Programming day two: functions and the apply family

Methods camp instructors

September 5th, 2018

Outline

- Go through detailed breakdown of two functions we've already written to use over the summer/in yesterday's class, using these examples to illustrate thought process for how to approach writing a function

Outline

- Go through detailed breakdown of two functions we've already written to use over the summer/in yesterday's class, using these examples to illustrate thought process for how to approach writing a function
- Discuss two problems that emerge with functions and how to avoid:

Outline

- Go through detailed breakdown of two functions we've already written to use over the summer/in yesterday's class, using these examples to illustrate thought process for how to approach writing a function
- Discuss two problems that emerge with functions and how to avoid:
  - Function is too specific to a particular case; need to generalize to accomodate a broader range of situations

Outline

- Go through detailed breakdown of two functions we've already written to use over the summer/in yesterday's class, using these examples to illustrate thought process for how to approach writing a function
- Discuss two problems that emerge with functions and how to avoid:
  - Function is too specific to a particular case; need to generalize to accomodate a broader range of situations
  - You incorrectly pass a function its arguments

Outline

- Go through detailed breakdown of two functions we've already written to use over the summer/in yesterday's class, using these examples to illustrate thought process for how to approach writing a function
- Discuss two problems that emerge with functions and how to avoid:
  - Function is too specific to a particular case; need to generalize to accomodate a broader range of situations
  - You incorrectly pass a function its arguments
- Introduce the apply family in base R, an indispensible tool to use in conjunction with functions that both highlight and overcome some problems with the for loop method we learned yesterday

Simple For Loop Example

```
# Create a vector filled with random values
# we want to use a loop to get the square of this vector
vector1 <- c(10, 20, 30, 40, 50)

# how to write a loop

vector.sq <- c()
# step 1, workshop the loop with a single data point
#for(i in 1:length(vector1)) {
  # 1st element of `vector11` squared into `1st`-th position of `ve
  vector.sq[1] <- vector1[1] * vector1[1]
#}

vector.sq
```

```
## [1] 100
```

```
# then try again with another data point, imitating how the loop wo
# step 2, workshop the loop with another data point
```

Data we'll be working with

In-class lecture example: same data as day 1 (AddHealth data)

Data we'll be working with

In-class lecture example: same data as day 1 (AddHealth data)

Homework example: CDC data where parents report that a vaccine caused autism in their child

Data we'll be working with

Figure 1:

When do we write our own functions?

Times when user-written functions, often in conjunction with the apply family, come in handy, are:

- **Repeat a process**: on day 1, we reviewed how to use a for loop to avoid having to copy/paste code to repeat some process; functions can provide a more efficient and flexible way to avoid this repetition

When do we write our own functions?

Times when user-written functions, often in conjunction with the apply family, come in handy, are:

- **Repeat a process**: on day 1, we reviewed how to use a for loop to avoid having to copy/paste code to repeat some process; functions can provide a more efficient and flexible way to avoid this repetition
- **Transparency about what the code is doing**: R has a plethora of packages that have built-in functions for many things we might want to do- for instance, the dplyr functions we reviewed yesterday that help with summary statistics; functions that facilitate "bootstrapped" quantities of interest. But what these functions are doing internally can be a black box, so to make sure we understand what's going on, we may want to write the function ourselves

How to approach writing a function

- Ask yourself: "what problem am I trying to use this function to solve?"

How to approach writing a function

- Ask yourself: "what problem am I trying to use this function to solve?"
- Once you've identified the problem, try writing code *outside of the function* to deal with a few *simple cases* of the problem

How to approach writing a function

- Ask yourself: "what problem am I trying to use this function to solve?"
- Once you've identified the problem, try writing code *outside of the function* to deal with a few *simple cases* of the problem
- Then, see what you can do to generalize the code from step two so that it can handle a variety of versions of the problem

Basic structure of a function

```
functionname <- function(argument1, argument2, argument3...)
{
what to do
return(what to return)
}
```

Basic structure of a function

Rather than discussing this structure in the abstract, let's review some functions we have already used in the homework assignment or first day's lectures and go through step by step what they were doing

- Example from day 1 programming lecture of finding unique responses on a scale across different categories of assignment

Basic structure of a function

Rather than discussing this structure in the abstract, let's review some functions we have already used in the homework assignment or first day's lectures and go through step by step what they were doing

- Example from day 1 programming lecture of finding unique responses on a scale across different categories of assignment
- On Day 4, we will have an upcoming math lecture about a log-likelihood function that we feed into another function (*optim*) that we will discuss in length on Friday.

For each example, we'll discuss as a class:

1. What problem is the function trying to solve?

For each example, we'll discuss as a class:

1. What problem is the function trying to solve?
2. What are the function's arguments in this case? (we may also for shorthand refer to these as a function's inputs)

For each example, we'll discuss as a class:

1. What problem is the function trying to solve?
2. What are the function's arguments in this case? (we may also for shorthand refer to these as a function's inputs)
3. What is the function doing with those arguments?

For each example, we'll discuss as a class:

1. What problem is the function trying to solve?
2. What are the function's arguments in this case? (we may also for shorthand refer to these as a function's inputs)
3. What is the function doing with those arguments?
4. What does the function return? What class is it? (feel free to check using R)

Programming lecture example: using a function to repeat a process for different subsets of data

Problem to solve: find the number of unique ratings of love's importance in four different categories of respondents: gender (women versus men) x debt status (debt versus no debt)

Programming lecture example: why write a function for this problem?

