Programming day 1: summer review, three tools (logical statements, control flow, for loops), and dplyr

Methods camp instructors

September 5th, 2018

Outline

- Programming Overview: Philosophies, Practices and Practicalities

Outline

- Programming Overview: Philosophies, Practices and Practicalities
- Indexing and manipulation of four main data structures: vectors, lists, matrices, and data.frames

Outline

- Programming Overview: Philosophies, Practices and Practicalities
- Indexing and manipulation of four main data structures: vectors, lists, matrices, and data.frames
- Three tools in base R useful for data manipulation

Outline

- Programming Overview: Philosophies, Practices and Practicalities
- Indexing and manipulation of four main data structures: vectors, lists, matrices, and data.frames
- Three tools in base R useful for data manipulation
    - Logical statements

Outline

- Programming Overview: Philosophies, Practices and Practicalities
- Indexing and manipulation of four main data structures: vectors, lists, matrices, and data.frames
- Three tools in base R useful for data manipulation
  - Logical statements
  - Control flow

Outline

- Programming Overview: Philosophies, Practices and Practicalities
- Indexing and manipulation of four main data structures: vectors, lists, matrices, and data.frames
- Three tools in base R useful for data manipulation
    - Logical statements
    - Control flow
    - For loops

Outline

- Programming Overview: Philosophies, Practices and Practicalities
- Indexing and manipulation of four main data structures: vectors, lists, matrices, and data.frames
- Three tools in base R useful for data manipulation
  - Logical statements
  - Control flow
  - For loops
- dplyr as a tool for data manipulation

Programming: Philosophies, Practices and Practicalities

- R originated from statisticians: maximize statistical performance.

Programming: Philosophies, Practices and Practicalities

- R originated from statisticians: maximize statistical performance.
- Most recently R is used by a much wider group, including computer scientists and social scientists.

Programming: Philosophies, Practices and Practicalities

- R originated from statisticians: maximize statistical performance.
- Most recently R is used by a much wider group, including computer scientists and social scientists.
- There is a distinct movement to push towards reproducible and readable code with a certain bent:

Programming: Philosophies, Practices and Practicalities

- R originated from statisticians: maximize statistical performance.
- Most recently R is used by a much wider group, including computer scientists and social scientists.
- There is a distinct movement to push towards reproducible and readable code with a certain bent:
  - Consistency between readability across languages

Programming: Philosophies, Practices and Practicalities

- R originated from statisticians: maximize statistical performance.
- Most recently R is used by a much wider group, including computer scientists and social scientists.
- There is a distinct movement to push towards reproducible and readable code with a certain bent:
  - Consistency between readability across languages
  - Reproducibility and Version Control (write code for humans, write data for computers), working with Git right away, commenting extensively, making use of logical names (no spaces), avoid duplicate/incremental files, produce markdown documents with all reproducible steps included etc.

Tidyverse

All of this culminates in Tidyverse (a philosophy and a collection of packages).

R packages for data science

The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying philosophy and common APIs.

Install the complete tidyverse with:

```
install.packages("tidyverse")
```

Programming: Philosophies, Practices and Practicalities

- Much of this philosophy culminates in teaching R in a particular fashion, for example, don't teach $ and [ ], loops and conditionals, data types, or *apply* family functions until much later, but instead start with dplyr, %>% from magrittr, and *ggplot2* immediately. Read more here[1].

[1] http://varianceexplained.org/r/teach-tidyverse/

Programming: Philosophies, Practices and Practicalities

- Much of this philosophy culminates in teaching R in a particular fashion, for example, don't teach $ and [ ], loops and conditionals, data types, or *apply* family functions until much later, but instead start with dplyr, %>% from magrittr, and *ggplot2* immediately. Read more here[1].
- A "classic" R course teaches much more base R solutions, with a focus on data types, loops and conditional statements right away.

[1] http://varianceexplained.org/r/teach-tidyverse/

Programming: Philosophies, Practices and Practicalities

- Much of this philosophy culminates in teaching R in a particular fashion, for example, don't teach $ and [ ], loops and conditionals, data types, or *apply* family functions until much later, but instead start with dplyr, %>% from magrittr, and *ggplot2* immediately. Read more here[1].
- A "classic" R course teaches much more base R solutions, with a focus on data types, loops and conditional statements right away.
- While we are philosophically all in on Tidyverse, we are going to teach you a combination.

[1] http://varianceexplained.org/r/teach-tidyverse/

Programming: Philosophies, Practices and Practicalities

- Much of this philosophy culminates in teaching R in a particular fashion, for example, don't teach $ and [ ], loops and conditionals, data types, or *apply* family functions until much later, but instead start with dplyr, %>% from magrittr, and *ggplot2* immediately. Read more here[1].
- A "classic" R course teaches much more base R solutions, with a focus on data types, loops and conditional statements right away.
- While we are philosophically all in on Tidyverse, we are going to teach you a combination.
  - You will encounter base R code and should know how to read this code.

[1] http://varianceexplained.org/r/teach-tidyverse/

Programming: Philosophies, Practices and Practicalities

- Much of this philosophy culminates in teaching R in a particular fashion, for example, don't teach $ and [ ], loops and conditionals, data types, or *apply* family functions until much later, but instead start with dplyr, %>% from magrittr, and *ggplot2* immediately. Read more here[1].
- A "classic" R course teaches much more base R solutions, with a focus on data types, loops and conditional statements right away.
- While we are philosophically all in on Tidyverse, we are going to teach you a combination.
  - You will encounter base R code and should know how to read this code.
  - While it's great that Tidyverse developers have poured so much time into making excellent wrappers for many base R functions, it is nonetheless useful to know the base R functions that they draw upon and be able to know why and how it is much more efficient.

[1] http://varianceexplained.org/r/teach-tidyverse/

Programming: Philosophies, Practices and Practicalities

- Much of this philosophy culminates in teaching R in a particular fashion, for example, don't teach $ and [ ], loops and conditionals, data types, or *apply* family functions until much later, but instead start with dplyr, %>% from magrittr, and *ggplot2* immediately. Read more here[1].
- A "classic" R course teaches much more base R solutions, with a focus on data types, loops and conditional statements right away.
- While we are philosophically all in on Tidyverse, we are going to teach you a combination.
  - You will encounter base R code and should know how to read this code.
  - While it's great that Tidyverse developers have poured so much time into making excellent wrappers for many base R functions, it is nonetheless useful to know the base R functions that they draw upon and be able to know why and how it is much more efficient.
  - Social science is really still more much base R than on the forefront of computational best practices (but that doesn't mean we shouldn't be the ones pushing for it!)

[1] http://varianceexplained.org/r/teach-tidyverse/

Schematic to understand matrices versus dataframes

	Homogeneous elements	**Heterogeneous elements**
1-dimensional	Vector	List
2-dimensional	Matrix	**Data.frame**

Source: Hadley Wickham's Advanced R

A slightly more complete way to look at it

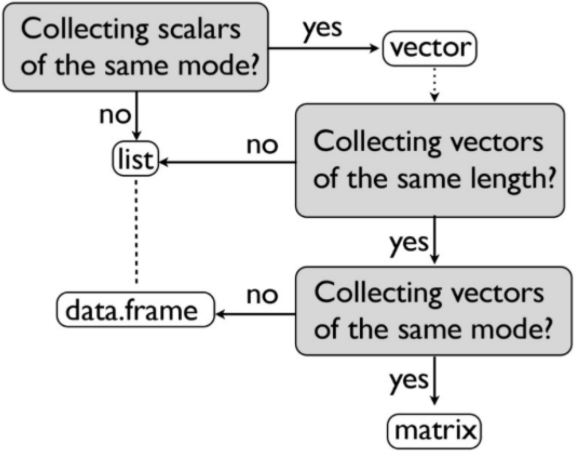
Simple objects and why we use *class*.

a simple view of simple R objects that will get you pretty far

Simple view	Technically correct R view		
	mode	class	typeof
character	character	character	character
logical	logical	logical	logical
numeric	numeric	integer or numeric	integer or double
factor	numeric	factor	integer

Complex Objects

a simple view of less simple objects that will get you pretty far



Source: Jenny

Bryan's Stat 545 Class

Data we'll be working with

In-class lecture example: data from 3rd wave of AddHealth containing people's ratings of how important the respondent believes the following are for a "successful marriage or serious committed relationship":

- love
- no cheating
- money

Data we'll be working with

In-class lecture example: data from 3rd wave of AddHealth on how demographic characteristics relate to how important the respondent believes the following are for a "successful marriage or serious committed relationship":

- love
- no cheating
- money

Today's Homework: data from the American National Election Studies (ANES) on how a respondent's degree of opposition to free trade is related to their views about three presidential candidates (at the time): Trump, Sanders, and Clinton

