



Paul Intrevado

# Exploratory Data Analysis with R

Summer 2017



UNIVERSITY OF  
SAN FRANCISCO | Master of Science  
in Analytics



# Table of Contents

- 1 A Brief Introduction
- 2 Introduction to R
- 3 RMarkdown
- 4 Data, Data Structures & Data Manipulation
- 5 Exploratory Data Analysis in R
- 6 Programming in R
- 7 Character Functions and Manipulation in R



## Section 1

### A Brief Introduction



# The Warden





# Who Am I?

- Ph.D. Operations Management (McGill University)
  - Research focused on service operations
  - Model and solve optimization problems (e.g., MIPs)
- B.Sc., M.Sc. Industrial Engineering (Purdue University)
- B. Commerce (McGill University)
- Assistant Prof @ USF as of August 2014
- I teach or have taught
  - MSAN 593 - Exploratory Data Analysis
  - MSAN 601 - Linear Regression Analysis
  - MSAN 605/625/627/632 - Practicum
- MS Analytics Practicum Director
- Associate Director of Data Institute



# What I Do

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

subject to

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

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

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

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

$$\eta_{it} \lambda_{it} + \psi_{i(t-1)}^{(\nu+1 \rightarrow \nu)} = \sum_{j \in \mathbb{J}} y_{ij\nu t} \quad \forall i \in \mathbb{I}, \nu \in \mathbb{V}, t \in \mathbb{T} \quad (4.6)$$

$$\eta_{it} \lambda_{it} \pi_{it} + \psi_{i(t-1)}^{(\nu+1 \rightarrow \nu)} \leq y_{ij\nu t} \quad \forall i = j \in \mathbb{J}, \nu \in \mathbb{V}, t \in \mathbb{T} \quad (4.7)$$



## 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++
- MATLAB
- CPLEX
- Gurobi
- R
- Python
- SQL
- CSS / HTML (web)
- MS Excel / MS Access / VBA



## Subsection 2

### About this Class



## Course Material

- All digital homework submissions and grades will be submitted/posted through Canvas
- All other course material can be found at the following link

<https://goo.gl/q5Kibn>



# Syllabus

**MSAN 593 – Exploratory Data Analysis with R**

**Instructor: Paul Intrevado**

**Course Syllabus**

**Summer 2017**

## **SUMMARY INFORMATION**

**Offices:** SFH 525 (Downtown) / McLaren Hall, Room 103 (Main Campus)

**Office Hours:** Wednesdays, 10.00 - 11.30 and 13.45 - 15.15 in SFH 525

**Office Phone:** 415/422.2527

**Email:** [pintrevado@usfca.edu](mailto:pintrevado@usfca.edu)

**Class Location:** 101 Howard Street, SFH 155/156

**Class Time:** 10:00 - 12:00 / 13:00 - 15:00 Thursdays and Fridays

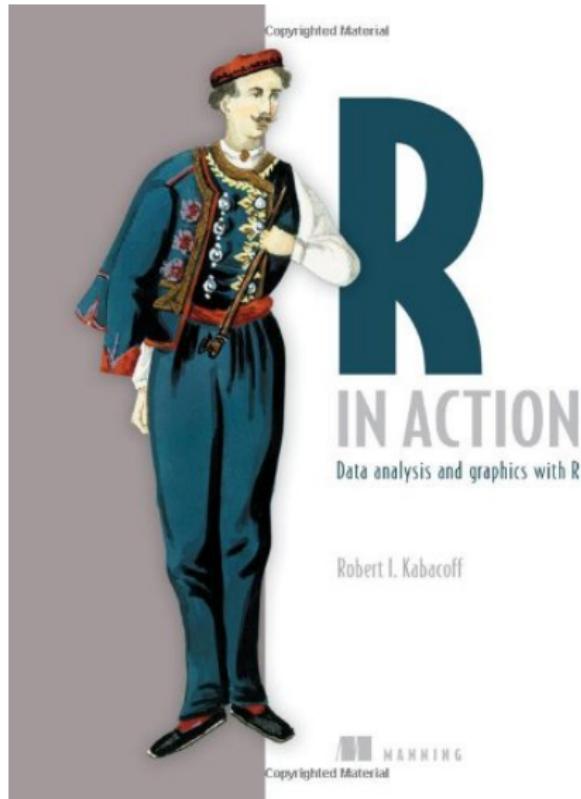
**Final Exam:** 13.00 - 15.00 Friday, August 11<sup>th</sup>, SFH 527/529

**Quizzes:** 09:15 - 10:00 Thursdays, SFH 155/156



## Subsection 3

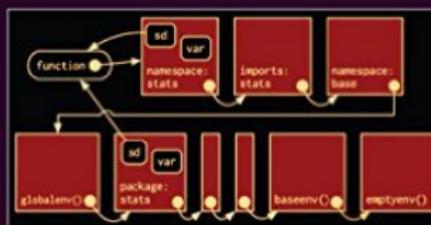
### Reference Textbooks





The R Series

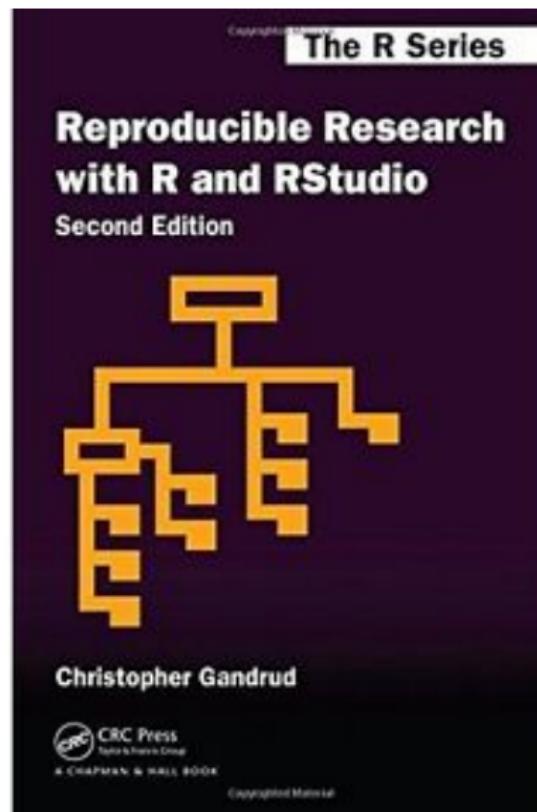
# Advanced R



Hadley Wickham



CRC Press  
Taylor & Francis Group  
A CHAPMAN & HALL BOOK





The R Series

# Dynamic Documents with R and knitr

Second Edition

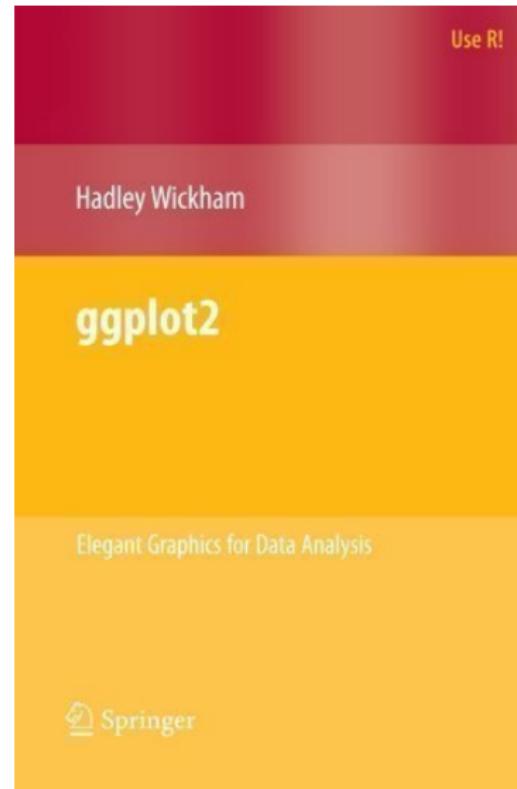


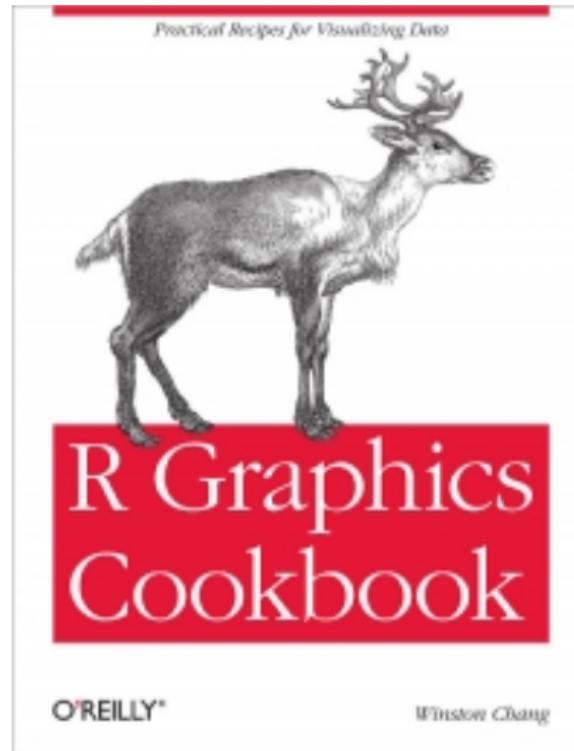
Yihui Xie

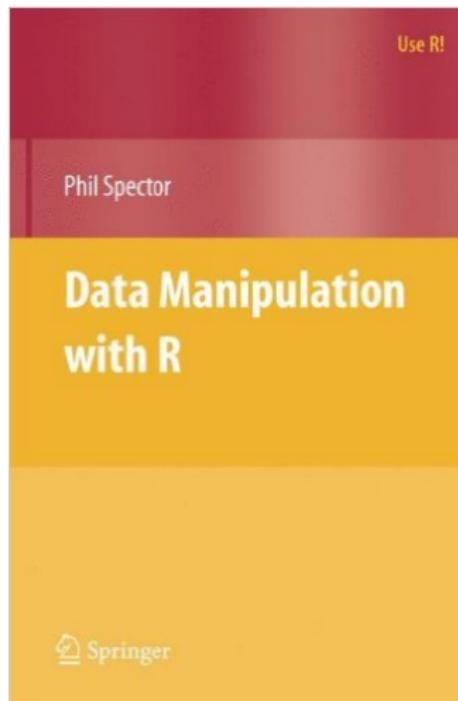


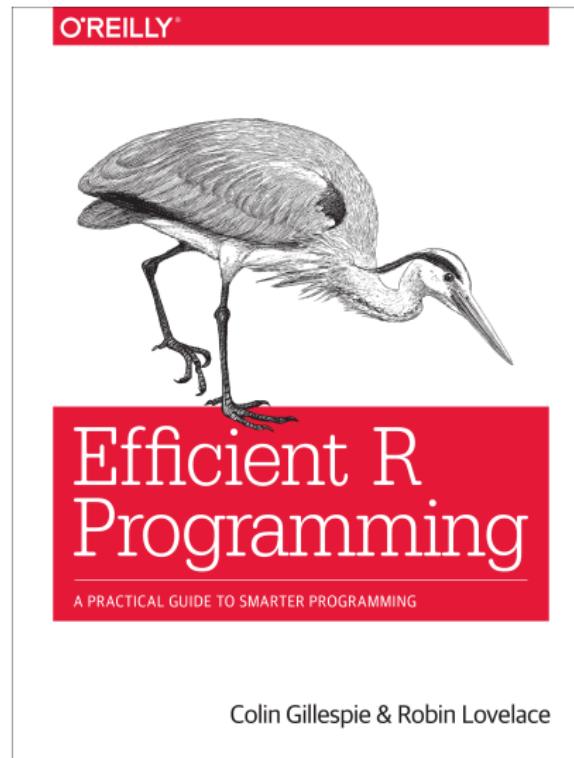
CRC Press  
Taylor & Francis Group

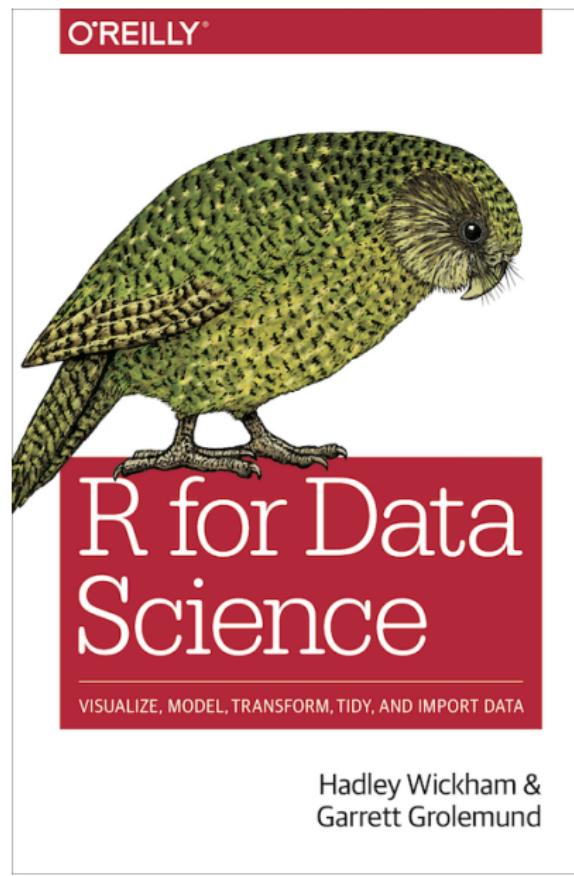
A CHAPMAN & HALL BOOK











Hadley Wickham &  
Garrett Grolemund



## Section 2

### Introduction to R



## What About Excel?





# Excel is Great for Certain Things...

Screenshot of Microsoft Excel showing a data table for student grades.

The table includes columns for Student ID, Name, Midterm, Final, and various assignment scores (Asn #1 through Asn #6), along with calculated V1 and V2 values, Final Points, and Letter Grade.

	A	B	C	D	E	F	G	H	I	J	K	L	M	
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  $\sim 1.05$  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  $\sim 1.05$  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

R version 3.2.4 (2016-05-10) -- very secure Vienna  
Copyright (C) 2016 The R Foundation for Statistical Computing  
Platform: x86\_64-apple-darwin13.4.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.  
You are welcome to redistribute it under certain conditions.  
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.  
Type 'contributors()' for more information and  
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or  
'help.start()' for an HTML browser interface to help.  
Type 'q()' to quit R.

[R.app GUI 1.67 (7152) x86\_64-apple-darwin13.4.0]  
[Workspace restored from /Users/Paul/.RData]  
[History restored from /Users/Paul/.Rapp.history]

> |



# This is what RStudio Looks Like

~/workbench - RStudio

userTrend.R\*    Q1Report.Rnw    userData

Source 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 pplot(state, age, color = active, data = userData,
21   main = "Breakdown of Users by Age and State") +
22   opts(plot.title = theme_text(size = 19))
23 |
```

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
> pplot(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
- It is best practice to use the assignment operator '`<-`' instead of the equality operator '`=`' for all code, even though both work, e.g.,

---

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

---

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



## Basic R 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?





# Useful R Keyboard Shortcuts: Autocomplete

The screenshot shows the RStudio interface with the following details:

- Console Area:** Shows the command `> q`. A dropdown menu is open over the command, listing completions for the function `q` from the `base` package. The first item in the list is `q {base}`.
- Message Area:** Below the dropdown, a message states: "The function quit or its alias q terminate the current R session. Press F1 for additional help".
- Workspace Area:** Shows a file tree with the following contents:

Name	Size	Modified
caratcut.png	291.7 KB	Feb 7, 2013, 9:37 AM
expensive.png	218.8 KB	Feb 7, 2013, 9:37 AM
blue.png	124 KB	Feb 7, 2013, 9:37 AM
caratbox.png	121.6 KB	Feb 7, 2013, 9:37 AM
blue2.png	121.6 KB	Feb 7, 2013, 9:37 AM
slides.md	6.6 KB	Feb 7, 2013, 9:37 AM
04-large-data		

[Tab]



# Useful R Keyboard Shortcuts: History

The screenshot shows the RStudio interface with the following details:

- Code Editor (Left Panel):** Displays R code in the "Untitled1" script. The last command in the history is highlighted with a blue selection bar:

```
qplot(table ~ depth, data = diamonds,
qplot(day, data = email)
qplot(day, mails, data = daily, geom = "line", colo
qplot(day, mails, data = daily, geom = "smooth", co
qplot(day, variants, data = daily, geom = "line", c
qplot(wday, hour, data = wh, size = freq)
qplot(mpg, wt, data = mtcars)
qplot(mpg, wt, data = mtcars, colour = cyl)
```
- History Tab (Top Bar):** Shows the "History" tab is active.
- File Browser (Right Panel):** Shows a folder structure under "04-large-data".

Name	Size	Modified
..		
04-large-data.html	4.2 MB	Feb 7, 2013, 9:37 AM
overplot.png	936.4 KB	Feb 7, 2013, 9:37 AM
transparent.png	863.2 KB	Feb 7, 2013, 9:37 AM
small.png	463.8 KB	Feb 7, 2013, 9:37 AM
caratcut.png	291.7 KB	Feb 7, 2013, 9:37 AM
B		Feb 7, 2013, 9:37 AM
B		Feb 7, 2013, 9:37 AM
B		Feb 7, 2013, 9:37 AM
B		Feb 7, 2013, 9:37 AM
- Keyboard Shortcut Overlay:** Large text at the bottom left indicates the keyboard shortcut: [Cmd/Ctrl + ↑].



# Useful R Keyboard Shortcuts: Execute Subset of Code

The screenshot shows the RStudio interface with the following details:

- Top Bar:** Shows the path `~/Documents/rstudio/training/Introduction to R/1-r-basics/1-basic-visualization/04-large-data - RStudio` and a tab labeled `04-large-data`.
- Left Panel:** A code editor window titled `Untitled1*` containing the R command `library(ggplot2)`. Below it is a `Console` window showing the same command being run.
- Middle Panel:** A large text area containing the text `[Cmd/ctrl + enter]`.
- Right Panel:** A file browser showing a list of files in the `04-large-data` directory. The files listed are:

Name	Size	Modified
..		
04-large-data.html	4.2 MB	Feb 7, 2013, 9:37 AM
overplot.png	936.4 KB	Feb 7, 2013, 9:37 AM
transparent.png	863.2 KB	Feb 7, 2013, 9:37 AM
small.png	463.8 KB	Feb 7, 2013, 9:37 AM
caratcut.png	291.7 KB	Feb 7, 2013, 9:37 AM
expensive.png	218.8 KB	Feb 7, 2013, 9:37 AM
blue.png	124 KB	Feb 7, 2013, 9:37 AM
caratbox.png	121.6 KB	Feb 7, 2013, 9:37 AM
blue2.png	121.6 KB	Feb 7, 2013, 9:37 AM
slides.md	6.6 KB	Feb 7, 2013, 9:37 AM
04-large-		



# Useful R Keyboard Shortcuts: Execute All Code

The screenshot shows the RStudio interface. In the top-left pane, the 'Console' window displays the command `> library(ggplot2)`. Below it, the 'File Explorer' pane shows a directory structure under '04-large-data'. The 'File' menu is open, and the option 'Execute All' is highlighted. A large bracket below the console area indicates the keyboard shortcut: [Cmd/ctrl + shift + enter].

[Cmd/ctrl + shift + enter]

Name	Size	Modified
..		Feb 7, 2013, 9:37 AM
04-large-data.html	4.2 MB	Feb 7, 2013, 9:37 AM
overplot.png	936.4 KB	Feb 7, 2013, 9:37 AM
transparent.png	863.2 KB	Feb 7, 2013, 9:37 AM
small.png	463.8 KB	Feb 7, 2013, 9:37 AM
caratcut.png	291.7 KB	Feb 7, 2013, 9:37 AM
expensive.png	218.8 KB	Feb 7, 2013, 9:37 AM
blue.png	124 KB	Feb 7, 2013, 9:37 AM
caratbox.png	121.6 KB	Feb 7, 2013, 9:37 AM
blue2.png	121.6 KB	Feb 7, 2013, 9:37 AM
slides.md	6.6 KB	Feb 7, 2013, 9:37 AM
04-large-		



# Useful R Keyboard Shortcuts: Restarting an R Session

The screenshot shows the RStudio interface. In the top-left pane, the code `library(ggplot2)` is entered. In the bottom-left pane, the console output shows:

```
1:17  (Top Level) : 
Console ~ /Documents/rstudio/training/Introduction to R/1-r-basics/1-basic-visualization/04-large-data > library(ggplot2)
> | 
Restarting R session...
```

In the bottom-right pane, a file browser lists several files:

Name	Size	Modified
..		
04-large-data.html	4.2 MB	Feb 7, 2013, 9:37 AM
overplot.png	936.4 KB	Feb 7, 2013, 9:37 AM
transparent.png	863.2 KB	Feb 7, 2013, 9:37 AM
small.png	463.8 KB	Feb 7, 2013, 9:37 AM
caratcut.png	291.7 KB	Feb 7, 2013, 9:37 AM
expensive.png	218.8 KB	Feb 7, 2013, 9:37 AM
blue.png	124 KB	Feb 7, 2013, 9:37 AM
caratbox.png	121.6 KB	Feb 7, 2013, 9:37 AM
blue2.png	121.6 KB	Feb 7, 2013, 9:37 AM
slides.md	6.6 KB	Feb 7, 2013, 9:37 AM
04-large-		



# IMPORTANT R Setting

Options

General

Code

Appearance

Pane Layout

Packages

Sweave

Spelling

Git/SVN

Publishing

Default working directory (when not in a project): ~

Restore most recently opened project at startup

Restore previously open source documents at startup

Restore .RData into workspace at startup

Save workspace to .RData on exit:

Always save history (even when not saving .RData)

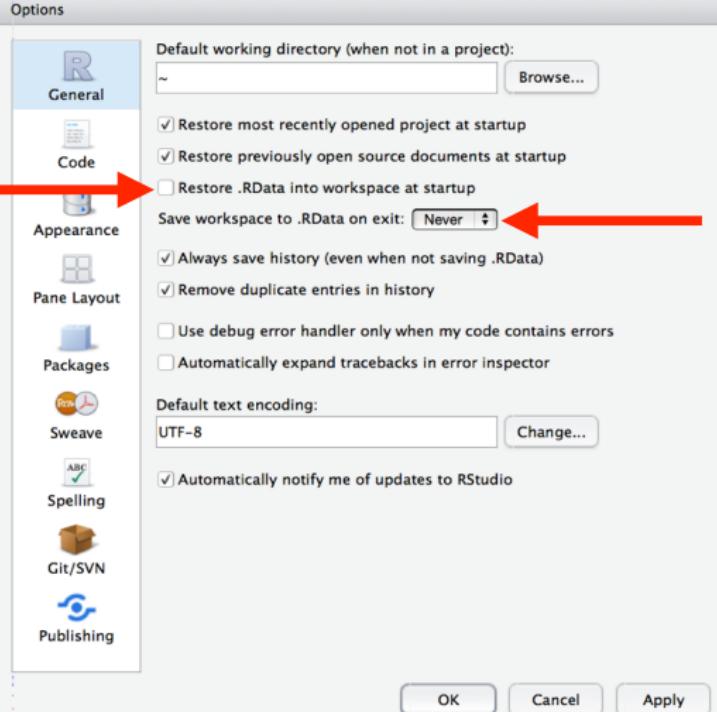
Remove duplicate entries in history

Use debug error handler only when my code contains errors

Automatically expand tracebacks in error inspector

Default text encoding: UTF-8

Automatically notify me of updates to RStudio





## TRY THIS [IN A SCRIPTING WINDOW]

- ➊  $10,352 + 987,653$
- ➋  $10,352 / 987,653$
- ➌  $2^6 \times 99$
- ➍  $91,4545 \text{ modulus } 33$



## TRY THIS [IN A SCRIPTING WINDOW]

- ①  $10,352 + 987,653$
- ②  $10,352 / 987,653$
- ③  $2^6 \times 99$
- ④  $91,4545 \text{ modulus } 33$

## SOLUTIONS

① > 10352 + 987653

② > 10352 / 987653

③ > > 2\*\*6 \* 99      OR      2^6 \* 99

④ > 914545 %% 33



## 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 For Mac, `cat("\014")` clears the console window
  - same as keystroke `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 ~ 10,400 R packages [April 2017]



## 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 ~ 10,400 R packages [April 2017]
- 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("dplyr")
    % Total    % Received % Xferd  Average Speed   Time     Time     Time   ...
      0          0          0          0          0          0          0 --::--::-- --::--::-- --::--::-- ...
The downloaded binary packages are in
 /var/folders/jm/3w7pqfms0nvg_ypvnvkkk83h0000gn/...

> summarise(iris, meanVal = mean(Sepal.Length))
Error: could not find function "summarise"

> dplyr::summarise(iris, meanVal = mean(Sepal.Length))
  meanVal
  1 5.843333
```



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

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

# it can now be called without the dplyr:: prefix
> summarise(iris, meanVal = mean(Sepal.Length))
  meanVal
1 5.843333
```

- 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 [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 tabs, there are two buttons: "Install" and "Update". A search bar and a refresh icon are also present. The main area displays a table of installed 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	



## Notes on R Packages

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

```
#load the package MatchIt  
> library(MatchIt)  
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



## 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"
```



## 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()`



## Notes on R Packages [EXAMPLE]

### TRY THIS

```
> library(magrittr)
> library(dplyr)

> iris %>% select(Petal.Length, Species) %>% group_by(Species) %>%
  summarize(meanVal = mean(Petal.Length))

> library(Hmisc)

> iris %>% select(Petal.Length, Species) %>% group_by(Species) %>%
  summarize(meanVal = mean(Petal.Length))

Error in summarize(., meanVal = mean(Petal.Length)) :
  argument "by" is missing, with no default
```



## Notes on R Packages [EXAMPLE]

### TRY THIS

```
> library(magrittr)
> library(dplyr)

> iris %>% select(Petal.Length, Species) %>% group_by(Species) %>%
  summarize(meanVal = mean(Petal.Length))

> library(Hmisc)

> iris %>% select(Petal.Length, Species) %>% group_by(Species) %>%
  summarize(meanVal = mean(Petal.Length))

Error in summarize(., meanVal = mean(Petal.Length)) :
  argument "by" is missing, with no default
```

### SOLUTION

```
> iris %>% select(Petal.Length, Species) %>% group_by(Species) %>%
  dplyr::summarize(meanVal = mean(Petal.Length))
```



## 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



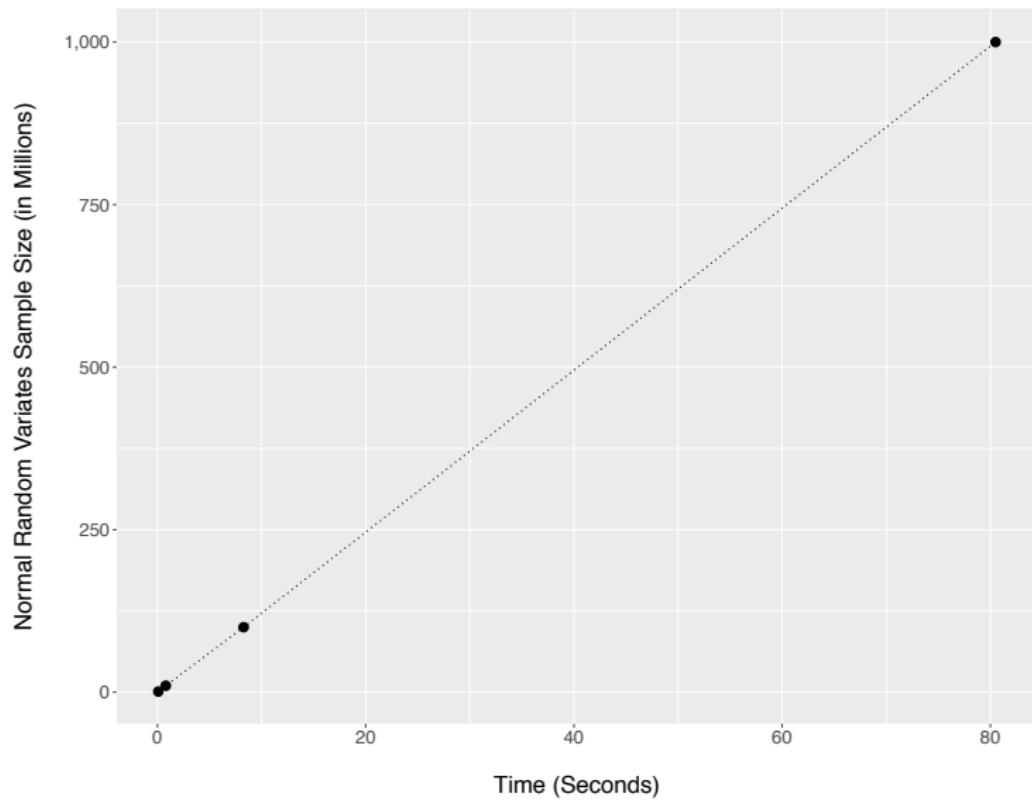
## How Big is Big Data in R? [CONT'D]

- 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



## TRY THIS

- ① Create a vector named `myVec` with 1,000,000 random  $\sim \mathcal{N}(10, 5)$  variates. How much time does this take?
- ② Repeat and time with 10,000,000 random variates
- ③ Repeat and time with 100,000,000 random variates
- ④ Repeat and time with 1,000,000,000 random variates
- ⑤ Generate a graph with *Sample Size* on the *y* axis and *Time* on the *x* axis





## SOLUTION

```
# create vector and time execution
system.time(
  myVec <- rnorm(1000000, mean = 10, sd = 5)
)

# load tidyverse to access tibble and ggplot packages
library(tidyverse)

myDF <- tibble(
  sampleSize = c(1, 10, 100, 1000),
  time = c(0.079, 0.805, 8.274, 80.5)
)

library(magrittr) # for piping
library(scales)   # for commas in y axis

# create graph in ggplot
myDF %>%
  ggplot(aes(x = time, y = sampleSize)) +
  geom_line(linetype = "dotted") +
  geom_point(size = 3) +
  xlab("\nTime (Seconds)") +
  ylab("Normal Random Variates Sample Size (in Millions)\n") +
  scale_y_continuous(labels = comma)
```



## 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



## Section 3

### RMarkdown



## Technical Reporting & Presentation Tools

- There are many ways to generate pdf- and web-based technical reports and presentations
  - ➊  $\text{\LaTeX}$  (pronounced *lay-tech*)
  - ➋ Lyx
  - ➌ Markdown (RMarkdown and other variations)
  - ➍ ...
- The (arguably) default, most generic and most flexible technical report/presentation tool is  $\text{\LaTeX}$
- 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

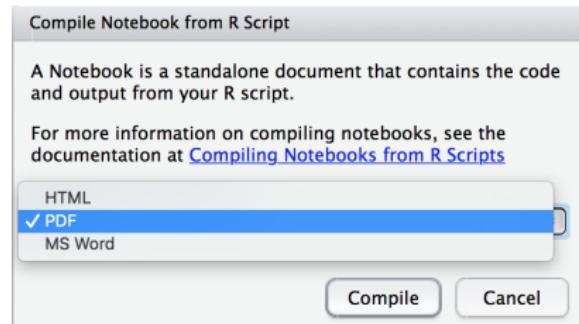


# The Simplest RMarkdown Execution

A screenshot of the RStudio interface. The top menu bar shows tabs for "codeOnSlides.R", "Untitled.Rmd", and "Untitled.R\*". Below the tabs is a toolbar with icons for back, forward, search, and file operations. 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
```

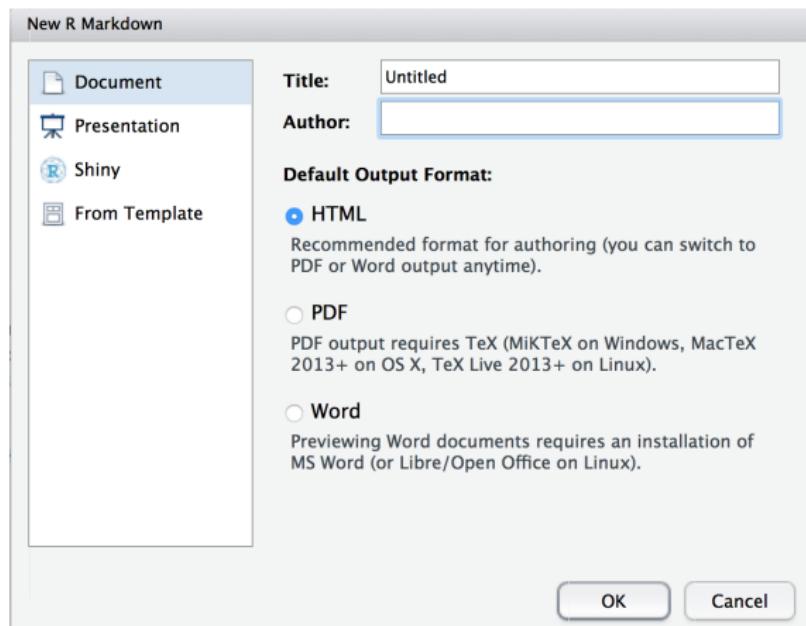
```
## [1] 3
```



# A Smarter RMarkdown Document

Create a new RMarkdown file

File ⇒ New File ⇒ RMarkdown...





## MacTex is Required for RMarkdown pdf Output





## RMarkdown: The Header

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

```
---
```

```
title: "My Title"
author: "Paul Intrevado"
date: "July 5, 2017"
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
```

- 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  $\text{\LaTeX}$  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  $\text{\LaTeX}$  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  $\text{\LaTeX}$  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  $\text{\LaTeX}$  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 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 = F`: prevents code from being evaluated
- `echo = F`: prevents code, but not results, from appearing in final output document
- `include = F`: runs code but doesn't show code **or** results in final output document
- `message = F / warning = F`: prevents messages or warning from appearing in final output document
- `cache = T`: will store the results of a code chunk in cache, so subsequent knits of the document don't need to re-execute computationally expensive code chunks (use only for static data)



## RMarkdown: Selected Global Chunk Options

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

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

- include = FALSE** is included so that the code is evaluated but the code chunk nor the results are printed to the output document

E.g. To have all code chunks in an RMarkdown document be suppressed, include **include = FALSE** 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



# Tables in RMarkdown

```
dplyr::slice(mtcars, 1:5)

## # A tibble: 5 x 11
##   mpg   cyl   disp    hp   drat    wt   qsec    vs    am   gear   carb
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 21.0     6    160   110   3.90  2.620  16.46     0     1     4     4
## 2 21.0     6    160   110   3.90  2.875  17.02     0     1     4     4
## 3 22.8     4    108    93   3.85  2.320  18.61     1     1     4     1
## 4 21.4     6    258   110   3.08  3.215  19.44     1     0     3     1
## 5 18.7     8    360   175   3.15  3.440  17.02     0     0     3     2
```

```
knitr:: kable(dplyr::slice(mtcars, 1:5))
```

| mpg  | cyl | disp | hp  | drat | wt    | qsec  | vs | am | gear | carb |
|------|-----|------|-----|------|-------|-------|----|----|------|------|
| 21.0 | 6   | 160  | 110 | 3.90 | 2.620 | 16.46 | 0  | 1  | 4    | 4    |
| 21.0 | 6   | 160  | 110 | 3.90 | 2.875 | 17.02 | 0  | 1  | 4    | 4    |
| 22.8 | 4   | 108  | 93  | 3.85 | 2.320 | 18.61 | 1  | 1  | 4    | 1    |
| 21.4 | 6   | 258  | 110 | 3.08 | 3.215 | 19.44 | 1  | 0  | 3    | 1    |
| 18.7 | 8   | 360  | 175 | 3.15 | 3.440 | 17.02 | 0  | 0  | 3    | 2    |



## 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](http://rmarkdown.rstudio.com/pdf_document_format.html)



## 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>



# LAB

## RMarkdown: Titanic Case Study

*Paul Intrevado*

*July 07, 2017*

This is my first R Markdown document for **MSAN 593**. I am required to submit all **MSAN 593** homework in RMarkdown.

Firstly, I am going to generate an `html` document as output, set in the YAML header section of this document. Secondly, I am **NOT** hardcoding the date, but rather using a function which will automatically print the current date on the day the document is knitted.

Now, I am going to import a dataset about passengers from the Titanic, using the following line of code:

```
read.csv("~/Desktop/titanic.csv")
```

This fails for a few reasons, namely, I read in the file and stored it nowhere. So I wasted my time waiting for R to read in the file, and then when it finally did, it printed the rows of data to the Console window, and voila, the data disappeared faster than it loaded. Now I know better.

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



## Section 4

# Data, Data Structures & Data Manipulation



# 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

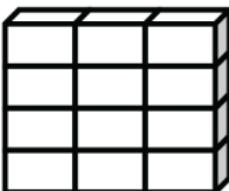


# Data Structures

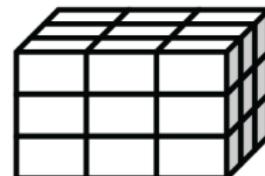
(a) Vector



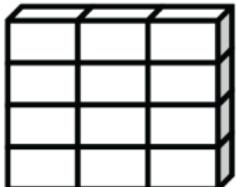
(b) Matrix



(c) Array



(d) Data frame



Columns can be different modes

(e) List

- Vectors
- Arrays
- Data frames
- Lists



# Vectors

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

## Properties of Vectors

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

**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 your 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



## TRY THIS TOO

- ➊ Guess and then check what types your vectors are.
- ➋ Check the length of each vector.
- ➌ Did you write the code in the console window or the editor?
- ➍ How do you execute a line of code in the editor?
- ➎ How do you execute multiple lines of code simultaneously in the editor?
- ➏ Did you leverage the TAB button for auto-completion?



## Accessing 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?



## 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)
```

## SOLUTION

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 <- c(1, 2, 3, NA)

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

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

> x[4]
[1] NA

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

**n.b.** `NA` and `NULL` are **NOT** the same



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()` and `glimpse()`

- 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
- Using the `tibble` package, `glimpse()` gives you a cleaner, easier-to-read view of your data
- Using `population` data from `tidyverse` package

```
> str(population)
Classes `tbl_df`, `tbl` and `data.frame`: 4060 obs. of  3 variables:
$ country    : chr  "Afghanistan" "Afghanistan" "Afghanistan" "Afghanistan" ...
$ year       : int  1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 ...
$ population: int  17586073 18415307 19021226 19496836 19987071 20595360 ...

> tibble::glimpse(population)
Observations: 4,060
Variables: 3
$ country    <chr> "Afghanistan", "Afghanistan", "Afghanistan", "Afghanistan"...
$ year       <int> 1995, 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004...
$ population <int> 17586073, 18415307, 19021226, 19496836, 19987071, 20595360...
```



## 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.** 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`



## 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)]`

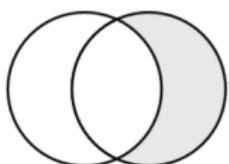
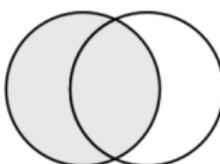
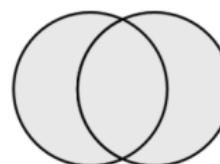
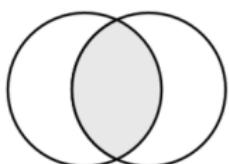
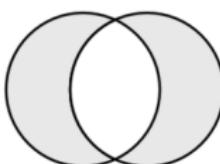
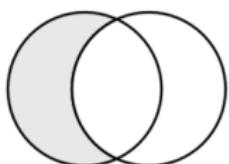
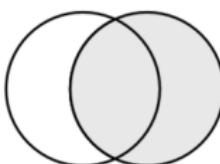


# Logical Operators

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



# Logical Operators

 $y \& !x$  $x$  $x | y$  $x \& y$  $xor(x, y)$  $x \& !y$  $y$



## PREAMBLE

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

## TRY THIS

- ➊ How many positive numbers ( $> 0$ ) are there in this vector?
- ➋ How many negative numbers ( $< 0$ ) are there in this vector?
- ➌ How many 0's are there in this vector?
- ➍ How many NAs are there in this vector?
- ➎ How many numbers in the vector are non-zero **and** not NAs?
- ➏ What is the sum of the positive numbers in this vector?
- ➐ 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(aA = 10:12, bB = "abc", cC = c(3.1415, 9), dD = c(T, F, F, F))

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

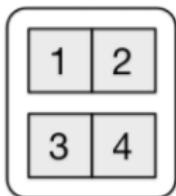
- Lists are recursive, i.e., a list can contain lists, making them fundamentally different from atomic vectors
- `is.list()` (test if list), `as.list()` (coerce to list), `unlist()` (convert to atomic vector + coercion)



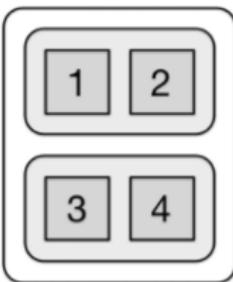
## Accessing List Elements [RDS, Wickham, 2017]

```
> x1 <- list(c(1, 2), c(3, 4))  
> x2 <- list(list(1, 2), list(3, 4))  
> x3 <- list(1, list(2, list(3)))
```

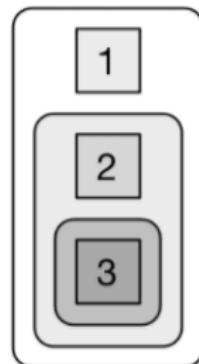
x1



x2



x3



**n.b.** Lists have rounded corners, atomic vectors have square corners



## Accessing List Elements [CONT'D]

- Using `[` accesses a sub-list, i.e., the result is a list (type preservation)

```
> x1 <- list(c(1, 2), c(3, 4))

> x2 <- list(list(1, 2), list(3, 4))

> str(x1[2])
List of 1
 $ : num [1:2] 3 4

> str(x2[2])
List of 1
 $ :List of 2
 ..$ : num 3
 ..$ : num 4
```



## Accessing List Elements [CONT'D]

- Using `[[` extracts elements of a list (type simplification)

```
> str(x1[2]) # generates a list
List of 1
$ : num [1:2] 3 4

> str(x1[[2]]) # generates a vector
num [1:2] 3 4

> str(x1[[2]][1]) # generates a singleton vector
num 3

> str(x1[2][1]) # generates a list
List of 1
$ : num [1:2] 3 4

> str(x1[2][[1]]) # generates a vector
num [1:2] 3 4

> str(x1[[2]][[1]]) # generates a singleton vector
num 3
```



## Why/When to Use Lists

- Lists are a catch all, and can therefore be used for just about anything (applications are almost infinite)
- Two important and frequently-used applications of lists are
  - ➊ Growing vectors to an unknown size, often in a `for` loop
  - ➋ With `JSON` data



## Lists: Growing Vectors to an Unknown Size

**SCEANRIO:** Imagine you are collecting user IP addresses across 100 websites, trying to figure out how many users pinged each website on a given day, and how many website pings there were in total

**OBJECTIVE:** Create a vector of IP addresses that pinged all 100 websites

- Clearly this number will change on a daily basis
- In this scenario you cannot preallocate the size of your vector as it is unknown
- Why is size preallocation important when coding?
- What happens when you don't preallocate?



## Lists: Growing Vectors to an Unknown Size [CONT'D]

- What does the following code do?
- n.b. For ease of exposition, instead of using IP addresses **doubles** are stored

```
set.seed(10)

means <- runif(100, 0, 10)

output <- double()

system.time(
for (i in seq_along(means)) {

  n <- sample(1000000, 1)
  output <- c(output, rnorm(n, means[i]))
}
)
```

- Use the above code and note the execution time on your machine



## PREAMBLE

```
set.seed(10)
means <- runif(100, 0, 10)
```

## TRY THIS

Using a `for` loop and a `list`, generate the same vector as the previous example, timing the code



## PREAMBLE

```
set.seed(10)
means <- runif(100, 0, 10)
```

## TRY THIS

Using a `for` loop and a `list`, generate the same vector as the previous example, timing the code

## SOLUTION

