1] "b" "b" "c" "c" "d" "d" "b" "b" "c" "c" "d" "d"
```

cut()

- `cut()` is a great function that takes a numeric vector, cuts it into n groups, and then converts those groups into factors
- `breaks` dictates how many evenly-spaced groups to create

```
> x <- seq(1,100, by = 3)

> y <- cut(x, breaks = 2)

> levels(y) # observe these are currently NOT ordered
[1] "(0.901,50.5]" "(50.5,100]"

> myDataFrame <- data.frame(x,y)

> myDataFrame[c(1, 2, nrow(myDataFrame) - 1, nrow(myDataFrame)), ]
      x           y
1    1 (0.901,50.5]
2    4 (0.901,50.5]
33  97   (50.5,100]
34 100   (50.5,100]
```

pretty()

- A prettier version of `cut()`, `pretty()` computes a sequence of about $n + 1$ equally spaced ‘round’ values that cover the range of the values of the vector passed to it
- Values are chosen so that they are 1, 2 or 5 times a power of 10
- `pretty()` can either decide the optimal number of breaks on its own, or it can be passed the argument `n`

```
> x <- seq(1,100, by = 3)

> pretty(x)
[1] 0 20 40 60 80 100

> pretty(x, 3)
[1] 0 50 100
```

any()

- Given a set of logical vectors, is at least one of the values **TRUE**?
- Returns a single logical value
- If relevant, can use **na.rm** option

```
> myAtomicVector <- c(1 ,2, 99.99, NA, sqrt(2))

> is.na(myAtomicVector)
[1] FALSE FALSE FALSE  TRUE FALSE

> any(is.na(myAtomicVector))
[1] TRUE
```

sample()

- `sample()` takes a sample of the specified size from the elements of a vector passed to it
- `replace` permits for sampling with or without replacement (default is `FALSE`, i.e., w/o replacement)
- `n` is non-negative, integral sample size

```
> sample(LETTERS, 5)
[1] "D" "R" "J" "I" "L"

> mtcars[sample(nrow(mtcars), 5), ]
          mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Chrysler Imperial 14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
Merc 450SLC       15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
Merc 280          19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
Merc 230          22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Datsun 710         22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
```

source()

- In its simplest form, `source()` permits running an external R script while inside another R script

E.g. The file `firstFile.R` contains a single line of code:

```
titanicData <- read.csv("~/titanic.csv")
```

A second file, `secondFile.R` can call `firstFile.R` via the `source()` function to run all code in `firstFile.R`

```
source("~/firstFile.R")
```

which, in this case, would result in loading the csv file and storing it in a data frame `titanicData`, where code from `secondFile.R` could then manipulate said data frame

LAB

Using iris dataset

- 1 How many observations there for each sepal length?
- 2 Generate eight equally-spaced values that cover the range of sepal length.
- 3 Calculate how many unique species there are.
- 4 Subset the data for flowers with petal width greater than 2.3
- 5 Create a sample of size three without replacement for odd numbers between 11 and 87.

Section 3

Advanced Techniques in R

Section Contents

3 Advanced Techniques in R

- Functions
- Writing Robust R Code
- Functional Programming & Functionals
- Evaluating the Performance of R Code
- Character Functions and Manipulation in R

Subsection 1

Functions

An Introduction to Rigorous Functional Programming

The focus of this section is to turn your existing, informal knowledge of functions into a rigorous understanding of what functions are and how they work.

The most important thing to understand about R is that functions are objects in their own right. You can work with them exactly the same way you work with any other type of object.

Function Components

All R functions have three parts

- the `body()`, the code inside the function
- the `formals()`, the list of arguments which controls how you can call the function
- the `environment()`, the *map* of the location of the function's variables

```
> myFunc <- function(x) x^2
```

```
> myFunc
function(x) x^2
```

```
> formals(myFunc)
$x
```

```
> body(myFunc)
x^2
```

```
> environment(myFunc)
<environment: R_GlobalEnv>
```

LAB

- 1 Write a function that takes two arguments, `a` and `b`, and returns rows `a` through `b` of `mtcars`
- 2 Write a function that takes a numeric vector as an input, squares every value in the vector, appends the squared vector to the original vector in the form of a data frame, and prints the first 10 rows of the data frame to the console
- 3 Write a function that takes a numeric vector as an input, squares every value in the vector, appends the squared vector to the original vector in the form of a data frame, and then returns and stores the data frame **over** the original vector, i.e., replace the old vector (which was input) with the new data frame (which is returned)

Lexical Scoping

- Scoping is the set of rules that govern how R looks up the value of a symbol
- There are four basic principles behind R's implementation of lexical scoping
 - 1 name masking
 - 2 functions vs. variables
 - 3 a fresh start
 - 4 dynamic lookup

A Brief Digression

- Whenever writing code, you want to be sure to clear your environment to ensure the fidelity of your results
- In each and every R script file I write, I always include the following two lines of code

```
rm(list=ls())
cat("\014")
```

- 1 `rm(list=ls())` removes all objects in the current environment
- 2 `ls` When invoked with no argument at the top level prompt, ls shows what data sets and functions a user has defined. When invoked with no argument inside a function, ls returns the names of the function's local variables.
- 2 On a Mac, `cat("\014")` clear the console windows (same as `ctrl + l`)

Name Masking

```
rm(list=ls())  
  
myFunc_01 <- function() {  
  x <- 1  
  y <- 2  
  c(x,y)  
}  
  
myFunc_01()
```

What does the preceding code return?

Name Masking

```
rm(list=ls())  
  
myFunc_01 <- function() {  
  x <- 1  
  y <- 2  
  c(x,y)  
}  
  
myFunc_01()
```

What does the preceding code return?

[1] 1 2

The function searches inside itself for `x` and `y`

Name Masking [CONT'D]

If a name isn't defined inside a function, R will look one level up

```
rm(list=ls())  
  
x <- 2  
  
myFunc_02 <- function() {  
  y <- 1  
  c(x,y)  
}  
  
myFunc_02()
```

What does the preceding code return?

Name Masking [CONT'D]

If a name isn't defined inside a function, R will look one level up

```
rm(list=ls())  
  
x <- 2  
  
myFunc_02 <- function() {  
  y <- 1  
  c(x,y)  
}  
  
myFunc_02()
```

What does the preceding code return?

```
[1] 2 1
```

- What would happen if you omitted `x <- 2` from the previous code?

Name Masking [CONT'D]

```
rm(list=ls())  
  
x <- 1  
myFunc_03 <- function() {  
  y <- 2  
  myFunc_04 <- function() {  
    z <- 3  
    c(x,y,z)  
  }  
  myFunc_04()  
}  
myFunc_03()
```

What does the preceding code return?

Name Masking [CONT'D]

```
rm(list=ls())  
  
x <- 1  
myFunc_03 <- function() {  
  y <- 2  
  myFunc_04 <- function() {  
    z <- 3  
    c(x,y,z)  
  }  
  myFunc_04()  
}  
myFunc_03()
```

What does the preceding code return?

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

In this case, the search begins inside the function, then where that function was defined, etc., up to the global environment, and finally to loaded packages.

Name Masking [CONT'D]

```
rm(list=ls())  
  
x <- 1  
myFunc_03 <- function() {  
  x <- 1000  
  y <- 2  
  myFunc_04 <- function() {  
    x <- 99  
    z <- 3  
    c(x,y,z)  
  }  
  myFunc_04()  
}  
myFunc_03()
```

What does the preceding code return?

Name Masking [CONT'D]

```
rm(list=ls())  
  
x <- 1  
myFunc_03 <- function() {  
  x <- 1000  
  y <- 2  
  myFunc_04 <- function() {  
    x <- 99  
    z <- 3  
    c(x,y,z)  
  }  
  myFunc_04()  
}  
myFunc_03()
```

What does the preceding code return?

[1] 99 2 3

Functions vs. Variables

The same principles apply for finding functions just as they do for finding variables

```
rm(list=ls())  
  
myFunc_04 <- function(x) x + 99  
  
myFunc_05 <- function() {  
  myFunc_04 <- function(x) x * 2  
  myFunc_04(20)  
}  
  
myFunc_05()
```

What does the preceding code return?

Functions vs. Variables

The same principles apply for finding functions just as they do for finding variables

```
rm(list=ls())  
  
myFunc_04 <- function(x) x + 99  
  
myFunc_05 <- function() {  
  myFunc_04 <- function(x) x * 2  
  myFunc_04(20)  
}  
  
myFunc_05()
```

What does the preceding code return?

```
[1] 40
```

- n.b.** Avoid coding confusion and unexpected results: **don't** give identical names to functions and variables

New Functional Environments for Each Execution

- Every time a function is called, a new environment is called to host execution; each invocation is completely independent
- The following function returns a value of 999 every time

```
# NOTE rm(list=ls()) is deleted

myFunc_06 <- function() {
  if(!exists("myAtomicVector")){
    myAtomicVector <- 999
  } else {
    myAtomicVector <- myAtomicVector + 1
  }
  print(myAtomicVector)
}

myFunc_06()
```

Real-Time Variable Lookup

A function will search for a value when it's run, **not** when it's created

```
> rm(list=ls())  
  
> myFunc_07 <- function() x  
  
> x <- 15  
  
> myFunc_07()  
[1] 15  
  
> x <- 20  
  
> myFunc_07()  
[1] 20
```

Self-Contained Functions

- Variables internal to a function, i.e., variables which are not passed to a function, should be locally scoped to ensure that a function is self-contained
- A function that is not self-contained can cause a pernicious error that can be difficult to identify
- Use the `findGlobals` function from the `codetools` package to identify global variables in a function

```
> rm(list=ls())  
  
> myFunc_08 <- function() x + 1  
  
# NOTE myFunc_08 is not self-contained  
  
> codetools::findGlobals(myFunc_08)  
[1] "+" "x"
```

LAB

- Write any function with locally-scoped variables, confirming there are locally scoped using the `codetools` package

Formal Arguments of a Function

- It is important to distinguish between the formal and actual arguments of a function
- **Formal arguments** are a property of the function

Arithmetic Mean

Description

Generic function for the (trimmed) arithmetic mean.

Usage

```
mean(x, ...)  
## Default S3 method:  
mean(x, trim = 0, na.rm = FALSE, ...)
```

Arguments

- x An R object. Currently there are methods for numeric/logical vectors and [date](#), [date-time](#) and [time interval](#) objects. Complex vectors are allowed for `trim = 0`, only.
- trim the fraction (0 to 0.5) of observations to be trimmed from each end of x before the mean is computed. Values of trim outside that range are taken as the nearest endpoint.
- na.rm a logical value indicating whether NA values should be stripped before the computation proceeds.
- ... further arguments passed to or from other methods.

Calling Arguments of a Function

- It is important to distinguish between the formal and actual arguments of a function
- **Actual or calling arguments** can vary each time you call a function

```
> mean(x = 1:10)  
[1] 5.5
```

```
> mean(x = 99:999)  
[1] 549
```

- In the above examples, the calling arguments are `1:10` and `99:999` respectively

Calling Arguments of a Function [CONT'D]

- When calling a function you can specify arguments by position, by complete name, or by partial name
- Arguments are matched in the following order
 - 1 Exact name (perfect matching)
 - 2 Prefix matching (imperfect/partial matching)
 - 3 Position

```
myFunc_09 <- function(arg1, my_arg2, my_arg3){  
  list(a = arg1, m1 = my_arg2, m2 = my_arg3)  
}
```

Calling Arguments of a Function [CONT'D]

- When calling a function you can specify arguments by position, by complete name, or by partial name
- Arguments are matched in the following order
 - 1 Exact name (perfect matching)
 - 2 Prefix matching (imperfect/partial matching)
 - 3 Position

```
# positional                                # joint partial matching and positional
> str(myFunc_09(1, 2, 3))                  > str(myFunc_09(2, 3, a = 1))
List of 3                                     List of 3
$ a : num 1                                  $ a : num 1
$ m1: num 2                                 $ m1: num 2
$ m2: num 3                                  $ m2: num 3

# exact matching and positional      # partial matching fails if match is ambiguous
> str(myFunc_09(2, 3, arg1 = 1))          > str(myFunc_09(1, 3, my = 1))
List of 3                                     Error in myFunc_09(1, 3, my = 1) :
$ a : num 1                                  argument 3 matches multiple formal arguments
$ m1: num 2
$ m2: num 3
```

Best Practices for Calling Arguments

- You only want to use positional matching for the first one or two arguments of a function call, i.e., the most commonly used arguments
 - Avoid using positional matching for infrequently used arguments
 - If a function uses ... (ellipsis), you can only specify arguments listed after the ... with their full name, i.e., exact matching
- n.b.** If you are writing code for a package to be published to CRAN, you are not permitted to use partial matching

Using a List of Arguments to Call a Function

- If you wish call a function with a list of arguments, use the following code

```
> funcArguments <- list(1:10, na.rm = TRUE)

> do.call(mean, funcArguments)
[1] 5.5

# equivalent to
> mean(1:10, na.rm = TRUE)
[1] 5.5
```

Default Arguments

Function arguments in R can have default values

```
# w/o default values
myFunc_10 <- function(a, b) {
  c(a, b)
}

> myFunc_10()
Error in myFunc_10() : argument "a" is missing, with no default
```

```
# with default values
myFunc_11 <- function(a = 1, b = 2) {
  c(a, b)
}

> myFunc_11()
[1] 1 2
```

Default Arguments [CONT'D]

Function arguments in R can be defined in terms of other arguments

```
myFunc_12 <- function(a = 1, b = a * 2) {  
  c(a, b)  
}  
  
> myFunc_12()  
[1] 1 2  
  
> myFunc_12(111)  
[1] 111 222  
  
> myFunc_12(99, 100)  
[1] 99 100
```

Missing Arguments

To determine whether or not an argument was supplied to a function, there are two common approaches

1 `missing()`

```
myFunc_13 <- function(arg1, arg2) {  
  c(missing(arg1), missing(arg2))  
}
```

```
> myFunc_13()  
[1] TRUE TRUE
```

```
> myFunc_13(arg1 = 1)  
[1] FALSE TRUE
```

```
> myFunc_13(arg2 = 99)  
[1] TRUE FALSE
```

```
> myFunc_13(1, 2)  
[1] FALSE FALSE
```

Missing Arguments [CONT'D]

To determine whether or not an argument was supplied to a function, there are two common approaches

- 2 Set default argument values to `NULL` and subsequently test if the argument is supplied using `is.null()`

```
> myFunc_14 <- function(arg1 = NULL, arg2 = NULL) {  
  c(is.null(arg1), is.null(arg2))  
}  
  
> myFunc_14()  
[1] TRUE TRUE  
  
> myFunc_14(arg1 = 1)  
[1] FALSE TRUE  
  
> myFunc_14(arg2 = 99)  
[1] TRUE FALSE  
  
> myFunc_14(1,2)  
[1] FALSE FALSE
```

Lazy Functional Evaluation of Calling Arguments

- R function arguments are only evaluated when they are used
- If you want to ensure that an argument is evaluated you can use `force()`