Today's Homework

Figure 2:

Preliminary: loading data

- Set working directory (file path to folder in which data was stored) and load data; remember that this path is *local* to your computer so you will need to edit whatever pathname we used in any file we provide.

```
# install.packages(tidyverse)
library("tibble")
library("readr")
library("ggplot2")
library('dplyr')
library('tidyr')
library('magrittr')
library('purrr')
library("tidyverse")
setwd("~/Dropbox/MethodsCamp/2018/Programming Lectures/Day1Programming")
```

```
## Error in setwd("~/Dropbox/MethodsCamp/2018/Programming Lectures/Day1Programming"): cannot change working directory
```

```
addh <- read.csv("addhealthlec1.csv")

# note here we would use readr and its read_ functions
#addh2<- read_csv("addhealthlec1.csv")
```

Preliminary: loading data

- Set working directory (file path to folder in which data was stored) and load data; remember that this path is *local* to your computer so you will need to edit whatever pathname we used in any file we provide.
- Can use "session" to set wd but make sure to copy/paste the code that's pasted in the console so your script can run from beginning -> end

```
# install.packages(tidyverse)
library("tibble")
library("readr")
library("ggplot2")
library('dplyr')
library('tidyr')
library('magrittr')
library('purrr')
library("tidyverse")
setwd("~/Dropbox/MethodsCamp/2018/Programming Lectures/Day1Programming")
```

```
## Error in setwd("~/Dropbox/MethodsCamp/2018/Programming Lectures/Day1Programming"): cannot change working directory
```

```
addh <- read.csv("addhealthlec1.csv")

# note here we would use readr and its read_ functions
#addh2<- read_csv("addhealthlec1.csv")
```

Preliminary: loading data

- Set working directory (file path to folder in which data was stored) and load data; remember that this path is *local* to your computer so you will need to edit whatever pathname we used in any file we provide.
- Can use "session" to set wd but make sure to copy/paste the code that's pasted in the console so your script can run from beginning -> end
- R's commands for reading in data are specific to the file type– the most common is read.csv for csv files, but on Thursday, we'll be discussing read commands specific to other file types (e.g., importing foreign data types like STATA .dta files)

```
# install.packages(tidyverse)
library("tibble")
library("readr")
library("ggplot2")
library('dplyr')
library('tidyr')
library('magrittr')
library('purrr')
library("tidyverse")
setwd("~/Dropbox/MethodsCamp/2018/Programming Lectures/Day1Programming")
```

```
## Error in setwd("~/Dropbox/MethodsCamp/2018/Programming Lectures/Day1Programming"): cannot change working directory
```

```
addh <- read.csv("addhealthlec1.csv")

# note here we would use readr and its read_ functions
#addh2<- read_csv("addhealthlec1.csv")
```

Common data structures you'll work with

Before reviewing the main data structures, how do you discern what *class* a particular object you've stored is, which affects how functions will interpret the object?

```
##check what the data are stored as
class(addh)
```

```
## [1] "data.frame"
```

```
# in tidyverse, we should use tibble instead of dataframes
addh.tibble <- as_tibble(addh)
class(addh.tibble)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

```
num.data <- as.matrix(addh)

##check what the "money's importance to a relationship"
##is stored as; remember we can index variables
##using brackets or $variablename
class(addh$money)
```

```
## [1] "integer"
```

In Tidyverse

```r
# in tidyverse
str(select(addh.tibble, money))
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    3050 obs. of  1 variable:
##  $ money: int  7 1 5 9 7 10 5 10 10 8 ...
```

```r
# or
addh %>%
  select(money) %>% # equv. to addh$money
  # head %>%
  # optional pipe to show you how you can really
  # nest pipes as much as you want
  str
```

```
## 'data.frame':    3050 obs. of  1 variable:
##  $ money: int  7 1 5 9 7 10 5 10 10 8 ...
```

Common data structures you'll work with

1. vector

For each, we'll review how to extract elements from that structure (indexing) as well as common operations performed with that structure in the context of data analysis/statistics

Common data structures you'll work with

1. vector
2. data.frame

For each, we'll review how to extract elements from that structure (indexing) as well as common operations performed with that structure in the context of data analysis/statistics

Common data structures you'll work with

1. vector
2. data.frame
3. matrix

For each, we'll review how to extract elements from that structure (indexing) as well as common operations performed with that structure in the context of data analysis/statistics

Common data structures you'll work with

1. vector
2. data.frame
3. matrix
4. list

For each, we'll review how to extract elements from that structure (indexing) as well as common operations performed with that structure in the context of data analysis/statistics

vectors: examples and basic commands

- **What is a vector?**: sequence of data elements of the same type

vectors: examples and basic commands

- **What is a vector?**: sequence of data elements of the same type
- Common examples of vectors:

vectors: examples and basic commands

- **What is a vector?**: sequence of data elements of the same type
- Common examples of vectors:
  - Any individual row or column in a data.frame or matrix

vectors: examples and basic commands

- **What is a vector?**: sequence of data elements of the same type
- Common examples of vectors:
  - Any individual row or column in a data.frame or matrix
  - Variable names: *colnames(data)*

vectors: examples and basic commands

- **What is a vector?**: sequence of data elements of the same type
- Common examples of vectors:
  - Any individual row or column in a data.frame or matrix
  - Variable names: *colnames(data)*
  - Vector of numeric indices to subset data (e.g., vector of randomly sampled numbers)

vectors: creating a vector

- How to create a vector: use *c* to string together the elements

```
##create a vector with three id's
sampidvec <- c("1690", "1370", "1121")
sampidvec
```

```
## [1] "1690" "1370" "1121"
```

```
addh[sampidvec, ]
```

```
##        id age gender income logincome    debt love nocheating money
## 1690 2169  24 female  24000 10.085809  nodebt   10         10     5
## 1370 1761  24   male  20000  9.903488  nodebt   10          7     7
## 1121 1439  23 female  24000 10.085809 yesdebt   10         10    10
##      paypercent logpaypercent
## 1690       84.0      4.430817
## 1370       76.8      4.341205
## 1121       83.9      4.429626
```

vectors: shortcuts to create a vector

- Depending on what you want in the vector, there are shortcuts to help you create the vector more efficiently:

```
##set a seed so we sample same ids each time
set.seed(123)

##create a vector with three randomly sampled id's
sampids <- sample(rownames(addh), size = 3)
sampids
```

```
## [1] "878"  "2404" "1247"
```

vectors: shortcuts to create a vector

- Depending on what you want in the vector, there are shortcuts to help you create the vector more efficiently:

1. *sample*: for vectors where we want to randomly sample from some larger pool

```
##set a seed so we sample same ids each time
set.seed(123)

##create a vector with three randomly sampled id's
sampids <- sample(rownames(addh), size = 3)
sampids
```

```
## [1] "878"  "2404" "1247"
```

Breaking down the sample command (will review more in probability lecture)

sample(vector to sample from, size of sample, replace or not? (default = no replacement))

In our use, we mixed feeding the sample function arguments by position (the vector argument) and arguments by name (will review more formally tomorrow) and we defaulted to sampling without replacement:

sample(rownames(addh), size = 3)

vectors: shortcuts to create a vector

2. *seq* for sequence patterns in numeric vectors

```
##create a sequence of the beginning of every decade
decades <- seq(from = 1900, to = 2000, by = 10)
decades
```

```
##  [1] 1900 1910 1920 1930 1940 1950 1960 1970 1980 1990 2000
```

vectors: shortcuts to create a vector

3. *paste*: for patterns in character vectors

```
##create names for decades spanning from 1900 to 2000 by 10
decadenames <- paste(c("decade", "birthday"),
                     seq(from = 1900, to = 2000, by = 10),
                     sep = "_")
decadenames
```

```
##  [1] "decade_1900"   "birthday_1910" "decade_1920"   "birthday_1930"
##  [5] "decade_1940"   "birthday_1950" "decade_1960"   "birthday_1970"
##  [9] "decade_1980"   "birthday_1990" "decade_2000"
```

vectors: extracting elements from a vector

- How to extract elements from a vector:

```
##extract first, second, and third element from sample id vector
sampids[1:3]
```

```
## [1] "878"  "2404" "1247"
```

```
##or...
sampids[c(1,2, 3)]
```

```
## [1] "878"  "2404" "1247"
```

```
##remove the first two elements
sampids[-1:-2]
```

```
## [1] "1247"
```

```
##remove the first and third elements
sampids[-c(1, 3)] # but remember this just prints it in the console, it doesn't overwrite sampid
```