```
set.seed(10)
means <- runif(100, 0, 10)

out <- list()

system.time({
  for (i in seq_along(means)){
    n <- sample(1000000, 1)
    out[[i]] <- rnorm(n, means[i])
  }
  out <- unlist(out)
})
```



## Lists: 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
- There are many packages in R that deal with JSON data including `jsonlite`, `rjson`, `RJSONIO`, `tidyjson`, etc.



## Lists: JSON Data [CONT'D]

JSON data that stores key/value pairs of information about people might look like this

```
[{  
    "first_name": "Laverna",  
    "last_name": "Marousek",  
    "email": "lmarousek0@pagesperso-orange.fr",  
    "gender": "Female",  
    "ip_address": "136.180.44.253"  
}, {  
    "first_name": "Merrill",  
    "last_name": "Matuska",  
    "email": "mmatuska1@businessinsider.com",  
    "gender": "Female",  
    "ip_address": "129.25.248.180"  
}]
```



## Lists: JSON Data [CONT'D]

JSON data may also be nested

```
[{  
    "first_name": "Steffen",  
    "last_name": "Biggs",  
    "child": [  
        {  
            "gender": "Female",  
            "name": "Alysa Terbeck"  
        }  
    ]  
, {  
    "first_name": "Ilyssa",  
    "last_name": "Padefield",  
    "child": [  
        {  
            "gender": "Male",  
            "name": "Haslett Rehor"  
        },  
        {  
            "gender": "Male",  
            "name": "Oliy McCambrois"  
        }  
    ]  
}]
```



## TRY THIS

Import `json_lab_data.json`, which contains nested data about parents and children

- ① How many children exist in this data set?
- ② How many female children are there? Male children?



## TRY THIS

Import `json_lab_data.json`, which contains nested data about parents and children

- ① How many children exist in this data set?
- ② How many female children are there? Male children?

## SOLUTION

```
> myJSONdata <- jsonlite::fromJSON("~/Desktop/json_lab_data.json")  
  
> flatJSONdata <- tibble::as_data_frame(jsonlite::flatten(myJSONdata))  
  
> unnestedJSONdata <- flatJSONdata %>% tidyrr::unnest(child)  
  
# Question 1  
> nrow(unnestedJSONdata)  
  
# Question 2  
> nrow(dplyr::filter(unnestedJSONdata, gender == "Male"))  
> nrow(dplyr::filter(unnestedJSONdata, gender == "Female"))
```



# Names

- A name is a vector **attribute**
- 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")))
[1] M F F M
Levels: F M

> class(x)
[1] "factor"

> typeof(x)
[1] "integer"

> str(x)
Factor w/ 2 levels "F", "M": 2 1 1 2
```

```
> levels(x)
[1] "F" "M"

> 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")))
[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))
```

- ➊ `c("h", "h", "h", "ob", "ov", "ov", "s")` is what we want to classify as a factor
- ➋ `levels = c("s", "h", "ov", "ob")` provides the levels
  - Omitting this enables R to order the levels itself
- ➌ `labels = c("Skinny", "Healthy", "Overweight", "Obese")` are the *nice* labels we want to see instead of the more obscurely-coded factors
- ➍ **n.b.** `labels` are mapped directly to `levels`
- ➎ `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
```



## PREAMBLE

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

## YOU TRY IT

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



## PREAMBLE

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

## YOU TRY IT

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

## SOLUTION

```
> myCyl <- factor(myCyl, labels = c("Small", "Medium", "Large"), ordered = T)
> length(myCyl[myCyl <= "Medium"])
```



## PREAMBLE

```
set.seed(4)
ageData <- round(runif(1000000, 0, 120))
```

## YOU TRY IT

Create the following ordered factors associated with ageData:

Infant (0-2 yrs); Toddler (3-5 yrs); Child (6-9 yrs); Tween (10-12 yrs); Teenager (13-19 yrs); Adult (20-65 yrs); Senior (66 yrs +).

- ① How many minors are there?
- ② Create a frequency table that shows the frequency by factor.



## PREAMBLE

```
set.seed(4)
ageData <- round(runif(1000000, 0, 120))
```

## SOLUTION

```
> ageData_class <- cut(ageData, breaks = c(-Inf, 2, 5, 9, 12, 19, 65, Inf),
  labels = c("infant", "toddler", "child", "tween", "teen", "adult", "senior"),
  levels = c("infant", "toddler", "child", "tween", "teen", "adult", "senior"),
  ordered = T)

> sum(ageData_class < "adult")
[1] 162804

> table(ageData_class)
ageData_class
 infant toddler     child    tween     teen    adult    senior
  20859     24991     33527     24909     58518    382780    454416
```



# 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)
%       # can drop ncol and nrow to shorten

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

```
> (y <- array(1:12, c(2, 3, 2)))
, , 1
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

, , 2
     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12
```



## 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
  a b c
d 1 2 3
e 4 5 6
f 7 8 9

> myMatrix[1, 1]
[1] 1

> myMatrix[1:3, 1]
d e f
1 4 7

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

```
> myMatrix[6]
[1] 8

# recycling logical row entries
> myMatrix[c(T, F), ]
  a b c
d 1 2 3
f 7 8 9

> myMatrix[1:2, "b"]
d e
2 5

# -----
# "[" will simplify results to the
# lowest possible dimensionality
# (more on this soon)
# -----
```



# Selected Functional Generalizations

## 1D

- ➊ `length()`
- ➋ `names()`
- ➌ `c()`

## nD

- ➊ `nrow()`, `ncol()`, `dim()`
- ➋ `rownames()`, `colnames()`,  
`dimnames()`
- ➌ `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

### Summary of Sample Data Frame

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 ...
 $ 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 ...  
> ?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"

...OR...

(xyz <- tibble::data_frame(1:3, c("a", "b", "c")))
# A tibble: 3 x 2
`1:3` `c("a", "b", "c")`<int><chr>
1     1             a
2     2             b
3     3             c
```



## Tibbles with tibble

- Data frames can be fickle, with their `stringsAsFactors = FALSE` requirements, etc.
- Tibbles—from the `tibble` package—are **data frames**, with some minor modifications which make operating with tibbles a more pleasant and worry-free experience than dealing with raw data frames
- Creating a tibble is the same as creating a data frame, save the suffix now changes to `tibble()` instead of `data.frame()`
- Coercing a data frame (or other data structure) is done with the `as_tibble()` function



## Tibbles with `tibble` [CONT'D]

- Tibbles have some very convenient advantages
- When importing (e.g., csv's) or coercing data, `tibble()` function:
  - ① **will not** automatically convert strings to factors
  - ② **will not** change the names of variables
  - ③ **will not** create row names
  - ④ **will not** do any partial matching when subsetting
- Subsetting tibbles can be done with either the `$` or `[[` operators



## 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 `is.data.frame()`
- An object can be coerced to a data frame using `as.data.frame()` or `tibble::as_data_frame()`



## 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]
  x
1 1
2 2
3 3

> myDataFrame[1, ]
  x  y  z
1 1 -4  A

> myDataFrame[c(1, 3)]
  x  z
1 1  A
2 2  B
3 3  C

> myDataFrame[2, 2]
[1] -5
```

```
> myDataFrame[, c(1, 3)]
  x  z
1 1  A
2 2  B
3 3  C

> myDataFrame["x"]
  x
1 1
2 2
3 3

> str(myDataFrame["x"]) # index as a list
'data.frame': 3 obs. of  1 variable:
 $ x: int  1 2 3

> str(myDataFrame[, "x"]) # index as a matrix
int [1:3] 1 2 3
```



## \$ for Data Frames

- \$ is a shorthand operator often used to access variables within a data frame

E.g. For the dataset `iris`, the code that returns the first three `Sepal.Length` values is `iris$Sepal.Length[1:3]`



## 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
2    2   -5    B
3    3   -6    C

# when the call returns more than one
#   column of a data frame, a value
#   of NULL is returned
> myDataFrame[["a", exact = F]]
NULL

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

```
# exact match to column name
> myDataFrame[["abd", exact = F]]
[1] -4 -5 -6

# partial matching b/c exact = F
#   call returns single column
#   therefore non-NULL return
> myDataFrame[["x", exact = F]]
[1] "A" "B" "C"

> 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
```



## Subsetting Operators

- We have seen the use of `[` to subset, but we can also use `[[` to subset
- Recall, when `[` is used to subset a list, it 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"
```



## \$ 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
```



## 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: ATOMIC VECTORS [EXAMPLE]

```
> 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: LISTS [EXAMPLE]

```
> 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: FACTORS [EXAMPLE]

```
> (myFactor <- factor(LETTERS[1:3]))
[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])
[1] "factor"

> myFactor[1, drop = TRUE] # drops unused levels
[1] A
Levels: A

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

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



## Simplifying vs. Preserving: MATRICES/ARRAYS [EXAMPLE]

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

> myMatrix[1, , drop = F]
   [,1] [,2] [,3]
[1,]    1    2    3

> str(myMatrix[1, , drop = F])
int [1:3] 1 2 3

> class(myMatrix[1, , drop = F])
[1] "matrix"

> typeof(myMatrix[1, , drop = F])
[1] "integer"
```

```
# if any dimension has length 1,
# dimension is dropped

> myMatrix[1, ]
[1] 1 2 3

> str(myMatrix[1, ])
int [1:3] 1 2 3

> class(myMatrix[1, ])
[1] "integer"

> typeof(myMatrix[1, ])
[1] "integer"

# why are
#  typeof(myMatrix[1, , drop = F])
#  and typeof(myMatrix[1, ])
#  both "integer" ?
```



## Simplifying vs. Preserving: DATA FRAMES [EXAMPLE] [1/2]

```
> (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] "list"

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

> str(myDataFrame[[1]])
int [1:3] 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
```



## Simplifying vs. Preserving: DATA FRAMES [EXAMPLE] [2/2]

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

> str(myDataFrame[, "x", drop = F])
'data.frame': 3 obs. of 1 variable:
 $ x: int 1 2 3

> typeof(myDataFrame[, "x", drop = F])
[1] "list"

> class(myDataFrame[, "x", drop = F])
[1] "data.frame"

> str(myDataFrame[, "x"])
int [1:3] 1 2 3

> typeof(myDataFrame[, "x"])
[1] "integer"

> class(myDataFrame[, "x"])
[1] "integer"

# if the output is a single column, returns a vector instead of a data frame
```



## TRY THIS

- 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.



## SOLUTION

1

```
> myMod$df.residual    # hint: str(myMod)
[1] 30

> summary(myMod)$r.squared  # hint: str(summary(myMod))
[1] 0.7528328

> summary(myMod)$adj.r.squared  # hint: str(summary(myMod))
[1] 0.7445939
```

2

```
> myDF[, sample(ncol(myDF), replace = F)]  # shuffle cols
```

3

```
> myDF[sample(nrow(myDF), replace = F), ]  # shuffle rows
```



## dplyr

- **dplyr** is a foundational package that facilitates data transformations for **data frames** (the **d** in **dplyr** refers to data frames)
- **n.b.** When loading **dplyr**, note that the base functions **filter()** and **lag()** are masked from the base R package **stats**
- To use functions **filter()** and **lag()** from the **stats** package, use **stats::filter()** and **stats::lag()**
- **dplyr** has a great number of useful functions, we will examine but a few of the most frequently used
- Don't forget to install and load **dplyr** before use



## Subsection 1

**dplyr**



## dplyr::filter() [ROW OPERATIONS]

A much simpler and intuitive way to subset data is to use the `filter()` function

```
> filter(mtcars, mpg > 32)
   mpg cyl disp hp drat    wt  qsec vs am gear carb
1 32.4    4 78.7 66 4.08 2.200 19.47  1  1     4    1
2 33.9    4 71.1 65 4.22 1.835 19.90  1  1     4    1

> filter(mtcars, mpg > 32, mpg < 33 )
   mpg cyl disp hp drat    wt  qsec vs am gear carb
1 32.4    4 78.7 66 4.08 2.2 19.47  1  1     4    1

> filter(mtcars, mpg > 32 & mpg < 33 )
   mpg cyl disp hp drat    wt  qsec vs am gear carb
1 32.4    4 78.7 66 4.08 2.2 19.47  1  1     4    1

> filter(mtcars, mpg > 32 & mpg < 33, wt > 2)
   mpg cyl disp hp drat    wt  qsec vs am gear carb
1 32.4    4 78.7 66 4.08 2.2 19.47  1  1     4    1
```



## dplyr::slice() [ROW OPERATIONS]

Select specific rows from a data frame

```
> head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5         1.4         0.2  setosa
2          4.9         3.0         1.4         0.2  setosa
3          4.7         3.2         1.3         0.2  setosa
4          4.6         3.1         1.5         0.2  setosa
5          5.0         3.6         1.4         0.2  setosa
6          5.4         3.9         1.7         0.4  setosa

> slice(iris, 3:5)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          4.7         3.2         1.3         0.2  setosa
2          4.6         3.1         1.5         0.2  setosa
3          5.0         3.6         1.4         0.2  setosa
```



## dplyr::select() [COLUMN OPERATIONS]

```
> glimpse(mtcars)
Observations: 32
Variables: 11
$ mpg <dbl> 21.0, 21.0, 22.8, 21.4, 18.7, 18.1, 14.3, 24.4, 22.8, 19.2, 17.8, ...
$ cyl  <dbl> 6, 6, 4, 6, 8, 6, 8, 4, 4, 6, 6, 8, 8, 8, 8, 8, 4, 4, 4, 4, 4, 8, ...
$ disp <dbl> 160.0, 160.0, 108.0, 258.0, 360.0, 225.0, 360.0, 146.7, 140.8, ...
$ hp   <dbl> 110, 110, 93, 110, 175, 105, 245, 62, 95, 123, 123, 180, 180, ...
$ drat <dbl> 3.90, 3.90, 3.85, 3.08, 3.15, 2.76, 3.21, 3.69, 3.92, 3.92, ...
$ wt   <dbl> 2.620, 2.875, 2.320, 3.215, 3.440, 3.460, 3.570, 3.190, 3.150, ...
$ qsec <dbl> 16.46, 17.02, 18.61, 19.44, 17.02, 20.22, 15.84, 20.00, 22.90, ...
$ vs   <dbl> 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, ...
$ am   <dbl> 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, ...
$ gear <dbl> 4, 4, 4, 3, 3, 3, 4, 4, 4, 4, 3, 3, 3, 3, 3, 4, 4, 4, 3, ...
$ carb <dbl> 4, 4, 1, 1, 2, 1, 4, 2, 2, 4, 4, 3, 3, 3, 4, 4, 4, 1, 2, 1, 1, ...

> glimpse(select(mtcars, mpg, cyl))
Observations: 32
Variables: 2
$ mpg <dbl> 21.0, 21.0, 22.8, 21.4, 18.7, 18.1, 14.3, 24.4, 22.8, 19.2, 17.8, ...
$ cyl  <dbl> 6, 6, 4, 6, 8, 6, 8, 4, 4, 6, 6, 8, 8, 8, 8, 8, 4, 4, 4, 4, 4, 8, ...
```



## dplyr::select() [COLUMN OPERATIONS][CONT'D]

**select(iris, contains("."))**

Select columns whose name contains a character string.

**select(iris, ends\_with("Length"))**

Select columns whose name ends with a character string.

**select(iris, everything())**

Select every column.

**select(iris, matches(".t."))**

Select columns whose name matches a regular expression.

**select(iris, num\_range("x", 1:5))**

Select columns named x1, x2, x3, x4, x5.

**select(iris, one\_of(c("Species", "Genus")))**

Select columns whose names are in a group of names.

**select(iris, starts\_with("Sepal"))**

Select columns whose name starts with a character string.

**select(iris, Sepal.Length:Petal.Width)**

Select all columns between Sepal.Length and Petal.Width (inclusive).

**select(iris, -Species)**

Select all columns except Species.



## Summarizing Data Using dplyr

The main `dplyr` functions used to summarize data are

- ① `summarise()`, which summarizes data into a single row of values
- ② `summarise_all()`, which applies multiple summary functions to each column



## dplyr::summarise()

- Applies function(s) to column(s) of data
- Following examples use `mammalSleep.csv`

```
> summarise(mammals, avgSleep = mean(sleep_total))
# A tibble: 1  1
  meanSleep
  <dbl>
1 10.43373

> summarise(mammals, avgSleep = mean(sleep_total), medSleep = median(sleep_total))
# A tibble: 1  2
  meanSleep medSleep
  <dbl>     <dbl>
1 10.43373    10.1

> summarise(mammals, avgSleep = mean(sleep_total),
  avgREM = mean(sleep_rem, na.rm = T))
# A tibble: 1  2
  meanSleep meanREM
  <dbl>     <dbl>
1 10.43373  1.87541
```





## dplyr::summarise\_all() [CONT'D]

- Multiple functions can be called on columns of data

E.g. Computing the variance and median of all columns of the iris data with column names that begin with the word Sepal

```
> summarise_all(select(iris, starts_with("Sepal")), funs(var, median))
   Sepal.Length_var Sepal.Width_var Sepal.Length_median Sepal.Width_median
1      0.6856935     0.1899794          5.8                  3
```



## dplyr::count()

- Counts the number of rows with each unique value of a variable
- Using the `hflights` dataset from the `hflights` package, compute the total number of flights per carrier

```
> count(hflights, UniqueCarrier)
# A tibble: 15 × 2
  UniqueCarrier     n
  <chr>      <int>
1 AA          3244
2 AS          365 
3 B6          695 
4 CO         70032
5 DL          2641
6 EV          2204
7 F9          838 
8 FL          2139
9 MQ          4648
10 OO         16061
11 UA          2072
12 US          4082
13 WN         45343
14 XE         73053
15 YV            79
```



## dplyr::group\_by()

- Groups data into rows with the same values
- Once data is identified as being grouped, it can be summarized by group

```
> summarise(group_by(mammals, vore), meanSleepTime = mean(sleep_total, na.rm = T))
# A tibble: 5 × 2
  vore    meanSleepTime
  <chr>      <dbl>
1 carni     10.378947
2 herbi     9.509375 
3 insecti   14.940000
4 omni     10.925000
5 <NA>     10.185714
```



## dplyr::group\_by()

- Groups data into rows with the same values
- Once data is identified as being grouped, it can be summarized by group

```
> summarise(group_by(mammals, vore), meanSleepTime = mean(sleep_total, na.rm = T))  
# A tibble: 5 × 2  
  vore    meanSleepTime  
  <chr>      <dbl>  
1 carni     10.378947  
2 herbi     9.509375  
3 insecti   14.940000  
4 omni     10.925000  
5 <NA>     10.185714
```

- Observe how cumbersome it is becoming to unpack the above code
- Thankfully, piping will help simplify our code



## A Brief Digression: `magrittr`

- `magrittr` is package which allows for the use of the piping operator `%>%`
- There are various piping operators, but for the purposes of this brief introduction, the focus will be solely on `%>%`; you are encouraged to read the `magrittr` documentation to learn more about the different piping operators
- Piping allows for a significant reduction in typing, as well as making code intuitive to external code reviewers
- Piping is very useful when using data manipulation packages such `tidyverse` and `dplyr`
- Piping operator keystroke: **SHIFT + COMMAND + M**



## A Brief Digression: magrittr [HOW IT WORKS]

- Data is piped into a function, and the data argument in the function can be dropped
  - Without the piping operator: `mean(x, na.rm = T)`
  - With the piping operator: `x %>% mean(na.rm = T)`

```
# w/o magrittr
> summarise(group_by(mammals, vore), meanSleepTime = mean(sleep_total, na.rm = T))

# -----
library(magrittr)

mammals %>%
  group_by(vore) %>%
  summarise(meanSleepTime = mean(sleep_total, na.rm = T))

# A tibble: 5 × 2
#   vore    meanSleepTime
#   <chr>      <dbl>
1 carni     10.378947
2 herbi     9.509375
3 insecti    14.940000
4 omni      10.925000
5 <NA>      10.185714
```



Using piping notation, **dplyr** functions and the **babynames** dataset from the **babynames** package,

## YOU TRY IT

- ① Compute the total number of births per year beginning in 1880, in 20-year increments; the output should be as follows

```
1 1880    201482
2 1900    450312
3 1920    2262732
...
```

- ② Compute the total number of **unique** names in each year beginning in 1880, in 20-year increments; the output should be as follows

```
1 1880    2000
2 1900    3731
3 1920    10756
...
```



## SOLUTION

1

```
babyNames %>%
  group_by(year) %>%
  summarise(totalBirthsPerYear = sum(n)) %>%
  slice(seq(1, nrow(.), by = 20))

# A tibble: 7 × 2
  year totalBirthsPerYear
  <dbl>          <int>
1 1880            201482
2 1900            450312
3 ...
4 ...
5 ...
6 ...
7 ...
```

2

```
babynames %>%
  count(year) %>%
  slice(seq(1, nrow(.), by = 20))

# A tibble: 7 × 2
  year   nn
  <dbl> <int>
1 1880    2000
2 1900    3731
3 ...
4 ...
5 ...
6 ...
7 ...
```



## dplyr::mutate()

- Computes **and** appends one or more new columns
- Using the **hflights** dataset from the **hflights** package, compute the percent of time spent airborne during a flight for each flight

```
myFlights <- hflights # create a local copy we can modify

myFlights <- myFlights %>%
  mutate(actualAirbornePct = AirTime / ActualElapsedTime)

# using a more advanced piping operator that pipes LHS data, but rather than
# returning the result of the entire chain, overwrites LHS with updated data

myFlights %<>%
  mutate(actualAirbornePct = AirTime / ActualElapsedTime)

glimpse(myFlights$actualAirbornePct)
num [1:227496] 0.667 0.75 0.686 0.557 0.71 ...
```



## Combine/Append Data Frames with rbind()

- When at least one data frame already exists, you can combine/append another data frame or a vector to the original data frame, but there are some caveats
  - Use `rbind()` to row-bind two data frames or a data frame with a vector

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

> 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, c(x = -99, y = "Zzz")))

      x     y
1    1     A
2    2     B
3    3     c
4 -99 <NA>

Warning message:
In `[<-factor`(`*tmp*`, ri, value = "Zzz") :
  invalid factor level, NA generated
```



## Combine/Append Data Frames rbind() [CONT'D]

- When at least one data frame already exists, you can combine/append another data frame or a vector to the original data frame, but there are some caveats
  - Use `rbind()` to row-bind a data frame with a vector