```
myFunc_15 <- function(x){  
  10  
}  
  
> myFunc_15()  
[1] 10  
  
> myFunc_15(thisIsNonsense)  
[1] 10  
  
> myFunc_15("nonsense")  
[1] 10
```

```
myFunc_16 <- function(x){  
  force(x)  
  10  
}  
  
> myFunc_16(thisIsNonsense)  
Error in force(x) : object  
  'thisIsNonsense' not found
```

Lazy Evaluation, Default & Missing Arguments

- Default arguments are evaluated inside the function
- If the expression depends on the current environment the results will differ depending on whether you use the default value or explicitly provide one

```
myFunc_17 <- function(a = ls()){  
  z <- 10  
  a  
}  
  
> myFunc_17()  
[1] "a" "z"  
  
> myFunc_17(ls())  
[1] "i"          "j"          "myFunc_13"  
[4] "myFunc_15"  "myFunc_17"
```

Return Values

The last expression evaluated in a function becomes the return value

```
myFunc_18 <- function(xyz) {  
  if (xyz < 10) {  
    0  
  } else {  
    10  
  }  
}
```

```
> myFunc_18(5)  
[1] 0
```

```
> myFunc_18(10)  
[1] 10
```

To return() or not to return()

- The last expression evaluated in a function is the return value
- You can always wrap the final expression in `return()` if you choose
- Calling `return()` is an additional call and will add to the execution time of your function, albeit minuscule for a single call
- In simplistic functions, R programmers will typically omit `return()`
- In longer, more complicated functions, `return()` is often used to distinguish “leaves” of code
- In sum, for the purposes of this class, I require the use of `return()` to make the code more legible for any functions with “leaves” of code

To return() or not to return() [EXAMPLE]

```
# simple function, does not require a return()

myFunc_15 <- function(x){
  10
}

# a more complex function benefits visually from having return()
#   but does not require return()

myFunc_18 <- function(xyz) {
  if (xyz < 10) {
    return(0)
  } else {
    return(10)
  }
}
```

LAB

- 1 Write a function that takes two arguments, `firstRow` and `lastRow`, and returns rows `firstRow` through `lastRow` of `iris`, and subsequently call the function with values `firstRow = 1` and `lastRow = 3`, using both positional matching and exact matching
- 2 In the question above, what are the formal and calling arguments of the function?
- 3 Is this function self-contained? Why or why not?
- 4 Rewrite the above function to include a data frame `myDataFrame` as an additional argument, such that it returns rows `firstRow` through `lastRow` of `myDataFrame`
- 5 Rewrite the function to use default arguments `firstRow = 1` and `lastRow = 10`, and evaluate all 3 arguments at the beginning of the function using `force`

Subsection 2

Writing Robust R Code

Writing Robust R Code

Debugging

How to fix unanticipated problems

Condition Handling

How functions communicate problems and how actions can be taken based on those communications

Defensive Programming

How to avoid common problems before they occur

Debugging Tools

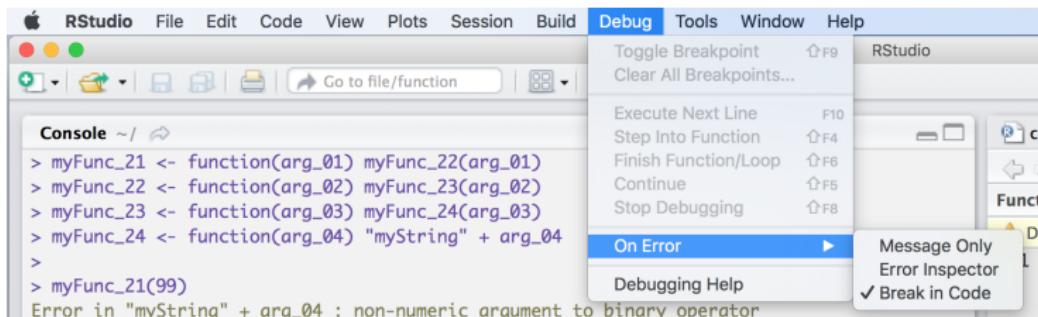
There are three key debugging tools

- 1 Error inspector and `traceback()` which lists a sequence of calls that lead to the error
- 2 “Return with Debug” tool and `options(error = browser)` which open an interactive session where the error occurred
- 3 Breakpoints and `browser()` which open an interactive session at an arbitrary location in the code

A Brief Digression

Depending on your selection in the menu bar, different actions will occur when R throws an error

- Selecting **Message Only** will simply print an error message to the console
- Selecting **Error Inspector** additionally provides links to *Show Traceback* and *Rerun with Debug*
- Selecting **Break in Code** additionally launches *Browse on Error*



Traceback & The Call Stack

- The call stack is the sequence of calls that lead up to an error
- For example, if we run the following code...

```
> rm(list=ls())  
  
> myFunc_21 <- function(arg_01) myFunc_22(arg_01)  
> myFunc_22 <- function(arg_02) myFunc_23(arg_02)  
> myFunc_23 <- function(arg_03) myFunc_24(arg_03)  
> myFunc_24 <- function(arg_04) "myString" + arg_04
```

Traceback & The Call Stack

- The call stack is the sequence of calls that lead up to an error
- For example, if we run the following code...

```
> rm(list=ls())  
  
> myFunc_21 <- function(arg_01) myFunc_22(arg_01)  
> myFunc_22 <- function(arg_02) myFunc_23(arg_02)  
> myFunc_23 <- function(arg_03) myFunc_24(arg_03)  
> myFunc_24 <- function(arg_04) "myString" + arg_04
```

- ... and then call `myFunc_21()`, we see the following error message

```
> myFunc_21(99)  
Error in "myString" + arg_04 : non-numeric argument to binary operator
```

Traceback & The Call Stack [CONT'D]

- Looking at the **Console** pane, you should see the following

```
> myFunc_21(99)
Error in "myString" + arg_04 : non-numeric argument to binary operator
  ⚡ Hide Traceback
  ⚡ Rerun with Debug

4 myFunc_24(arg_03)
3 myFunc_23(arg_02)
2 myFunc_22(arg_01)
1 myFunc_21(99)
```

- The call stack is to be read from bottom to top:
 - The initial call is to `myFunc_21()`
 - `myFunc_21()` calls `myFunc_22()`
 - `myFunc_22()` calls `myFunc_23()`
 - `myFunc_23()` calls `myFunc_24()` which triggers the error
- The **Traceback** window shows you where the error occurred, **not** why it occurred

Browsing on Error

- Selecting *Rerun with Debug* allows you to enter the interactive debugger
- This reruns the command that created the error, pausing the execution where the error occurred
- This puts you in an interactive state inside the function, and you can interact with any objects defined there
- You will observe
 - 1 A **Traceback** pane with the call stack
 - 2 An **Environment** pane with all objects in the current environment
 - 3 A **Code Browser** pane (icon of glasses) listing the statement that will be run next highlighted in yellow
 - 4 A `Browse[1]>` prompt in the console window which allows you to run arbitrary code

Browsing on Error [CONT'D]

The screenshot shows the RStudio interface with several panes:

- Console** pane: Displays R code execution and errors. It shows two error messages related to the `myFunc_21` function, which is called from `myFunc_24`.
- Code** pane: Shows the source code for `myFunc_24` and `codeOnSlides.R`.
- Environment** pane: Shows the variable `arg_04` with value `99`.
- Traceback** pane: Shows the call stack leading to the error.

```
Console ~ / 
> myFunc_21(99)
Error in "myString" + arg_04 : non-numeric argument to binary operator
> myFunc_21(99)
Error in "myString" + arg_04 : non-numeric argument to binary operator
Called from: myFunc_24(arg_03)
Browse[1]>
```

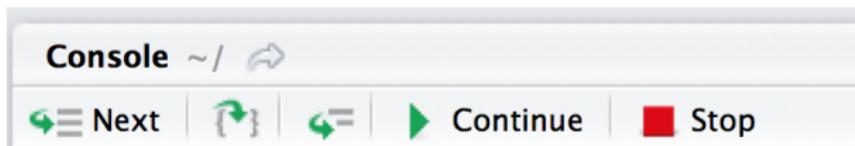
```
codeOnSlides.R * myFunc_24 * 
Function: myFunc_24 (.GlobalEnv)
(Debug-only)
1 function(arg_04) "myString" + arg_04
```

```
Environment History Plots 
myFunc_240 +
Values
arg_04      99
```

```
Traceback
eval(expr, envir, enclos)
myFunc_24(arg_03)
myFunc_23(arg_02)
myFunc_22(arg_01)
myFunc_21(99)
```

Browsing on Error [CONT'D]

A few special commands can be accessed in the toolbar on the **Console** pane (from left to right)



- Next executes the next step in the function
- Step Into works similarly to Next, except if the next line is a function, it will also step into that function
- Finish completes execution of current loop or function
- Continue leaves interactive debugging and continues regular execution of the function
- Stop stops debugging, terminates function, and returns to the global workspace

Condition Handling

- The task of handling expected errors, e.g., when your function is expecting an atomic vector as an argument but is passed a data frame
- In R, there are two main tools for handling conditions (including errors) programmatically
 - 1 `try()` gives you the ability to continue execution even when an error occurs
 - 2 `tryCatch()` lets you specify *handler* functions that control what happens when a condition is signaled

Ignoring Errors with `try()`

Whereas running a function that throws an error terminates the execution, wrapping code in the statement `try()` results in an error message printing **but** execution will continue

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

myFunc_25 <- function(z){
  log(z)
  print("Made it here")
}

> myFunc_25("abc")
Error in log(z) : non-numeric argument to mathematical function

myFunc_26 <- function(z){
  try(log(z))
  print("Made it here")
}

> myFunc_26("abc")
Error in log(z) : non-numeric argument to mathematical function
[1] "Made it here"
```

Ignoring Errors with `try()` [CONT'D]

- If you prefer, you can suppress the error message with
`try(..., silent = TRUE)`
- Large blocks of code can be wrapped in `try()`
- The output of `try()` can also be captured
 - 1 If the execution of code within `try()` is successful, the result will be the last result evaluated (just as in a function)
 - 2 If the execution of code in unsuccessful, the (invisible) result will be of class `try-error`

```
> successful <- try(1 + 99)

> class(successful)
[1] "numeric"

> unsuccessful <- try("a" + "b")
Error in "a" + "b" : non-numeric argument to binary operator

> class(unsuccessful)
[1] "try-error"
```

Handling Conditions with `tryCatch()`

- `tryCatch()` is a general tool for handling conditions
- `tryCatch()` can handle
 - 1 errors (made by `stop()`)
 - 2 warnings (`warning()`)
 - 3 message (`message()`)
 - 4 interrupts (user-terminated code execution, e.g., `ctrl + C`)
- `tryCatch()` maps conditions to **handlers**, i.e., named functions that are called with the condition as an argument
- If a condition is signaled, `tryCatch()` will call the first handles whose name matches one of the classes of the condition

Handling Conditions with `tryCatch()` [EXAMPLE]

```
show_condition <- function(code) {  
  tryCatch(code,  
    error = function(x) "myError",  
    warning = function(x) "myWarning",  
    message = function(x) "myMessage"  
  )  
}  
  
> show_condition(stop("!"))  
[1] "myError"  
  
> show_condition(warning("?!"))  
[1] "myWarning"  
  
> show_condition(message("?"))  
[1] "myMessage"  
  
> show_condition(10)  
[1] 10
```

Handling Conditions with `tryCatch()` [EXAMPLE CONT'D]

Let's follow the execution of the function `show_condition()` step by step

- 1 `show_condition(stop("!"))` calls the function `show_condition()`, passing `stop("!")` as the argument, represented in the function as `code`
- 2 `code` is executed in the `tryCatch()` block, where `code == stop("!")`
- 3 the function `stop()` "*stops execution of the current expression and executes an error action*"
- 4 when `stop()` executes an error action, `tryCatch()` maps the `error` condition to a function `error = function(x)` "`myError`", which prints the word `myError` to the console
- 5 execution of the function terminates

Handling Conditions with `tryCatch()` [EXAMPLE CONT'D]

- When a condition is mapped to a function, what is being passed to that function?
- Let's modify the previous code and explore the inner workings of condition handling

```
show_condition <- function(code) {  
  tryCatch(code,  
          error = function(x) y <<- x  
  )  
}  
  
> show_condition(stop("!"))  
## this call generates no message in the console
```

- This is the first time we observe the `<<-` operator, which makes an assignment to a global variable
- In the above function, `x` exists only in the functional environment whereas `y` exists in the global environment

Handling Conditions with `tryCatch()` [EXAMPLE CONT'D]

```
> y
<simpleError in doTryCatch(return(expr), name, parentenv, handler): !>

> str(y)
List of 2
 $ message: chr "!"
 $ call    : language doTryCatch(return(expr), name, parentenv, handler)
 - attr(*, "class")= chr [1:3] "simpleError" "error" "condition"

> attributes(y)
$names
[1] "message" "call"

$class
[1] "simpleError" "error"      "condition"

> y$message
[1] "!"

> y$call
doTryCatch(return(expr), name, parentenv, handler)
```

Handling Conditions with `tryCatch()` [EXAMPLE CONT'D]

- The function mapped to a particular condition in `tryCatch()` may be customized

```
show_condition <- function(code) {  
  tryCatch(code,  
    error = function(x) {  
      print(x$message)  
      print(x$call)  
      writeLines("\nSilly errors like this make\n Paul very angry")  
    }  
  )  
}  
  
> show_condition(stop("!"))  
[1] "!"  
doTryCatch(return(expr), name, parentenv, handler)  
  
Silly errors like this make  
Paul very angry
```

LAB

Write a function employing error handling techniques that takes a single vector as input, take the natural log of each element in that vector, and print the result of each to the console

Defensive Programming

- Defensive programming is the art of making code fail in a well-defined manner even when something unexpected occurs
- A key principle of defensive programming is to *fail fast*: as soon as something wrong is discovered, signal an error
- This *fail fast* behavior is more work up front for the programmer, but results in easier debugging for the user, as they receive errors earlier rather than later, before the error has been potentially digested by multiple functions

Implementing the *Fail Fast* Principle

- 1 Be strict about what a function accepts
 - If a function is not vectorized in inputs but uses functions that are, build in a check to ensure that inputs are scalars
 - Use `stopifnot()` or the `assertthat` package

Implementing the *Fail Fast* Principle

- 1 Be strict about what a function accepts
 - If a function is not vectorized in inputs but uses functions that are, build in a check to ensure that inputs are scalars
 - Use `stopifnot()` or the `assertthat` package
- 2 Avoid functions that use non-standard evaluation such as `subset`, `transform` and `with`
 - These functions save time when working with R interactively, but they typically fail uninformatively
 - Non-standard evaluation is the ability of a computing language to access not only the value(s) of a function's argument but also the code used to compute them (Advanced R, Chapter 13)