```
## [1] "2404"
```

```
answers <- factor(c(1,2,3),
                  levels = c("yes", "no", "maybe"))
levels(answers)
```

vectors: different element types within a vector

- Different element types contained within vector:
    1. numeric (encompasses integer and double)

vectors: different element types within a vector

- Different element types contained within vector:
    1. numeric (encompasses integer and double)
    2. character: when creating, use quotes around each element

vectors: different element types within a vector

- Different element types contained within vector:
    1. numeric (encompasses integer and double)
    2. character: when creating, use quotes around each element
    3. logical: TRUE/FALSE statements

vectors: different element types within a vector

- Different element types contained within vector:
    1. numeric (encompasses integer and double)
    2. character: when creating, use quotes around each element
    3. logical: TRUE/FALSE statements
    4. Non-"atomic" type- factor: R treats differently than numeric– e.g., can't calculate averages or other things that do not make sense for categories. Has a "levels" attribute that codes levels of the factor and that you might label

vectors: different element types within a vector

```
##check class of sample ids vector
as.character(sampids)
```

```
## [1] "878"  "2404" "1247"
```

```
class(sampids)
```

```
## [1] "character"
```

```
##convert to numeric identifiers
numsampids <- as.numeric(sampids)
numsampids
```

```
## [1]  878 2404 1247
```

```
class(numsampids)
```

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

```
##convert back to string identifiers
stringsampids <- as.character(numsampids)
stringsampids
```

```
## [1] "878"  "2404" "1247"
```

```
##how to use a vector to extract elements from another vector
```

Combining vectors into two data structures: data.frames and matrices

Two methods of combination:

1. Stack rows one on top of the other -> becomes matrix or data.frame

Combining vectors into two data structures: data.frames and matrices

Two methods of combination:

1. Stack rows one on top of the other -> becomes matrix or data.frame
2. Stack columns side by side -> becomes matrix or data.frame

Combining vectors into data.frames and matrices: stacking rows

Binding rows: *rbind* and *rbind.data.frame*

```
##combine each observation's age with its id into
## 1 x 2 vector that has an id and age
obs1 <- c(numsampids[1], ages[1])
obs1
```

```
## [1] 878  24
```

```
obs2 <- c(numsampids[2], ages[2])
obs3 <- c(numsampids[3], ages[3])

##stack as rows in a matrix
obs1to3 <- rbind(obs1, obs2, obs3)
obs1to3
```

```
##      [,1] [,2]
## obs1  878   24
## obs2 2404   21
## obs3 1247   20
```

```
class(obs1to3)
```

```
## [1] "matrix"
```

```
##stack as rows into a data.frame
obs1to3df <- as.data.frame(obs1to3)
obs1to3df
```

```
##        V1 V2
```

Combining vectors into data.frames and matrices: placing columns side by side

Binding columns: *cbind* and *cbind.data.frame*

```
##another way to arrive at same answer:
##(if they're in same order), putting id
##vector side by side with age vector by binding them as columns
obs1to3cols <- cbind(numsampids, ages)
obs1to3cols
```

```
##      numsampids ages
## [1,]        878   24
## [2,]       2404   21
## [3,]       1247   20
```

```
##data.frame form
obs1to3colsdf <- as.data.frame(obs1to3cols)
obs1to3colsdf
```

```
##   numsampids ages
## 1        878   24
## 2       2404   21
## 3       1247   20
```

matrices versus data.frames

The previous slides showed that you can combine rows/columns into *either* a matrix or a data.frame

When might we use each and what are the advantages/ disadvantages of each as a way to store multiple vectors?

matrices: advantages and disadvantages

- Stores elements of the *same type*– e.g., a matrix of character elements or a matrix of numeric elements

matrices: advantages and disadvantages

- Stores elements of the *same type*– e.g., a matrix of character elements or a matrix of numeric elements
- The above can cause problems for putting together vectors of different types, with R defaulting to turning all vectors into a character vector if any vectors contain a string (might have noticed in summer assignment when we converted the skin cancer dataset into a matrix, since state names were strings)

matrices: advantages and disadvantages

- Stores elements of the *same type*– e.g., a matrix of character elements or a matrix of numeric elements
- The above can cause problems for putting together vectors of different types, with R defaulting to turning all vectors into a character vector if any vectors contain a string (might have noticed in summer assignment when we converted the skin cancer dataset into a matrix, since state names were strings)
- On the plus side, matrices are needed for linear algebra operations like matrix multiplication (which is used in the context of linear regression) and take up less memory in R

matrices: advantages and disadvantages

Illustration of care needed when combining different types of vectors into a matrix

```
##print the two vectors
stringsampids; ages
```

```
## [1] "878"  "2404" "1247"
```

```
## [1] 24 21 20
```

```
ages <- c(22, 10, 19)
##combine into a matrix
agestringmat <- cbind((stringsampids),
                      (ages))
agestringmat
```

```
##      [,1]   [,2]
## [1,] "878"  "22"
## [2,] "2404" "10"
## [3,] "1247" "19"
```

```
class(agestringmat)
```

```
## [1] "matrix"
```

```
mean(agestringmat[, 2])
```

data.frames: advantages and disadvantages

- Can store elements of *different types*- e.g., a character vector; a factor vector; a numeric vector, and most data in social science are composed of heterogeneous types

data.frames: advantages and disadvantages

- The downside, because the elements are of different types, can't use for linear algebra operations so need to convert characters and factors into numeric type before changing a data.frame into a matrix (e.g., if have a variable, gender, coded as male and female, assign a numeric value to each level)

Where we're going next

We've reviewed:

1. Vectors: how to create, how to extract elements, how to check/change their type

Where we're going next

We've reviewed:

1. Vectors: how to create, how to extract elements, how to check/change their type
2. How to stack row vectors (*rbind*) or place column vectors side by side (*cbind*) into a data.frame or matrix

Where we're going next

We've reviewed:

1. Vectors: how to create, how to extract elements, how to check/change their type
2. How to stack row vectors (*rbind*) or place column vectors side by side (*cbind*) into a data.frame or matrix
3. Rough sketch of differences between data.frame and matrix for storing two-dimensional data

What's up next

Now, we're going to:

1. Review common operations performed on data.frames, since that's the form in which you'll keep the majority of your data for research– many of these operations can also be performed on matrices, and before concluding, we'll review some small differences that emerge when working with data.frames versus matrices
2. Discuss dplyr, a package that aims to simplify some of the data manipulation we review in the first part of the lecture

What's up next

Now, we're going to:

1. Review common operations performed on data.frames, since that's the form in which you'll keep the majority of your data for research– many of these operations can also be performed on matrices, and before concluding, we'll review some small differences that emerge when working with data.frames versus matrices
2. Discuss dplyr, a package that aims to simplify some of the data manipulation we review in the first part of the lecture
3. Discuss three tools useful for manipulating data.frames and matrices:

What's up next

Now, we're going to:

1. Review common operations performed on data.frames, since that's the form in which you'll keep the majority of your data for research– many of these operations can also be performed on matrices, and before concluding, we'll review some small differences that emerge when working with data.frames versus matrices
2. Discuss dplyr, a package that aims to simplify some of the data manipulation we review in the first part of the lecture
3. Discuss three tools useful for manipulating data.frames and matrices:
   3.1 Logical statements

What's up next

Now, we're going to:

1. Review common operations performed on data.frames, since that's the form in which you'll keep the majority of your data for research– many of these operations can also be performed on matrices, and before concluding, we'll review some small differences that emerge when working with data.frames versus matrices
2. Discuss dplyr, a package that aims to simplify some of the data manipulation we review in the first part of the lecture
3. Discuss three tools useful for manipulating data.frames and matrices:
    3.1 Logical statements
    3.2 Control structures (if, ifelse, etc.)

What's up next

Now, we're going to:

1. Review common operations performed on data.frames, since that's the form in which you'll keep the majority of your data for research– many of these operations can also be performed on matrices, and before concluding, we'll review some small differences that emerge when working with data.frames versus matrices
2. Discuss dplyr, a package that aims to simplify some of the data manipulation we review in the first part of the lecture
3. Discuss three tools useful for manipulating data.frames and matrices:
    3.1 Logical statements
    3.2 Control structures (if, ifelse, etc.)
    3.3 Focus on for loops

data.frame: extracting columns

Indexing:

data.frame[row index, column index]

For most data structures, this will be:

data.frame[observations, variables]

How do we extract columns from the data.frame?

data.frame[observations, variables]

data.frames: extracting columns

Ways to extract columns:

- *Less recommended*: use column index (less recommended because position might change as you add/remove columns from the data)

data.frames: extracting columns

Ways to extract columns:

- *Less recommended*: use column index (less recommended because position might change as you add/remove columns from the data)
- *More recommended*: use name of columns

data.frames: extracting columns

Ways to extract columns:

- *Less recommended*: use column index (less recommended because position might change as you add/remove columns from the data)
- *More recommended*: use name of columns
- If the columns of interest share a naming pattern, can use regular expressions (see Intermediate R module in DataCamp if want more info on regex)

data.frames: extracting columns

Ways to extract columns:

- *Less recommended*: use column index (less recommended because position might change as you add/remove columns from the data)
- *More recommended*: use name of columns
- If the columns of interest share a naming pattern, can use regular expressions (see Intermediate R module in DataCamp if want more info on regex)
- If extracting a column to use in some operation, often use: *data*$*variable*

data.frame: extracting columns

Example: extract the age, gender, income, and three relationship-related columns from the data