How we would do manually- have to copy and paste repeatedly!:

```
##how we would do manually
femdebt <-  length(unique(addh$love[addh$gender == "female" &
                                    addh$debt == "yesdebt"]))
femdebt
```

```
## [1] 7
```

```
femnodebt <- length(unique(addh$love[addh$gender == "female" &
                                     addh$debt == "nodebt"]))
femnodebt
```

```
## [1] 8
```

```
maledebt <- length(unique(addh$love[addh$gender == "male" &
                                    addh$debt == "yesdebt"]))
maledebt
```

```
## [1] 8
```

```
malenodebt <- length(unique(addh$love[addh$gender == "male" &
                                      addh$debt == "nodebt"]))
malenodebt
```

Programming lecture example: how to generalize the specific code into a function

```
#focusing on two cases
femdebt <-  length(unique(addh$love[addh$gender == "female" &
                                    addh$debt == "yesdebt"]))
maledebt <- length(unique(addh$love[addh$gender == "male" &
                                    addh$debt == "yesdebt"]))
```

To generalize, *keep* the part of the expression we're holding constant during each copy (in green) and *replace* the part of the expression we're changing during each copy (in red) with a more general argument:

length(unique(addh$love[addh$gender == "female" & addh$debt == "debt"]))

becomes...

length(unique(**x**))

Translating into R

length(unique(**x**))

```
# Our custom function!
# x here is our placeholder for things we want to function to take in
nunique <- function(x){
  length(unique(x))
  }
```

Programming lecture example: feeding the function into another command to execute the function more efficiently

We have:

1. Identified the problem: finding the number of unique responses on a variable for four groups in the data

Programming lecture example: feeding the function into another command to execute the function more efficiently

We have:

1. Identified the problem: finding the number of unique responses on a variable for four groups in the data
2. Created a generalized function to solve the problem: the function can measure the number of unique elements of *any* vector x

Programming lecture example: feeding the function into another command to execute the function more efficiently

We have:

1. Identified the problem: finding the number of unique responses on a variable for four groups in the data
2. Created a generalized function to solve the problem: the function can measure the number of unique elements of *any* vector x
3. Now, we can think of ways to feed the function the vector with ratings of love's importance for each of the four groups (men with no debt, women with no debt, etc..) most efficiently

Programming lecture example: feeding the function into another command to execute the function more efficiently

```
# less efficient way to feed the function the appropriate vectors
nunique(x = addh$love[addh$gender == "female" & addh$debt == "nodebt"])
```

```
## [1] 8
```

```
nunique(x = addh$love[addh$gender == "male" & addh$debt == "nodebt"])
```

```
## [1] 10
```

```
# etc...
```

Programming lecture example: feeding the function into another command to execute the function more efficiently

We can make the function more efficient by embedding it in another command (tapply, which divides a vector based on one or more grouping variables and which we will discuss in detail later today):

```
# more efficient way to apply, specify name of function
Tapply.output <- tapply(addh$love, list(addh$gender, addh$debt),
                  nunique)

class(Tapply.output)
```

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

```
# include function directly in command
tapply(addh$love, list(addh$gender, addh$debt),
                  function(x){length(unique(x))})
```

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

```
# how we would do this in Tidyverse?
tidyverse.output <- addh %>%
  split(list(.$gender, .$debt)) %>%
  map(~nunique(.$love))
```

Apply vs. Tidyverse

- For the first case, we used the *apply* family of functions in R, which we will now review more formally.

Apply vs. Tidyverse

- For the first case, we used the *apply* family of functions in R, which we will now review more formally.
- For anything you might want to do with *apply*, you can probably accomplish the same thing in Tidyverse, with *Purrr* with *Dplyr*, as we see above.

Apply vs. Tidyverse

- For the first case, we used the *apply* family of functions in R, which we will now review more formally.
- For anything you might want to do with *apply*, you can probably accomplish the same thing in Tidyverse, with *Purrr* with *Dplyr*, as we see above.
- We will go into *apply* family, because they are useful, and you will see them a lot. And the syntex is confusing so you should know how to read it.

Apply vs. Tidyverse

- For the first case, we used the *apply* family of functions in R, which we will now review more formally.
- For anything you might want to do with *apply*, you can probably accomplish the same thing in Tidyverse, with *Purrr* with *Dplyr*, as we see above.
- We will go into *apply* family, because they are useful, and you will see them a lot. And the syntex is confusing so you should know how to read it.
- We already saw briefly how *Purrr* and how it fits in with *Dplyr* (and of course with *magrittr*). As a good exercise, we can rewrite some *apply* examples here into *Purrr* code.