Implementing the *Fail Fast* Principle

- 1 Be strict about what a function accepts
 - If a function is not vectorized in inputs but uses functions that are, build in a check to ensure that inputs are scalars
 - Use `stopifnot()` or the `assertthat` package
- 2 Avoid functions that use non-standard evaluation such as `subset`, `transform` and `with`
 - These functions save time when working with R interactively, but they typically fail uninformatively
 - Non-standard evaluation is the ability of a computing language to access not only the value(s) of a function's argument but also the code used to compute them (Advanced R, Chapter 13)
- 3 Avoid functions that return different output depending on their input
 - Two big offenders are `[` and `sapply()`
 - Whenever subsetting a data frame in a function, **always** use the option `drop = F` to maintain the data structure, e.g., to avoid converting a one-column data frame to an atomic vector

stop() versus stopifnot()

```
> myVec <- c("a", "bcd", "efgh")  
  
> if(length(unique(nchar(myVec))) != 1) {  
    stop("Error: Elements of your input vector do not have the same length!")  
}  
Error: Error: Elements of your input vector do not have the same length!  
  
> stopifnot(length(unique(nchar(myVec))) != 1,  
           "Error: Elements of your input vector HAVE the same length!")  
Error: "Error: Elements of your input vector HAVE the same length!" is not TRUE  
  
> stopifnot(1 == 1, all.equal(pi, 3.14159265), 1 < 2) # all TRUE  
  
> stopifnot(1 == 2, all.equal(pi, 3.14159265), 1 < 2) # all first is FALSE  
Error: 1 == 2 is not TRUE
```

Subsection 3

Functional Programming & Functionals

Functional Programming

- Assume you are given the following data frame

```
> myDataFrame_01
  A  B C  D  E F
1 1  6 1  5 -99 1
2 10 4 4 -99  9 3
3 7  9 5  4  1 4
4 2  9 3  8  6 8
5 1 10 5  9  8 6
6 6  2 1  3  8 5
```

- Your objective is to replace all of the -99s with NAs

Functional Programming

- Assume you are given the following data frame

```
> myDataFrame_01
  A  B  C  D  E  F
1 1  6  1  5 -99  1
2 10 4  4 -99   9  3
3 7  9  5  4   1  4
4 2  9  3  8   6  8
5 1 10  5  9   8  6
6 6  2  1  3   8  5
```

- Your objective is to replace all of the -99s with NAs
- You could—but shouldn't—iterate through each column manually, e.g.

```
myDataFrame_01$A[myDataFrame_01$A == -99] <- NA
myDataFrame_01$B[myDataFrame_01$B == -99] <- NA
...
myDataFrame_01$F[myDataFrame_01$F == -99] <- NA
```

Problems with Brute-Force Approaches

- 1 It's easy to make copy-paste mistakes
 - 2 It makes bugs more likely
 - 3 It makes updating code a HUGE pain in the arse
 - 4 etc.
-
- Employ the **Do Not Repeat Yourself (DRY)** Principle

"Every piece of knowledge must have a single, unambiguous, authoritative representation within a system" [Thomas & Hunt, <http://pragprog.com>]

Functional Programming [EXAMPLE 1]

Let's write a function with the objective of replacing all -99s in a single column with NAs

```
fix99s_byCol <- function(myCol) {  
  myCol[myCol == -99] <- NA  
}
```

- Will the code above work as intended?

Functional Programming [EXAMPLE 1]

Let's write a function with the objective of replacing all -99s in a single column with NAs

```
fix99s_byCol <- function(myCol) {  
  myCol[myCol == -99] <- NA  
}
```

- Will the code above work as intended? Hint: **no**. Why not?

Functional Programming [EXAMPLE 1]

Let's write a function with the objective of replacing all -99s in a single column with NAs

```
fix99s_byCol <- function(myCol) {  
  myCol[myCol == -99] <- NA  
}
```

- Will the code above work as intended? Hint: **no**. Why not?
- The following **does** work as intended

```
fix99s_byCol <- function(myCol) {  
  myCol[myCol == -99] <- NA  
  myCol  
}  
  
myDataFrame_01$A <- fix99s_byCol(myDataFrame_01$A)  
...  
myDataFrame_01$F <- fix99s_byCol(myDataFrame_01$F)
```

- This reduces but doesn't eliminate the potential for errors
- There is no gain in efficiency (repetitive code is still required)

Functional Programming [EXAMPLE 1 REVISITED]

- What if there were a function that could iterate not only across all rows of a column checking for NAs, but also across all columns of a data frame?
- `lapply()`—from the generic family of `apply()` functionals—takes three inputs
 - 1 A list
 - 2 A function (applied to each element of the list)
 - 3 ... (other arguments to pass to the function)
- `lapply()` applies the function to each element of a list a returns the new list
- n.b.** We can employ `lapply()` here because data frames are lists

Functional Programming [EXAMPLE 1 REVISITED]

- What if there were a function that could iterate not only across all rows of a column checking for NAs, but also across all columns of a data frame?
 - `lapply()`—from the generic family of `apply()` functionals—takes three inputs
 - 1 A list
 - 2 A function (applied to each element of the list)
 - 3 ... (other arguments to pass to the function)
 - `lapply()` applies the function to each element of a list and returns the new list
- n.b.** We can employ `lapply()` here because data frames are lists

Definition `lapply()` returns a list of the same length as (the list) X, each element of which is the result of applying a function to the corresponding element of X.

Functional Programming [EXAMPLE 1 REVISITED]

```
fix99s_byCol <- function(myCol) {  
  myCol[myCol == -99] <- NA  
  myCol  
}  
  
> myDataFrame_01 <- lapply(myDataFrame_01, fix99s_byCol)  
  
> str(myDataFrame_01)  
List of 6  
 $ A: num [1:6] 1 10 7 2 1 6  
 $ B: num [1:6] 6 4 9 9 10 2  
 $ C: num [1:6] 1 4 5 3 5 1  
 $ D: num [1:6] 5 NA 4 8 9 3  
 $ E: num [1:6] NA 9 1 6 8 8  
 $ F: num [1:6] 1 3 4 8 6 5
```

- This almost worked...but not quite
- As the name implies, `lapply()` returns a list, not a data frame

Functional Programming [EXAMPLE 1 REVISITED] [CONT'D]

Here are two ways to correct the previous function call so that it returns a data frame

- 1 > myDataFrame_01 <-
as.data.frame(lapply(myDataFrame_01,
fix99s_byCol))
- 2 > myDataFrame_01[] <- lapply(myDataFrame_01,
fix99s_byCol)

Functional Programming [EXAMPLE 1 REVISITED]

Employing functional programming, as in the previous example, has many advantages

- 1 It is very compact
- 2 If the code for a missing value changes, it only needs to be updated in a single location
- 3 It works for any number of columns, so you don't need to specify the number of columns, therefore avoiding potential mistakes
- 4 All columns are evaluated uniformly
- 5 You can generalize the technique to a subset of columns if preferred

```
> myDataFrame_01[1:3] <- lapply(myDataFrame_01[1:3], fix99s_byCol)
```

Adding Arguments

- What if different columns employed different coding schemes for missing values, e.g., -99, -999 and -8888888?
- You could end up copy/pasting the function

```
fix99s_byCol <- function(myCol) {  
  myCol[myCol == -99] <- NA  
  myCol  
}
```

and replacing the -99 in `myCol[myCol == -99] <- NA`, with a updated values in each copy/paste so that you end up with three different functions, but we know better than that

Adding Arguments

- What if different columns employed different coding schemes for missing values, e.g., -99, -999 and -8888888?
- You could end up copy/pasting the function

```
fix99s_byCol <- function(myCol) {  
  myCol[myCol == -99] <- NA  
  myCol  
}
```

and replacing the -99 in `myCol[myCol == -99] <- NA`, with a updated values in each copy/paste so that you end up with three different functions, but we know better than that

- We can simply add an argument to the previous code as follows

```
fixMissing <- function(myCol, myValue) {  
  myCol[myCol == myValue] <- NA  
  myCol  
}
```

Anonymous Functions

- The following code is equivalent and permissible

```
myFunc_26 <- function(myCol) {  
  myCol <- myCol + 3  
  myCol  
}  
  
lapply(myDataFrame_01, myFunc_26)  
  
#####  
  
lapply(myDataFrame_01, function(x) x + 3)
```

- What you are observing in the lower half of the code is an anonymous function, i.e., a function that does not have a name
- This is distinct from other languages that require you to bind a function to a name, e.g., C++, Python, Ruby, etc.

LAB

- Create a compact and robust function which, when passed an $n \times m$ numeric data frame, returns, for each column, the
 - 1 Mean
 - 2 Median
 - 3 Standard Deviation
 - 4 Variance
 - 5 Quantiles
 - 6 IQR

LAB

- Create a compact and robust function which, when passed an $n \times m$ numeric data frame, returns, for each column, the
 - 1 Mean
 - 2 Median
 - 3 Standard Deviation
 - 4 Variance
 - 5 Quantiles
 - 6 IQR
- NOT THE BEST SOLUTION (but it works)

```
mySummaryFunc <- function(myCols) {  
  c(mean(myCols), median(myCols), sd(myCols), var(myCols),  
    quantile(myCols), IQR(myCols))  
}  
  
lapply(myDataFrame_01, mySummaryFunc)
```

Functionals

A functional is a function that takes a function as an input and returns a vector as an output

```
myFunctional_01 <- function(myFuncArg) myFuncArg(runif(1000), na.rm = TRUE)
```

This works as follows

- 1 We create a functional named `myFunctional_01`
- 2 We pass the argument `myFuncArg` to `myFunctional_01`,
where the argument is itself a function
- 3 The argument `myFuncArg` is then called using the parameters
defined in the inline, anonymous function, namely,
`runif(1000)`, `na.rm = TRUE`, which generates 1,000
random variates $\sim \mathcal{U}[0, 1]$, with all `NA` values excluded from
whatever calculation is executed by the argument `myFuncArg`

Functionals [CONT'D]

When we call the functional

```
myFunctional_01 <- function(myFuncArg) myFuncArg(runif(1000), na.rm = TRUE)
```

we get the following results

```
> myFunctional_01(mean)  
[1] 0.5194186
```

```
> myFunctional_01(mean)  
[1] 0.5038302
```

```
> myFunctional_01(min)  
[1] 0.000956069
```

```
> myFunctional_01(max)  
[1] 0.9990114
```

```
> myFunctional_01(sd)  
[1] 0.2846844
```

Why use Functionals?

- A common use of functionals is as an alternative to *for* loops
- *For* loops have a reputation for being slow in R, which is only partly true
- The real advantage of using functionals is the ability to express a clear, specific objective in a single statement
- Loops are far more abstracted from their objective
- As a consequence of the clearly-articulated objective of a functional, the probability of generating bugs in your code decreases

The `apply()` Family of Functionals

The `apply()` family of functionals are often used in lieu of `for` loops, coming in a variety of flavors (not exhaustive)

| Functional | Input | Output |
|-----------------------|--------------|--------------|
| <code>apply()</code> | Array/Matrix | Vector/Array |
| <code>lapply()</code> | Vector/List | List |
| <code>sapply()</code> | Vector/List | ... |
| <code>vapply()</code> | Vector/List | Vector |

Let's examine a few examples to convince ourselves that the `apply()` family of functionals are truly useful

LAB

Using state.x77

- 1 Write code that **does not contain** functionals that computes the mean of each column of data
- 2 Employ your functional of choice to write code that computes the mean of each column of data

LAB [SOLUTION]

- 1 Write code that **does not contain** functionals that computes the mean of each column of data

```
> myColMeans_01 <- numeric(ncol(state.x77))

for (i in 1:8) {
  myColMeans_01[i] <- mean(state.x77[ , i])
}

> myColMeans_01
[1] 4246.4200 4435.8000      1.1700    70.8786     7.3780      ...
```

- 2 Employ your functional of choice to write code that computes the mean of each column of data

```
> (myColMeans_02 <- apply(state.x77, 2, mean))
Population      Income Illiteracy   Life Exp      Murder    HS Grad ...
4246.4200 4435.8000      1.1700    70.8786     7.3780 53.1080 ...
```

LAB [SOLUTION] [CONT'D]

Observations about the use of `apply()`

- 1 What data type is being passed to `apply()`?

LAB [SOLUTION] [CONT'D]

Observations about the use of `apply()`

- 1 What data type is being passed to `apply()`?

```
> class(state.x77)
[1] "matrix"
```

- 2 What data type is being passed to the `mean` function?

LAB [SOLUTION] [CONT'D]

Observations about the use of `apply()`

- 1 What data type is being passed to `apply()`?

```
> class(state.x77)
[1] "matrix"
```

- 2 What data type is being passed to the `mean` function?

```
> apply(state.x77, 2, class)
Population      Income Illiteracy     Life Exp     Murder     HS Grad    ...
  "numeric"    "numeric"   "numeric"    "numeric"   "numeric"   "numeric"   ...
```

```
> apply(state.x77, 2, is.matrix)
Population      Income Illiteracy     Life Exp     Murder     HS Grad    ...
  FALSE        FALSE       FALSE      FALSE      FALSE      FALSE      ...
```

```
> apply(state.x77, 2, is.vector)
Population      Income Illiteracy     Life Exp     Murder     HS Grad    ...
  TRUE         TRUE       TRUE      TRUE      TRUE      TRUE      ...
```

LAB [SOLUTION] [CONT'D]

Observations about the use of `apply()`

- 3 What data type is returned by `apply()`?

LAB [SOLUTION] [CONT'D]

Observations about the use of `apply()`

- 3 What data type is returned by `apply()`?

```
> class(apply(state.x77, 2, mean))
[1] "numeric"

> is.matrix(apply(state.x77, 2, mean))
[1] FALSE

> is.vector(apply(state.x77, 2, mean))
[1] TRUE
```

LAB [SOLUTION] [CONT'D]

Observations about the use of `apply()`

- 3 What data type is returned by `apply()`?

```
> class(apply(state.x77, 2, mean))  
[1] "numeric"
```

```
> is.matrix(apply(state.x77, 2, mean))  
[1] FALSE
```

```
> is.vector(apply(state.x77, 2, mean))  
[1] TRUE
```

- 4 `apply()` coerces input to either a matrix (in 2 dimensions) or an array (in > 2 dimensions), therefore the second argument indicates the dimension over which to apply the function

EXAMPLE

What if we feed a data frame to `apply()`?

```
> str(iris)
'data.frame': 150 obs. of  5 variables:
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species      : Factor w/ 3 levels "setosa","versicolor",...: 1 1 1 1 1 ...
 
> apply(iris, 2, mean)
Sepal.Length  Sepal.Width Petal.Length  Petal.Width       Species
                  NA          NA          NA          NA          NA
Warning messages:
1: In mean.default(newX[, i], ...) :
  argument is not numeric or logical: returning NA
...
5: In mean.default(newX[, i], ...) :
  argument is not numeric or logical: returning NA

> class(iris[1:4])
[1] "data.frame"
```

EXAMPLE [CONT'D]