```
##way 1: using the column indices
addh2 <- addh[, c(2:4, 7:9)]
head(addh2, 3)
```

```
##   age gender income love nocheating money
## 1  20   male  15000   10         10     7
## 2  22   male  30000   10          5     1
## 3  19 female   1500   10         10     5
```

```
##way 2: using the column names
addh2 <- addh[, c("age",  "gender", "income",
         "love", "nocheating", "money")]
head(addh2, 3)
```

```
##   age gender income love nocheating money
## 1  20   male  15000   10         10     7
## 2  22   male  30000   10          5     1
## 3  19 female   1500   10         10     5
```

data.frame: extracting columns

```r
##way 3: illustration of indexing using data$variable
source("/Users/xyd/Desktop/IHS.R")
```

```
## Warning in file(filename, "r", encoding = encoding): cannot open file '/
## Users/xyd/Desktop/IHS.R': No such file or directory
```

```
## Error in file(filename, "r", encoding = encoding): cannot open the connection
```

```r
IHS(addh2$money)
```

```
## Error in IHS(addh2$money): could not find function "IHS"
```

```r
addh2$logmoney <- log(addh2$money)
head(addh2, 2)
```

```
##   age gender income love nocheating money logmoney
## 1  20   male  15000   10         10     7  1.94591
## 2  22   male  30000   10          5     1  0.00000
```

data.frame: extracting rows and logical operators

How do we extract rows from the data.frame?

data.frame[observations, variables]

Usually, relies on the main logical operators:

- equals: $==$

data.frame: extracting rows and logical operators

How do we extract rows from the data.frame?

data.frame[observations, variables]

Usually, relies on the main logical operators:

- equals: ==
- not equals: != (or ! in front of an identity statement)

data.frame: extracting rows and logical operators

How do we extract rows from the data.frame?

data.frame[observations, variables]

Usually, relies on the main logical operators:

- equals: $==$
- not equals: $!=$ (or ! in front of an identity statement)
- comparison: $<$, $\leq$, $>$, $\geq$

data.frame: extracting rows and logical operators

How do we extract rows from the data.frame?

data.frame[observations, variables]

Usually, relies on the main logical operators:

- equals: $==$
- not equals: $!=$ (or ! in front of an identity statement)
- comparison: $<$, $\leq$, $>$, $\geq$
- and: &

data.frame: extracting rows and logical operators

How do we extract rows from the data.frame?

data.frame[observations, variables]

Usually, relies on the main logical operators:

- equals: $==$
- not equals: $!=$ (or ! in front of an identity statement)
- comparison: $<$, $\leq$, $>$, $\geq$
- and: &
- or: $|$

How are logical operators useful?

Example: you've been running your analyses including the full AddHealth sample, but your adviser tells you that you should only include those older than college years. You want to subset the data to include only persons aged 22 and older.

How might we do this without logical operators?

```
#rank the respondents in order of age and manually look at which
#row forms the cutoff
#between 21 and 22, and then manually restrict to that row and higher
orderedaddh <- addh[order(addh$age), ]

#trying to find the 21/22 year old cutoff
orderedaddh[300:303, ]
```

```
##        id age gender income logincome    debt love nocheating money
## 2945 3790  19 female  10000  9.210340 yesdebt   10         10     1
## 2946 3791  19 female  15000  9.615805 yesdebt   10         10     4
## 2956 3805  19   male  15000  9.615805  nodebt   10         10     2
## 2960 3810  19   male   4000  8.294050  nodebt   10          9     5
##      paypercent logpaypercent
## 2945       54.8      4.003690
## 2946       68.4      4.225373
## 2956       68.4      4.225373
## 2960       31.4      3.446808
```

```
#not high enough, try again:
orderedaddh[1300:1303, ]
```

```
##     id age gender income logincome    debt love nocheating money
## 81 107  22 female  10000  9.210340 yesdebt   10         10     6
## 83 110  22 female  28000 10.239960  nodebt   10         10    10
## 85 112  22   male  33000 10.404263  nodebt   10         10     7
## 88 116  22 female   3000  8.006368 yesdebt   10         10     8
##    paypercent logpaypercent
```

Logical operators: advantages

Help us easily subset the data, and can also apply multiple criteria at once

data.frame: extracting rows and logical operators

```
##only want people 22 and over
head(addh[addh$age >= 22, ], 3)
```

```
##   id age gender income logincome   debt love nocheating money paypercent
## 2  4  22   male  30000 10.308953 nodebt   10          5     1       90.8
## 4  6  22 female  12000  9.392662 nodebt   10         10     9       56.3
## 6  8  25   male  30000 10.308953 nodebt   10         10    10       90.8
##   logpaypercent
## 2      4.508659
## 4      4.030695
## 6      4.508659
```

```
##want people whose income < 20000 but have no debt
head(addh[addh$debt == "nodebt" &
          addh$income < 20000, ], 3)
```

```
##   id age gender income logincome   debt love nocheating money paypercent
## 1  2  20   male  15000  9.615805 nodebt   10         10     7       64.2
## 3  5  19 female   1500  7.313220 nodebt   10         10     5       19.4
## 4  6  22 female  12000  9.392662 nodebt   10         10     9       56.3
##   logpaypercent
## 1      4.162003
## 3      2.965273
## 4      4.030695
```

data.frame: extracting rows and logical operators

```
##want people who either have income >= 80000 OR
##are over the age of 25
head(addh[addh$income >= 80000 |
          addh$age > 25,  ], 3)
```

```
##       id age gender income logincome   debt love nocheating money
## 71    95  27   male   3000  8.006368 nodebt    7         10     5
## 87   114  26 female  26800 10.196157 nodebt   10         10     9
## 104 138  26   male  14000  9.546813 nodebt   10         10     1
##      paypercent logpaypercent
## 71         25.1      3.222868
## 87         88.6      4.484132
## 104        62.4      4.133565
```

```
##no missing data in this particular cleaned data, but
##say there was, and want people not missing data
##for any of the relationship attitude variables
nomissrel <- addh[!is.na(addh$love) &
                  !is.na(addh$nocheating) &
                  !is.na(addh$money), ]

##more detail on what is.na(vector) is doing:
testvec <- c(NA, 5, NA, 6, 6)
is.na(testvec)
```

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

What's next

- We've reviewed how to create the different structures, how to extract elements (aka indexing), and how to manipulate in various other ways

What's next

- We've reviewed how to create the different structures, how to extract elements (aka indexing), and how to manipulate in various other ways
- All of the manipulation we've done thus far (with the exception of the ggplot graph) uses commands in *base R* (R's built-in functions)

What's next

- We've reviewed how to create the different structures, how to extract elements (aka indexing), and how to manipulate in various other ways
- All of the manipulation we've done thus far (with the exception of the ggplot graph) uses commands in *base R* (R's built-in functions)
- The Tidyverse way for manipulating data that can be more efficient/readable than base R in certain cases is to use the *dplyr* package

What's next

- We've reviewed how to create the different structures, how to extract elements (aka indexing), and how to manipulate in various other ways
- All of the manipulation we've done thus far (with the exception of the ggplot graph) uses commands in *base R* (R's built-in functions)
- The Tidyverse way for manipulating data that can be more efficient/readable than base R in certain cases is to use the *dplyr* package
- We'll review basics of how to translate base R into dplyr, for the most part using examples from the previous slides, but to become more "fluent", can practice with DataCamp dplyr module

Outline of dplyr review

- dplyr “verbs”:

Outline of dplyr review

- dplyr “verbs”:
  - select

Outline of dplyr review

- dplyr “verbs”:
  - select
  - filter

Outline of dplyr review

- dplyr "verbs":
  - select
  - filter
  - arrange

Outline of dplyr review

- dplyr “verbs”:
  - select
  - filter
  - arrange
  - mutate

Outline of dplyr review

- dplyr "verbs":
  - select
  - filter
  - arrange
  - mutate
  - group_by

Outline of dplyr review

- dplyr "verbs":
  - select
  - filter
  - arrange
  - mutate
  - group_by
  - summarise

Outline of dplyr review

- dplyr "verbs":
  - select
  - filter
  - arrange
  - mutate
  - group_by
  - summarise
- rename

Outline of dplyr review

- dplyr “verbs”:
  - select
  - filter
  - arrange
  - mutate
  - group_by
  - summarise
- rename
- chaining together verbs with pipe operator %>%

dplyr: basic structure of verbs

verb(name of data.frame or object, operation 1 to perform, operation 2 to perform...)

select: a way to extract columns

Can be used in combination with other dplyr verbs such as: contains, starts_with, and ends_with

Example: extract any column with the word “pay”: paypercent and logpaypercent