Apply family: general motivation

- Main motivation: apply a function, either one we write ourselves or a built in function in R, repeatedly over some structured input

Apply family: general motivation

- Main motivation: apply a function, either one we write ourselves or a built in function in R, repeatedly over some structured input
- Didn't we use a for loop to do that? Some problems with for loops that motivate our use of the *apply* family:

Apply family: general motivation

- Main motivation: apply a function, either one we write ourselves or a built in function in R, repeatedly over some structured input
- Didn't we use a for loop to do that? Some problems with for loops that motivate our use of the *apply* family:
  - for loops often slower (less important)

Apply family: general motivation

- Main motivation: apply a function, either one we write ourselves or a built in function in R, repeatedly over some structured input
- Didn't we use a for loop to do that? Some problems with for loops that motivate our use of the *apply* family:
  - for loops often slower (less important)
  - More important: for loop saves all objects associated with intermediate steps in your environment, which can create unnecessary objects when you only care about the last one

Example of for loop cluttering our environment

Example: we want to clean the elements of a vector of place names in multiple steps (could combine into one step but the multiple are clearer). All we care about is the final vector of cleaned names.

Problem: using a *for loop* results in intermediate objects (*nonum* and *nospace*) that are stored/clutter our environment but that we don't want to be there

Example of for loop cluttering our environment

```
set.seed(1234)
sampmat <- matrix(NA, nrow = 15, ncol = 10)
# 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
  drawof10 <- sample(addh$money, size = 10, replace= FALSE)
  # 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, ] <- drawof10
}
head(sampmat, 2) #object we want stored
```

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

```
head(drawof10, 2) #useless intermediate objects, could use remove() to clean
```

```
## [1] 10  1
```

Apply-family: Global View

- apply

Apply-family: Global View

- apply
- Takes in array (1D or 2D, aka matrix), apply a function to it, then outputs a matrix/array, can set if you want function to apply to column or row or both.

Apply-family: Global View

- apply
- Takes in array (1D or 2D, aka matrix), apply a function to it, then outputs a matrix/array, can set if you want function to apply to column or row or both.
- tapply

Apply-family: Global View

- apply
- Takes in array (1D or 2D, aka matrix), apply a function to it, then outputs a matrix/array, can set if you want function to apply to column or row or both.
- tapply
- Apply a function to each cell of vectors (can be of differing length), which can be an unique combination of grouping factors. This can often also be accomplished in dplyr::group_by and summarise.

Apply-family: Global View

- apply
- Takes in array (1D or 2D, aka matrix), apply a function to it, then outputs a matrix/array, can set if you want function to apply to column or row or both.
- tapply
- Apply a function to each cell of vectors (can be of differing length), which can be an unique combination of grouping factors. This can often also be accomplished in dplyr::group_by and summarise.
- sapply

Apply-family: Global View

- apply
- Takes in array (1D or 2D, aka matrix), apply a function to it, then outputs a matrix/array, can set if you want function to apply to column or row or both.
- tapply
- Apply a function to each cell of vectors (can be of differing length), which can be an unique combination of grouping factors. This can often also be accomplished in dplyr::group_by and summarise.
- sapply
- flexibily takes in arrays or lists, does function on it, then returns simpliest form back

Apply-family: Global View

- apply
- Takes in array (1D or 2D, aka matrix), apply a function to it, then outputs a matrix/array, can set if you want function to apply to column or row or both.
- tapply
- Apply a function to each cell of vectors (can be of differing length), which can be an unique combination of grouping factors. This can often also be accomplished in dplyr::group_by and summarise.
- sapply
- flexibily takes in arrays or lists, does function on it, then returns simpliest form back
- *Briefly: lapply (complex version of sapply, returns list of the same length as whatever you input), mapply (a multivariate version of sapply, applies function to the first elements of each input data, then second and so on. Arguments are recycled if necessary.)*

apply: structure

- **Takes in**: arrays (0d = element, 1d = vector, 2d = matrix. . .)

apply(array to apply function to, whether to apply over rows or columns, function)

apply: structure

- **Takes in**: arrays (0d = element, 1d = vector, 2d = matrix. . .)
- **How to set up**:

apply(array to apply function to, whether to apply over rows or columns, function)

apply: structure

- **Takes in**: arrays (0d = element, 1d = vector, 2d = matrix. . .)
- **How to set up**:

apply(array to apply function to, whether to apply over rows or columns, function)

- For *whether to apply over rows or columns*: 1 = rows, 2 = columns, c(1, 2) = all elements

apply: structure

- **Takes in**: arrays (0d = element, 1d = vector, 2d = matrix. . .)
- **How to set up**:

apply(array to apply function to, whether to apply over rows or columns, function)

- For *whether to apply over rows or columns*: 1 = rows, 2 = columns, c(1, 2) = all elements
- **What it returns**: array

apply: apply function to each row

- apply(array to apply function to, 1, function)

$$\begin{bmatrix} id & age & gender \\ 1 & 22 & female \\ 2 & 23 & male \\ 3 & 21 & male \end{bmatrix}$$

apply: apply function to each row

- apply(array to apply function to, 1, function)
- Red = first iteration, orange = second iteration, blue = third iteration...

$$\begin{bmatrix} id & age & gender \\ 1 & 22 & female \\ 2 & 23 & male \\ 3 & 21 & male \end{bmatrix}$$

apply: apply function to each column,

- apply(array to apply function to, MARGIN = 2, function)

$$\begin{bmatrix} id & age & gender \\ 1 & 22 & female \\ 2 & 23 & male \\ 3 & 21 & male \end{bmatrix}$$

apply: apply function to each column,

- apply(array to apply function to, MARGIN = 2, function)
- Red = first iteration, orange = second iteration, blue = third iteration...