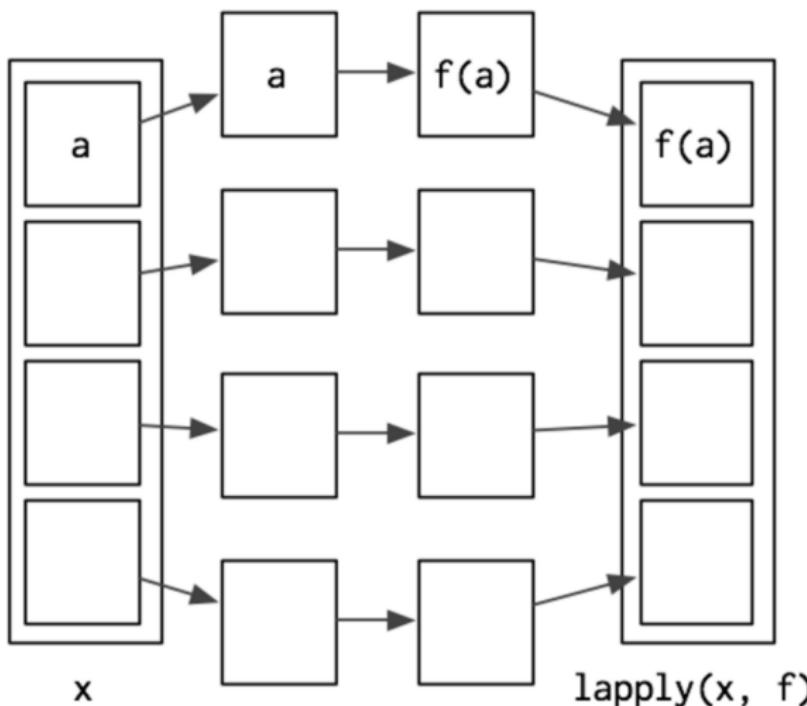
What if we feed a data frame to `apply()`?

```
> class(apply(iris[1:4], 2, mean))
[1] "numeric"

> apply(iris[1:4], 2, is.vector)
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
      TRUE        TRUE        TRUE        TRUE

> is.vector(apply(iris[1:4], 2, mean))
[1] TRUE
```

lapply()



How `lapply()` Works

`lapply()` is a wrapper for a common loop pattern

- It creates a container for output
- Applies the function `f()` to each element of a list
- Fills the container with the results
- Returns a list
- Use `unlist()` to convert the list to a vector

n.b. `lapply()` is particularly useful for working with data frames as data frames are lists

sapply() and vapply()

- Both operate similarly to `lapply()`, taking similar inputs, but they differ on output
- `sapply()` will guess at what type of output it should generate
 - `sapply()` is good for interactive coding as it minimizes typing and the coder is able to observe and rectify any unexpected output types
 - **do not** bury an `sapply()` in a function where it can generate an odd and difficult to trace error
- `vapply()` requires an additional argument, specifying the output type
 - More verbose than `sapply()`, it always generates consistent output based on argument specification, gives more informative error messages, and never fails silently, and is therefore more appropriate for use inside functions

Final Notes on Functionals

- For multiple varying arguments, use `Map()`
- Leveraging the fact that each iterations of `apply()` functionals is isolated from all others, this lends themselves well to parallelisation using `mclapply()` and `mcMap()` from the `parallel` package
- May also want to read up on the `purrr` package

LAB

Using state.x77

- 1 Use `lapply()` to return the correlation of each numeric variable with population. What type of output is returned?
- 2 Repeat with `sapply()`. What type of output is returned?
- 3 Find the sum of area by region (*hint*: search for an `apply()` that we have not yet used)

LAB

Using state.x77

- 1 Use `lapply()` to return the correlation of each numeric variable with population. What type of output is returned?

```
c <- state.x77
lapplyResult <- lapply(3:8, function(i) return(cor(s[,2], s[,i])))
str(lapplyResult)
```

- 2 Repeat with `sapply()`. What type of output is returned?

```
sapplyResult = sapply(3:8, function(i) return(cor(s[, 2], s[, i])))
str(sapplyResult)
```

- 3 Find the sum of area by region (*hint*: search for an `apply()` that we have not yet used)

```
tapply(s[, "Area"], state.region, sum)
```

Subsection 4

Evaluating the Performance of R Code

The Purpose of R

- R is not a fast computer language
- R is not meant nor designed to be a fast language
- R is designed to make data analysis and statistics easier for the user
- In this section, we will take a brief look at what makes R slow, and how you can systematically make code a little faster

How to Quantify Code Efficiency

- A precise way to measure the speed of small blocks of code is microbenchmarking
- R has a package called `microbenchmark` which provides a range of tools for evaluating code efficiency

```
> microbenchmark(sqrt(x), x^0.5, times = 1000)
Unit: microseconds
      expr     min      lq     mean   median      uq     max neval
sqrt(x) 3.959  4.2420  6.507298  6.607  7.3975  64.095  1000
x^0.5 24.730 26.7105 32.004310 28.484 35.3140 110.297  1000
```

- By default, `neval = 100`

Some Context on Code Efficiency

It is useful to think about how many times a function needs to run before it takes one second

| Microbenchmark | Interpretation |
|----------------|--------------------------------------|
| 1 ms | 1,000 calls takes one second |
| 1 μ s | 1,000,000 calls takes one second |
| 1 ns | 1,000,000,000 calls takes one second |

Practical Interpretation

It takes roughly 800 ns to compute the square root of 100 numbers using `sqrt()`. That means that if you repeated that operation a million times, i.e., compute the square root on 10,000,000 numbers, it would take 0.8 seconds.

system.time()

Wrapping code in `system.time()` will also give you the system time required to process code, but

- 1 `microbenchmark()` is far more precise
- 2 `system.time()` only runs the block of code once, therefore you need to manually wrap `system.time()` in a loop to generate meaningful statistics

```
> microbenchmark(sqrt(x), x^0.5, times = 1000)
Unit: microseconds
      expr    min     lq    mean   median     uq    max neval
sqrt(x) 3.834 3.971 6.823777 4.213 7.7275 639.492 1000
x^0.5 24.094 24.804 30.690782 26.232 31.9885 100.101 1000
>
> system.time(for (i in 1:1000) x^0.5) / 1000
    user  system elapsed
2.7e-05 1.0e-06 2.8e-05
```

Subsection 5

Character Functions and Manipulation in R

Character Functions

- R is not known for its prowess in dealing with textual analysis and natural language processing, but it does have some useful features and functions that give it some of the textual functionality of Python and Perl

length()

- Be sure to thoroughly understand the `length()` function, whose results can be unexpected or confusing (or both), without throwing any warning
- For vectors, matrices, arrays, factors, data frames and lists, `length()` returns the **number of elements**

```
> myStr_01 <- "abcdef"
> length(myStr_01)
[1] 1

> myStr_02 <- "abcdefghijklmnopqrstuvwxyz"
> length(myStr_02)
[1] 1

> myVec_01 <- 1:5
> length(myVec_01)
[1] 5
```

nchar()

- The vectorized `nchar()` function will return the number of characters in a character value

```
> myStr_01 <- "abcdef"  
> nchar(myStr_01)  
[1] 6
```

```
> myStr_02 <- "abcdefghijklmnopqrstuvwxyz"  
> nchar(myStr_02)  
[1] 26
```

```
> myVec_02 <- 12:8  
> nchar(myVec_02)  
[1] 2 2 2 1 1
```

- Note that `nchar()` coerces numeric values to characters

cat()

- The `cat()` function will combine character values and print them to the screen or to a file
- `cat()` coerces its arguments to character values, then concatenates and displays them

paste()

- The `paste()` function will accept an unlimited number of scalars, and join them together, separating each scalar with a space by default
- To use a character string other than a space as a separator, the `sep=` argument can be used

```
> x <- 99
```

```
> paste('My age is: ', 1 + x, "...boy am I old", sep="***")
[1] "My age is: ***100***...boy am I old"
```

paste() [CONT'D]

- If you pass a character **vector** to **paste()**, the **collapse=** argument can be used to specify a character string to place between each element of the vector

n.b. Using **sep=** has no effect when exclusively passing a character vector to **paste()**

```
> paste(c("abc", "abcdef", "ZzZzZZzZzZ"))
[1] "abc"         "abcdef"       "ZzZzZZzZzZ"

> paste(c("abc", "abcdef", "ZzZzZZzZzZ"), sep = "")
[1] "abc"         "abcdef"       "ZzZzZZzZzZ"

> paste(c("abc", "abcdef", "ZzZzZZzZzZ"), collapse = "") 
[1] "abcabcdefZzZzZZzZzZ"

> paste(c("abc", "abcdef", "ZzZzZZzZzZ"), collapse = " ")
[1] "abc abcdef ZzZzZZzZzZ"
```

paste() [CONT'D]

- When multiple arguments are passed to `paste()`, it will vectorize the operations, recycling shorter elements when necessary

```
> paste("x", 1:5, sep = "_")
[1] "x_1" "x_2" "x_3" "x_4" "x_5"
```

- Including `collapse=` collapses individual elements into a single string

```
> paste("x", 1:5, sep = "_", collapse = "")
[1] "x_1x_2x_3x_4x_5"
```

```
> paste("x", 1:5, sep = "_", collapse = " ")
[1] "x_1 x_2 x_3 x_4 x_5"
```

substring()

- The `substring()` function can be used either to extract parts of character strings, or to change the values of part of character strings
- `substring()` accepts the arguments `first=` and `last=`, identifying the location of the first and last character (with an integer), respectively, in the string
- The `first=` is required, but `last=` may be omitted
- `substring()` coerces inputs to characters

```
> (myStr_03 <- paste(LETTERS[1:8], letters[1:8], sep = "", collapse = ""))
[1] "AaBbCcDdEeFfGgHh"

> substring(myStr_03, 6, 12)
[1] "cDdEeFf"

> myNum <- 123456789
> substring(myNum, 3)
[1] "3456789"
```

substring() [CONT'D]

- **substring()** is vectorized

- for **first=** and **last=** arguments

```
> (myStr_04 <- c("paul", "john", "sally"))
[1] "paul"  "john"  "sally"
```

```
> substring(myStr_04, 3, 4)
[1] "ul"   "hn"   "ll"
```

- for character vectors passed to **substring()**

```
> (myStr_05 <- 'my tiny bed')
[1] "my tiny bed"
```

```
> substring(myStr_05, first = c(1, 4, 9), last = c(2, 7, 11))
[1] "my"    "tiny"  "bed"
```

substring() [CONT'D]

- `substring()` may also be used for assignment

```
> myStr_06 <- "my big dog"

> substring(myStr_06, 8, 10) <- "cat"

> myStr_06
[1] "my big cat"

> substring(myStr_06, 4, 6) <- "gigantic"

> myStr_06
[1] "my gig cat"
```

- There is also a function `substr()` which operates similarly to `substring()`, but the latter is more robust

Regular Expressions in R

- Regular expressions are a method of expressing patterns in character values which can then be used to extract parts of strings or to modify those strings in some way
- The most common functions that support working with regular expressions in R are `strsplit()`, `grep()`, `grepl()`, `sub()`, `gsub()`, `regexpr()` and `gregexpr()`
- The backslash character (\) is used in regular expressions to signal that certain characters with special meaning in regular expressions should be treated as normal characters
- In R, this means that two backslash characters need to be entered into an input string anywhere that special characters need to be escaped
- Although the double backslash will display when the string is printed, the use of `nchar()` or `cat()` will confirm that only a single backslash is actually included in the string

A Brief Introduction to Regular Expressions

- Regular expression are composed of three components
 - literal characters, which are matched by a single character
 - character classes, which can be matched by any number of characters
 - modifiers, which operate on literal characters or character classes
- As many punctuation marks are regular expression modifiers, the following characters must always be preceded by a backslash to retain their literal meaning

. ^ \$ + ? * () [] { } | \

Forming Character Classes

- To form a character class, use square brackets ([]) surrounding the characters to be matched

E.g. to create a character class that will be matched either by x, y, z or the number 1, use [xyz1]

- Dashes are used inside of character classes to represent a range of values, e.g., [a-z] or [2-9]
- If a dash is to be literally included in a character class, it should be either the first character in the class or it should be preceded by a backslash
- Other special characters, save square brackets, do not need to be preceded by a backslash when used in a character class

Modifiers for Regular Expressions

| Modifier | Meaning |
|----------|---|
| ^ | anchors expression to the beginning of target |
| \$ | anchors expression to end of target |
| . | matches any single character except newline |
| | matches any single character except newline |
| () | groups patterns together |
| * | matches 0 or more occurrences of preceding entity |
| ? | matches 0 or 1 occurrence of preceding entity |
| + | matches 1 or more occurrences of preceding entity |
| {n} | matches exactly n occurrences of preceding entity |
| {n, } | matches n or more occurrences of preceding entity |
| {n, m} | matches between n and m occurrences of preceding entity |

Modifiers for Regular Expressions [EXAMPLE]

- Modifiers operate on whatever entity then follow, using parentheses for grouping if necessary

E.g. To identify a string with two digits, followed by one or more letters, the matching regular expression would be

'[0-9] [0-9] [a-zA-Z]+'
[0-9] [0-9] [a-zA-Z]+

E.g. For two consecutive appearances of the string 'photo' the matching regular expression could be
'(photo){2}'

Modifiers for Regular Expressions [EXAMPLE] [CONT'D]

- E.g. For jpg filename consisting exclusively of letters, the matching regular expression could be

'^ [a-zA-Z]+\\.jpg\$'

- Observe how this regular expression is constructed
 - ^ explicitly states that the file must **begin** with whatever proceeds it, in this case, [a-zA-Z]
 - + allows for any number of letters, so long as there is at least one
 - The second backslash, i.e., the backslash on the right, is required so that . can retain its literal meaning (recall . is a regular expression modifier)
 - The first backslash, i.e., the backslash on the left, is a required R quirk
 - \$ ensures that the **final** four characters in the file name are .jpg

strsplit()

- The `strsplit()` function can use a character string or regular expression to divide up a character string into smaller pieces
- `strsplit()` returns its results in a list, regardless of input
- To break up a sentence into its constituent words

```
> myString_07 <- "I enjoy reading books"

> strsplit(myString_07, "")  
[[1]]  
[1] "I" " " "e" "n" "j" "o" "y" " " "r" "e" "a" "d" "i" "n" "g"  
[16] " " "b" "o" "o" "k" "s"

> strsplit(myString_07, " ")  
[[1]]  
[1] "I"       "enjoy"    "reading"  "books"
```

strsplit() with Regular Expressions

- Given `strsplit()` accepts regular expressions to determine where to split a string, the function is very versatile

E.g. It commonly occurs that when customers input free-form text on surveys, they may accidentally include more than once space between words, thereby requiring a regular expression to appropriately handle the variable inputs

```
> myString_08 <- "I      enjoy reading    books"  
  
> strsplit(myString_08, " ")  
[[1]]  
[1] "I"      ""      ""      ""      ""      "enjoy"  
[7] "reading" ""      ""      "books"  
  
> strsplit(myString_08, " +")  
[[1]]  
[1] "I"      "enjoy"  "reading" "books"
```

Using Regular Expression in R

- Sufficiently versed in the construction of regular expressions, those expressions can now be implemented for use with specific functions
- `grep()` and `grepl()` are used to test for the presence of a regular expression
- `regexpr()` and `gregexpr()` can pinpoint and potentially extract those parts of a string that were matched by a regular expression

grep()

- `grep()` accepts a regular expression and a character string or vector of character strings, and returns the **indices** of those elements of the string which are matched by the regular expression
- If the `value = TRUE` argument is passed to `grep()`, it will return the actual strings which matched the expression instead of the indices
- To match literal strings (instead of regular expressions), the `fixed = TRUE` argument should be passed to `grep()`

```
> firstNames <-read.csv("~/Desktop/firstNames.csv", stringsAsFactors = F)
```

```
> grep('^Ai', firstNames$firstname)
[1] 54 55 56 57 58 59 60 61
```

```
> grep('^Ai', firstNames$firstname, value = T)
[1] "Ai"      "Aida"    "Aide"    "Aiko"    "Aileen"   "Ailene"
[7] "Aimee"   "Aisha"
```

grep() [CONT'D]