```
> (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, abc = -1:-3))
   x   y   z
1 1  98 1000
2 2  99 1001
3 3 100 1002
abc -1  -2  -3
```

n.b. behavior can quickly become quirky if the length of your vector is not equal to the number of columns in the data frame



## PREAMBLE

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

## TRY THIS

Based on the `myDataFrame_06` code, what happens if we replace `???` with:

- (a) `abc = -1`
- (b) `abc = -1:-2`
- (c) `abc = -1:-99`
- (d) `abc = c(-1, -2)`
- (e) `abc = c("-1", -2)`
- (f) `abc = c("a", -2, -3))`



## PREAMBLE

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

## SOLUTION

- (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**



## dplyr::bind\_rows

`bind_rows()` from the `dplyr` package the faster and more predictable cousin of `rbind()`, although it is not without its own quirks

- `bind_rows()` **only permits the binding of data frames to each other**; `rbind()` allows the binding of a vector to a data frame
- Even though it is part of the `tidyverse`, the result of a call to `bind_rows()` creates a data frame, not a tibble



## PREAMBLE

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

## TRY THIS

Based on the `myDataFrame_08` code, what happens if we replace `???` with:

- (a) `data.frame(-1)`
- (b) `data.frame(-1:-2)`
- (c) `data.frame(c(-1:-2))`
- (d) `data.frame(-1:-99)`
- (e) `data.frame("-1", -2)`
- (f) `data.frame(c("-1", -2))`



## SOLUTION

(a)

```
> (myDataFrame_06 <- bind_rows(myDataFrame_05, data.frame(-1)))
```

|   | x  | y   | z    | X.1 |
|---|----|-----|------|-----|
| 1 | 1  | 98  | 1000 | NA  |
| 2 | 2  | 99  | 1001 | NA  |
| 3 | 3  | 100 | 1002 | NA  |
| 4 | NA | NA  | NA   | -1  |

(b)

```
> (myDataFrame_06 <- bind_rows(myDataFrame_05, data.frame(-1:-2)))
```

|   | x  | y   | z    | X.1..2 |
|---|----|-----|------|--------|
| 1 | 1  | 98  | 1000 | NA     |
| 2 | 2  | 99  | 1001 | NA     |
| 3 | 3  | 100 | 1002 | NA     |
| 4 | NA | NA  | NA   | -1     |
| 5 | NA | NA  | NA   | -2     |

(c)

```
> (myDataFrame_06 <- bind_rows(myDataFrame_05, data.frame(c(-1:-2))))
```

|   | x  | y   | z    | c..1..2 |
|---|----|-----|------|---------|
| 1 | 1  | 98  | 1000 | NA      |
| 2 | 2  | 99  | 1001 | NA      |
| 3 | 3  | 100 | 1002 | NA      |
| 4 | NA | NA  | NA   | -1      |
| 5 | NA | NA  | NA   | -2      |



## SOLUTION [CONT'D]

(d)

```
> (myDataFrame_06 <- bind_rows(myDataFrame_05, data.frame(-1:-99)))  
   x     y     z X.1..99  
1  1    98  1000      NA  
2  2    99  1001      NA  
3  3   100  1002      NA  
4  NA   NA    NA     -1  
5  NA   NA    NA     -2  
...  
102 NA   NA    NA     -99
```

(e)

```
> (myDataFrame_06 <- bind_rows(myDataFrame_05, data.frame("-1", "-2)))  
   x     y     z X..1. X.2  
1  1    98  1000    <NA>  NA  
2  2    99  1001    <NA>  NA  
3  3   100  1002    <NA>  NA  
4  NA   NA    NA     -1  -2
```

(f)

```
> (myDataFrame_06 <- bind_rows(myDataFrame_05, data.frame(c("-1", "-2"))))  
   x     y     z c...1....2.  
1  1    98  1000    <NA>  
2  2    99  1001    <NA>  
3  3   100  1002    <NA>  
4  NA   NA    NA     -1  
5  NA   NA    NA     -2
```



## Combine/Append Data Frames with cbind()

- When at least one data frame already exists, you can combine/append another data frame or a vector to the original data frame, but there are some caveats
  - Use `cbind()` to column-bind two data frames or a data frame with a vector

```
> (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, z = -1:-3))
   x y  z
1 1 A -1
2 2 B -2
3 3 c -3
```



## Combine/Append Data Frames with cbind() [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
  - 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, abc = -1:-3))
   x     y     z abc
1 1    98 1000  -1
2 2    99 1001  -2
3 3   100 1002  -3
```



## PREAMBLE

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

## TRY THIS

Based on the `myDataFrame_08` code, what happens if we replace `???` with:

- (a) `abc = -1`
- (b) `abc = -1:-2`
- (c) `abc = -1:-99`
- (d) `abc = c("-1", -2)`
- (e) `abc = c("a", -2, -3))`



## PREAMBLE

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

## SOLUTION

- (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)



## dplyr::bind\_cols

`bind_cols()` from the `dplyr` package the faster and more predictable cousin of `cbind()`, although it is not without its own quirks

- `bind_cols()` only permits the binding of data frames to each other; `cbind()` allows the binding of a vector to a data frame
- Even though it is part of the `tidyverse`, the result of a call to `bind_cols()` creates a data frame, not a tibble



## PREAMBLE

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

## TRY THIS

Based on the `myDataFrame_08` code, what happens if we replace `???` with:

- (a) `abc = data.frame(-1)`
- (b) `abc = data.frame(-1:-2)`
- (c) `abc = data.frame(-1:-99)`
- (d) `abc = data.frame(c("-1", -2))`
- (e) `abc = data.frame(c("a", -2, -3))`



## SOLUTION

(a)

```
> bind_cols(myDataFrame_07, data.frame(abc = -1))  
Error: incompatible number of rows (1, expecting 3)
```

(b)

```
> bind_cols(myDataFrame_07, data.frame(abc = -1:-2))  
Error: incompatible number of rows (2, expecting 3)
```

(c)

```
> bind_cols(myDataFrame_07, data.frame(abc = -1:-99))  
Error: incompatible number of rows (99, expecting 3)
```

(d)

```
> bind_cols(myDataFrame_07, data.frame(abc = c("-1", -2)))  
Error: incompatible number of rows (2, expecting 3)
```

(e)

```
> bind_cols(myDataFrame_07, data.frame(abc = -1))  
Error: incompatible number of rows (1, expecting 3)
```

(f)

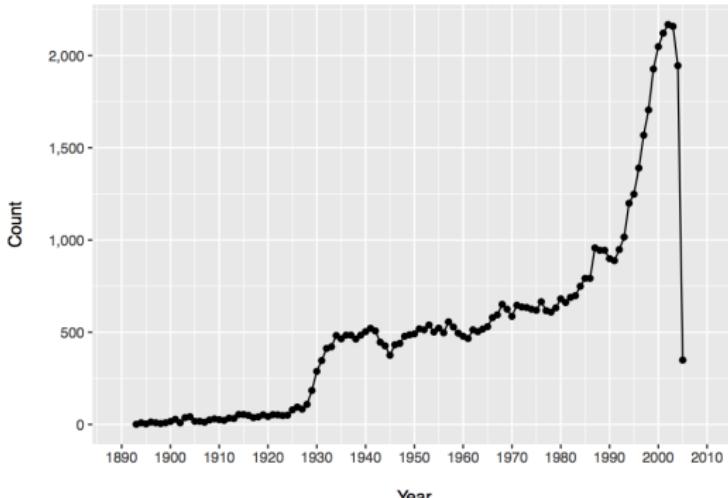
```
> bind_cols(myDataFrame_07, data.frame(abc = c("a", -2, -3)))  
   x   y   z  qqq  
1 1  98 1000    a  
2 2  99 1001   -2  
3 3 100 1002   -3
```



## LAB

Case Study in `dplyr`: `ggplot2` Movies*Paul Intrevado**July 19, 2017*

The `ggplot2movies` package contains a dataset named `movies`, which provides information on 58,788 movies, dating as far back as 1893! Here is a quick graphical summary of how many movies were released each year:





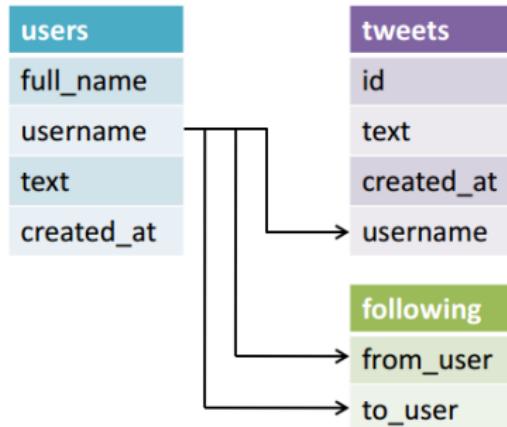
## Subsection 2

### Relational Data in R



## Relational Data with dplyr

- It often happens that the data being analyzed exists in multiple tables of data
- The difficulty arises when trying to combine these disparate tables into a single, cohesive table on which analyses can be conducted





## Relational Data with dplyr [CONT'D]

- A very common place for these various data tables is in a relational database, e.g., an SQL database (structured query language)
- Although R can connect directly and query from a variety of relational databases, this is beyond the scope of this course
- Using functions from the `dplyr` package, data frames can be joined to conduct additionally complex analyses
- There are three types of operations on relational data
  - mutating joins
  - filtering joins
  - set operations

**n.b.** Due to relational nature of the data examined in this section, the terms data frame/tibble and table will be used interchangeably



## Mutating Joins

- A mutating join combines variables from two tables based on a matched **key** value

E.g. To join the following two tables based on the key **x1**, there are a number of possibilities

| a  |    | b  |    |
|----|----|----|----|
| x1 | x2 | x1 | x3 |
| A  | 1  | A  | T  |
| B  | 2  | B  | F  |
| C  | 3  | D  | T  |

- Retain rows of both tables **only** for rows with matching keys
- Join matching rows from table **b** to **a** by matching key
- Join matching rows from table **a** to **b** by matching key
- Retain all values in all rows by matching key



## Mutating Joins [CONT'D]

| x1 | x2 | x3 |
|----|----|----|
| A  | 1  | T  |
| B  | 2  | F  |
| C  | 3  | NA |

**dplyr::left\_join(a, b, by = "x1")**

Join matching rows from b to a.

| x1 | x3 | x2 |
|----|----|----|
| A  | T  | 1  |
| B  | F  | 2  |
| D  | T  | NA |

**dplyr::right\_join(a, b, by = "x1")**

Join matching rows from a to b.

| x1 | x2 | x3 |
|----|----|----|
| A  | 1  | T  |
| B  | 2  | F  |

**dplyr::inner\_join(a, b, by = "x1")**

Join data. Retain only rows in both sets.

| x1 | x2 | x3 |
|----|----|----|
| A  | 1  | T  |
| B  | 2  | F  |
| C  | 3  | NA |
| D  | NA | T  |

**dplyr::full\_join(a, b, by = "x1")**

Join data. Retain all values, all rows.



## Sample Customer/Transaction Tibbles for Example Joins

```
> (custTbl <- read_csv("mutatingJoins_ex01_table_01.csv"))
# A tibble: 6 × 4
  cust_id first_name   last_name gender
    <int>     <chr>      <chr>   <chr>
1 149201      Starla Godilington     F
2 136127       Barris   Stivani     M
3 389922      Rubetta Schermick     F
4 836399      Elnore    Upham     F
5 83025        Merrel   Braney     M
6 763354      Fosser    Ian       M

> (transactionTbl <- read_csv("mutatingJoins_ex01_table_02.csv"))
# A tibble: 6 × 5
  id creditCardNum creditCardTypes transactionAmt cust_id
    <int>        <dbl>          <chr>           <chr>    <int>
1 1  3.575026e+15          jcb            $3.82    149201
2 2  3.553533e+15          jcb            $7.90    136127
3 3  3.050191e+13         blanche          $9.18    389922
4 4  4.912087e+12          visa            $2.91    836399
5 5  3.560840e+15          jcb            $9.94    83025
6 6  4.565765e+15          visa            $6.22    39021
```

- How much did each customer spend?



## Sample Customer/Transaction Tibbles for Example Joins

|   | cust_id | first_name | last_name   | gender |
|---|---------|------------|-------------|--------|
|   | <int>   | <chr>      | <chr>       | <chr>  |
| 1 | 149201  | Starla     | Godilington | F      |
| 2 | 136127  | Barris     | Stivani     | M      |
| 3 | 389922  | Rubetta    | Schermick   | F      |
| 4 | 836399  | Elnore     | Upham       | F      |
| 5 | 83025   | Merrel     | Braney      | M      |
| 6 | 763354  | Fosser     | Ian         | M      |

|   | id    | creditCardNum | creditCardTypes | transactionAmt | cust_id |
|---|-------|---------------|-----------------|----------------|---------|
|   | <int> | <dbl>         | <chr>           | <chr>          | <int>   |
| 1 | 1     | 3.575026e+15  | jcb             | \$3.82         | 149201  |
| 2 | 2     | 3.553533e+15  | jcb             | \$7.90         | 136127  |
| 3 | 3     | 3.050191e+13  | blanche         | \$9.18         | 389922  |
| 4 | 4     | 4.912087e+12  | visa            | \$2.91         | 836399  |
| 5 | 5     | 3.560840e+15  | jcb             | \$9.94         | 83025   |
| 6 | 6     | 4.565765e+15  | visa            | \$6.22         | 39021   |

- `cust_id` 763354 exists in `custTbl` but does not have any transactions recorded in `transactionTbl`
- `id == 6` in `transactionTbl` indicates a transaction by `cust_id` 39021, but `cust_id` 39021 does not exist in `custTbl`



## dplyr::inner\_join()

- Inner joins join **only** the rows of two data frames that share a matched key

```
inner_join(custTbl, transactionTbl, by = "cust_id") %>% select(-id, -creditCardNum)

# A tibble: 5 × 6
  cust_id first_name   last_name gender creditCardTypes transactionAmt
    <int>     <chr>       <chr>   <chr>        <chr>          <chr>
1 149201      Starla     Godilington     F           jcb        $3.82
2 136127      Barris     Stivani       M           jcb        $7.90
3 389922      Rubetta   Schermick      F           blanche     $9.18
4 836399      Elnore    Upham        F           visa       $2.91
5 83025       Merrel    Braney        M           jcb        $9.94
```

- The same result could have been obtained without using a join, but it would have been far more lengthy and cumbersome

**n.b.** Tibbles were joined on `cust_id` key, therefore the record in `custTbl` with `cust_id == 763354` has been omitted from the join, as well as the record in `custTbl` which does not have any transactions associated with `cust_id == 763354`



## Outer Joins

Whereas **inner** joins keep rows that have key matches in both tables being joined, **outer** joins keep rows that exist in one or the other (or both) tables being joined.

| a  |    | b  |    |
|----|----|----|----|
| x1 | x2 | x1 | x3 |
| A  | 1  | A  | T  |
| B  | 2  | B  | F  |
| C  | 3  | D  | T  |

Outer joins come in three variants (assume the key is **x1**):

- A **left** join would keep all rows in **a**
- A **right** join would keep all rows in **b**
- A **full** join would keep all rows in both **a** and **b**



## dplyr::left\_join()

```
left_join(custTbl, transactionTbl, by = "cust_id") %>% select(-id, -creditCardNum)

# A tibble: 6 × 6
  cust_id first_name   last_name gender creditCardTypes transactionAmt
    <int>     <chr>      <chr>    <chr>        <chr>        <chr>
1 149201      Starla   Godilington     F          jcb       $3.82
2 136127      Barris    Stivani      M          jcb       $7.90
3 389922      Rubetta  Schermick     F        blanche      $9.18
4 836399      Elnore   Upham       F          visa      $2.91
5 83025       Merrel   Braney       M          jcb       $9.94
6 763354      Fosser    Ian         M        <NA>       <NA>
```

- All rows in `custTbl` have been retained
- Where there were no matches for `cust_id` from `custTbl` to `transactionTbl`, `<NA>`s were inserted
- Where there were no matches for `cust_id` from `transactionTbl` to `custTbl`, rows were omitted



## dplyr::right\_join()

```
right_join(custTbl, transactionTbl, by = "cust_id") %>% select(-id, -creditCardNum)

# A tibble: 6 × 6
  cust_id first_name   last_name gender creditCardTypes transactionAmt
    <int>     <chr>      <chr>    <chr>        <chr>        <chr>
1 149201      Starla   Godilington     F          jcb       $3.82
2 136127      Barris    Stivani      M          jcb       $7.90
3 389922      Rubetta  Schermick     F        blanche     $9.18
4 836399      Elnore    Upham       F          visa      $2.91
5 83025       Merrel    Braney       M          jcb       $9.94
6 39021        <NA>      <NA>      <NA>        visa      $6.22
```

- All rows in `transactionTbl` have been retained
- Where there were no matches for `cust_id` from `transactionTbl` to `custTbl`, `<NA>`s were inserted
- Where there were no matches for `cust_id` from `custTbl` to `transactionTbl`, rows were omitted



## left\_join() vs. right\_join()

The following code snippets generate identical output, by rearranging the order of the arguments passed to the \*\_join's

```
> left_join(transactionTbl, custTbl, by = "cust_id") %>%
  select(-id, -creditCardNum)
# A tibble: 6 × 6
  creditCardTypes transactionAmt cust_id first_name last_name gender
  <chr>           <chr>     <int>      <chr>      <chr>   <chr>
1 jcb             $3.82     149201    Starla     Godilington F
2 jcb             $7.90     136127    Barris     Stivani   M
3 blanche        $9.18     389922    Rubetta   Schermick F
4 visa            $2.91     836399    Elnore     Upham    F
5 jcb             $9.94     83025     Merrel     Braney   M
6 visa            $6.22     39021     <NA>       <NA>     <NA>

> right_join(custTbl, transactionTbl, by = "cust_id") %>%
  select(-id, -creditCardNum)
# A tibble: 6 × 6
  cust_id first_name last_name gender creditCardTypes transactionAmt
  <int>      <chr>      <chr>   <chr>      <chr>           <chr>
1 149201    Starla     Godilington F          jcb           $3.82
2 136127    Barris     Stivani   M          jcb           $7.90
3 389922    Rubetta   Schermick F          blanche      $9.18
4 836399    Elnore     Upham    F          visa          $2.91
5 83025     Merrel     Braney   M          jcb           $9.94
6 39021     <NA>       <NA>     <NA>       visa          $6.22
```



## dplyr::full\_join()

```
full_join(custTbl, transactionTbl, by = "cust_id") %>% select(-id, -creditCardNum)

# A tibble: 7 × 6
  cust_id first_name   last_name gender creditCardTypes transactionAmt
    <int>      <chr>       <chr>   <chr>        <chr>          <chr>
1 149201      Starla     Godilington     F           jcb        $3.82
2 136127      Barris     Stivani       M           jcb        $7.90
3 389922      Rubetta    Schermick      F           blanche     $9.18
4 836399      Elnore     Upham        F           visa       $2.91
5 83025       Merrel     Braney        M           jcb        $9.94
6 763354      Fosser     Ian          M           <NA>      <NA>
7 39021       <NA>       <NA>       <NA>       visa       $6.22
```

- A full join retains all rows from both tables, inserting <NA>s where there were no `cust_id` matches



## Filtering Joins and Set Operations

`dplyr` offers other functions that may help in manipulating relational data

- `semi_join(a, b)` **keeps** only rows in `a` that match with `b`  
(but variables from `b` are not included)
- `anti_join(a, b)` **omits** only rows in `a` that match with `b`  
(but variables from `b` are not included)
- `intersect(a, b)` returns rows that are in **both** `a` and `b`
- `union(a, b)` returns rows that are in `a` or `b` **or both**
- `setdiff(a, b)` returns rows that are in `a` but not in `b`



# LAB

## LAB: dplyr and \*\_joins

First we download the JSON data set `dplyr_joins_lab.json` and explore it.

This is nested JSON data, which requires a little more code to flatten into a workable data frame.

We want to explore, among other things, the classes of purchases made by customer, so we will import join `classes.csv` to our JSON data we might get more descriptive data on purchases.

We can now explore our data and answer some interesting questions.

The categories, ordered by frequency of purchase, arranged in descending order and excluding NAs are

| Class            | Frequency |
|------------------|-----------|
| cleaningSupplies | 748       |
| netSupplies      | 736       |

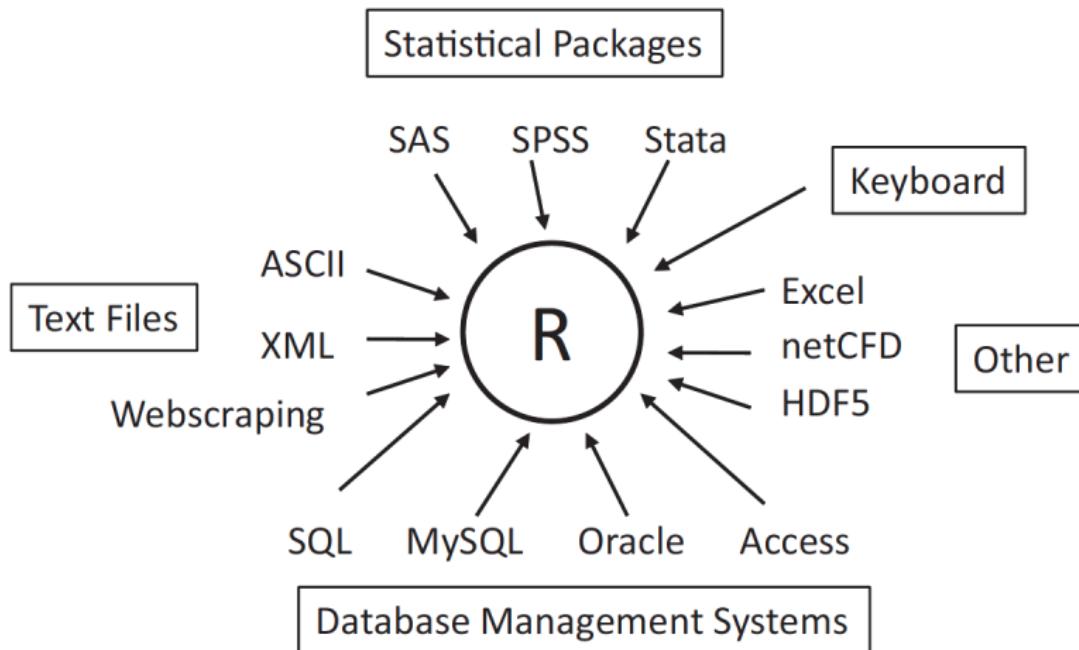


## Subsection 3

### Importing Data



## R can Import Data from a Multitude of Sources



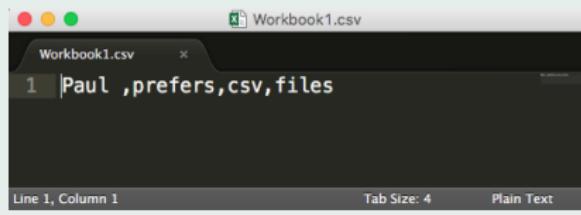


# Why csv Files?

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

csv File (.csv)

23 bytes



A screenshot of a text editor window titled "Workbook1.csv". The window shows a single line of text: "1 |Paul ,prefers,csv,files". The status bar at the bottom indicates "Line 1, Column 1", "Tab Size: 4", and "Plain Text".