```
##base R
paycol <- addh[, c("paypercent", "logpaypercent")]
head(paycol, 3)
```

```
##   paypercent logpaypercent
## 1       64.2      4.162003
## 2       90.8      4.508659
## 3       19.4      2.965273
```

```
##dplyr
paycold <- select(addh, dplyr::contains("pay"))
# note here I used namespace b/c of dyplr/purrr conflict
head(paycold, 3)
```

```
##   paypercent logpaypercent
## 1       64.2      4.162003
## 2       90.8      4.508659
## 3       19.4      2.965273
```

filter: a way to extract rows

Can be used in combination with logical statements we learned earlier

Example: extract observations with an income <20,000 year but no debt

```
##base R
nodebt <- addh[addh$debt == "nodebt" &
      addh$income < 20000, ]
nrow(nodebt)
```

```
## [1] 1275
```

```
##dplyr
nodebtd <- filter(addh, debt == "nodebt" &
                    income < 20000)
nrow(nodebtd)
```

```
## [1] 1275
```

arrange: a way to arrange rows by the order of their column values

Example: find the two observations who think money is extremely important for a relationship (10 on money variable) but who pay for the fewest percentage of dates (paypercent)

```
##base R
moneyint <- addh[addh$money == 10, ]
moneyint[order(moneyint$paypercent), ][1:2, ]
```

```
##        id age gender income logincome    debt love nocheating money
## 2379 3058  21 female   1044  6.950815 yesdebt   10         10    10
## 390   506  24 female   1200  7.090077  nodebt   10         10    10
##      paypercent logpaypercent
## 2379       18.7      2.928524
## 390        19.0      2.944439
```

```
##dplyr
arrange(filter(addh, money == 10), paypercent)[1:2, ]
```

```
##     id age gender income logincome    debt love nocheating money
## 1 3058  21 female   1044  6.950815 yesdebt   10         10    10
## 2  506  24 female   1200  7.090077  nodebt   10         10    10
##   paypercent logpaypercent
## 1       18.7      2.928524
## 2       19.0      2.944439
```

mutate: a way to add new variables to the data.frame

Example: add a variable with the average rating for nocheating, money, and love's importance for a relationship (sum divided by 3) and another variable that logs that rating

```
##base R
addh$rateavg <- rowSums(addh[, c("love", "money", "nocheating")])/3
addh$rateavglog <- log(addh$rateavg)
head(addh[, c("love", "money", "nocheating", "rateavg", "rateavglog")], 3)
```

```
##   love money nocheating  rateavg rateavglog
## 1   10     7         10 9.000000   2.197225
## 2   10     1          5 5.333333   1.673976
## 3   10     5         10 8.333333   2.120264
```

```
##dplyr
addhd <- mutate(addh,
               rateavg = (love + money + nocheating)/3,
               rateavglog = log(rateavg))
head(select(addh, love, money, nocheating, rateavg, rateavglog), 3)
```

```
##   love money nocheating  rateavg rateavglog
## 1   10     7         10 9.000000   2.197225
## 2   10     1          5 5.333333   1.673976
## 3   10     5         10 8.333333   2.120264
```

```
# can you guys put this into tidyverse/pipe syntex?
addhd %>%
```

group_by and summarise: a way to collapse data by category and generate summary statistics

Example:

1. Group by gender
2. Generate a summary statistic of not cheating's importance on that grouped data

```
##base R- will learn apply family tomorrow
tapply(addh$nocheating, addh$gender, mean)
```

```
##   female     male
## 9.852698 9.612203
```

group_by and summarise: a way to collapse data by category and generate summary statistics

```
gender_group <- group_by(addh, gender)
summarise(gender_group,
          meannocheat = mean(nocheating))
```

```
## # A tibble: 2 x 2
##   gender meannocheat
##   <fct>        <dbl>
## 1 female        9.85
## 2 male          9.61
```

group_by and summarise: a way to collapse data by category and generate summary statistics

Summarise also has a number of verbs for creating summary statistics:

1. n(): count the elements in a group

group_by and summarise: a way to collapse data by category and generate summary statistics

Summarise also has a number of verbs for creating summary statistics:

1. n(): count the elements in a group
2. n_distinct(): count the distinct elements in a group

group_by and summarise: a way to collapse data by category and generate summary statistics

Summarise also has a number of verbs for creating summary statistics:

1. n(): count the elements in a group
2. n_distinct(): count the distinct elements in a group
3. first: list the first element (would usually use in combo with arrange)

group_by and summarise: a way to collapse data by category and generate summary statistics

Summarise also has a number of verbs for creating summary statistics:

1. n(): count the elements in a group
2. n_distinct(): count the distinct elements in a group
3. first: list the first element (would usually use in combo with arrange)
4. last: list the lest element (same as above)

group_by and summarise

Example: find: 1) the number of females and males by debt status, 2) the percentage in each debt x gender category as a fraction of all observations; 3) the number of distinct ratings of love's importance in each of these debt x gender categories

```
##base R
table(addh$gender, addh$debt); prop.table(table(addh$gender, addh$debt))
```

```
##
##          nodebt yesdebt
##   female    828     747
##   male      907     568
```

```
##
##             nodebt   yesdebt
##   female 0.2714754 0.2449180
##   male   0.2973770 0.1862295
```

```
##distinct
tapply(addh$love, list(addh$gender, addh$debt),
                  function(x){length(unique(x))})
```

```
##        nodebt yesdebt
## female      8       7
## male       10       8
```

group_by and summarise: a way to collapse data by category and generate summary statistics

```r
##dplyr
genderdebt <- group_by(addh, gender, debt)
# note I use namespace explictly, dplyr/plyr conflict
dplyr::summarise(genderdebt,
          count = n(),
          percent = n()/nrow(addh),
          distinctlove = n_distinct(love))
```

```
## # A tibble: 4 x 5
## # Groups:   gender [2]
##   gender debt    count percent distinctlove
##   <fct>  <fct>   <int>   <dbl>        <int>
## 1 female nodebt    828   0.271            8
## 2 female yesdebt   747   0.245            7
## 3 male   nodebt    907   0.297           10
## 4 male   yesdebt   568   0.186            8
```

Summary: base R versus dplyr

Goal	base R	dplyr
Extract columns	data[, c("col1", "col2"...)]	select(data, col1, col2...)
Extract rows	data[variable == condition,]	filter(data, variable == condition)
Arrange by column value (default = ascending)	data[order(variable),]	arrange(data, variable)
Add new variables to data.frame	data$newvar <- log(data$oldvar)	mutate(data, newvar = log(oldvar))
Grouped summary statistics	tapply(data$outcomevar, list(data$groupvar1, data$groupvar2...), function to perform)	summarise(group_by(data, groupvar1, groupvar2), stat1 = function 1 to perform, stat2 = function 2 to perform...)

Rename

- Rename: you can use plyr::rename() as a function to modify names by name, not position.