- To find regular expressions regardless of case, use `ignore.case = TRUE`
- To search for a regular expression that is a single word, e.g., `shoes`, which is not preceded nor proceeded by any other characters except for punctuation, whitespace or the beginning or ending of a line, wrap the string with escaped angled brackets (`\\\<` and `\\\>`)

```
> myStr_09 <- c("run to the store", "running", "run...quickly!", "run!")
```

```
> grep('\\<run\\>', myStr_09, value = T)
[1] "run to the store" "run...quickly!"      "run!"
```

- If the regular expression passed to `grep()` is not matched, `grep()` returns an empty numeric vector, which can be easily evaluated to be `TRUE` or `FALSE` using the `any()` function

grep() [CONT'D]

- To find regular expressions regardless of case, use `ignore.case = TRUE`
- To search for a regular expression that is a single word, e.g., `shoes`, which is not preceded nor proceeded by any other characters except for punctuation, whitespace or the beginning or ending of a line, wrap the string with escaped angled brackets (`\\\<` and `\\\>`)

```
> myStr_09 <- c("run to the store", "running", "run...quickly!", "run!")
```

```
> grep('\\<run\\>', myStr_09, value = T)
[1] "run to the store" "run...quickly!"      "run!"
```

- If the regular expression passed to `grep()` is not matched, `grep()` returns an empty numeric vector, which can be easily evaluated to be `TRUE` or `FALSE` using the `any()` function

`grepl()` will return a logical vector the length of the input vector, identifying the elements which matched the regular expression

regexpr()

- `regexpr()` pinpoints and can extract those parts of a string that are matched by a regular expression
- `regexpr()` outputs a vector of *starting positions* of the regular expressions found; if none are found, -1 is returned
- `match.length` provides information about the length of each match
- `regexpr()` will only provide information on the **first** match in a given input string

```
> myStr_10 <- c("94112 94117", "H8P 2S5", " 90210", "47907-1233")  
  
> regexpr('[0-9]{5}', myStr_10)  
[1] 1 -1 2 1  
attr(),"match.length")  
[1] 5 -1 5 5  
attr(),"useBytes")  
[1] TRUE
```

gregexpr()

- `gregexpr()` operates similarly to `regexpr()`, but returns information on **all** matches found (not just the first)
- `gregexpr()` always returns its result in the form of a list

```
> myStr_10 <- c("94112 94117", "H8P 2S5", " 90210", "47907-1233")
```

```
> gregexpr('[0-9]{5}', myStr_10)
[[1]]                               [[3]]
[1] 1 7                             [1] 2
attr(),"match.length")           attr(),"match.length")
[1] 5 5                           [1] 5
attr(),"useBytes")                 attr(),"useBytes")
[1] TRUE                          [1] TRUE

[[2]]                               [[4]]
[1] -1                            [1] 1
attr(),"match.length")           attr(),"match.length")
[1] -1                           [1] 5
attr(),"useBytes")                 attr(),"useBytes")
[1] TRUE                          [1] TRUE
```

Substitutions

- To substitute text based on regular expressions, R provides two functions, `sub` and `gsub`
- Both functions accept
 - 1 a regular expression
 - 2 a string containing what will be substituted for the regular expression
 - 3 a string (or strings) to operate on
- Like `regexpr()` and `gregexpr()`, `sub` only changes **only** the first occurrence of the regular expression, whereas `gsub` changes all occurrences

gsub

- A common application of `gsub` is the scrubbing of financial data

```
> financialData_01 <- c("$11,345.65", "$99,125.22", "$13,321.99")  
  
> str(financialData_01)  
chr [1:3] "$11,345.65" "$99,125.22" "$13,321.99"  
  
> as.numeric(financialData_01)  
[1] NA NA NA  
Warning message:  
NAs introduced by coercion  
  
> sub('[,$,]', '', financialData_01)  
[1] "11,345.65" "99,125.22" "13,321.99"  
  
> gsub('[,$,]', '', financialData_01)  
[1] "11345.65" "99125.22" "13321.99"
```

- The `gsub` function replaces all instances of \$ and , with nothing (''), effectively deleting them from the string

LAB

Using tweets.csv

- 1 Identify all tweets with the word 'flight' in them
- 2 How many tweets end in a question mark?
- 3 How many tweets have airport codes in them (assume any three subsequent capital letters are airport codes)
- 4 Identify all tweets with URLs in them
- 5 Replace all instances of repeated exclamation points with a single exclamation point
- 6 Replace consecutive exclamation points, question marks, and periods with a single period, split the tweet on periods, and create a list where each element is a vector of the split strings from each tweet

Section 4

Exploratory Data Analysis in R

Section Contents

4 Exploratory Data Analysis in R

- Introduction to Exploratory Data Analysis in R
- Introduction to ggplot2

Subsection 1

Introduction to Exploratory Data Analysis in R

summary()

- `summary()` is a generic function used to produce result summaries of the results of various model fitting functions; the function invokes particular methods which depend on the class of the first argument

```
> summary(mtcars[1:3])
      mpg              cyl              disp
Min.   :10.40   Min.   :4.000   Min.   : 71.1
1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8
Median  :19.20   Median  :6.000   Median  :196.3
Mean    :20.09   Mean    :6.188   Mean    :230.7
3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0
Max.   :33.90   Max.   :8.000   Max.   :472.0
```

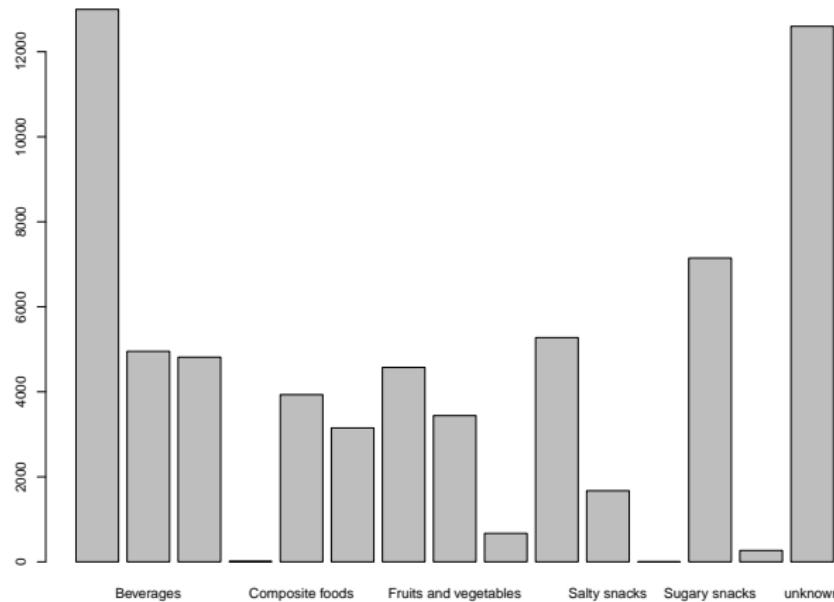
- `summary()` of an `lm` object will be different

Graphics in Base R

- The base R graphics package, while being quick to code and functional, is not particularly aesthetically pleasing, is syntactically cumbersome, and is outshone by what can be achieved using the `ggplot2` package
- While I would not recommend generating graphical output for external consumption using the base R graphics package, it does offer a quick and dirty way to graphically examine data

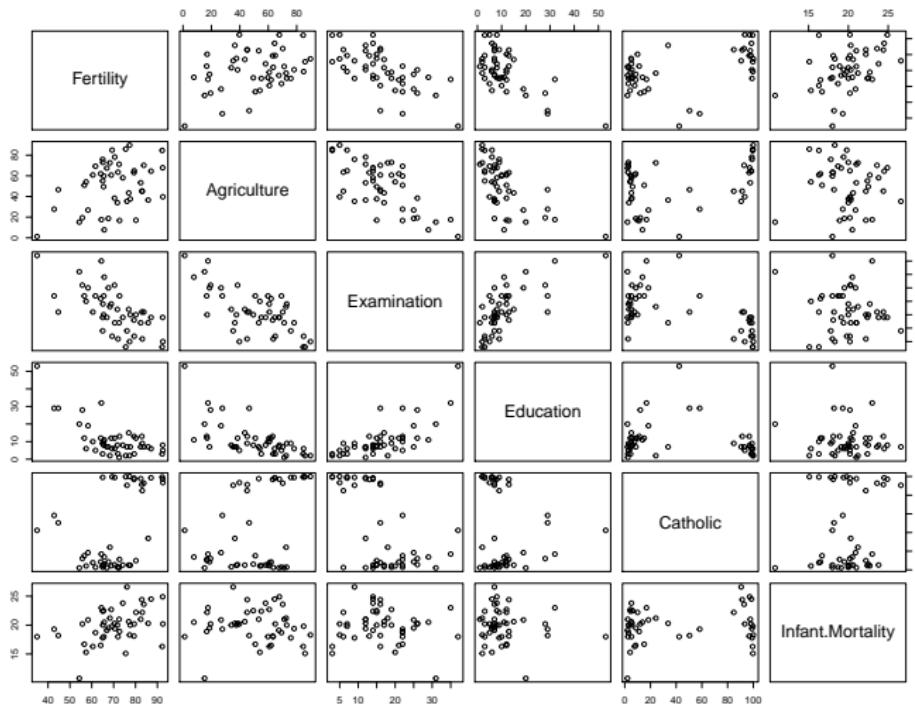
plot() (bar graph)

```
> plot(food$pnnns_groups_1) ## World Wide Food Facts Data (Kaggle)
```



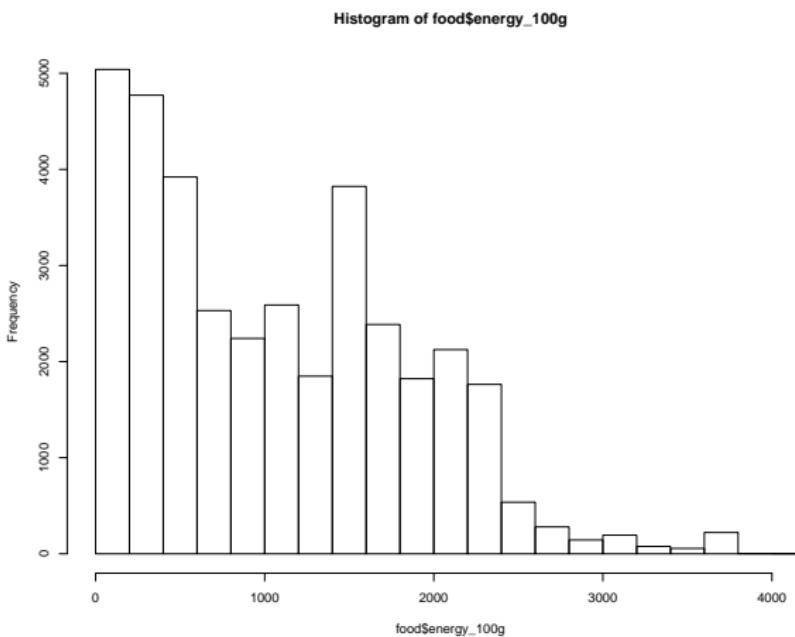
plot() (scatter plot matrix)

```
> plot(swiss) ## Base R Data
```



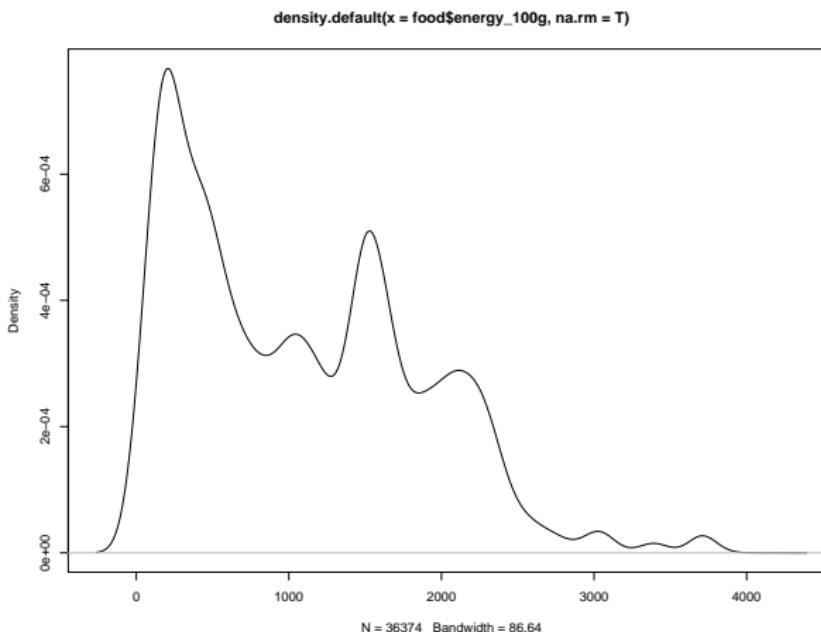
hist()

```
> hist(food$energy_100g) ## World Wide Food Facts Data (Kaggle)
```



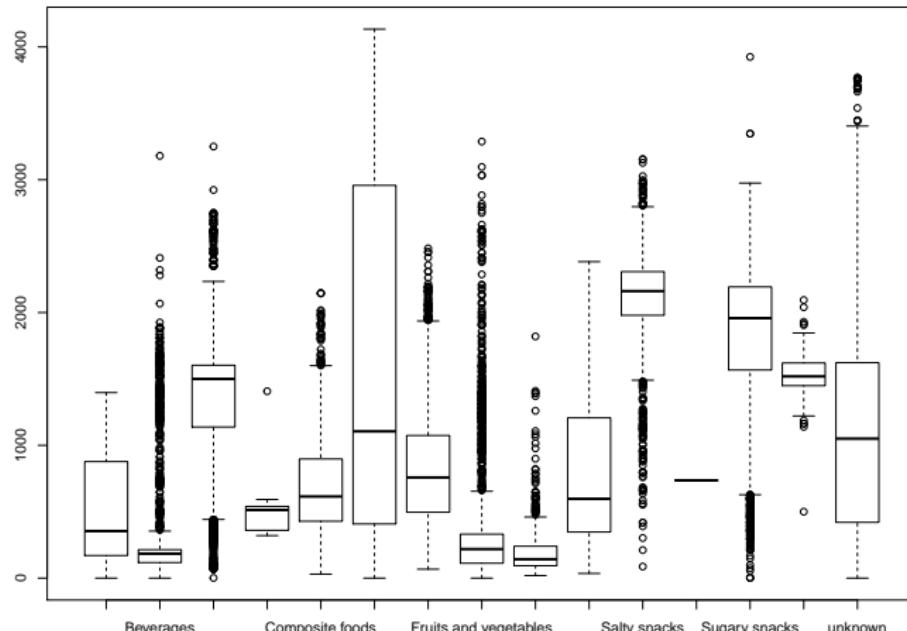
plot() (kernel density)

```
> plot(density(food$energy_100g, na.rm = T))
```



boxplot()

```
> with(food, boxplot(energy_100g ~ pnns_groups_1))
```

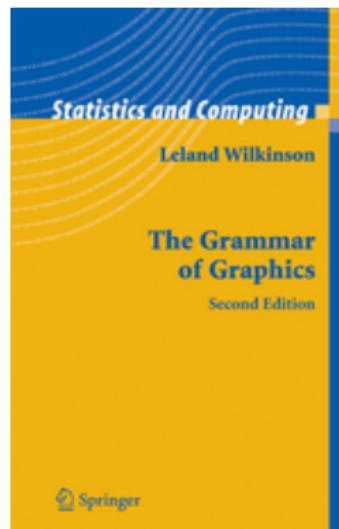


Subsection 2

Introduction to ggplot2

What is ggplot2

- **ggplot2** is an R package for producing graphics
- **ggplot2** differentiates itself from other packages as it is based on a deep underlying grammar, adopted from Wilkinson's *The Grammar of Graphics*