# Why csv Files?

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

csv File (.csv)  
23 bytes

Workbook1.csv

1 |Paul ,prefers,csv,files

Line 1, Column 1

Tab Size: 4 Plain Text

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

Home Insert Page Layout Formulas Data

Cut Copy Format

G3

|   |      |         |     |       |
|---|------|---------|-----|-------|
| 1 | Paul | prefers | csv | files |
| 2 |      |         |     |       |
| 3 |      |         |     |       |



## 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 delimited text 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- (`.tsv`) and space-delimited files
  - these files may alternatively have a `.txt` file extension



## 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 = ","`



## readr

Part of the `tidyverse`, the `readr` package offers the ability to read data into R far more quickly and conveniently than base R functions

- `read_csv()` reads comma-delimited files
- `read_csv2()` reads semicolon-separated files
- `read_tsv()` reads tab-delimited files
- `read_delim()` reads files in any types of delimiter
- `read_tsv()` reads fixed-width files
- These functions have been shown to be  $\sim 10\times$  faster than their base R counterparts, and also produce tibbles instead of data frames (and therefore don't automatically convert character vectors to factors)
- Need even more speed? Try `fread()` from the `data.table` package



## Beyond Delimited Files

- **haven** is a great package to read SPSS, Stata and SAS files (which can get messy)
- An alternative package to read in Excel files is **readxl**
- **DBI** facilitates writing SQL queries for a backend database
- **jsonlite** is great when working with JSON data



## Subsection 4

### 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
```



## 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?

```
> unique(mtcars$cyl)  
[1] 6 4 8
```

- A similar function, `dplyr::distinct()`, will remove all duplicate rows from a data frame



## which()

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

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

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



## paste()

- Converts arguments to characters and then concatenates them

```
> library(magrittr)

> iris %>%
  dplyr::filter(Sepal.Length >= 7.7) %>%
  dplyr::mutate(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"
```



## 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(from = 1, by = 7, length.out = 10)
[1] 1 8 15 22 29 36 43 50 57 64
```



## 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"
```



## 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
```

- Passing a data structure to **anyNA()** will return a single **logical** indicating whether or not **NAs** are in said data structure; this is an alternative to the equivalent compound statement **any(is.na())**



## 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



## Section 5

# 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



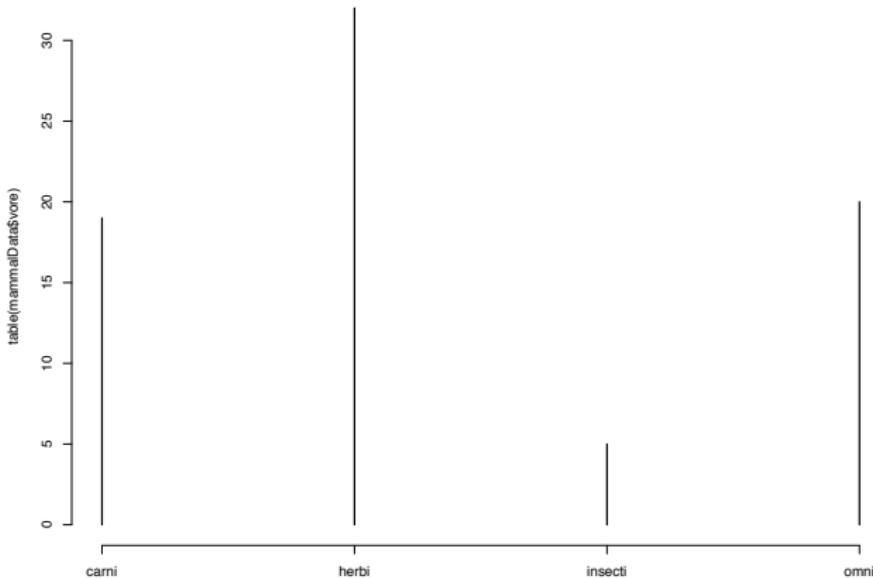
## Graphics in Base R

- The base R graphics package, while being quick to code and functional, is not particularly aesthetically pleasing, is syntactically cumbersome, and is far 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() [DYNAMIC]

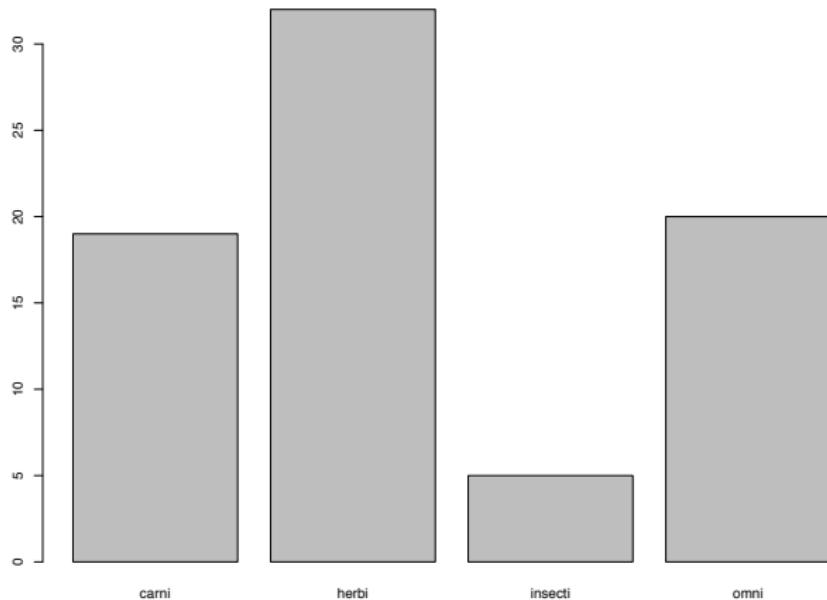
```
> plot(table(mammalData$vore)) ## mammalSleep.csv
```





## barplot() [BAR GRAPH]

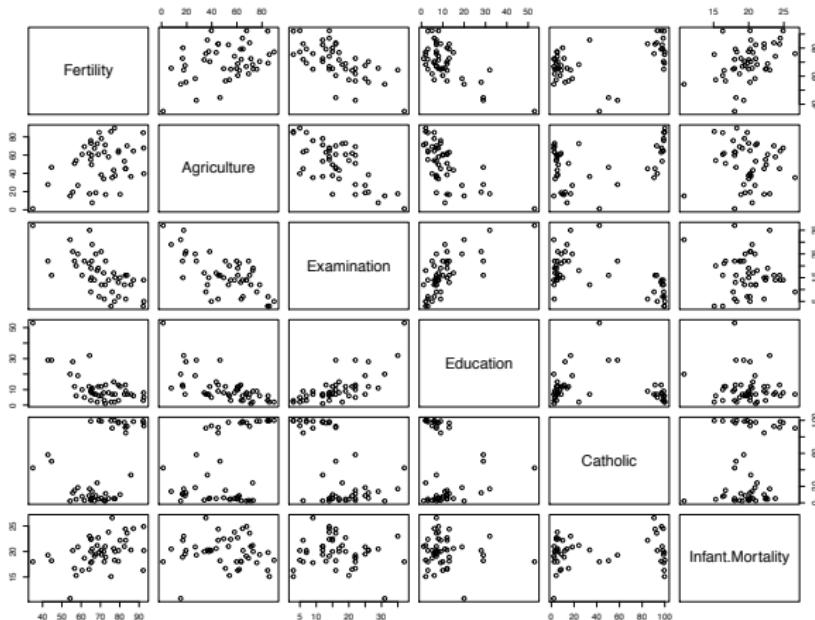
```
> barplot(table(mammalData$vore)) ## mammalSleep.csv
```





## plot() [SCATTER PLOT MATRIX]

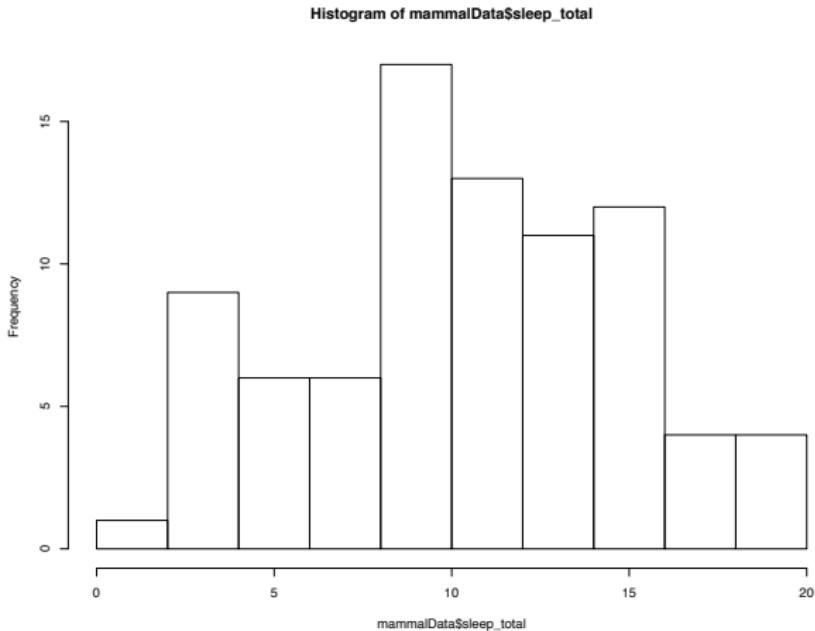
```
> plot(swiss) ## Base R Data
```





## hist() [HISTOGRAM]

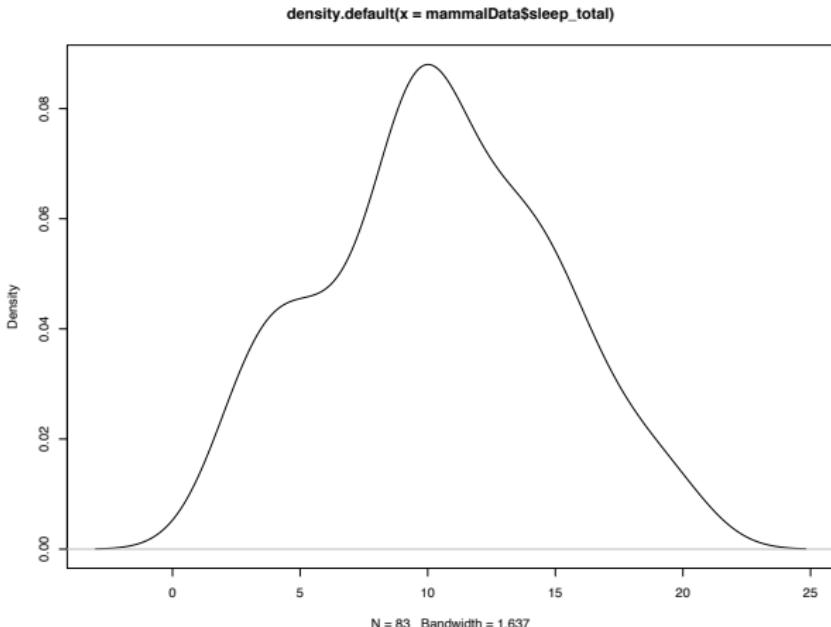
```
> hist(mammalData$sleep_total) ## mammalSleep.csv
```





## plot() [KERNEL DENSITY PLOT]

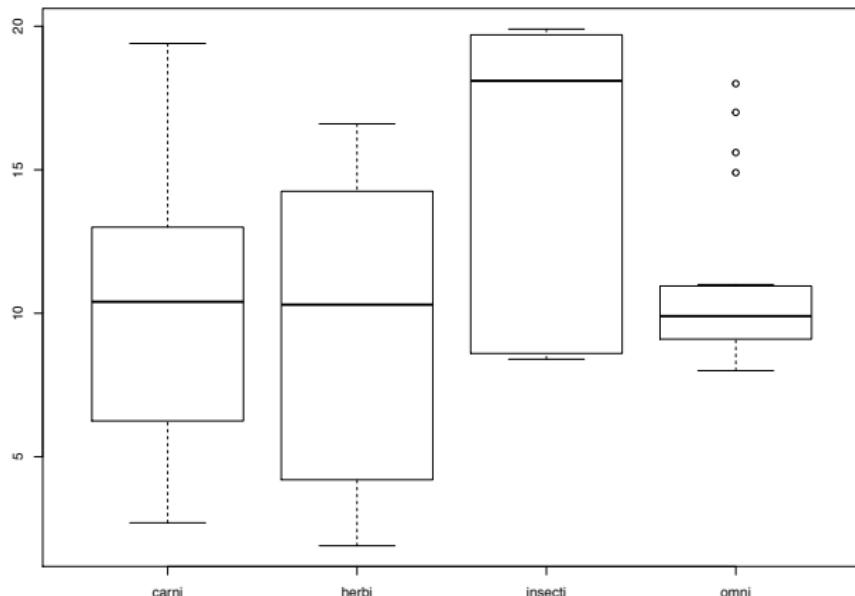
```
> plot(density(mammalData$sleep_total)) ## mammalSleep.csv
```





## boxplot() [BOX PLOT]

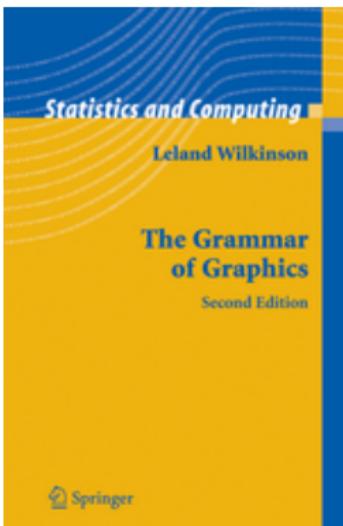
```
> boxplot(sleep_total ~ vore, data = mammalData) ## mammalSleep.csv
```





## 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]

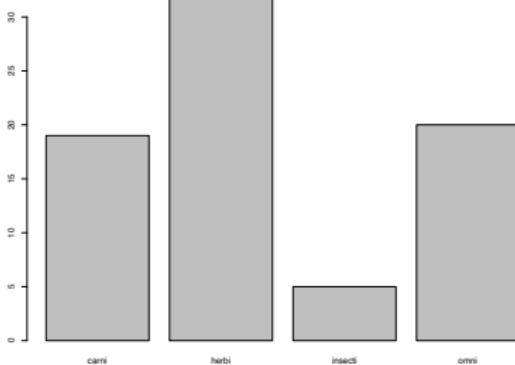
- This grammar of graphics is composed of a set of independent components that can be composed in many different ways
- This makes What is `ggplot2` very powerful, because you are not limited to a set of pre-specified graphics, but you can create new graphics that are precisely tailored to your problem
- Do not be fooled: even though `ggplot2` has a high level of flexibility and complexity, publication-quality graphs can be coded in seconds, with many of the extraneous details (e.g., legends) taken care of by `ggplot2` default behavior



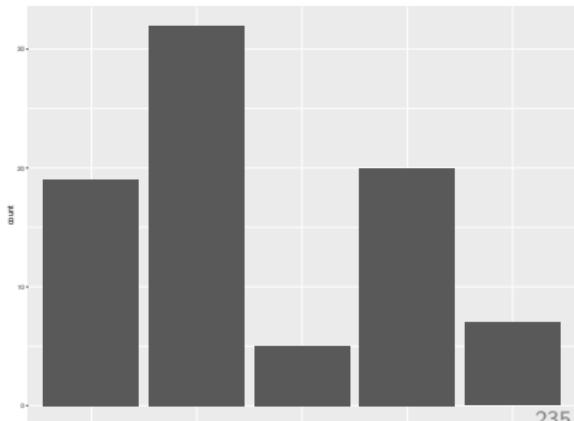
## qplot() versus plot()

- `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

```
> barplot(table(mammalData$vore))
```



```
> qplot(mammalData$vore)
```





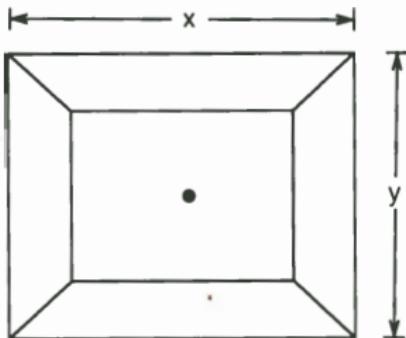
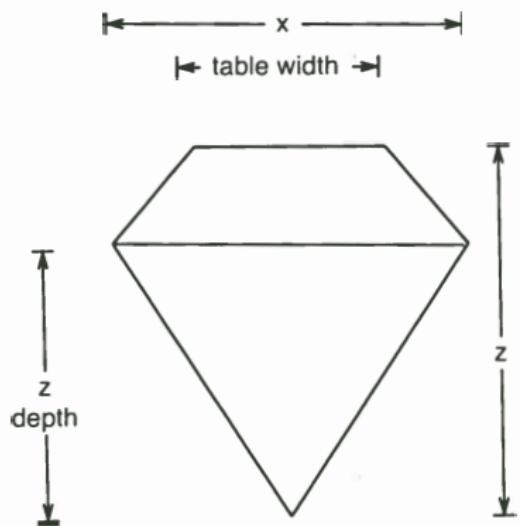
## diamonds dataset

A data frame with 53940 rows and 10 variables:

- price: price in US dollars ( $\$326\text{--}\$18,823$ )
- 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)



## iamonds dataset [CONT'D]



$$\begin{aligned} \text{depth} &= z \text{ depth} / z * 100 \\ \text{table} &= \text{table width} / x * 100 \end{aligned}$$



## A First Example in ggplot

```
> library(tidyverse)
> library(magrittr)

> set.seed(101)

> diamondsSubset <- diamonds[sample(nrow(diamonds), 200), ]

> diamondPlot <- diamondsSubset %>% ggplot(aes(x = carat, y = price))
# the above line of code does not generate a graph

> summary(diamondPlot)

data: carat, cut, color, clarity, depth, table, price, x, y, z [200x10]
mapping: x = carat, y = price
faceting: <ggproto object: Class FacetNull, Facet>
...
...
...
```



# ggplot Geometric Objects

| Geoms - Use a geom to represent data points, use the geom's aesthetic properties to represent variables. Each function returns a layer.   |  |  |
|---|--|--|
| One Variable  | Two Variables  | Three Variables  |
| <b>Continuous</b><br><pre>a + geom_area(stat = "bin")</pre><br><pre>a + geom_density(kernel = "gaussian")</pre><br><pre>a + geom_dotplot(binwidth = .05)</pre><br><pre>a + geom_freqpoly(binwidth = 1)</pre><br><pre>a + geom_histogram(binwidth = 5)</pre><br><b>Discrete</b><br><pre>b &lt;- ggplot(mpg, aes(fct))</pre><br><pre>b + geom_bar()</pre> <p><code>x, alpha, color, fill, linetype, size, weight</code></p> | <b>Continuous X, Continuous Y</b><br><pre>f + geom_blank()</pre><br><pre>f + geom_jitter()</pre><br><pre>f + geom_point()</pre><br><pre>f + geom_quantile()</pre><br><pre>f + geom_rug(sides = "tl")</pre><br><pre>f + geom_smooth(model = lm)</pre><br><pre>f + geom_text(label = cyl)</pre><br><b>Continuous Bivariate Distribution</b><br><pre>i &lt;- ggplot(movies, aes(year, rating))</pre><br><pre>i + geom_bin2d(binwidth = c(.5, .5))</pre><br><pre>i + geom_density2d()</pre><br><pre>i + geom_hex()</pre><br><b>Continuous Function</b><br><pre>j &lt;- ggplot(economics, aes(date, unemployment))</pre><br><pre>j + geom_area()</pre><br><pre>j + geom_line()</pre><br><pre>j + geom_step(direction = "hv")</pre><br><b>Visualizing error</b><br><pre>df &lt;- data.frame(group = c("A", "B"), fit = 4.5, se = 1.2)</pre> <pre>k &lt;- ggplot(df, aes(group, fit, ymin = fit - se, ymax = fit + se))</pre><br><pre>k + geom_crossbar()</pre><br><pre>k + geom_errorbar()</pre><br><pre>k + geom_linerange()</pre><br><pre>k + geom_pointrange()</pre><br><b>Maps</b><br><pre>data &lt;- data.frame(murder = USArestates\$Murder,</pre> <pre>state = tolower(rownames(USArestates)))</pre> <pre>map &lt;- ggplot(data, aes(state))</pre> <pre>i &lt;- ggplot(data, aes(state) + murder)</pre> <pre>i + geom_map(aes(map_id = state), map = map) +</pre> <pre>expand_limits(map_id = map\$state) +</pre> <pre>map_id, alpha, color, fill, linetype, size</pre> | <b>Three Variables</b><br><pre>m &lt;- width(seals, sqrt(delta_long^2 + delta_lat^2))</pre> <pre>m &lt;- ggplot(seals, aes(long, lat))</pre><br><pre>m + geom_segment(aes(xend = long + delta_long,</pre> <pre>yend = lat + delta_lat))</pre><br><pre>m + geom_rect(aes(xmin = long, ymin = lat,</pre> <pre>xmax = long + delta_long,</pre> <pre>ymax = lat + delta_lat))</pre><br><pre>m + geom_contour(aes(z = z))</pre><br><pre>x, y, z, alpha, color, linetype, size, weight</pre> |