$$\begin{bmatrix} id & age & gender \\ 1 & 22 & female \\ 2 & 23 & male \\ 3 & 21 & male \end{bmatrix}$$

apply: apply function to each element

- apply(array to apply function to, c(1, 2), function)

id	*age*	*gender*
1	22	*female*
2	23	*male*
3	21	*male*

apply: apply function to each element

- apply(array to apply function to, c(1, 2), function)
- Red = first iteration, orange = second iteration, blue = third iteration, green = fourth iteration, yellow = fifth iteration...

$$\begin{bmatrix} id & age & gender \\ 1 & 22 & female \\ 2 & 23 & male \\ 3 & 21 & male \end{bmatrix}$$

apply: example of applying a function to every column

We can create our own version of colMeans, applying it to the age and percentage of dates paid for columns in our Addhealth data

```
##extract relevant columns
addh2 <- addh[, c("age", "paypercent")]

##find the mean of the columns using apply
apply(addh2, 2, function(x){mean(x)})
```

```
##         age paypercent
##    21.86984    60.27872
```

```
mean(addh2$age)
```

```
## [1] 21.86984
```

```
mean(addh2$paypercent)
```

```
## [1] 60.27872
```

```
##could also subset directly inside the apply
##to do in one step
apply(addh[, c("age", "paypercent")], 2, function(x){mean(x)})
```

```
##         age paypercent
##    21.86984    60.27872
```

apply: examples

- That example showed how we can replicate R's built-in functions using apply...but now let's use it for a fairly common practice within data analysis

apply: examples

- That example showed how we can replicate R's built-in functions using apply...but now let's use it for a fairly common practice within data analysis
- Say we had many skewed variables and we wanted to log those variables to make their distributiosn more symmetric, but we want to do this logging without going through each variable one by one

apply: examples

- That example showed how we can replicate R's built-in functions using apply...but now let's use it for a fairly common practice within data analysis
- Say we had many skewed variables and we wanted to log those variables to make their distributiosn more symmetric, but we want to do this logging without going through each variable one by one
- **Example with the AddHealth data**: log the income variable and the percentage of dates people pay for variable

apply: example of applying to every element

```
# create a logged pay percent and log income
loginclogpay <- apply(addh[, c("income", "paypercent")], c(1, 2), log)
# view the output of apply and check its class
head(loginclogpay, 3)
```

```
##          income paypercent
## [1,]  9.615805   4.162003
## [2,] 10.308953   4.508659
## [3,]  7.313220   2.965273
```

```
# Do this in Tidyverse
TV.loginclogpay <- addh %>%
  select(income, paypercent) %>%
  log
head(TV.loginclogpay, 3)
```

```
##      income paypercent
## 1  9.615805   4.162003
## 2 10.308953   4.508659
## 3  7.313220   2.965273
```

```
# can append to your original dataset by cbind
addh <- cbind(addh, TV.loginclogpay)
```

apply: example with a user-defined function

- In the previous example, we structured the apply command as follows:

apply(array, rows/columns/or both, built-in R function)

apply(array, rows/columns/or both, function we write ourselves)

apply: example with a user-defined function

- In the previous example, we structured the apply command as follows:

  apply(array, rows/columns/or both, built-in R function)

- But we can also use apply, and structure it in the same way, with a function we write ourselves:

  apply(array, rows/columns/or both, function we write ourselves)

apply: example with a user-defined function

Example with AddHealth data: say we wanted to center/standardize all the numeric variables to be in the unit of standard deviations with $\mu = 0$ and $\sigma = 1$ in the data by doing the following:

$$x\ rescaled_i = \frac{x_i - mean(x)}{sd(x)}$$

We could create a function to do this and then apply to all elements in our data (R has a built-in function called scale but it is relatively easy to do ourselves)

apply: use to transform all variables

```
##create normalize function
normalizefunc <- function(x){(x - mean(x))/sd(x)}

##apply normalize function to columns of dataframe restricted to
##numeric variables
addhnormalized <- apply(addhnumeric, 2, normalizefunc)

##check that it worked by manually normalizing the age variable and comparing
addh$normage <- (addh$age - mean(addh$age))/sd(addh$age)
head(addh[, "normage"], 3)
```

```
## [1] -1.04847063  0.07298665 -1.60919927
```

```
head(addhnormalized[, "age"], 3)
```

```
## [1] -1.04847063  0.07298665 -1.60919927
```

apply: use to transform all variables

```
##could also do in one step
addhnormalized2 <- apply(addhnumeric, 2, function(x){x- mean(x)/sd(x)})
head(addhnormalized2)
```

```
##            age    income logincome     love nocheating     money
## [1,]  7.736957 14998.499 -1.813336 1.200856  0.7030654  4.668633
## [2,]  9.736957 29998.499 -1.120189 1.200856 -4.2969346 -1.331367
## [3,]  6.736957  1498.499 -4.115921 1.200856  0.7030654  2.668633
## [4,]  9.736957 11998.499 -2.036479 1.200856  0.7030654  6.668633
## [5,]  6.736957 11998.499 -2.036479 1.200856 -0.2969346  4.668633
## [6,] 12.736957 29998.499 -1.120189 1.200856  0.7030654  7.668633
##      paypercent logpaypercent  income.1 paypercent.1
## [1,]   61.54019     -4.986063 -1.813336    -4.986063
## [2,]   88.14019     -4.639406 -1.120189    -4.639406
## [3,]   16.74019     -6.182793 -4.115921    -6.182793
## [4,]   53.64019     -5.117371 -2.036479    -5.117371
## [5,]   53.64019     -5.117371 -2.036479    -5.117371
## [6,]   88.14019     -4.639406 -1.120189    -4.639406
```

```
# Now let's try in Tidyverse! With Purrr
TV.addhnorm <- addh %>%
  map_if(is.numeric, ~normalizefunc(.)) %>%
  as.data.frame

head(TV.addhnorm[, "age"], 3)
```

```
## [1] -1.04847063  0.07298665 -1.60919927
```