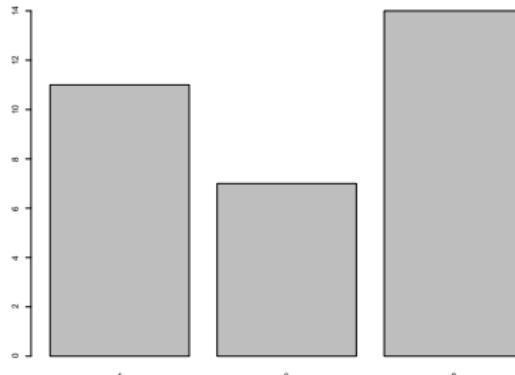
What is ggplot2 [CONT'D]

- This grammar of graphics is composed of a set on independent components that can be composed in many different ways
- This makes What is `ggplot2` very powerful, because you are not limited to a set of pre-specified graphics, but you can create new graphics that are precisely tailored to your problem
- Do not be fooled: even though `ggplot2` has a high level of flexibility and complexity of, publication-quality graphs can be coded in seconds, with many of the extraneous details (e.g., legends) taken care of by `ggplot2` default behavior

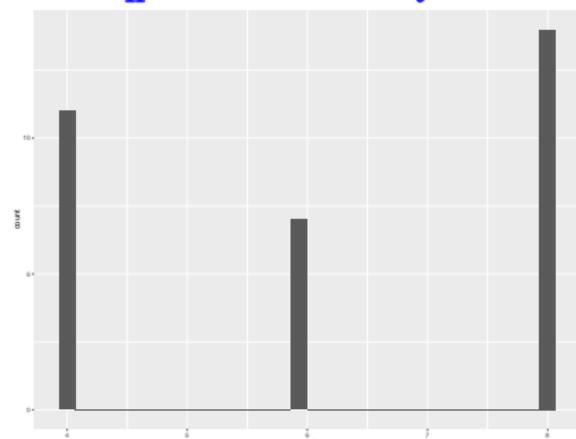
qplot() versus plot()

- `plot()` is the base R graphics plotting package
- `qplot()` is a function from the `ggplot2` package meant to mimic and improve upon the simplicity and speed of plotting in base R

```
plot(as.factor(mtcars$cyl))
```



```
qplot(mtcars$cyl)
```

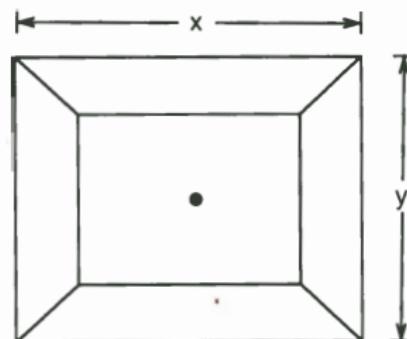
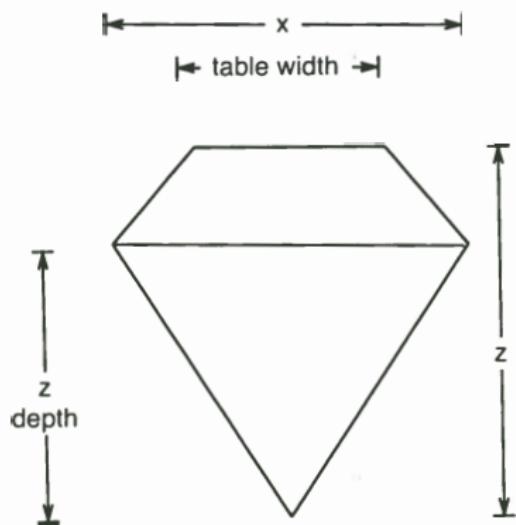


diamonds dataset

A data frame with 53940 rows and 10 variables:

- price: price in US dollars (\$326–\$18,823)
- carat: weight of the diamond (0.2–5.01)
- cut: quality of the cut (Fair, Good, Very Good, Premium, Ideal)
- color: diamond colour, from J (worst) to D (best)
- clarity: a measurement of how clear the diamond is (I1 (worst), SI1, SI2, VS1, VS2, VVS1, VVS2, IF (best))
- x: length in mm (0–10.74)
- y: width in mm (0–58.9)
- z: depth in mm (0–31.8)
- depth: total depth percentage = $z / \text{mean}(x, y) = 2 * z / (x + y)$ (43–79)
- table: width of top of diamond relative to widest point (43–95)

diamonds dataset [CONT'D]

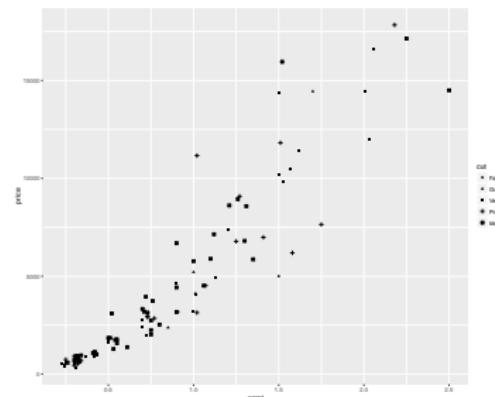
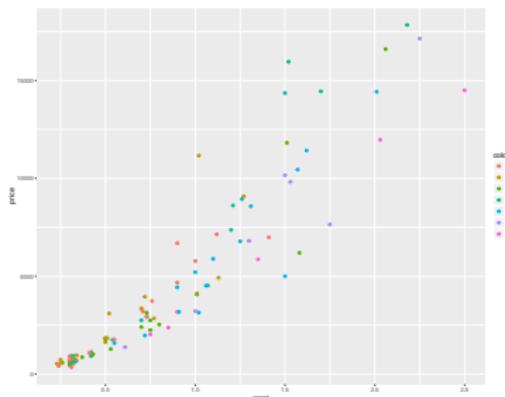


$$\text{depth} = z \text{ depth} / z * 100$$
$$\text{table} = \text{table width} / x * 100$$

Aesthetic Attributes

- The first major difference when using `qplot()` instead of base graphing in R is the when assigning colors, sizes, shapes, etc.
- Use a subset of the diamonds data for ease of exposition

```
> set.seed(1410)
> diamonds_subset_01 <- diamonds[sample(nrow(diamonds), 100), ]
> qplot(carat, price, data = diamonds_subset_01, colour = color)
> qplot(carat, price, data = diamonds_subset_01, shape = cut)
```



Aesthetic Attributes & Scales

- Color, size and shape are all examples of aesthetic attributes, visual properties that affect the way observations are displayed
- For every aesthetic attribute, there is a function, called a **scale**, which maps data values to valid values for that aesthetic
- It is the scale that controls the appearance of the points and associated legend
- In the example on the previous slide, it is the color **scale** that maps J to **purple** and F to **green**
- Other common aesthetics include alpha to control for point opacity

Geometric Objects

- Geometric objects, or geoms as they are commonly known and used, describe the type of object used to display the data
- Common one-dimensional geoms include

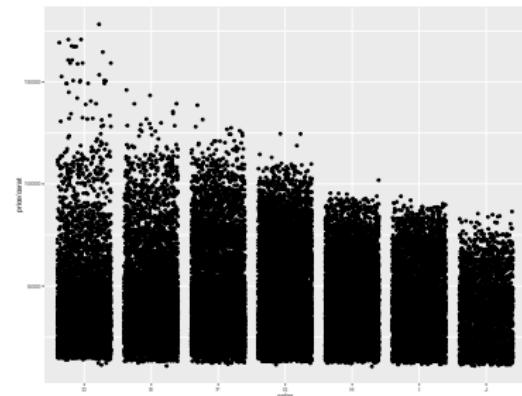
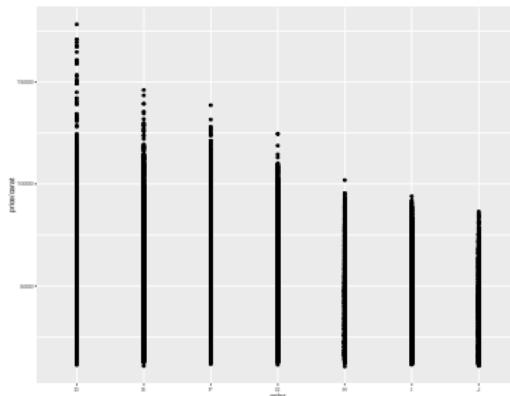
- 1 geom = "histogram"
- 2 geom = "freqpoly"
- 3 geom = "density"
- 4 geom = "bar"

- Common two-dimensional geoms include

- 1 geom = "point"
- 2 geom = "boxplot"
- 3 geom = "path" (line in any direction)
- 4 geom = "line" (line from left to right)
- 5 geom = "smooth" (smoothed line with standard error)

geom = "jitter"

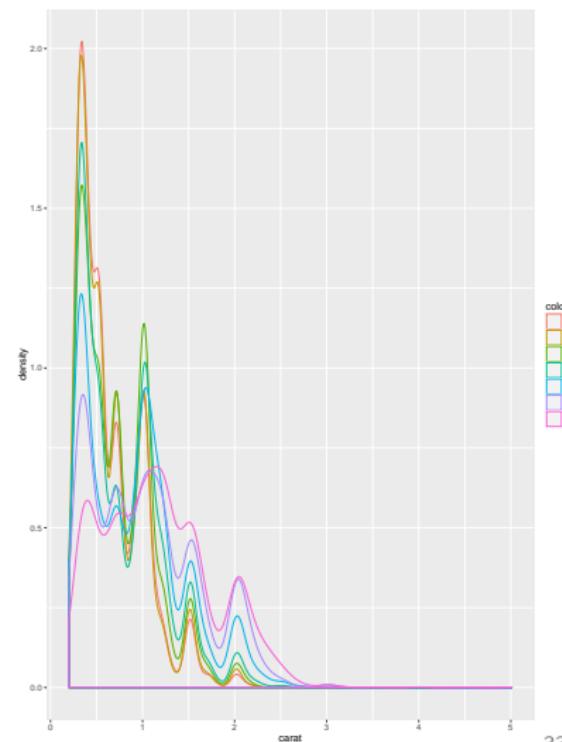
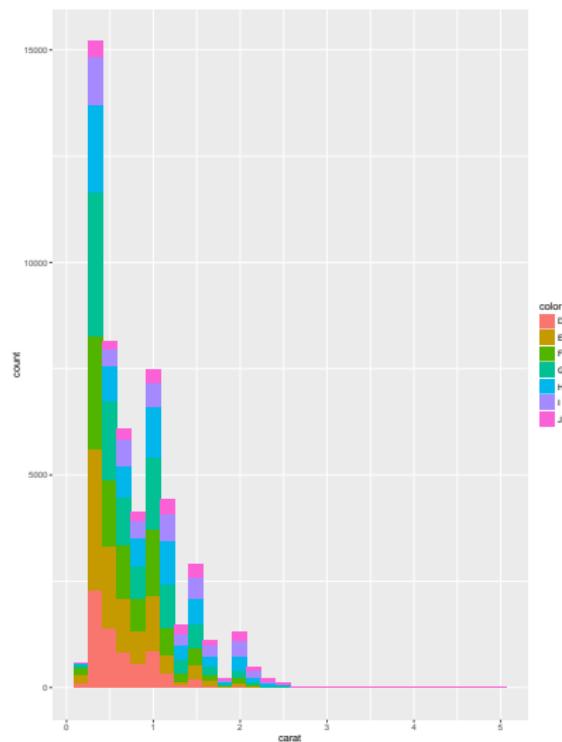
- When plotting a significant number of points with a continuous response variable (y) and a categorical predictor (x), plots can often suffer from overplotting, i.e., multiple superimposed points
- geom = "jitter"** spreads points in an effort get a better sense of how many points exist at any given response level



`geom = "histogram" & geom = "density"`

- With histograms, be sure to play with the `binwidth` argument to select the bin size that best communicates the structure of the data
- Similarly, iterate through various values of `adjust` for density plots to find the right smoothing factor (larger values of `adjust` generate smoother lines)
- Mapping a *categorical* variable to an aesthetic will automatically split the geom by that variable (see proceeding slide)

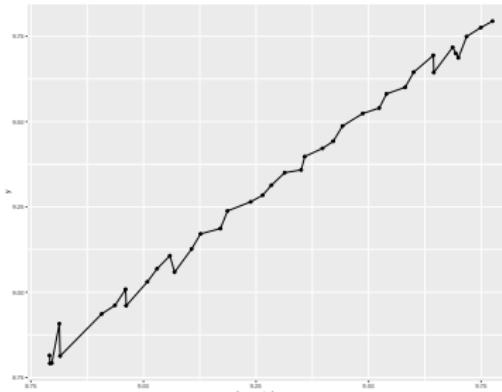
geom = "histogram" & geom = "density" [CONT'D]



Using Multiple geoms

- Time series graphs are often nicer not just with a line connecting adjacent data, but with points representing the data as well
- Multiple geoms can be employed in a single `qplot()` by creating a vector of geoms, e.g., `geom = c("line", "point")`

```
qplot(lag.quarterly.revenue, y, data = freeny, geom = c("line", "point"))
```