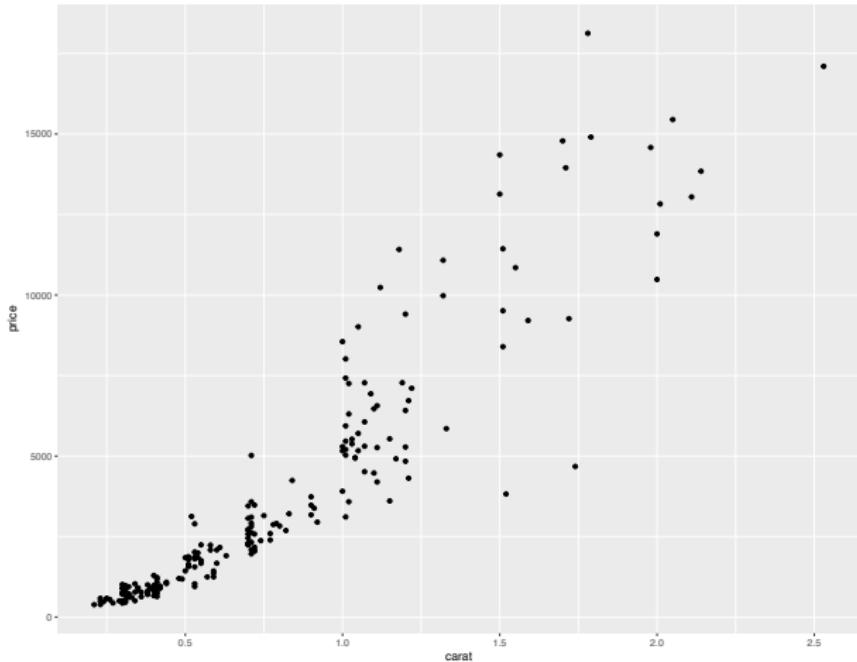
## ggplot Geometric Objects

- 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
  - ➊ `geom_histogram`
  - ➋ `geom_freqpoly`
  - ➌ `geom_density`
  - ➍ `geom_bar`
- Common two-dimensional geoms include
  - ➊ `geom_point`
  - ➋ `geom_boxplot`
  - ➌ `geom_path` (line in any direction)
  - ➍ `geom_line` (line from left to right)
  - ➎ `geom_smooth` (smoothed line with standard error)



## A First Example in ggplot [REVISITED]

```
diamondsSubset %>% ggplot(aes(x = carat, y = price)) + geom_point()
```





# 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()`



## Generic ggplot Syntax

```
> ggplot(data = <myData>) + <geom_*>(mapping = aes(<myPreferredMapping>))  
...OR, with piping operator...  
> <myData> %>% ggplot() + <geom_*>(mapping = aes(<myPreferredMapping>))
```



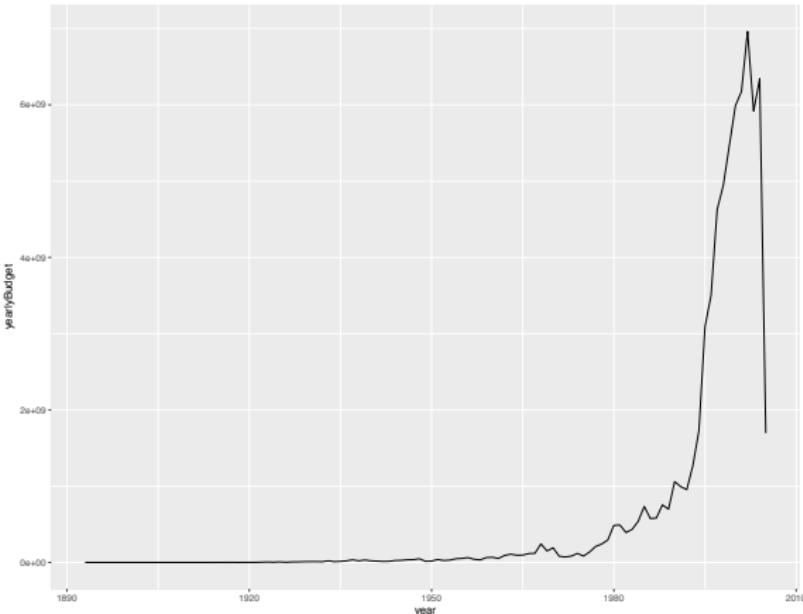
## aes(): Aesthetic Mappings

- To map variables to different parts of the plot, we use the `aes()` function, which takes a list of aesthetic-variable pairs
- For a one-variable graph, e.g., a histogram, the minimal information required to be supplied to `aes(x = <myXdata>)`
- In a two variable graph, aesthetic information required include `aes(x = <myXdata>, y = <myYdata>)`
- Aesthetics can be mapped at the `ggplot()` level or at the `geom_*`()



## aes(): Aesthetic Mappings [EXAMPLE 1/2]

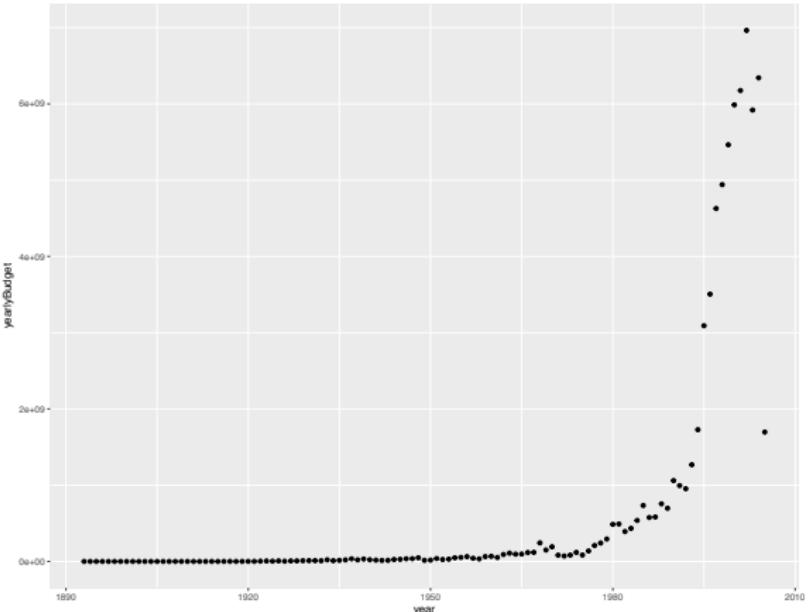
```
> ggplot2movies::movies %>% group_by(year) %>%  
  summarize(yearlyBudget = sum(as.numeric(budget), na.rm = T)) %>%  
  ggplot() + geom_line(aes(x = year, y = yearlyBudget))
```





## aes(): Aesthetic Mappings [EXAMPLE 2/2]

```
> ggplot2movies::movies %>% group_by(year) %>%  
  summarize(yearlyBudget = sum(as.numeric(budget), na.rm = T)) %>%  
  ggplot() + geom_point(aes(x = year, y = yearlyBudget))
```





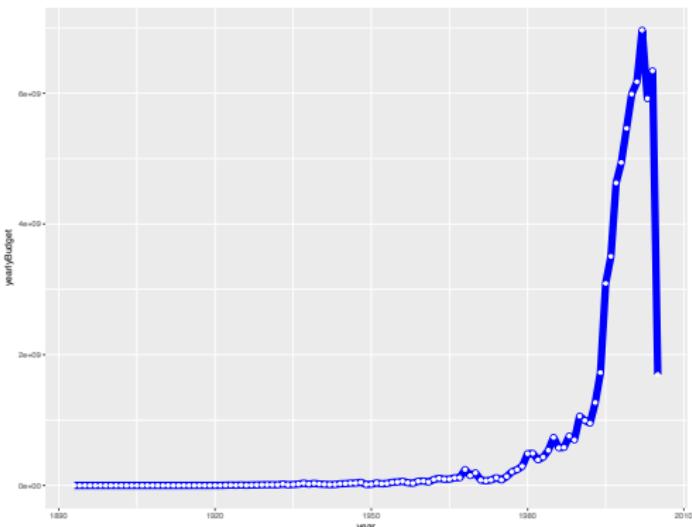
## Layers

- Perhaps the most powerful aspect of `ggplot()` is the ability to construct graphics from layers
- Creating the `ggplot()` object creates a base graphical object
- Creating a `geom` or layering multiple `geoms` provides the user an empty canvas with which to create highly-stylized graphics to convey information
- The order in which `geoms` are layered will affect the output



## Layering geoms [EXAMPLE 1/3]

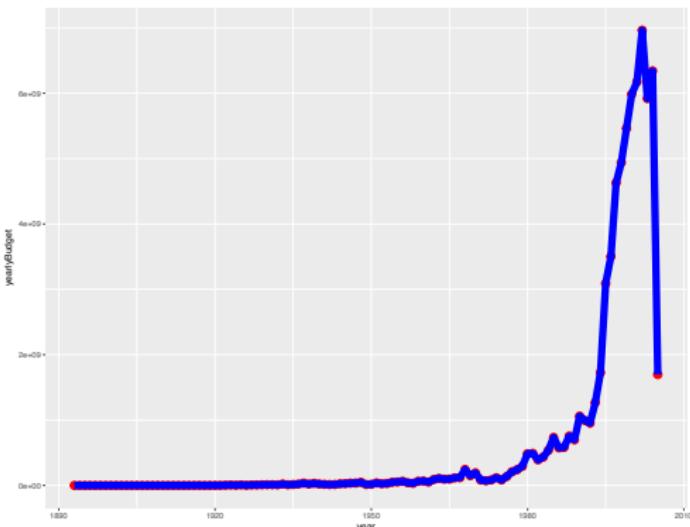
```
> ggplot2movies::movies %>%  
  group_by(year) %>% summarize(yearlyBudget = sum(as.numeric(budget),  
  na.rm = T)) %>% ggplot() +  
  geom_line(aes(x = year, y = yearlyBudget), color = "blue", size = 4) +  
  geom_point(aes(x = year, y = yearlyBudget))
```





## Layering geoms [EXAMPLE 2/3]

```
> ggplot2movies::movies %>%  
  group_by(year) %>% summarize(yearlyBudget = sum(as.numeric(budget),  
  na.rm = T)) %>% ggplot() +  
  geom_point(aes(x = year, y = yearlyBudget), color = "red", size = 4) +  
  geom_line(aes(x = year, y = yearlyBudget), color = "blue", size = 4)
```





## Layering geoms [EXAMPLE 3/3]

```
# create DF of mean yearly budgets
meanYearlyBudget <- ggplot2movies::movies %>%
  group_by(year) %>% summarize(meanYearlyBudget = mean(as.numeric(budget),
    na.rm = T))

# create DF of min/max for each year
summaryStats <- ggplot2movies::movies %>%
  group_by(year) %>% summarize(minBudget = min(as.numeric(budget), na.rm = T),
    maxBudget = max(as.numeric(budget), na.rm = T))

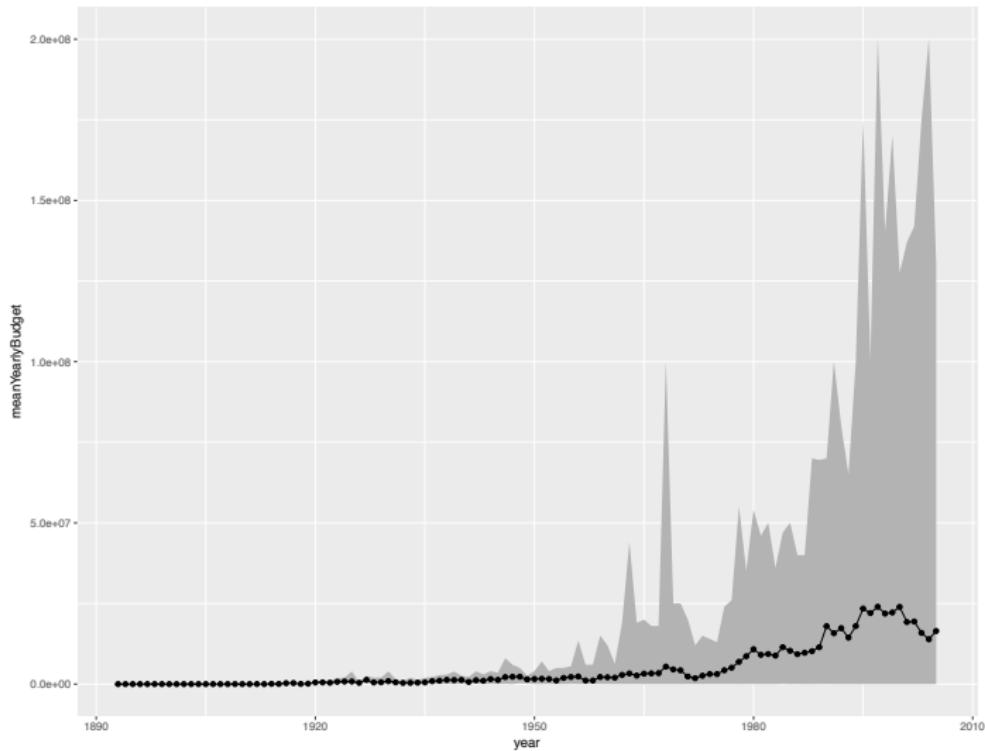
# remove NaNs from DF
meanYearlyBudget$meanYearlyBudget[is.nan(meanYearlyBudget$meanYearlyBudget)] <- 0

# remove Inf/-Inf from DF
summaryStats[summaryStats == Inf | summaryStats == -Inf] <- 0

# plot mean line (line + point) with underlying ribbon for min/max
# ordering of layers is important here
ggplot() +
  geom_ribbon(data = summaryStats, aes(x = year, ymin = minBudget,
    ymax = maxBudget), fill = "grey70") +
  geom_point(data = meanYearlyBudget, aes(x = year, y = meanYearlyBudget)) +
  geom_line(data = meanYearlyBudget, aes(x = year, y = meanYearlyBudget))
```



## Layering geoms [EXAMPLE 3/3] [CONT'D]





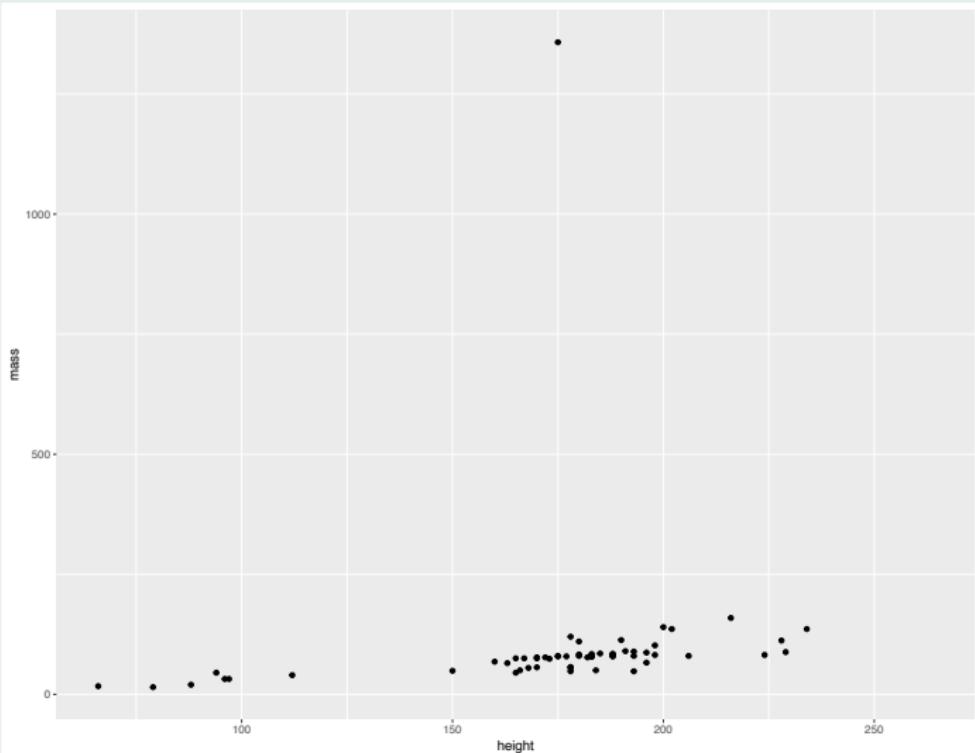
## TRY THIS

Is there a relationship between `height` and `mass` in the star wars universe?

- ① a
- ② Using `dplyr::starwars`, create a scatter plot in `ggplot` with  $x = \text{height}$  ,  $y = \text{mass}$
- ③ Create the same graph as but on this one, highlight all of the characters from Tatooine in red

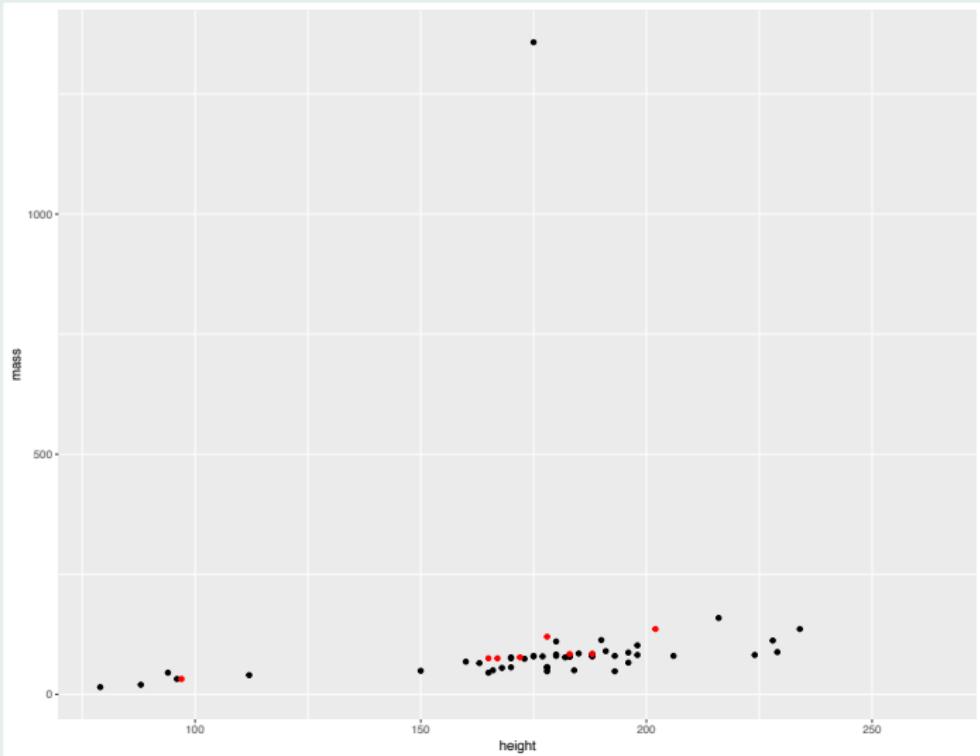


## FINAL OUTPUT 1





## FINAL OUTPUT 2





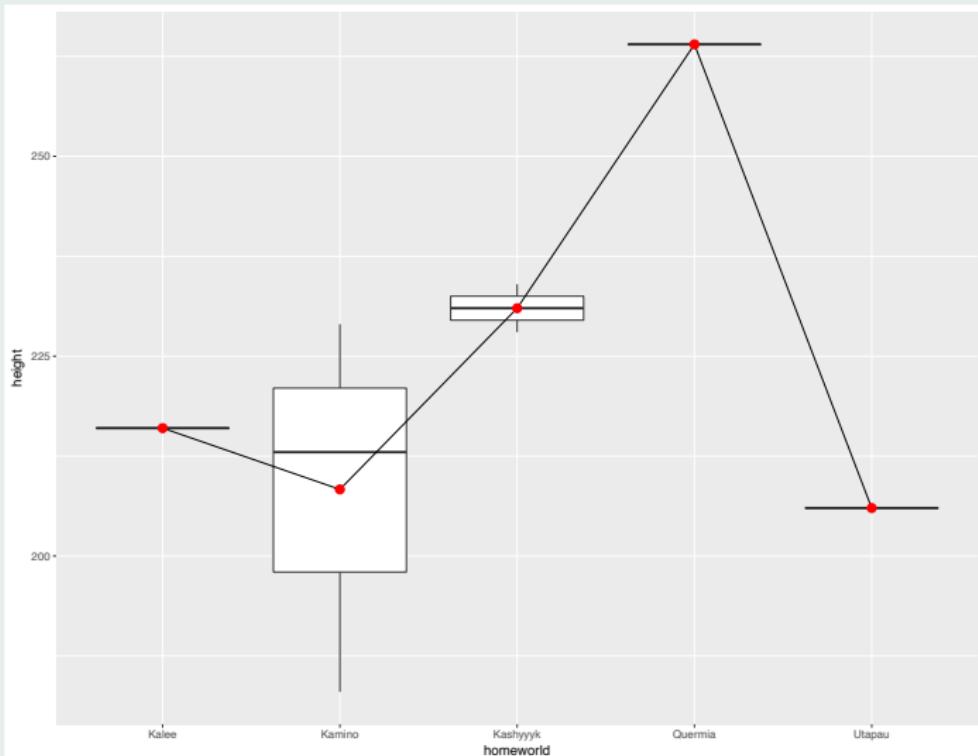
## TRY THIS

Using `dplyr::starwars`, create a graph in `ggplot` with

- ① Find the planets with the top five mean heights
- ② Create a box plot of the distributions of those heights
- ③ Overlay a line connecting the mean heights of those five planets (*hint: use group = 1*)