tapply: structure

- **Takes in**: what's called a "ragged array"; in other words, will often be vectors of different lengths because we take a vector of interest and split it using one or more grouping variables into multiple vectors that likely have different lengths (recall homework examples where you wanted to group by educ level, then get mean of free trade)

tapply(vector of interest, grouping variable, function)

tapply(vector of interest, list(grouping variable 1, grouping variable 2...), function)

tapply: structure

- **Takes in**: what's called a "ragged array"; in other words, will often be vectors of different lengths because we take a vector of interest and split it using one or more grouping variables into multiple vectors that likely have different lengths (recall homework examples where you wanted to group by educ level, then get mean of free trade)
- **How to set up with one grouping variable**: \

tapply(vector of interest, grouping variable, function)

tapply(vector of interest, list(grouping variable 1, grouping variable 2...), function)

tapply: structure

- **Takes in**: what's called a "ragged array"; in other words, will often be vectors of different lengths because we take a vector of interest and split it using one or more grouping variables into multiple vectors that likely have different lengths (recall homework examples where you wanted to group by educ level, then get mean of free trade)
- **How to set up with one grouping variable**: \

tapply(vector of interest, grouping variable, function)

- **How to set up with multiple grouping variables**:

tapply(vector of interest, list(grouping variable 1, grouping variable 2...), function)

tapply: structure

- **Takes in**: what's called a "ragged array"; in other words, will often be vectors of different lengths because we take a vector of interest and split it using one or more grouping variables into multiple vectors that likely have different lengths (recall homework examples where you wanted to group by educ level, then get mean of free trade)
- **How to set up with one grouping variable**: \

tapply(vector of interest, grouping variable, function)

- **How to set up with multiple grouping variables**:

tapply(vector of interest, list(grouping variable 1, grouping variable 2...), function)

- **Returns**: by default, returns an array (a vector if you group by one variable, a matrix if you group by multiple variables)

tapply: structure

- **Takes in**: what's called a "ragged array"; in other words, will often be vectors of different lengths because we take a vector of interest and split it using one or more grouping variables into multiple vectors that likely have different lengths (recall homework examples where you wanted to group by educ level, then get mean of free trade)
- **How to set up with one grouping variable**: \

tapply(vector of interest, grouping variable, function)

- **How to set up with multiple grouping variables**:

tapply(vector of interest, list(grouping variable 1, grouping variable 2...), function)

- **Returns**: by default, returns an array (a vector if you group by one variable, a matrix if you group by multiple variables)
- Note that using tapply is similar to using the combination of group_by and summarise in dplyr

tapply: example with function in R

We went over many examples in programming lecture 1 when introducing dplyr's group_by and summarize, but here is another

Example: we want to construct a plot that shows the mean percentage of dates paid for by gender and rating of money's importance

```
##use tapply to construct the matrix, index is like group_by
payresults <- tapply(addh$paypercent, INDEX = list(addh$money, addh$gender), FUN = mean)
head(payresults, 3)
```