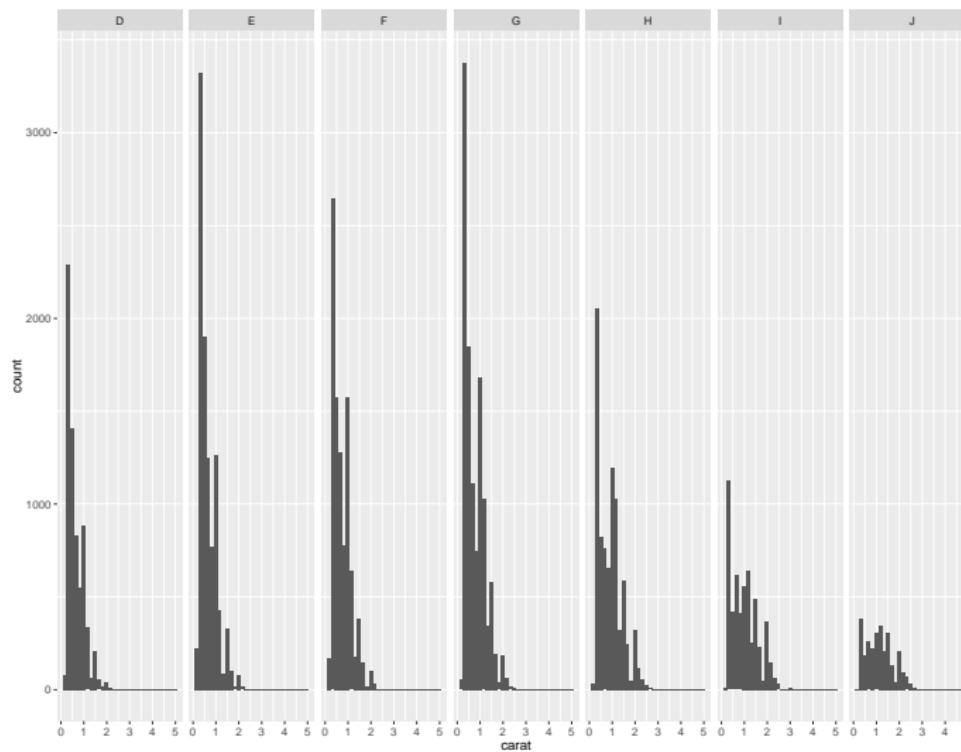


Faceting

- Although aesthetics may be employed to compare subgroups, all groups are drawn on the same plot
- Faceting takes an alternative approach, creating tables of graphics by splitting data into subsets, and displaying sub-graphs (typically) in a grid arrangement
- Use the `facet = <rowVar> ~ <colVar>` option—where both `rowVar` and `colVar` should be categorical variables
- Use `.` notation as an optional placeholder
- The following code generates the graph on the proceeding slide

```
> qplot(carat, data = diamonds, facets = . ~ color, geom = "histogram")
```

Faceting [CONT'D]



qplot() versus ggplot()

- Plots can be created in two ways:
 - 1 all at one with `qplot()`
 - 2 piece-by-piece using `ggplot()` and layer functions, which provide far more control and complexity over a graph
- `ggplot()` allows us to build a plot layer by layer, where each layer can come from a different data sets and have a different aesthetic mappings
- This advanced behavior differs from `qplot()` which permits only a single data set and a single set of aesthetic mappings

Creating a Plot

- Plots can be created in two ways:
 - 1 all at one with `qplot()`
 - 2 piece-by-piece using `ggplot()` and layer functions, which provide far more control and complexity over a graph
- `ggplot()` allows us to build a plot layer by layer, where each layer can come from a different data sets and have a different aesthetic mappings
- This advanced behavior differs from `qplot()` which permits only a single data set and a single set of aesthetic mappings

From qplot() to ggplot()

- Using `qplot()` we can create a graph, e.g., a scatter plot, with carat on the x-axis, price on the y-axis, with cut mapped to color using the following code

```
> qplot(carat, price, data = diamonds, colour = cut)
```

- With `ggplot()`, this are additionally complicated
- All aesthetics in `ggplot()` are wrapped in the `aes()` function (short for aesthetics)
- The `ggplot()` code that generates a *plot object*—**not** the plot itself—is

```
> ggplot(diamonds, aes(carat, price, colour = cut))
```

- n.b.** This line of code will generate an empty plot because there are no geoms, i.e., layers

Layers

- A minimal layer may do nothing more than specify a geom
- By adding a geom to the code on the previous slide, we can fill the empty graph with data

```
> ggplot(diamonds, aes(carat, price, colour = cut)) + geom_point()

# note that a ggplot plot object can be stored in a variable
# the following code generate identical results

> myPlot <- ggplot(diamonds, aes(carat, price, colour = cut))

> (myPlot <- myPlot + geom_point())
```

- Layers allow for writing clean, concise and reusable code, e.g., to include a thick, blue line of best fit w/o standard errors, can create the generic layer, and then add it to any plot which for which it is appropriate

```
> ggplot(diamonds, aes(carat, price, colour = cut)) + geom_point()

# note that a ggplot plot object can be stored in a variable
# the following code generate identical results
```

Layers [CONT'D]

- Layers allow for writing clean, concise and reusable code

E.g. To include a thick, blue line of best fit w/o standard errors and with 50% opacity, create the generic layer, and then add it to any plot which for which it is appropriate

```
> myBestFitLine <- geom_smooth(method = "lm", se = F,  
                                colour = alpha("steelblue", 0.5), size = 2)  
  
> qplot(sleep_rem, sleep_total, data = msleep) + myBestFitLine  
  
> qplot(awake, brainwt, data = msleep) + myBestFitLine  
  
> qplot(bodywt, brainwt, data = msleep, log= "xy") + myBestFitLine
```

The Plot Object as a Variable

- When storing a plot object in a variable, you can inspect the structure of the plot without actually plotting it

```
> myPlot <- ggplot(diamonds, aes(carat, price, colour = cut)) +  
    geom_point()  
  
> summary(myPlot)  
data: carat, cut, color, clarity, depth, table, price, x, y, z  
[53940x10]  
mapping: x = carat, y = price, colour = cut  
faceting: facet_null()  
-----  
geom_point: na.rm = FALSE  
stat_identity: na.rm = FALSE  
position_identity
```

Data

- Input data for `ggplot()` **must be a data frame**
- Do not reference data that is not passed to `ggplot()` as the default data frame, as this makes it impossible to encapsulate all of the data needed for plotting in a single object

```
> ggplot(mtcars, aes(x = cyl, y = diamonds$cut[1:32])) + geom_point()
```

- You could make it work (as the above does), but it is bad form and, moreover, it defeats the purpose, strength and flexibility of `ggplot()`

aes()

- To map variables to different parts of the plot, we use the `aes()` function, which takes a list of aesthetic-variable pairs

E.g. `aes(x = weight, y = height, colour = age)` maps weight to `x`, height to `y` and `colour` to `age`

- `x =` and `y =` can be dropped

n.b. Any variables in an `aes()` specification **must** be contained inside the plot or layer data

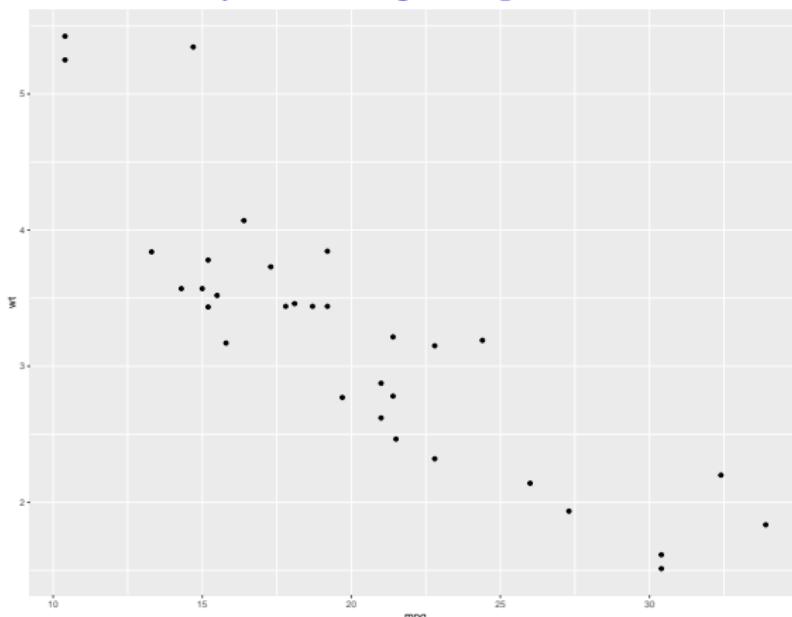
- Default aesthetic mappings can either be set when the plot is initialized or modified at a later time

```
> (myPlot <- ggplot(mtcars, aes(x = mpg, y = wt)) + geom_point())  
  
> myPlot + geom_point(aes(colour = factor(cyl)))  
  
> myPlot + geom_point(aes(y = disp))
```

- Aesthetic mappings specified in a layer affect only that layer
- For that reason, unless you modify the default scales, axis labels and legend titles will be based on plot defaults (see

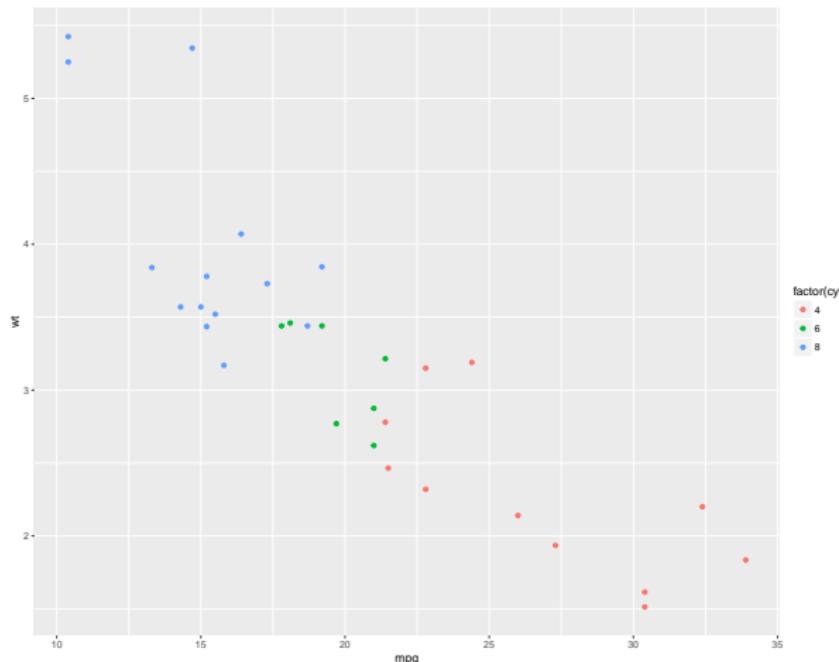
Updating Aesthetics on Layers [EXAMPLE 1/3]

```
> myPlot <- ggplot(mtcars, aes(x = mpg, y = wt))  
      > myPlot + geom_point()
```



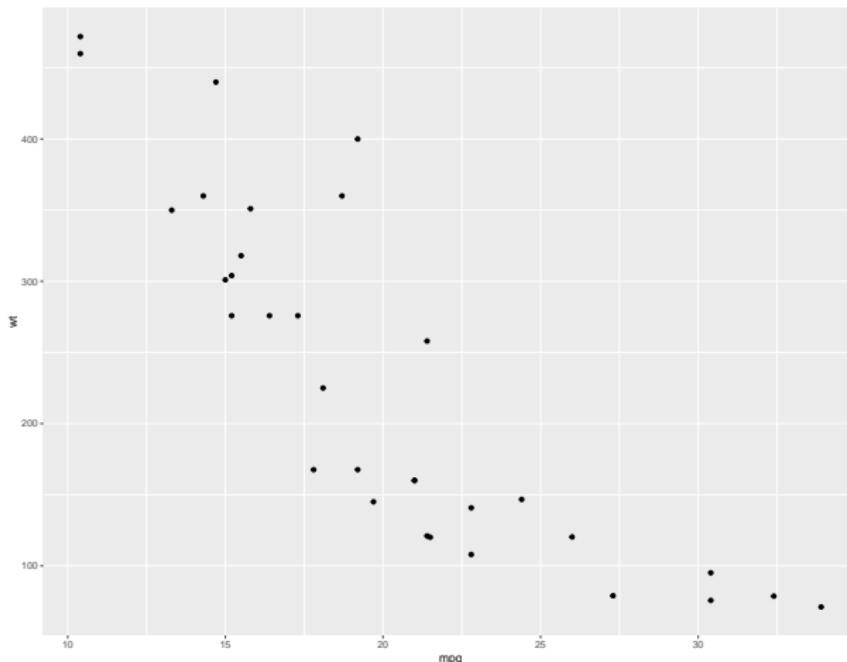
Updating Aesthetics on Layers [EXAMPLE 2/3]

```
> myPlot + geom_point(aes(colour = factor(cyl)))
```



Updating Aesthetics on Layers [EXAMPLE 3/3]

```
> myPlot + geom_point(aes(y = disp))
```



Grouping

- geoms can be roughly divided into two groups: individual and collective
- An individual geom has a distinctive graphical object for each observations in a data frame, e.g., `geom_point()` has single point for each observation
- Collective geoms represent multiple observations, the result of a statistical summary or just fundamental to the display of a particular geom, e.g., polygons
- The `group` aesthetic controls which observations go into which individual graphical element

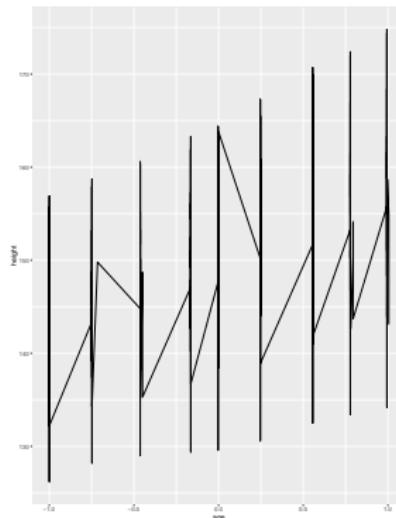
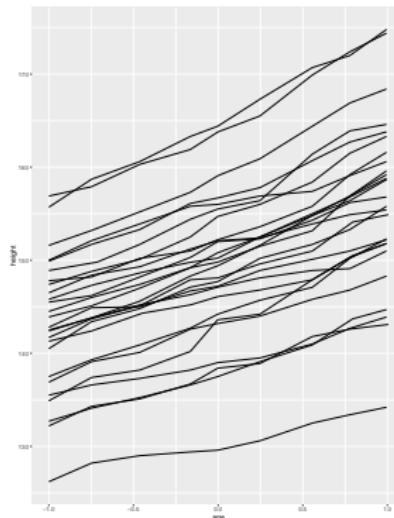
Grouping [CONT'D]

- By default, the `group` is set to the interaction of all discrete variables in the plot, which typically generates the expected results
- In the event it does not (or when no discrete variables are used in the plot), the `group` can be mapped to a variable that has a different value for each group
- `interaction()` is useful if a single pre-existing variable doesn't cleanly separate groups, but a combination does
- Grouping examples on the following slides will use longitudinal data from `nlme` package called `Oxboys`, which records height, age and subject (26 boys) measured at nine occasions

Multiple Groups / One Aesthetic

- Objective: separate the data into groups in an effort to distinguish between individual subjects, but render all of them in the same way

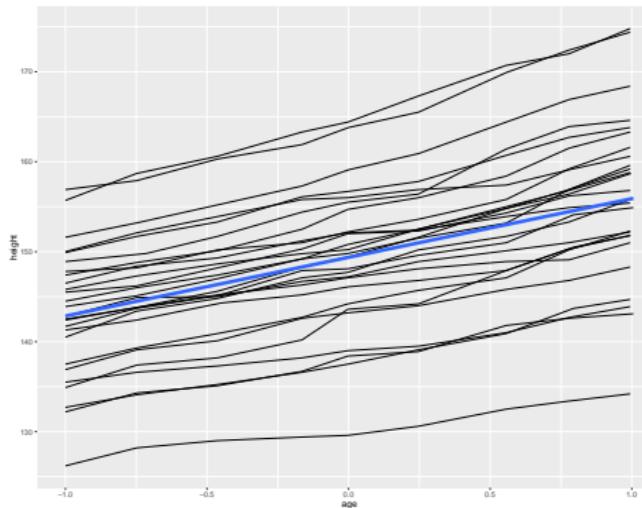
LEFT `myPlot <- ggplot(Oxboys, aes(age, height, group = Subject)) + geom_line()`
RIGHT `myPlot <- ggplot(Oxboys, aes(age, height)) + geom_line()`