## FINAL OUTPUT





## SOLUTION

```
df1 <- dplyr::starwars %>%
  select(height, homeworld) %>%
  group_by(homeworld) %>%
  summarise(meanHeight = mean(height, na.rm = T)) %>%
  arrange(desc(meanHeight)) %>%
  ungroup() %>%
  slice(1:5)

df2 <- dplyr::starwars %>%
  select(height, homeworld) %>%
  filter(homeworld %in% x[["homeworld"]])

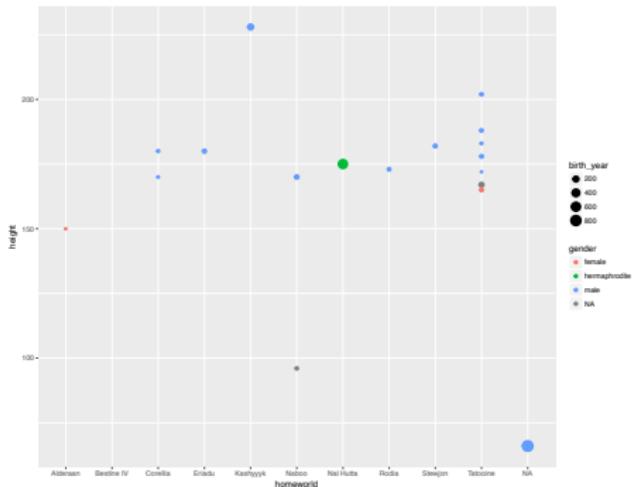
ggplot() +
  geom_boxplot(data = df2, aes(x = homeworld, y = height)) +
  geom_line(data = df1, aes(x = homeworld, y = meanHeight), group = 1) +
  geom_point(data = df1, aes(x = homeworld, y = meanHeight),
             size = 3, color = "red")
```



## 2D Graphic BUT nD Information

- A third variable can be added to a plot by mapping it to an aesthetic, such as color, shape, size

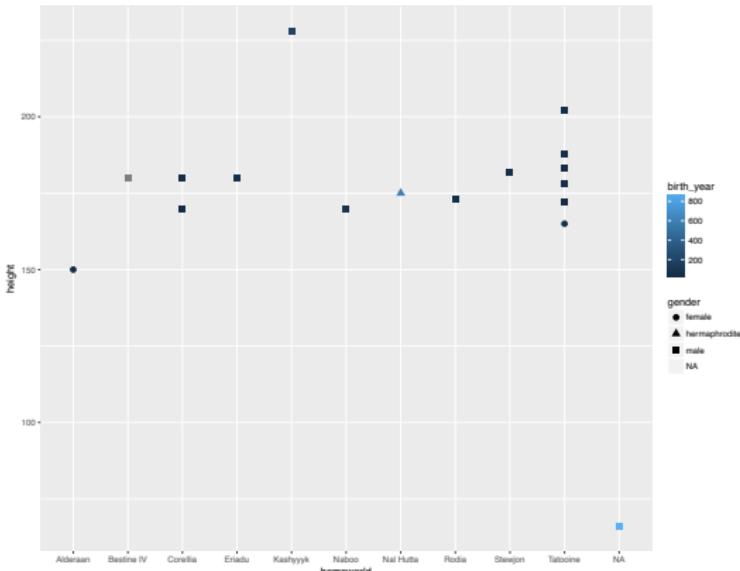
```
dplyr::starwars[1:20, ] %>% ggplot() +  
  geom_point(aes(x = homeworld, y = height, color = gender, size = birth_year))
```





## 2D Graphic BUT nD Information [CONT'D]

```
dplyr::starwars[1:20, ] %>% ggplot() +  
  geom_point(aes(x = homeworld, y = height,  
                 shape = gender, color = birth_year), size = 5)
```





## 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             |



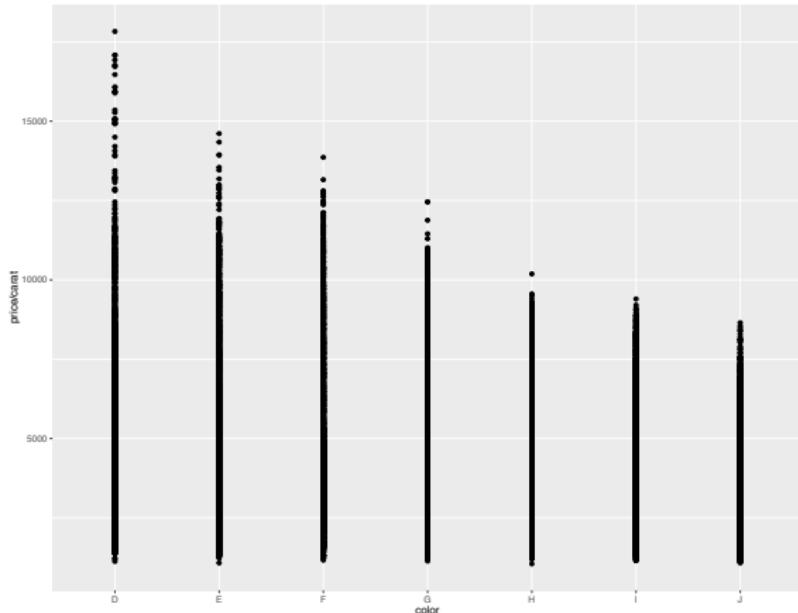
```
geom = "jitter"
```

- 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 = "jitter" [EXAMPLE 1/2]

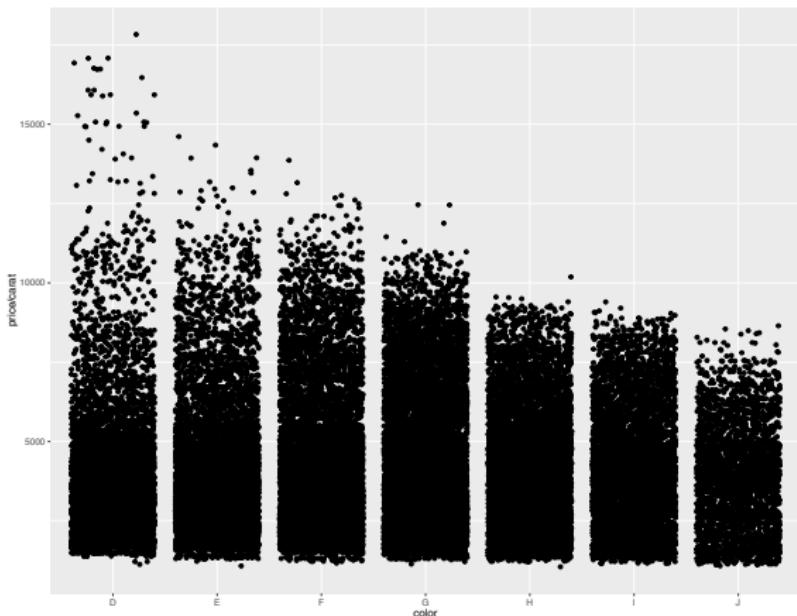
```
diamonds %>% mutate(pricePerCarat = price / carat)
myDiamonds %>% ggplot(aes(x = color, y = pricePerCarat)) +
  geom_point() ### no jitter here
```





## geom = "jitter" [EXAMPLE 2/2]

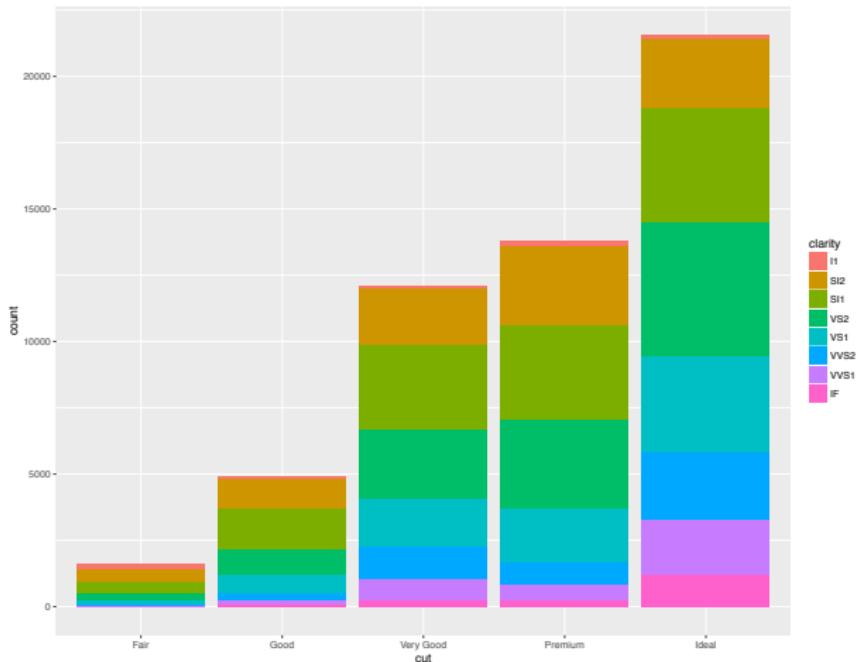
```
diamonds %>% mutate(pricePerCarat = price / carat)
myDiamonds %>% ggplot(aes(x = color, y = pricePerCarat)) +
  geom_point(position = "jitter")
```





## Useful Plots [1/2]

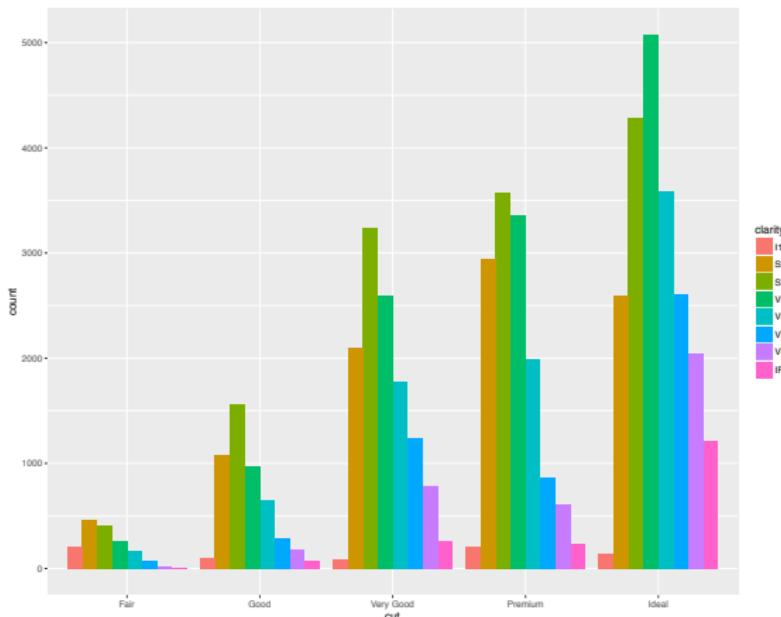
```
ggplot2::diamonds %>% ggplot() + geom_bar(aes(x = cut, fill = clarity)))
```





## Useful Plots [2/2]

```
ggplot2::diamonds %>% ggplot() + geom_bar(aes(x = cut, fill = clarity),  
                                         position = "dodge")
```





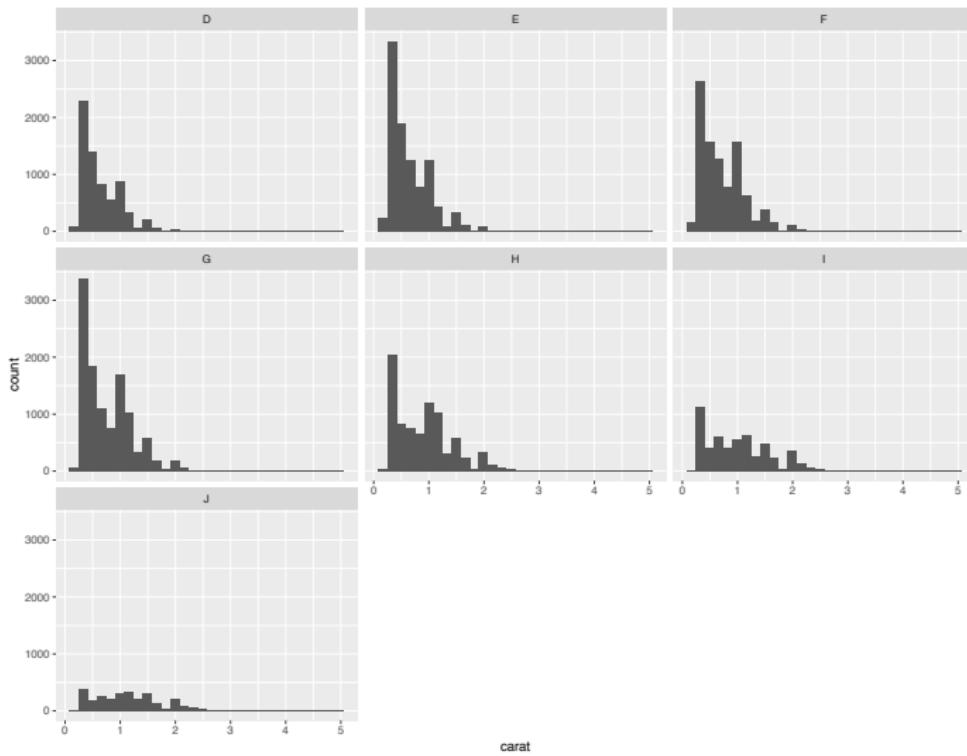
## Faceting

- 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
- The following code generates the graph on the proceeding slide

```
> diamonds %>% ggplot() + geom_histogram(aes(carat)) + facet_wrap(~ color)
```



## Faceting [CONT'D]





## 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]

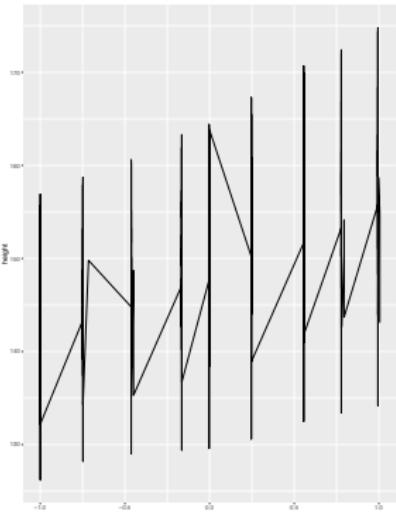
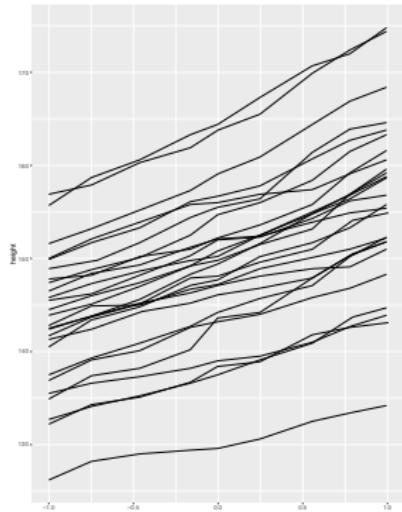
- By default, the `group` is set to the interaction of all discrete variables in the plot, which typically generates the expected results
- In the event it does not (or when no discrete variables are used in the plot), the `group` can be mapped to a variable that has a different value for each group
- `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

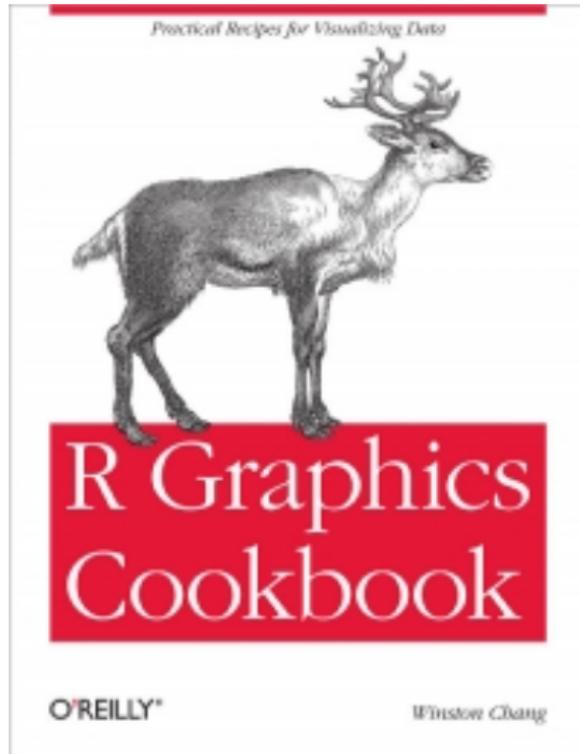


## Multiple Groups / One Aesthetic

- 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(nlme::Oxboys, aes(age, height, group = Subject)) + geom_line()`  
RIGHT `myPlot <- ggplot(nlme::Oxboys, aes(age, height)) + geom_line()`





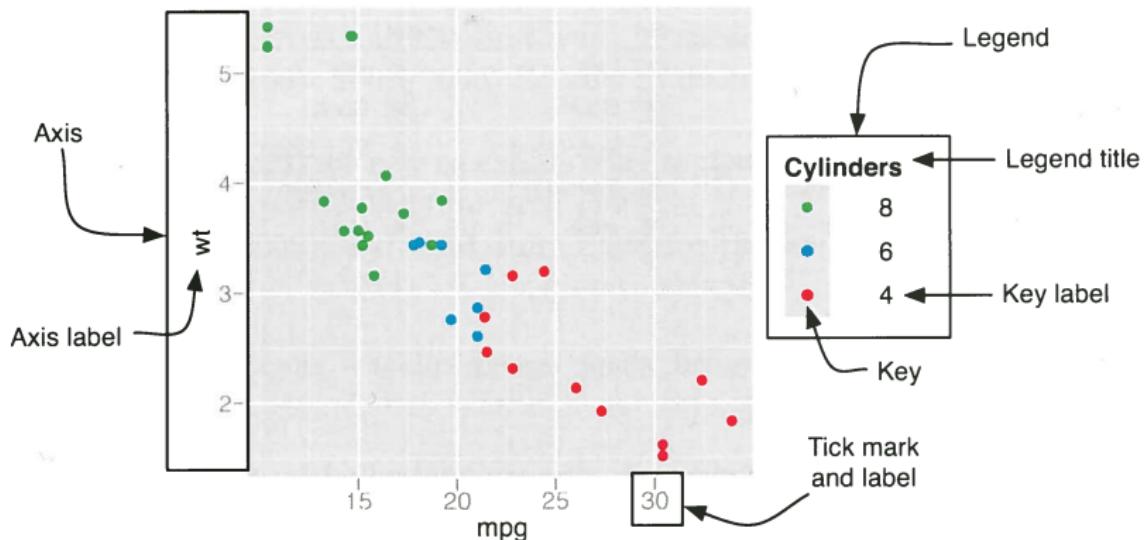


## Annotating Plots

- **ggplot2** makes it very easy to annotate a plot with
  - ➊ Vertical bars
  - ➋ Horizontal bars
  - ➌ Shaded regions
  - ➍ Text
  - ➎ Mathematical symbols and equations
  - ➏ Arrows



## Components of the Axes and Legend





## Notes

- 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



## Subsection 1

`tidyverse`



## Additionally Complex Analysis

- Often times, the data we are trying to plot simply is not in the format we need it to be
- It is not uncommon that a majority of the time spent visualizing data is not writing the visualization code, e.g., `ggplot`, but rather restructuring data (mainly data frames) so as to be able to visualize the data
- There are functions in base R such as `aggregate()` and `merge()`, etc., which can help with this, as well as functions such as `melt()` and `cast()` from the `reshape2` package
- This section will examine the use of `dplyr`



## Tidy Data with `tidyverse`

- Data should always be tidy before you begin to work with it
- All data should be organized such that
  - ➊ **each column is a variable**
  - ➋ **each row is an observation**
- `tidyverse` provides four main functions for tidying your messy data
  - ➌ `gather()`
  - ➍ `spread()`
  - ➎ `separate()`
  - ➏ `unite()`



## tidyverse::gather()

- `gather()` takes multiple columns and gathers them into key-value pairs
- It makes *wide* data *longer*
- The equivalent function in the popular `reshape2` package is `melt()`



## tidyverse::gather() [EXAMPLE]

- A company wants to compare sales of customers who have member cards and those who don't in five different cities

```
> (sales <- read_csv("tidyverse_gather_memberCardSales.csv"))
Parsed with column specification:
cols(
  city = col_character(),
  memberCardSales = col_integer(),
  noMemberCardSales = col_integer()
)
# A tibble: 5 × 3
      city   memberCardSales   noMemberCardSales
      <chr>        <int>            <int>
1 New York       297783           756185
2 San Francisco  761946           485681
3 Chicago        769215           155030
4 Austin          604328           962551
5 Las Vegas       174663           420949
```



## tidyverse::gather() [EXAMPLE]

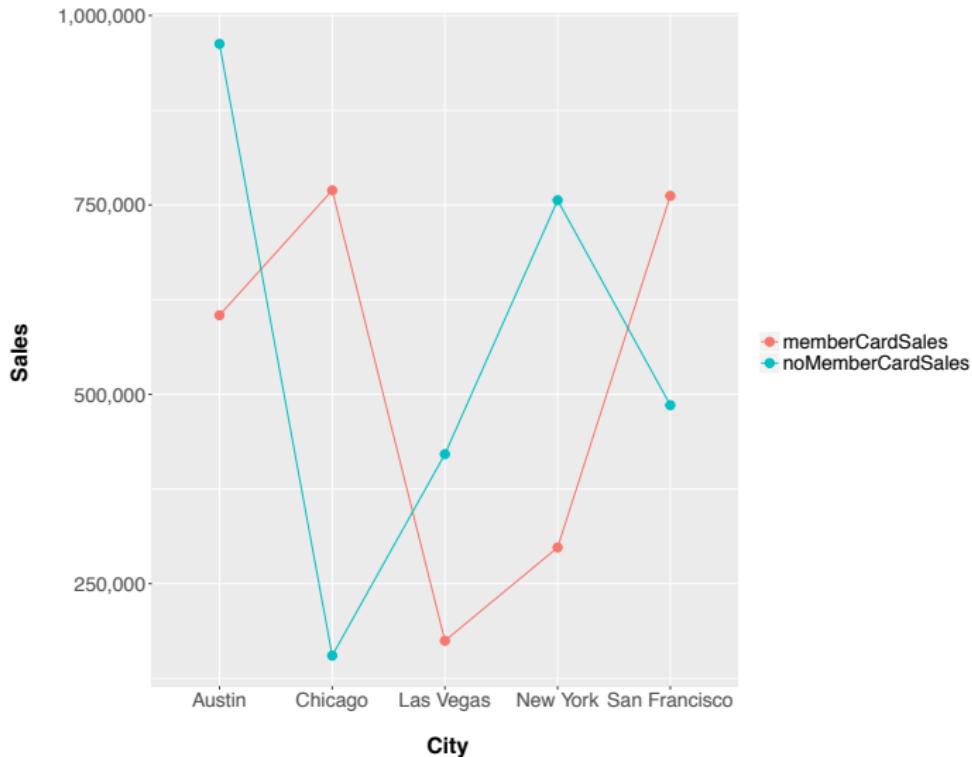
- A company wants to compare sales of customers who have member cards and those who don't in five different cities

```
> (sales <- read_csv("tidyverse_gather_memberCardSales.csv"))
Parsed with column specification:
cols(
  city = col_character(),
  memberCardSales = col_integer(),
  noMemberCardSales = col_integer()
)
# A tibble: 5 × 3
      city   memberCardSales   noMemberCardSales
      <chr>        <int>            <int>
1 New York       297783           756185
2 San Francisco  761946           485681
3 Chicago        769215           155030
4 Austin          604328           962551
5 Las Vegas       174663           420949
```