```
# let's use one of the built in R datasets mtcars
head(mtcars, 3)
```

```
##                mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4     21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710    22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
```

```
# what if we want the names to be more informative,
better.mtcars <- plyr::rename(mtcars,
  c(disp = "displacement",
  mpg = 'Miles per Gallon'))
head(better.mtcars, 3)
```

```
##               Miles per Gallon cyl displacement  hp drat    wt  qsec vs am
## Mazda RX4                 21.0   6          160 110 3.90 2.620 16.46  0  1
## Mazda RX4 Wag             21.0   6          160 110 3.90 2.875 17.02  0  1
## Datsun 710                22.8   4          108  93 3.85 2.320 18.61  1  1
##               gear carb
## Mazda RX4        4    4
## Mazda RX4 Wag    4    4
```

Rename

- Rename: you can use plyr::rename() as a function to modify names by name, not position.
- You can rename numerous columns by using c() to produce a 1-D array to pass to the replace position

```
# let's use one of the built in R datasets mtcars
head(mtcars, 3)
```

```
##                mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4     21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710    22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
```

```
# what if we want the names to be more informative,
better.mtcars <- plyr::rename(mtcars,
  c(disp = "displacement",
  mpg = 'Miles per Gallon'))
head(better.mtcars, 3)
```

```
##               Miles per Gallon cyl displacement  hp drat    wt  qsec vs am
## Mazda RX4                 21.0   6          160 110 3.90 2.620 16.46  0  1
## Mazda RX4 Wag             21.0   6          160 110 3.90 2.875 17.02  0  1
## Datsun 710                22.8   4          108  93 3.85 2.320 18.61  1  1
##               gear carb
## Mazda RX4        4    4
## Mazda RX4 Wag    4    4
```

Combining multiple verbs with piping

- You'll notice that the example on the previous slides combines multiple actions:

Combining multiple verbs with piping

- You'll notice that the example on the previous slides combines multiple actions:
- Pipes provide a way to chain together multiple verbs in a specified order.

Combining multiple verbs with piping

- You'll notice that the example on the previous slides combines multiple actions:
- Pipes provide a way to chain together multiple verbs in a specified order.
- Pipes (%>%) comes from the *magrittr* package with two aims: to decrease development time and to improve readability and maintainability of code.

Combining multiple verbs with piping

- You'll notice that the example on the previous slides combines multiple actions:
- Pipes provide a way to chain together multiple verbs in a specified order.
- Pipes (%>%) comes from the *magrittr* package with two aims: to decrease development time and to improve readability and maintainability of code.
- This operator %>% allow you to pipe a value forward into an expression or function call; something along the lines of *x %>% f*, rather than *f(x)*. It might be helpful to think of this as ... then...

Piping Functional Sequence

The basic (pseudo) usage of the pipe operator goes something like this:

```
awesome_data <-
raw_interesting_data %>%
transform(somehow) %>%
filter(the_good_parts) %>%
finalize
```

This takes an input, an output, and a sequence transformations. That's suprisingly close to the definition of a function, so magrittr is really just a convenient way of of defining and applying a function. (Also try command + shift + m for a nice short cut!)

Example of combining multiple verbs with piping

Example: - Group the data by gender and debt status - Find the average rating of love, no cheating, and money's importance for a relationship in each group - Arrange the groups by their rating of money's importance to a relationship from the highest to rating to the lowest rating

What would this look like, still using dplyr, but without piping? A nested mess...

```
arrange(summarise(group_by(addh, gender, debt),
                       nocheatavg = mean(nocheating),
                       loveavg = mean(love),
                       moneyavg = mean(money)), desc(moneyavg))
```

```
## # A tibble: 4 x 5
## # Groups:   gender [2]
##   gender debt    nocheatavg loveavg moneyavg
##   <fct>  <fct>        <dbl>   <dbl>    <dbl>
## 1 male   nodebt        9.60    9.54     6.40
## 2 female nodebt        9.84    9.76     6.40
## 3 female yesdebt       9.87    9.83     6.38
## 4 male   yesdebt       9.62    9.60     6.25
```

Piping: begin from the "most nested"/first operation and move to the last

1) Group the data by gender and debt status; 2) find the avg. rating of love, no cheating, and money's importance; 3) arrange the groups from rating money's importance the highest to rating it the lowest

- Without piping:

```
arrange(summarise(group_by(addh, gender, debt),
nocheatavg = mean(nocheating), loveavg = mean(love), moneyavg =
mean(money)), desc(moneyavg))
```

```
addh %>%
group_by(gender, debt) %>%
summarise(nocheatavg = mean(nocheating), loveavg = mean(love), moneyavg =
mean(money)) %>%
arrange(desc(moneyavg))
```

Piping: begin from the "most nested"/first operation and move to the last

1) Group the data by gender and debt status; 2) find the avg. rating of love, no cheating, and money's importance; 3) arrange the groups from rating money's importance the highest to rating it the lowest

- Without piping:

```
arrange(summarise(group_by(addh, gender, debt),
nocheatavg = mean(nocheating), loveavg = mean(love), moneyavg =
mean(money)), desc(moneyavg))
```

- With piping:

```
addh %>%
group_by(gender, debt) %>%
summarise(nocheatavg = mean(nocheating), loveavg = mean(love), moneyavg =
mean(money)) %>%
arrange(desc(moneyavg))
```

Implementing in R with pipes

And as a bonus, rename the columns to be more elaborative

```
addh %>%
  group_by(gender, debt) %>%
  summarise(nocheatavg = mean(nocheating),
            loveavg = mean(love),
            moneyavg = mean(money)) %>%
  arrange(desc(moneyavg)) %>%
  plyr::rename(c("nocheatavg" = "no cheating average", "loveavg" = "love average", "moneyavg" =
```

```
## # A tibble: 4 x 5
## # Groups:   gender [2]
##   gender debt    `no cheating average` `love average` `money average`
##   <fct>  <fct>                   <dbl>          <dbl>           <dbl>
## 1 male   nodebt                   9.60           9.54            6.40
## 2 female nodebt                   9.84           9.76            6.40
## 3 female yesdebt                  9.87           9.83            6.38
## 4 male   yesdebt                  9.62           9.60            6.25
```

data.frame: using control structures for variable construction

Brief disclaimer: control structures are not specific to data.frames– usually, we use them to check vectors and we can also use them in the context of matrices. But here, we're lumping them in with data.frames because we often use these statements to construct new variables of interest to add to our data

data.frame: using control structures for variable construction

How do we construct new variables based on more complicated sequences of logical operators?

Example Task: want to create a new variable, *loveormoney*, that takes on one of three values:

- Equally important: respondent ranked the two as equally important

How would we do the Example Task without the use of control structures?

data.frame: using control structures for variable construction

How do we construct new variables based on more complicated sequences of logical operators?

Example Task: want to create a new variable, *loveormoney*, that takes on one of three values:

- Equally important: respondent ranked the two as equally important
- Love more important: respondent ranked love as more important than money

How would we do the Example Task without the use of control structures?

data.frame: using control structures for variable construction

How do we construct new variables based on more complicated sequences of logical operators?

Example Task: want to create a new variable, *loveormoney*, that takes on one of three values:

- Equally important: respondent ranked the two as equally important
- Love more important: respondent ranked love as more important than money
- Money more important: respondent ranked money as more important than love

How would we do the Example Task without the use of control structures?

data.frame: using control structures for variable construction

Not really possible– imagine doing the following by hand for all 3050 observations

```
addhtest <- addh
#create a love or money variable filled with missing data
addhtest$loveormoney <- NA

#manually code a new variable based on comparing
#the ratings
addhtest[c(1, 6, 9), c("id", "love", "money")] #obs to code
```

```
##   id love money
## 1  2   10     7
## 6  8   10    10
## 9 11    5    10
```

```
addhtest$loveormoney[1] <- "lovemoreimport"
addhtest$loveormoney[6] <- "equal"
addhtest$loveormoney[9] <- "moneymoreimport"
addhtest[c(1, 6, 9), c("id", "love", "money", "loveormoney")]
```

```
##   id love money     loveormoney
## 1  2   10     7  lovemoreimport
## 6  8   10    10           equal
## 9 11    5    10 moneymoreimport
```

How would we do the example task without the use of control structures?

Rather than checking the condition by hand (whether a respondent's rating for the love variable exceeded, equaled, or was less than their rating for the money's importance variable), use R to check conditions by combining the logical operators we previously reviewed with "control structures"

If Else: common control structures for variable construction:

ifelse(logical test, what to do if true, what to do if false)

if(logical test, what to do if true), else(what to do if false)

if(logical test, what to do if true), else(logical test, what to do if true), else(...)...

data.frame: using control structures for variable construction

```
##truncated ifelse for use with one logical statement
##here, coding 1 = money more important, 0 = equal or less
addh2$moneymoreimport <- ifelse(addh2$money >
                                addh2$love, 1, 0)
head(addh2[, c("love", "money", "moneymoreimport")], 3)
```

```
##   love money moneymoreimport
## 1   10     7               0
## 2   10     1               0
## 3   10     5               0
```

```
##can combine multiple into a nested ifelse
addh2$loveormoney <- ifelse(addh2$money >
                            addh2$love, "moneymoreimport",
                            ifelse(addh2$money ==
                                   addh2$love,
                                    "moneyequal",
                                   "lovemoreimport"))
head(addh2[, c("love", "money", "loveormoney")], 3)
```