Different Groups on Different Layers

- Objective: plot data based on different layers of aggregation
- In this example, a second layer is created whereby a line of best-fit is overlaid using **all** data, hence **group = 1**

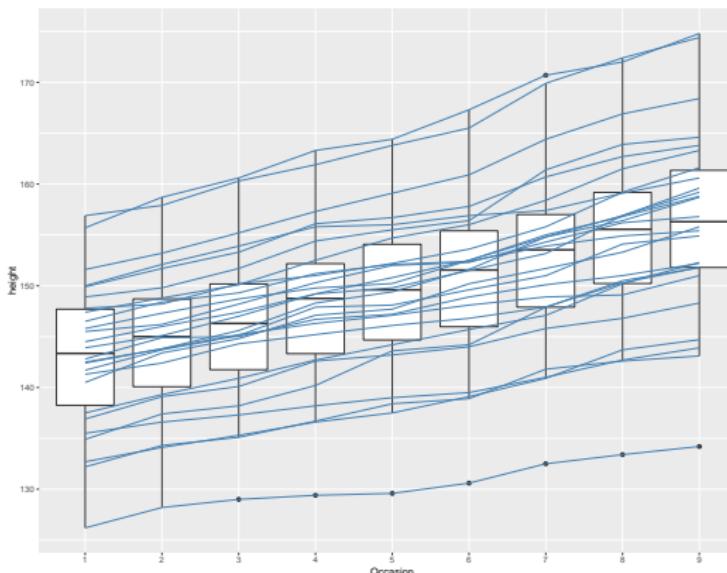
```
myPlot <- ggplot(Boys, aes(age, height, group = Subject)) + geom_line()  
myPlot + geom_smooth(aes(group = 1), method = "lm", size = 2, se = F)
```



Overriding Default Groupings

- Objective: a plot has a discrete scale, but there is a desire to draw lines that connect across groups (think interaction plots)

```
myPlot <- ggplot(Oxboys, aes(Occasion, height)) + geom_boxplot()  
myPlot + geom_line(aes(group = Subject), colour = "steelblue")
```



More on geoms

- Geometric objects, or **geoms**, perform the actual rendering of the layer
- Each **geom**s has a set of aesthetics it *can* determine without direction, and a set that are **required** for drawing
 - `geom_point()` **requires** both an x and y coordinate, but will automatically process color, size and shape aesthetics
 - `geom_bar()` **requires** a height `ymin`, but will automatically process width, border color and fill color
- Every **geom** has a default statistic and every statistic a default **geom**
E.g. The bin statistic automatically defaults to using the bar **geom** to create a histogram

ggplot2 geoms [Wickham, 2009]

| Name | Description |
|------------|---|
| abline | Line, specified by slope and intercept |
| area | Area plots |
| bar | Bars, rectangles with bases on y-axis |
| blank | Blank, draws nothing |
| boxplot | Box-and-whisker plot |
| contour | Display contours of a 3d surface in 2d |
| crossbar | Hollow bar with middle indicated by horizontal line |
| density | Display a smooth density estimate |
| density_2d | Contours from a 2d density estimate |
| errorbar | Error bars |
| histogram | Histogram |
| hline | Line, horizontal |
| interval | Base for all interval (range) geoms |
| jitter | Points, jittered to reduce overplotting |
| line | Connect observations, in order of x value |
| linerule | An interval represented by a vertical line |
| path | Connect observations, in original order |
| point | Points, as for a scatterplot |
| pointrange | An interval represented by a vertical line, with a point in the middle |
| polygon | Polygon, a filled path |
| quantile | Add quantile lines from a quantile regression |
| ribbon | Ribbons, y range with continuous x values |
| rug | Marginal rug plots |
| segment | Single line segments |
| smooth | Add a smoothed condition mean |
| step | Connect observations by stairs |
| text | Textual annotations |
| tile | Tile plot as densely as possible, assuming that every tile is the same size |
| vline | Line, vertical |

Statistical Transformations

- A statistical transformation or **stat** transforms data, typically by summarizing it in some manner
- A **stat** takes a dataset as input and returns a dataset as output
- **stat** can add new variables to the original dataset, to which aesthetics may also be mapped

E.g. **stat_bin**, used to generate histograms, produces the count, density and x statistics

- An online search of a specific **stat** will list new variables generated by using that **stat**

Default ggplot2 Statistics & Aesthetics [Wickham, 2009]

| Name | Default stat | Aesthetics |
|------------|--------------|--|
| abline | abline | colour, linetype, size |
| area | identity | colour, fill, linetype, size, x , y |
| bar | bin | colour, fill, linetype, size, weight, x |
| bin2d | bin2d | colour, fill, linetype, size, weight, xmax , xmin , ymax , ymin |
| blank | identity | |
| boxplot | boxplot | colour, fill, lower , middle , size , upper , weight, x , ymax , ymin |
| contour | contour | colour, linetype, size, weight, x , y |
| crossbar | identity | colour, fill, linetype, size, x , y , ymax , ymin |
| density | density | colour, fill, linetype, size, weight, x , y |
| density2d | density2d | colour, linetype, size, weight, x , y |
| errorbar | identity | colour, linetype, size, width, x , ymax , ymin |
| freqpoly | bin | colour, linetype, size |
| hex | binhex | colour, fill, size, x , y |
| histogram | bin | colour, fill, linetype, size, weight, x |
| hline | hline | colour, linetype, size |
| jitter | identity | colour, fill, shape, size, x , y |
| line | identity | colour, linetype, size, x , y |
| linerule | identity | colour, linetype, size, x , y , ymax , ymin |
| path | identity | colour, linetype, size, x , y |
| point | identity | colour, fill, shape, size, x , y |
| pointrange | identity | colour, fill, linetype, shape, size, x , y , ymax , ymin |
| polygon | identity | colour, fill, linetype, size, x , y |
| quantile | quantile | colour, linetype, size, weight, x , y |
| rect | identity | colour, fill, linetype, size, xmax , xmin , ymax , ymin |
| ribbon | identity | colour, fill, linetype, size, x , ymax , ymin |
| rug | identity | colour, linetype, size |
| segment | identity | colour, linetype, size, x , xend , y , yend |
| smooth | smooth | alpha, colour, fill, linetype, size, weight, x , y |
| step | identity | colour, linetype, size, x , y |
| text | identity | angle, colour, hjust , label , size, vjust , x , y |
| tile | identity | colour, fill, linetype, size, x , y |
| vline | vline | colour, linetype, size |

Statistical Transformations [EXAMPLE]

- The following code generates a `ggplot` object

```
> my_ggplotObj_01 <- ggplot(diamonds, aes(carat))
```

- This code generates the default histogram...

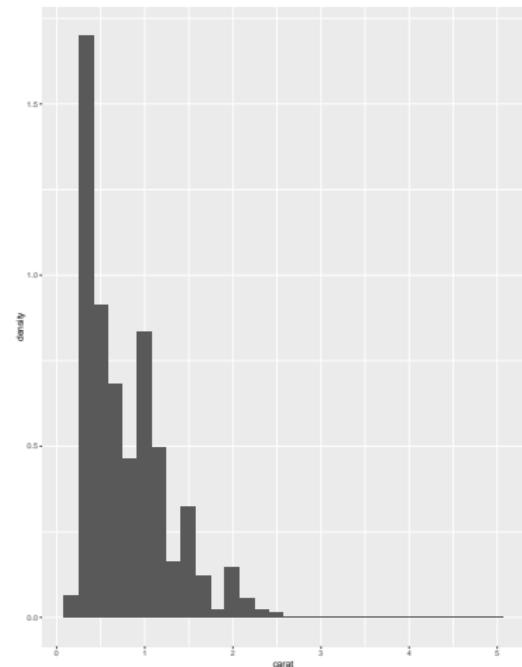
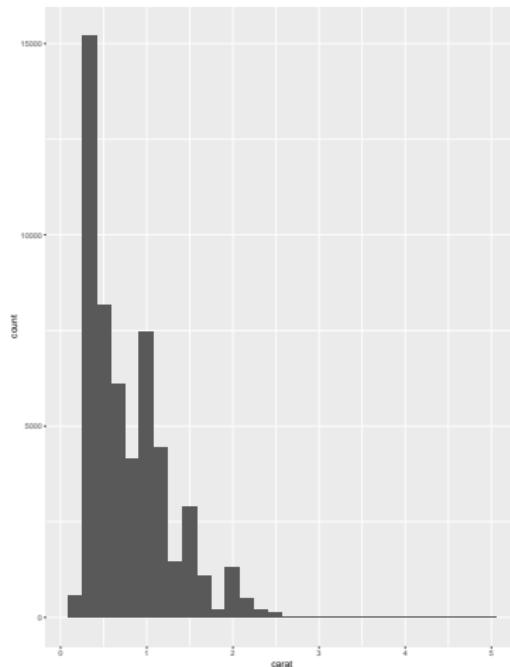
```
> my_ggplotObj_01 + geom_histogram()
```

- ...whereas this code generates a histogram with density on the y axis instead of the default count

```
> my_ggplotObj_01 + geom_histogram(y = ..density..)
```

- n.b.** When referencing the names of variables generated by a `stat` (as above), variable names **must** be surrounded by ..

Statistical Transformations [EXAMPLE CONT'D]



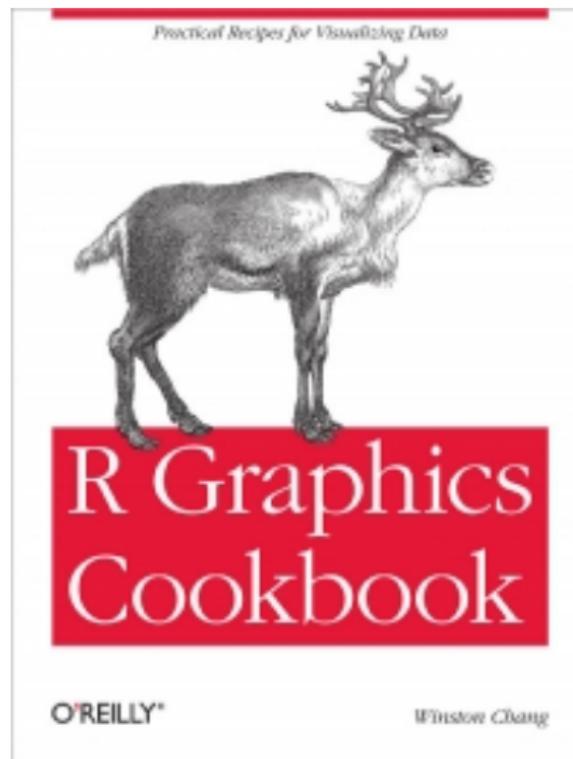
stats in ggplot2 [Wickham, 2009]

| Name | Description |
|------------|---|
| bin | Bin data |
| boxplot | Calculate components of box-and-whisker plot |
| contour | Contours of 3d data |
| density | Density estimation, 1d |
| density_2d | Density estimation, 2d |
| function | Superimpose a function |
| identity | Don't transform data |
| qq | Calculation for quantile-quantile plot |
| quantile | Continuous quantiles |
| smooth | Add a smoother |
| spoke | Convert angle and radius to xend and yend |
| step | Create stair steps |
| sum | Sum unique values. Useful for overplotting on scatter-plots |
| summary | Summarise y values at every unique x |
| unique | Remove duplicates |

Position Adjustments

- Position adjustments apply minor tweaks to the position of elements within a layer
- Position adjustments are usually used with discrete data
- Position can be changed with the argument `position=`

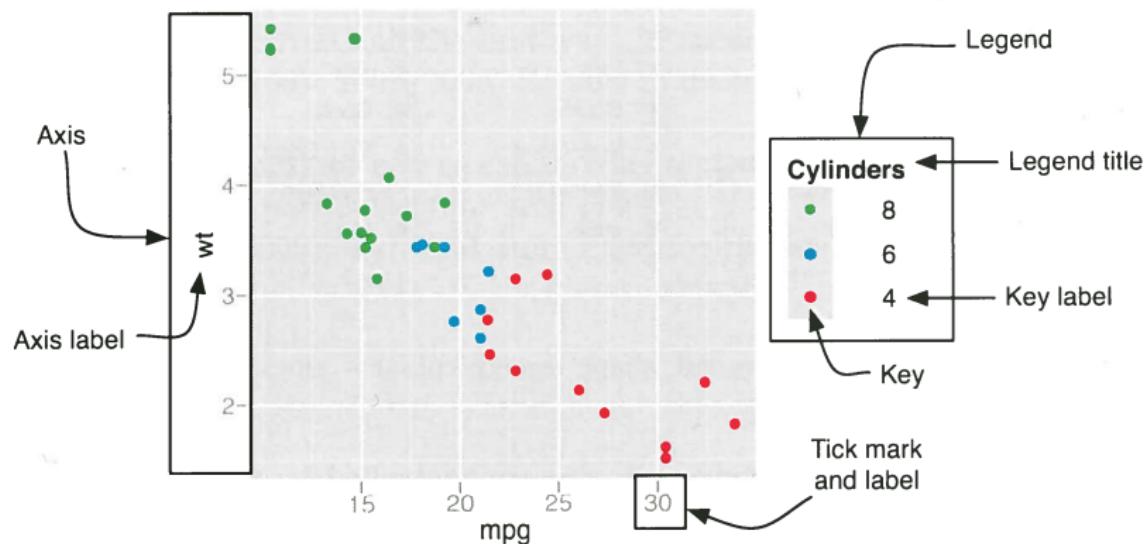
| Adjustment | Description |
|------------|---|
| dodge | Adjust position by dodging overlaps to the side |
| fill | Stack overlapping objects and standardise have equal height |
| identity | Don't adjust position |
| jitter | Jitter points to avoid overplotting |
| stack | Stack overlapping objects on top of one another |



Annotating Plots

- **ggplot2** makes it very easy to annotate a plot with
 - 1 Vertical bars
 - 2 Horizontal bars
 - 3 Shaded regions
 - 4 Text
 - 5 Mathematical symbols and equations
 - 6 Arrows

Components of the Axes and Legend



Axes

- Be sure to set a font size that is **legible** on your graphs, particularly when dealing with axis labels and axis ticks
- When dealing with large orders of magnitude on an axis, either reduce the order of magnitude manually **or** use the **scales** package and include commas
 - 1 If you have \$15,000,000,000 as an axis tick, it might be more succinct and digestible for the reader to simply have \$15, and in the axis title include a parenthetical reference (\$B)
 - 2 If you have \$2500000 on an axis tick, you force the reader to parse the text to figure out the order of magnitude; in lieu add commas using the **scales** package, resulting in \$2,500,000

LAB

Using airquality

- 1 Create a scatterplot of day (x) vs. temperature (y) and draw lines that connect months, Color the lines by month as well and display the month labels with the full name of the month (e.g., "January") (be sure the months appear in order)
- 2 Create a barchart where each bar is a month, and the height is the mean wind for that month