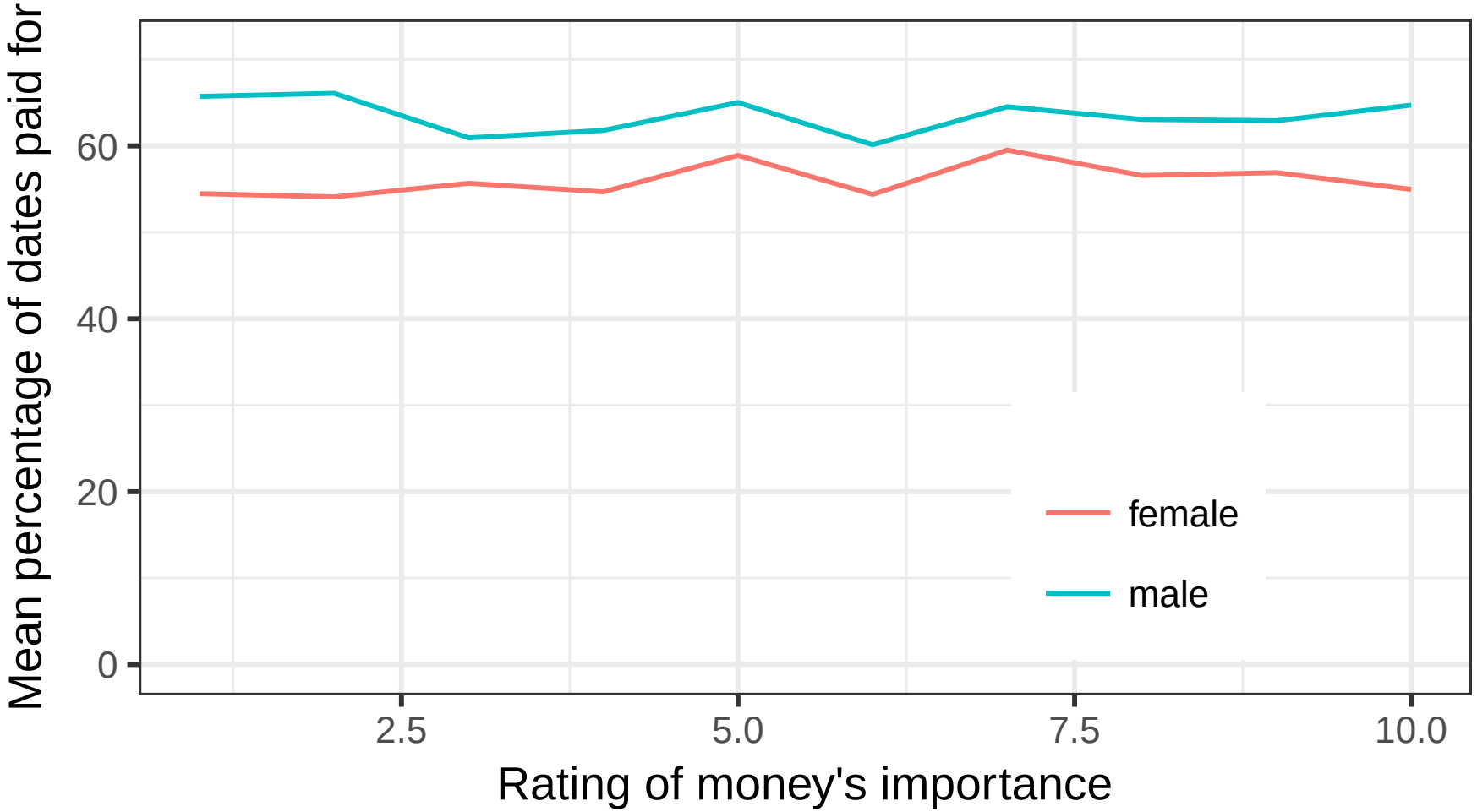
```
##     female     male
## 1 54.47708 65.72548
## 2 54.09815 66.08400
## 3 55.67222 60.93881
```

```
# dyplr
addh %>%
  group_by(money, gender) %>%
  summarize(pay.mean = mean(paypercent)) %>%
  head(3)
```

```
## Error in eval(expr, envir, enclos): found duplicated column name: income, paypercent
```

Applying ggplot skills learned over the summer to plot results

Code for 1) transforming into a data.frame from the matrix the function returns and 2) cleaning up to plot is in .rmd and we'll review similar examples on plotting day



sapply: structure

- **Takes in**: list, data.frame, or vector (we'll call these object below)

sapply(object to apply function over, function, argument 2, argument3...)

sapply: structure

- **Takes in**: list, data.frame, or vector (we'll call these object below)
- **How to set up, generally**:

sapply(object to apply function over, function, argument 2, argument3...)

sapply: structure

- **Takes in**: list, data.frame, or vector (we'll call these object below)
- **How to set up, generally**:

sapply(object to apply function over, function, argument 2, argument3...)

- **Returns**: either vector, matrix, or list– whichever is the *simplest* format for the output (hence the *s* in sapply) (so vector if all the elements are scalar; matrix if all the elements are of the same length)

sapply: example of "translating" structure of for loop from day 1 into function + sapply

Example: using the AddHealth data, remove each observation one by one and regress the person's rating of money's importance on their age, storing the coefficient on age from the regression

1. First, we'll do using a for loop (exact same code as the example yesterday but with different data)

sapply: example of "translating" structure of for loop from day 1 into function + sapply

Example: using the AddHealth data, remove each observation one by one and regress the person's rating of money's importance on their age, storing the coefficient on age from the regression

1. First, we'll do using a for loop (exact same code as the example yesterday but with different data)
2. Then, we'll produce same results using a function + sapply, and show why this not only runs faster but is also much more flexible/useful

How to set up using for loop

```
# initialize empty vector, though generally preallocate memory if possible
agecoef <- c()
# alternative vector creation
# agecoef <- rep(NA, nrow(addh))
# iterate through each observation in the data,
for(i in 1:nrow(addh)){
  # removing one observation at a time
  dataminus1 <- addh[-1, ]
  # feed this modified data into a regression func
  regminus1 <- lm(money ~ age,
                  data = dataminus1)
  # save the regression output coefficient for age
  agecoefsingle <- coef(regminus1)["age"]
  # append the output coef into our empty vector at the end
  agecoef <- c(agecoef, agecoefsingle)
  # alternative way to save loop output
  # agecoef[i] <- agecoefsingle
}
```

Translating the for loop into a function

Basic idea: can take the "meat" inside the for loop sandwich and embed inside a function, changing indexing where appropriate and specifying "return" (if you don't specify return, the function defaults to returning the last evaluated expression, but it's usually safer to do this explicitly)

Step one: generalizing the code inside the for loop

Can take code inside the "meat" of the for loop and highlight things we should replace with more general arguments