```
##   love money    loveormoney
## 1   10     7 lovemoreimport
## 2   10     1 lovemoreimport
## 3   10     5 lovemoreimport
```

data.frame: using control structures for variable construction

- What happens if the sequence of conditionals within the "ifelse" statement gets too long and complicated?

data.frame: using control structures for variable construction

- What happens if the sequence of conditionals within the "ifelse" statement gets too long and complicated?
- Then, can chain together the following:

data.frame: using control structures for variable construction

- What happens if the sequence of conditionals within the “ifelse” statement gets too long and complicated?

if(logical statement){

what to do

}

data.frame: using control structures for variable construction

- What happens if the sequence of conditionals within the "ifelse" statement gets too long and complicated?
- Then, can chain together the following:

if(logical statement){

what to do

}

data.frame: using control structures for variable construction

- What happens if the sequence of conditionals within the "ifelse" statement gets too long and complicated?
- Then, can chain together the following:

if(logical statement){

what to do

} else if (logical statement){

what to do

}

data.frame: using control structures for variable construction

- What happens if the sequence of conditionals within the "ifelse" statement gets too long and complicated?
- Then, can chain together the following:

if(logical statement){

what to do

} else if (logical statement){

what to do

} else if (logical statement){

what to do

}

data.frame: using control structures for variable construction

- What happens if the sequence of conditionals within the "ifelse" statement gets too long and complicated?
- Then, can chain together the following:

if(logical statement){

  what to do

} else if (logical statement){

  what to do

} else if (logical statement){

  what to do

} else {

  what to do with values that didn't meet any of the above conditions

}

data.frame: using control structures for variable construction

Example: create a variable, *loveormoney2*, that takes on the following values:

- "extreme" if person either codes love or money as 9 or 10

data.frame: using control structures for variable construction

Example: create a variable, *loveormoney2*, that takes on the following values:

- "extreme" if person either codes love or money as 9 or 10
- lovegreater if love $>$ money

data.frame: using control structures for variable construction

Example: create a variable, *loveormoney2*, that takes on the following values:

- "extreme" if person either codes love or money as 9 or 10
- lovegreater if love $>$ money
- same if love $==$ money

data.frame: using control structures for variable construction

Example: create a variable, *loveormoney2*, that takes on the following values:

- "extreme" if person either codes love or money as 9 or 10
- lovegreater if love $>$ money
- same if love $==$ money
- moneygreater if money $>$ love

data.frame: using control structures for variable construction

If we were doing this using the *ifelse* command, we would feed that command the love and money vectors and it would perform the logical check/operation on *all elements of those vectors*. So in our example:

```
addh2$moneymoreimport <- ifelse(addh2$money >
                                addh2$love, 1, 0)
```

- The above command is looking through *each and every observation's* love and money values, seeing whether money is greater, and then coding the new variable appropriately

data.frame: using control structures for variable construction

If we were doing this using the *ifelse* command, we would feed that command the love and money vectors and it would perform the logical check/operation on *all elements of those vectors*. So in our example:

```
addh2$moneymoreimport <- ifelse(addh2$money >
                                addh2$love, 1, 0)
```

- The above command is looking through *each and every observation's* love and money values, seeing whether money is greater, and then coding the new variable appropriately
- In contrast, if, else if, else sequences only look at the FIRST element of a vector, so in order to do this sort of checking for every observation they need to be embedded in what's called a *for loop*

data.frame: using control structures for variable construction

If we were doing this using the *ifelse* command, we would feed that command the love and money vectors and it would perform the logical check/operation on *all elements of those vectors*. So in our example:

```
addh2$moneymoreimport <- ifelse(addh2$money >
                                addh2$love, 1, 0)
```

- The above command is looking through *each and every observation's* love and money values, seeing whether money is greater, and then coding the new variable appropriately
- In contrast, if, else if, else sequences only look at the FIRST element of a vector, so in order to do this sort of checking for every observation they need to be embedded in what's called a *for loop*
- How we'll proceed: 1) show how the if, else if, else sequence works outside the loop for one element/one observation; 2) generalize it into a for loop so that it checks every observation in the data

data.frame: using control structures for variable construction- logical with one observation

```
##choose an observation to test
participant1 <- addh2[1, ]
participant1[, c("age", "gender", "income", "love", "money")]
```

```
##   age gender income love money
## 1  20   male  15000   10     7
```

```
##run logical sequence
part1result <- c()
if(participant1$love >= 9 | participant1$money >= 9){
  part1result <- "extreme"
} else if (participant1$love > participant1$money){
  part1result <- "lovegreater"
} else if (participant1$love == participant1$money){
  part1result <- "same"
} else {
  part1result <- "moneygreater"
}
part1result
```

```
## [1] "extreme"
```

data.frame: for loops

To generalize so that it goes through all observations (all elements of the love and money vectors), we can embed the logical sequence inside another control structure, a *for loop*, which helps us iterate through elements of a vector, matrix, data.frame, or list

Tomorrow, we'll learn how to replicate most of the things that a for loop does using functions and the "apply" family (which have many advantages), but for now, we'll review *for* loops

data.frame: for loops

Two main ways to construct:

1. Go through every element of a vector:

   for(i in vector){

      what to do

   }

data.frame: for loops

Two main ways to construct:

1. Go through every element of a vector:

   for(i in vector){

   what to do

   }
2. Iterate through a set number of elements:

data.frame: for loops

Two main ways to construct:

1. Go through every element of a vector:

   for(i in vector){

   what to do

   }
2. Iterate through a set number of elements:
   2.1 Another way of writing the for loop from number 1:

       for(i in 1:length(vector)){

       what to do

       }

data.frame: for loops

Two main ways to construct:

1. Go through every element of a vector:

   for(i in vector){

      what to do

   }
2. Iterate through a set number of elements:
   2.1 Another way of writing the for loop from number 1:

       for(i in 1:length(vector)){

          what to do

       }
   2.2 A for loop that iterates starting at the second element:
       for(i in 2:length(vector)){

          what to do

       }

data.frame: for loops

Using the 2.1 way of for loop construction, how do we generalize the expression that we applied to one observation? Here's what the for loop will do with $i = 1$ and $i = 2$

```
part1result <- c()
##run logical sequence
if(participant1$love >= 9 | participant1$money >= 9){
  part1result <- "extreme"
} else if (participant1$love > participant1$money){
  part1result <- "lovegreater"
} else if (participant1$love == participant1$money){
  part1result <- "same"
} else {
  part1result <- "moneygreater"
}
##what if we wanted to repeat with participant 2?
participant2 <- addh2[2, ]
##run sequence with participant 2
part2result <- c()
if(participant2$love >= 9 | participant2$money >= 9){
  part2result <- "extreme"
} else if (participant2$love > participant2$money){
  part2result <- "lovegreater"
} else if (participant2$love == participant2$money){
  part2result <- "same"
} else {
  part2result <- "moneygreater"
}
##combine the two answers
part1andpart2 <- c(part1result, part2result)
part1andpart2
```

```
## [1] "extreme" "extreme"
```

data.frame: for loops

Now copy and paste that code, altering where appropriate, 3048 more times (the number of remaining observations in the data)

data.frame: for loops

Now copy and paste that code, altering where appropriate, 3048 more times (the number of remaining observations in the data)... just kidding

data.frame: for loops

Steps to turn into a for loop:

1. Initialize a vector to store results- this time it will store an entire vector of results rather than one result. Can either do:

data.frame: for loops

Steps to turn into a for loop:

1. Initialize a vector to store results- this time it will store an entire vector of results rather than one result. Can either do:
   1.1 Initialize a vector of a certain length: *vec <- vector(length = desired length)*

data.frame: for loops

Steps to turn into a for loop:

1. Initialize a vector to store results- this time it will store an entire vector of results rather than one result. Can either do:
   1.1 Initialize a vector of a certain length: *vec* $<$- *vector(length* $=$ *desired length)*
   1.2 Initialize an empty vector: *vec* $<$- *c()*

data.frame: for loops

Steps to turn into a for loop:

1. Initialize a vector to store results- this time it will store an entire vector of results rather than one result. Can either do:
   1.1 Initialize a vector of a certain length: *vec* <- *vector(length = desired length)*
   1.2 Initialize an empty vector: *vec* <- *c()*
2. Use the for statement to tell the loop what to iterate through- in this case, we're iterating through $i = 1$, $i = 2 \ldots i = 3050$ (the number of observations in the data), pulling out each observation one by one and testing it via the logical sequence

data.frame: for loops

Steps to turn into a for loop:

1. Initialize a vector to store results- this time it will store an entire vector of results rather than one result. Can either do:
    1.1 Initialize a vector of a certain length: *vec <- vector(length = desired length)*
    1.2 Initialize an empty vector: *vec <- c()*
2. Use the for statement to tell the loop what to iterate through- in this case, we're iterating through $i = 1$, $i = 2 ... i = 3050$ (the number of observations in the data), pulling out each observation one by one and testing it via the logical sequence
3. Copy and paste the code from the single-observation case into the "meat" part of the *for* loop sandwich

data.frame: for loops

Steps to turn into a for loop:

1. Initialize a vector to store results- this time it will store an entire vector of results rather than one result. Can either do:
    1.1 Initialize a vector of a certain length: *vec <- vector(length = desired length)*
    1.2 Initialize an empty vector: *vec <- c()*
2. Use the for statement to tell the loop what to iterate through- in this case, we're iterating through $i = 1$, $i = 2 ... i = 3050$ (the number of observations in the data), pulling out each observation one by one and testing it via the logical sequence
3. Copy and paste the code from the single-observation case into the "meat" part of the *for* loop sandwich
4. For step three, make sure to add indexing where appropriate

data.frame: for loops