- **OBJECTIVE** Create a line plot in `ggplot` of Sales (y) on City (x), with one line for card status



## `tidyverse::gather()` [EXAMPLE CONT'D]





## tidyverse::gather() [EXAMPLE CONT'D]

- Recall that in a tidy data set, each column is a variable and each row is an observation, which is not the case here
- Use `gather()` to organize the columns into key-value pairs of cardStatus and Sales

```
> (myTidySales <- sales %>% gather(cardStatus,
  Sales, memberCardSales, noMemberCardSales))

# A tibble: 10 x 3
#   city      cardStatus     Sales
#   <chr>    <chr>       <int>
1 New York memberCardSales 297783
2 San Francisco memberCardSales 761946
3 Chicago    memberCardSales 769215
4 Austin     memberCardSales 604328
5 Las Vegas  memberCardSales 174663
6 New York  noMemberCardSales 756185
7 San Francisco noMemberCardSales 485681
8 Chicago    noMemberCardSales 155030
9 Austin     noMemberCardSales 962551
10 Las Vegas noMemberCardSales 420949
```



## tidyverse::gather() [GGPLOT CODE]

```
library(scales)

ggplot(myTidySales,
       aes(x = city, y = Sales, group = cardStatus, colour = cardStatus)) +
  geom_line() +
  geom_point(size = 3) +
  scale_y_continuous(labels = comma) +
  ylab("Sales \n") +
  xlab("\n City") +
  theme(legend.title = element_blank(),
        axis.text = element_text(size = 16),
        axis.title = element_text(size = 18, face = "bold"),
        legend.text = element_text(size = 16))
```

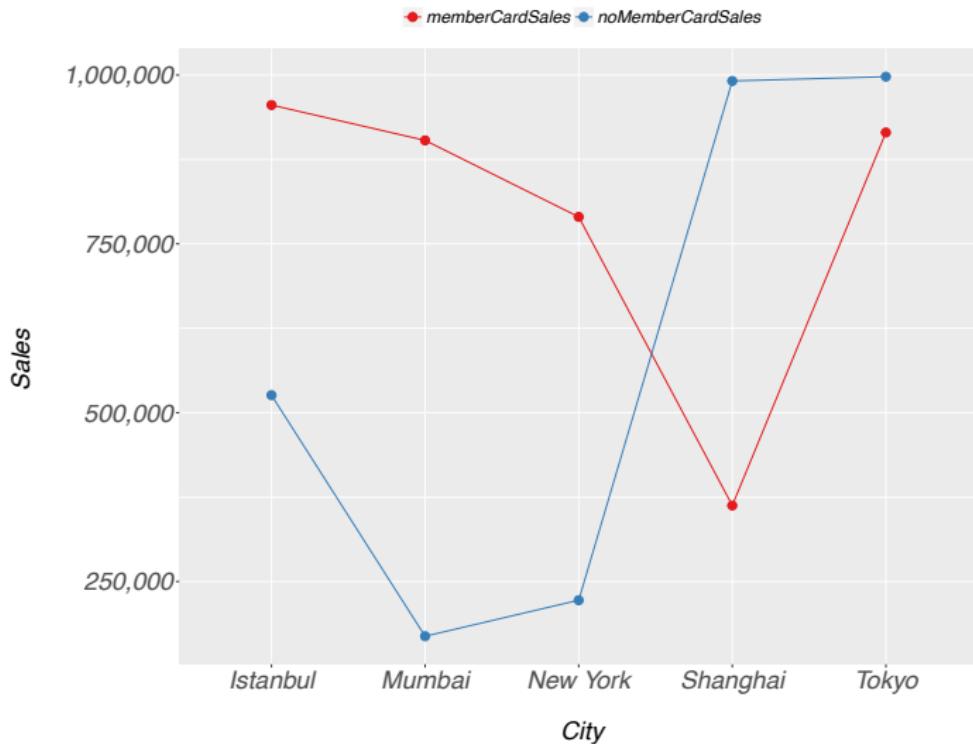


## `tidyverse::separate()`

- Sometimes variables are clumped together in a single column
- `separate()` allows you to parse them
- `extract()` works similarly but uses regexp groups instead of a splitting pattern or position
- **EXAMPLE** A company wants to compare sales of customers from *five international cities* who have member cards and those who don't
- **OBJECTIVE** Create a line plot in `ggplot` of Sales (y) on City (x), with one line for card status



## `tidyverse::separate()` [EXAMPLE CONT'D]





## tidyverse::separate() [EXAMPLE]

```
> (sales <- read_csv("tidyverse_separate_memberCardSales.csv"))
Parsed with column specification:
cols(
  city = col_character(),
  noMemberCardSales = col_character(),
  memberCardSales = col_character()
)
# A tibble: 5 × 3
  city      noMemberCardSales memberCardSales
  <chr>      <chr>           <chr>
1 New York  222355USD     789924USD
2 Tokyo      997512JPY    914879JPY
3 Mumbai     169019INR    903189INR
4 Istanbul   525930TRY    955508TRY
5 Shanghai   991163CNY    362563CNY
```

- The first challenge is how to create a numeric value out of a currency column that contains alphanumeric characters



## tidyverse::separate() [EXAMPLE CONT'D]

- Firstly, observe from the raw data (previous slide) that it is not tidy, i.e., each column should be a variable and each row should be an observation
- **STEP 1** Make the data tidy using `gather()`

```
> (myTidySales <- sales %>% gather(cardStatus, Sales,
  memberCardSales, noMemberCardSales))

# A tibble: 10 × 3
  city      cardStatus     Sales
  <chr>      <chr>      <chr>
1 New York  memberCardSales 789924USD
2 Tokyo      memberCardSales 914879JPY
3 Mumbai     memberCardSales 903189INR
4 Istanbul   memberCardSales 955508TRY
5 Shanghai   memberCardSales 362563CNY
6 New York  noMemberCardSales 222355USD
7 Tokyo      noMemberCardSales 997512JPY
8 Mumbai     noMemberCardSales 169019INR
9 Istanbul   noMemberCardSales 525930TRY
10 Shanghai  noMemberCardSales 991163CNY
```



## tidyverse::separate() [EXAMPLE CONT'D]

- STEP 2 Split `Sales` into `sales` and `currency` w/ `separate()`

```
> (myTidierDataFrame <- myTidySales %>% separate(Sales,
  into = c("sales", "currency"), sep = -4, convert = T))
# A tibble: 10 × 4
  city      cardStatus sales currency
  <chr>      <chr>    <int>   <chr>
1 New York memberCardSales 789924 USD
2 Tokyo     memberCardSales 914879 JPY
3 Mumbai    memberCardSales 903189 INR
4 Istanbul   memberCardSales 955508 TRY
5 Shanghai   memberCardSales 362563 CNY
6 New York  noMemberCardSales 222355 USD
7 Tokyo     noMemberCardSales 997512 JPY
8 Mumbai    noMemberCardSales 169019 INR
9 Istanbul   noMemberCardSales 525930 TRY
10 Shanghai  noMemberCardSales 991163 CNY
```

- The `sep = -4` option counts from the far right -3 spaces (a quirk)
- The `convert = T` option tells `separate()` to convert the new columns into types it thinks is most appropriate



## tidyverse::separate() [GGPLOT CODE]

```
library(scales)

ggplot(myTidierDataFrame, aes(x = city, y = sales)) +
  geom_line(aes(group = cardStatus, colour = cardStatus)) +
  geom_point(aes(group = cardStatus, colour = cardStatus), size = 3) +
  labs(x = "\nCity", y = "Sales \n") +
  scale_y_continuous(labels = comma) +
  theme(legend.title = element_blank(),
        axis.title = element_text(size = 20, face = "italic"),
        axis.text = element_text(size = 20, face = "italic"),
        legend.text = element_text(size = 14, face = "italic"),
        legend.position = "top") +
  scale_colour_brewer(palette = "Set1")
```



## tidyverse::spread() & tidyverse::unite()

- The opposite of the `gather()` function is the `spread()` function, which takes *long* data and makes it *wide*
- The opposite of the `separate()` function is the `unite()` function, which combines multiple columns into a single column



## Section 6

# Programming in R



## Section Contents

### 6 Programming in R

- Control Flow
- Functions
- Writing Robust R Code
- Functional Programming & Functionals
- Evaluating the Performance of R Code



## Subsection 1

### Control Flow



# 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()`



## 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"
```



## TRY THIS

- Write a loop which iterates through the numbers 1 to 10, and prints the following statement, e.g., "This is the number 3"



## TRY THIS

- Write a loop which iterates through the numbers 1 to 10, and prints the following statement, e.g., "This is the number 3"

## SOLUTION

- ```
> for (i in 1:10) {  
+   print(paste("This is the number", i))  
+ }  
  
[1] "This is the number 1"  
[1] "This is the number 2"  
[1] "This is the number 3"  
[1] "This is the number 4"  
...
```



## TRY THIS

- Write a loop which iterates through the numbers 1 to 10, and prints the following statement, e.g., "This is the number 3"

## SOLUTION

- ```
> for (i in 1:10) {  
+   print(paste("This is the number", i))  
+ }  
  
[1] "This is the number 1"  
[1] "This is the number 2"  
[1] "This is the number 3"  
[1] "This is the number 4"  
...
```

- The multi-line form of the `for()` loop

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



## TRY THIS

- Write a loop which iterates through the numbers 1 to 10, and prints whether that number is even or odd, e.g., "The number 3 is odd"
- Be sure to actively test whether each number is even or odd, don't just alternate between even and odd using code

## SOLUTION

```
for(i in 1:10){  
  if (i %% 2 == 0) {  
    print(paste("The number", i, "is even"))  
  } else {  
    print(paste("The number", i, "is odd"))  
  }  
}
```



## 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



## TRY THIS

- ➊ Print the numbers 10 to 0 in decrements of 2 using a `while()` loop
- ➋ Write a `while()` that can find the largest prime number between 1 and any number you choose to input



## TRY THIS

- ➊ Print the numbers 10 to 0 in decrements of 2 using a `while()` loop
- ➋ Write a `while()` that can find the largest prime number between 1 and any number you choose to input

## SOLUTION

➊ > while (n > 0) print(n); n <- n - 2

➋ n <- 5632  
thisNumberIsPrime <- F  
  
while (!isTRUE(thisNumberIsPrime)){  
 n <- n-1  
 thisNumberIsPrime <- isPrime(n)  
}



## 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

OK

```
x <- -3

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

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



## TRY THIS

- ① Set a variable `x <- 3` and write an `if() else()` statement that determines whether or not the number is greater than 5, printing the result to the screen
- ② Create a vector of length 10, populated with 10 randomly selected integers from 1 to 100. Write an `if() else()` loop, iterating through all values of the vector to determine which are greater than 50 and which are less than 50, printing the result to the screen.



## SOLUTION

1

```
x <- 3
if( x > 5) {
  print("x is greater than 5")
} else {
  print("x is less than 5")
}
```

2

```
x <- sample(1:100, 10, replace = F)
for(i in 1:length(x)) {
  if( x[i] > 50) {
    print("x is greater than 50")
  } else {
    print("x is less than 50")
  }
}
```



## 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"
```



## TRY THIS

- Use `ifelse()` to determine which values of `x` are greater than 50 and which are less than 50, printing the result to the screen.



## TRY THIS

- Use `ifelse()` to determine which values of `x` are greater than 50 and which are less than 50, printing the result to the screen.

## SOLUTION

```
> ifelse(x > 50, "x is greater than 50", "x is less than 50")
```



## switch()

- `switch()` chooses statements based on the discrete value of 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**

- ➊ Using a `for()` loop, recode the entries in the Survived variable with "Survived" and "Perished"
- ➋ 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;  $18 \text{ yrs} \leq$  "Adult"  $\leq 65 \text{ yrs}$ ; and "Senior" if older than 65 yrs
- ➌ Ordering the passengers in descending order by last name, use a `while()` loop to identify the name of the 100<sup>th</sup> surviving passenger



# 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
  - 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 a message that says "Press Enter for next Graph" to add a pause in the sequential execution



## Subsection 2

### 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

- ➊ Write a function that takes two arguments, `a` and `b`, and returns rows `a` through `b` of `mtcars`
- ➋ 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
- ➌ 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



# 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"
```



# 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
  - Exact name (perfect matching)
  - Prefix matching (imperfect/partial matching)
  - Position

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

# exact matching and positional
> str(myFunc_09(2, 3, arg1 = 1))
List of 3
$ a : num 1
$ m1: num 2
$ m2: num 3
```

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

# partial matching fails if match is ambiguous
> str(myFunc_09(1, 3, my = 1))
Error in myFunc_09(1, 3, my = 1) :
  argument 3 matches multiple formal arguments
```



## 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



# 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)
  }
}
```



## Copy-on-Modify Semantics

- R protects you from a potentially nasty side-effect, namely, that the modification of a function argument does not change the original value
- This behavior differs from other language such as Java

```
rm(list=ls())  
  
myFunc_19 <- function(x) {  
  x$var1 <- 99  
  x  
}  
  
> x <- list(var1 = 1)  
  
> myFunc_19(x)  
$var1  
[1] 99  
  
> x$var1  
[1] 1
```



# LAB

- ➊ 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
- ➋ In the question above, what are the formal and calling arguments of the function?
- ➌ Is this function self-contained? Why or why not?
- ➍ Rewrite the above function to include a data frame `myDataFrame` as an additional argument, such that it returns rows `firstRow` through `lastRow` of `myDataFrame`



## Subsection 3

### 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*

The screenshot shows the RStudio interface with the 'Debug' menu open. The 'On Error' option is highlighted with a blue arrow. A tooltip below the menu lists three options: 'Message Only', 'Error Inspector', and 'Break in Code', with 'Break in Code' checked.

Console ~ / ↵

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



## 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]

Screenshot of RStudio interface showing error handling and debugging.

**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]>

```

**Code Editor:**

```

codeOnSlides.R  myFunc_24.R
Function: myFunc_24 (GlobalEnv)
Debug location is approximate because the source is not available.
1 function(arg_04) "myString" + arg_04

```

**Environment:**

| Name   | Value |
|--------|-------|
| arg_04 | 99    |

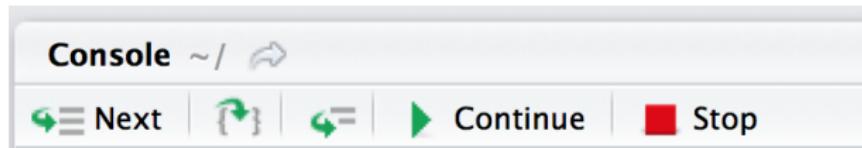
**Traceback:**

- 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
  - ➊ `try()` gives you the ability to continue execution even when an error occurs
  - ➋ `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
  - ➊ errors (made by `stop()`)
  - ➋ warnings (`warning()`)
  - ➌ message (`message()`)
  - ➍ 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

- ➊ `show_condition(stop("!"))` calls the function `show_condition()`, passing `stop("!")` as the argument, represented in the function as `code`
- ➋ `code` is executed in the `tryCatch()` block, where `code == stop("!")`
- ➌ the function `stop()` “*stops execution of the current expression and executes an error action*”
- ➍ 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
- ➎ 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
- n.b.** 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")
    }
  )
}

> \textcolor{blue}{\textbf{show_condition(stop("!"))}}
1 "!"
doTryCatch(return(expr), name, parentenv, handler)

Silly errors like this make
 Paul very angry
```



## TRY THIS

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

- ➊ 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

- ➊ 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
- ➋ 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

- ➊ 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
- ➋ 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)
- ➌ Avoid functions that return different output depending on their input
  - 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) # first is FALSE

Error: 1 == 2 is not TRUE
```



## Subsection 4

### 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

- ➊ It's easy to make copy-paste mistakes
- ➋ It makes bugs more likely
- ➌ It makes updating code a HUGE pain in the arse
- ➍ 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 and 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

- ➊ It is very compact
- ➋ If the code for a missing value changes, it only needs to be updated in a single location
- ➌ It works for any number of columns, so you don't need to specify the number of columns, therefore avoiding potential mistakes
- ➍ All columns are evaluated uniformly
- ➎ 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

```
fixMising <- 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 in 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.



## TRY THIS

- 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



## TRY THIS

- 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



## TRY THIS with Using state.x77

- ➊ Write code that **does not contain** functionals or `dplyr` code that computes the mean of each column of data
- ➋ Employ your functional of choice to write code that computes the mean of each column of data



## 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    ...
```



## SOLUTION [CONT'D]

- ➊ What data type was passed to `apply()`?



## SOLUTION [CONT'D]

- ➊ What data type was passed to `apply()`?

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

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



## SOLUTION [CONT'D]

- ① What data type was 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 ...
```



## SOLUTION [CONT'D]

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



## SOLUTION [CONT'D]

- 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
```



## SOLUTION [CONT'D]

- 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()`?

```
> tibble::as.tibble(iris)
# A tibble: 150 x 5
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
    <dbl>       <dbl>      <dbl>       <dbl>   <fctr>
1          NA         NA         NA         NA   setosa
2          NA         NA         NA         NA   setosa
3          NA         NA         NA         NA   setosa
4          NA         NA         NA         NA   setosa
5          NA         NA         NA         NA   setosa
6          NA         NA         NA         NA   setosa
7          NA         NA         NA         NA   setosa
8          NA         NA         NA         NA   setosa
9          NA         NA         NA         NA   setosa
10         NA         NA         NA         NA   setosa
# ...
```

```
> apply(iris, 2, mean)
Sepal.Length Sepal.Width Petal.Length Petal.Width Species
      NA           NA           NA           NA   setosa
Warning messages:
1: In mean.default(newX[, i], ...) :
  argument is not numeric or logical: returning NA
...

```

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

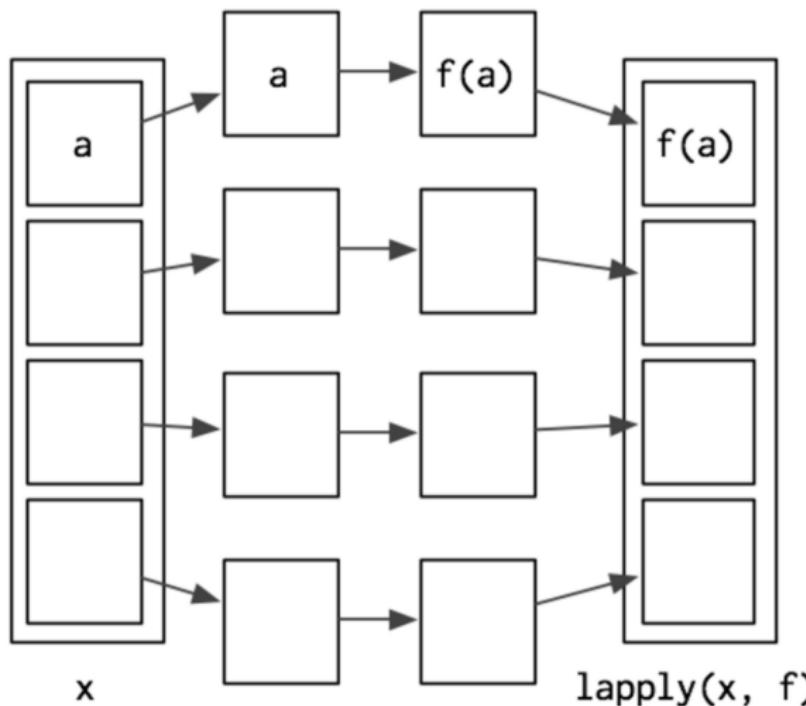
```
> class(apply(iris[1:4], 2, mean))
[1] "numeric"
```

```
> 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



## TRY THIS with `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)



## TRY THIS with `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(c[3:8], function(i) return(cor(s[,2], s[,i])))  
> str(lapplyResult)
```

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

```
> sapplyResult = sapply(c[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(c[, "Area"], state.region, sum)
```



## Subsection 5

### 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 can make 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 $\mu$ 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

- ➊ `microbenchmark()` is far more precise
- ➋ `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
```



## Section 7

# 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", "ZzZzZZzzZ"))
[1] "abc"        "abcdef"      "ZzZzZZzzZ"

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

> paste(c("abc", "abcdef", "ZzZzZZzzZ"), collapse = "")
[1] "abcabcdefZzZzZZzzZ"

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



## 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        | <i>Meaning</i>  |
|-----------------|---|
| ^               | anchors expression to the beginning of target                         |
| \$              | anchors expression to end of target                                   |
| .               | matches any single character except newline                           |
|                 | separates alternative patterns  |
| ( )             | 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                     |
| { <i>n</i> }    | matches exactly <i>n</i> occurrences of preceding entity              |
| { <i>n, m</i> } | matches <i>n</i> or more occurrences of preceding entity              |
| { <i>n, m</i> } | matches between <i>n</i> and <i>m</i> 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]+'`

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" "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 `grep1()` 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 that 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 that 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 elements that 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]]
[1] 1 7
attr("match.length")
[1] 5 5
attr("useBytes")
[1] TRUE
```

```
[[2]]
[1] -1
attr("match.length")
[1] -1
attr("useBytes")
[1] TRUE
```

```
[[3]]
[1] 2
attr("match.length")
[1] 5
attr("useBytes")
[1] TRUE
```

```
[[4]]
[1] 1
attr("match.length")
[1] 5
attr("useBytes")
[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 `regexp()` 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



## TRY THIS 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