- **Old**:

```
dataminus1 <- addh[-i, ]
regminus1 <- lm(money ~ age, data = dataminus1)
agecoefsingle <- coef(regminus1)["age"]
agecoef <- (agecoef, agecoefsingle)
return(agecoef)
```

Step one: generalizing the code inside the for loop

Can take code inside the "meat" of the for loop and highlight things we should replace with more general arguments

- **Old**:
  dataminus1 <- addh[-i,]
  regminus1 <- lm(money ~ age, data = dataminus1)
  agecoefsingle <- coef(regminus1)["age"]
  agecoef <- (agecoef, agecoefsingle)
  return(agecoef)
- Four things to generalize: 1) name of data.frame, 2) formula for regression, 3) which coefficient to extract, 4) vector in which to store coefficients

Step one: generalizing the code inside the for loop

Can take code inside the "meat" of the for loop and highlight things we should replace with more general arguments

- **Old**:
  dataminus1 <- addh[-i,]
  regminus1 <- lm(money ~ age, data = dataminus1)
  agecoefsingle <- coef(regminus1)["age"]
  agecoef <- (agecoef, agecoefsingle)
  return(agecoef)
- Four things to generalize: 1) name of data.frame, 2) formula for regression, 3) which coefficient to extract, 4) vector in which to store coefficients
- **New**:
  dataminus1 <- data[-i,]
  regminus1 <- lm(formula, data = dataminus1)
  coefsingle <- coef(regminus1)[coeftoextract]
  vectofill <- (vectofill, coefsingle)
  return(vectofill)

Translating the for loop into a function: steps one and two

Step two: check the indexing inside the function and make sure the index you want to iterate over is specified as an argument (in this case, we're specifying it as *i*)

```
# make a function called leaveoneout.moregeneral that takes i,
# data, formula, coeftoextract, and vectofill parameters
leaveoneout.moregeneral <- function(i, data, formula,
                                    coeftoextract,
                                    vectofill){
  # remove data points 1 at a time, indexed by i
  dataminus1 <- data[-i, ]
  # run regression with this data
  regminus1 <- lm(formula = formula,
                  data = dataminus1)
  coefsingle <- coef(regminus1)[coeftoextract]
  vectofill <- c(vectofill, coefsingle)
  return(vectofill)
}
```

Translating the for loop into a function: applying the function we created using sapply

- Create a vector with 1:number of iterations to apply the function to

```
# define i vector to iterate over
i <- 1:nrow(addh)
# use sapply to apply the function to i, and also feed it the other inputs
agecoefs.func.output <- sapply(i,
        leaveoneout.moregeneral,
       # define data as the data we want to feed it
        data = addh,
       # define what formula we want to feed into lm
        formula = money ~ age, coeftoextract = "age",
       # and store result in a vector;
        vectofill = c())
head(agecoefs.func.output)
```

```
##         age         age         age         age         age         age
## 0.07753542 0.07745794 0.07704594 0.07735052 0.07763852 0.07629035
```

Translating the for loop into a function: applying the function we created using sapply

- Create a vector with 1:number of iterations to apply the function to
- Use sapply with that vector using the structure: sapply(vector with length = iterations, function, other argument 1, other argument2...)

```
# define i vector to iterate over
i <- 1:nrow(addh)
# use sapply to apply the function to i, and also feed it the other inputs
agecoefs.func.output <- sapply(i,
        leaveoneout.moregeneral,
       # define data as the data we want to feed it
        data = addh,
       # define what formula we want to feed into lm
        formula = money ~ age, coeftoextract = "age",
       # and store result in a vector;
        vectofill = c())
head(agecoefs.func.output)
```

```
##         age         age         age         age         age         age
## 0.07753542 0.07745794 0.07704594 0.07735052 0.07763852 0.07629035
```

Advantages of this more general function

Can apply to different variables in the same data.frame by changing the formula argument, but may be a bit tedious.

```
#look at gender controlling for age and debt status
coefsgender <- sapply(i,
        leaveoneout.moregeneral,
        data = addh,
       formula = money ~ gender + age + debt,
       coeftoextract = "gendermale",
       vectofill = c())
head(coefsgender)
```

```
##  gendermale  gendermale  gendermale  gendermale  gendermale  gendermale
## -0.07212000 -0.06822225 -0.07238763 -0.06982896 -0.07219284 -0.07350948
```

```
#look at debt status controlling for income
coefsdebt <- sapply(i,
        leaveoneout.moregeneral,
        data = addh,
       formula = money ~ debt + income,
       coeftoextract = "debtyesdebt",
       vectofill = c())
head(coefsdebt)
```

```
## debtyesdebt debtyesdebt debtyesdebt debtyesdebt debtyesdebt debtyesdebt
## -0.07356308 -0.07778541 -0.07457649 -0.07245704 -0.07357929 -0.07132838
```

Advantages of this more general function

This is tedious.

```
# to do with the for loop, we would have had to
# rewrite the entire loop each time we changed
# what variables we wanted to look at
for(i in 1:nrow(addh)){
  dataminus1 <- addh[-i, ]
  # involves changing formula and coef to extract inside coefficient
  regminus1 <- lm(money ~ gender,
                  data = dataminus1)
  agecoefsingle <- coef(regminus1)["gendermale"]
  agecoef <- c(agecoef, agecoefsingle)
}
# repeat again
for(i in 1:nrow(addh)){
  dataminus1 <- addh[-i, ]
  regminus1 <- lm(money ~ gender,
                  data = dataminus1)
  agecoefsingle <- coef(regminus1)["debtyesdebt"]
  agecoef <- c(agecoef, agecoefsingle)
}
```