```
##initalize an empty vector
loveormoney2 <- c()
##for loop iterating through
##obs 1, obs 2...obs 3050
for(i in 1:nrow(addh2)){
  # index i^th element of addh2$love, check if statement is true
  if(addh2$love[i] >= 9 | addh2$money[i] >= 9){
  # if true, then in our empty vector, put "extreme"
  loveormoney2[i] <- "extreme"
  # else, if not true, then check if i^th love is greater than i^th money
} else if (addh2$love[i] > addh2$money[i]){
  # if that is true, then put "lovegreater"
  loveormoney2[i] <- "lovegreater"
} else if (addh2$love[i] == addh2$money[i]){
  loveormoney2[i] <- "same"
  # else for all other cases, money is greater
} else {
  loveormoney2[i] <- "moneygreater"
}
}
##append vector to data as new variable
addh2$loveormoney2 <- loveormoney2
##view some selected results
addh2[2505:2507, c("love", "money", "loveormoney2")]
```

```
##      love money loveormoney2
## 2505    6     8 moneygreater
## 2506   10    10      extreme
```

What comes up next in schematic. . .

	Homogeneous elements	**Heterogeneous elements**
1-dimensional	Vector	
2-dimensional	Matrix	Data.frame

Will review more after linear algebra lecture, for now, will just discuss as a container for the results of for loops

For now, review matrices as a container for the outcomes of for loops

- In previous example, each iteration of the for loop generated a *single element*. The result was a vector of length i

For now, review matrices as a container for the outcomes of for loops

- In previous example, each iteration of the for loop generated a *single element*. The result was a vector of length *i*
- What happens when we want each iteration to generate a *vector*, with the result being a matrix of dimensions: *iterations* x *length of vector*?

For now, review matrices as a container for the outcomes of for loops

- In previous example, each iteration of the for loop generated a *single element*. The result was a vector of length i
- What happens when we want each iteration to generate a *vector*, with the result being a matrix of dimensions: *iterations* x *length of vector*?
- Can use a matrix to use store for loop results

matrices: using for loops to populate a matrix

Example: want to calculate distribution around mean rating of money's importance. To do that, we want to:

1. Draw 1000 samples of size 3050 (number of observations in data) with replacement and store the samples in a matrix where each row is a draw and each column is an element of that draw (so the matrix will be 1000 x 3050) inside the loop

matrices: using for loops to populate a matrix

Example: want to calculate distribution around mean rating of money's importance. To do that, we want to:

1. Draw 1000 samples of size 3050 (number of observations in data) with replacement and store the samples in a matrix where each row is a draw and each column is an element of that draw (so the matrix will be 1000 x 3050) inside the loop
2. Find mean of each of the 1000 samples outside the loop

matrices: using for loops to populate a matrix

Example: want to calculate distribution around mean rating of money's importance. To do that, we want to:

1. Draw 1000 samples of size 3050 (number of observations in data) with replacement and store the samples in a matrix where each row is a draw and each column is an element of that draw (so the matrix will be 1000 x 3050) inside the loop
2. Find mean of each of the 1000 samples outside the loop
3. Plot the distribution of that mean outside the loop

matrices: using for loops to populate a matrix

```
# initialize empty matrix, good to preallocate space
set.seed(1234)
sampmat <- matrix(NA, nrow = 1000, ncol = 3050)
# iterate through each row of the matrix
for(i in 2:nrow(sampmat)){
  # and fill it with a sample of size 10 from the data
  draws <- sample(addh$money, size = 3050, replace= TRUE)
  # note that because each i-th sample is filling a row,
  # we add that sample to the matrix by indexing the i-th row
  sampmat[i, ] <- draws
}

# this is basically bootstrapping!
# check to make sure the for loop properly populated the matrix
sampmat[1:2, 1:10]
```

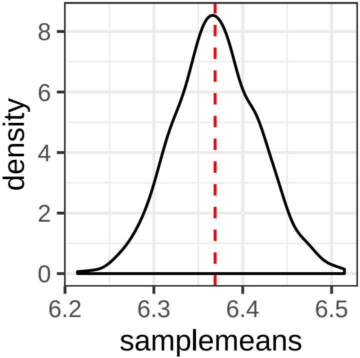
```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]   NA   NA   NA   NA   NA   NA   NA   NA   NA    NA
## [2,]    5    2    7    5    7    5    5    8    5    10
```

```
# find mean of each 1000 samples
samplemeans <- rowMeans(sampmat)
```

matrices: using for loops to populate a matrix

```r
# plot distribution of mean ratings
# adding a vertical line for observed mean
ggplot(as.data.frame(samplemeans), aes(x = samplemeans)) +
  geom_density() +
  geom_vline(xintercept = mean(addh$money),
             col = "red", linetype = "dashed") +
  theme_bw()
```

```
## Warning: Removed 1 rows containing non-finite values (stat_density).
```



Briefly: lists

- Like data.frames but unlike matrices/vectors, can handle elements of different types (e.g., character and numeric)

Briefly: lists

- Like data.frames but unlike matrices/vectors, can handle elements of different types (e.g., character and numeric)
- Unlike matrices and data.frames, can handle elements of different-lengths (e.g., you can't have a data.frame where one column is 1 x 49 and the other column is 1 x 10)

Lists: how to create a list

Just a general tip that a lot of R package outputs, including the standard output from regression lm, are in forms of lists, so this may be helpful for you to learn how to extract elements from those outputs.

```
##basic way to create a list
listofthree <- list("one", c(1, 2), FALSE)
```

lists: how to extract elements

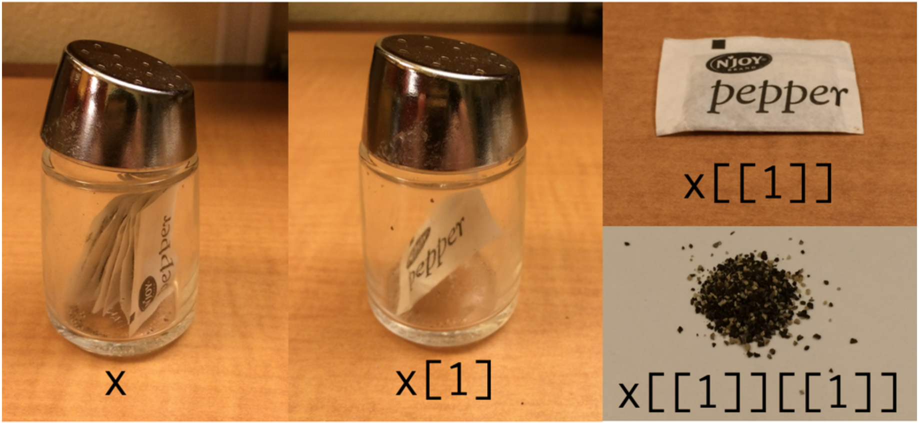


Figure 3:

Source: Hadley Wickham

lists: how to extract elements

```r
# how might we extract list 2, from the list of 3
listofthree[2]
```

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

```r
# what is this extraction?
str(listofthree[2])
```

```
## List of 1
##  $ : num [1:2] 1 2
```

```r
# what if we want to extract the numeric element
listofthree[[2]]
```

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

```r
# what is this extraction
str(listofthree[[2]])
```

```
##  num [1:2] 1 2
```

```r
# what is we want to extract the number 2 from the second list
listofthree[[2]][2]
```

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

Summing up

In this lecture, we've reviewed: - Indexing and manipulation of four main data structures: vectors, lists, matrices, and data.frames - Three tools in base R useful for data manipulation - Logical statements - Control structures - Focus on for loops - dplyr and pipes as a tool for data manipulation. Tidyverse overview.

Now: we'll practice integrating these concepts with the homework assignment looking at opposition to free trade and support for presidential candidates. We'll now draw pairs for the homework. Only one of you needs to submit the assignment via Piazza by the start of tomorrow's day (9 AM)

P.S. Many of you will encounter latex/knitting errors (the worst kind!), please try this guide[2].

[2] (https://www.eng.famu.fsu.edu/~dommelen/l2h/errors.html#misdol)

Drawing groups

For homework, optional