Advantages of this more general function

Can apply to different data, but make sure to change the index to iterate through as there will now be a different number of observations, in .rmd, change working directory to where the ANES data from the activity is located

```
i <- 1:nrow(anesdf)
freetradecoefs <- sapply(i, leaveoneout.moregeneral,
        data = anesdf,
       formula = fttrump ~ freetrade,
       coeftoextract = "freetrade",
       vectofill = c())
head(freetradecoefs)
```

```
## freetrade freetrade freetrade freetrade freetrade freetrade
##  1.226334  1.225436  1.223042  1.239754  1.225935  1.246853
```

Can now better see advantages of function + apply over loop other than processing time

- If we had kept the leave one out process in a for loop:

Can now better see advantages of function + apply over loop other than processing time

- If we had kept the leave one out process in a for loop:
  - When we wanted to change the model specification, we would have had to copy/paste the for loop and change the formula inside *lm*

Can now better see advantages of function + apply over loop other than processing time

- If we had kept the leave one out process in a for loop:
  - When we wanted to change the model specification, we would have had to copy/paste the for loop and change the formula inside *lm*
  - If we wanted to change the data we applied the function to, we would have had to copy/paste and rewrite it using different data

Can now better see advantages of function + apply over loop other than processing time

- If we had kept the leave one out process in a for loop:
  - When we wanted to change the model specification, we would have had to copy/paste the for loop and change the formula inside *lm*
  - If we wanted to change the data we applied the function to, we would have had to copy/paste and rewrite it using different data
- With the function, we can keep the function stored and apply it over a range of different situations- could also modify to return varying numbers of coefficients, to run different models than lm, etc.

Other apply and tidyverse functions that we did not review in depth

For more information, see the DataCamp modules on the apply family

- lapply: returns a list rather than simplifying to a matrix or vector as in sapply

Other apply and tidyverse functions that we did not review in depth

For more information, see the DataCamp modules on the apply family

- lapply: returns a list rather than simplifying to a matrix or vector as in sapply
- mapply: multivariate version of sapply, use when you have several data structures and want to apply some function to 1st element of each data structure listed, 2nd element of each data structure listed, etc. . .

Other apply and tidyverse functions that we did not review in depth

For more information, see the DataCamp modules on the apply family

- lapply: returns a list rather than simplifying to a matrix or vector as in sapply
- mapply: multivariate version of sapply, use when you have several data structures and want to apply some function to 1st element of each data structure listed, 2nd element of each data structure listed, etc. . .
- Also check out Dplyr modules (including cleaning data and joining data) in Datacamp

Other apply and tidyverse functions that we did not review in depth

For more information, see the DataCamp modules on the apply family

- lapply: returns a list rather than simplifying to a matrix or vector as in sapply
- mapply: multivariate version of sapply, use when you have several data structures and want to apply some function to 1st element of each data structure listed, 2nd element of each data structure listed, etc. . .
- Also check out Dplyr modules (including cleaning data and joining data) in Datacamp
- Purrr is still relatively less known and no modules on Datacamp, which is why we taught you Apply. Apply is covered in Intermediate R.

Brief example with mapply

```
# example of mapply to use t.tests to compare responses
# about what's important for relationship for two different factor variables
grouping <- rep(c("debt", "gender"), each = 3)
outcome <- c("love", "nocheating", "money")

custom.t <- function(x, y){
  formula <- as.formula(paste(y, x, sep = "~"))
  lm(formula, data = addh)
}


mapply(custom.t, x = grouping, y = outcome)
```

Brief example with mapply: output (run in .rmd to see more clearly)

```
##
## Call:
## lm(formula = formula, data = addh)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -8.7285  0.2715  0.2715  0.3533  0.3533
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  9.64669    0.02640 365.366   <2e-16 ***
## debtyesdebt  0.08183    0.04021   2.035   0.0419 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.1 on 3048 degrees of freedom
## Multiple R-squared:  0.001357,   Adjusted R-squared:  0.001029
## F-statistic: 4.142 on 1 and 3048 DF,  p-value: 0.04193
```

```
##                debt          debt          debt          gender
## call           Expression    Expression    Expression    Expression
## terms          Expression    Expression    Expression    Expression
## residuals      Numeric,3050  Numeric,3050  Numeric,3050  Numeric,3050
## coefficients   Numeric,8     Numeric,8     Numeric,8     Numeric,8
## aliased        Logical,2     Logical,2     Logical,2     Logical,2
## sigma          1.099764      1.047144      2.732006      1.094577
```

do.call: similar in structure to the apply family, that's useful for applying a function repeatedly but that needs to be fed a list

do.call(function to apply, list to apply over)

Example: you're putting together a time-series dataset, where the file for each ACS year is stored separately (e.g.: one file for 2008, another for 2009, etc. . .). You've already read in the data (we'll go over how to do that more efficiently on Friday) and stored them as a list and want to use rbind.data.frame to combine all the data.frames into one mega data.frame containing all years

do.call: similar in structure to the apply family, that's useful for applying a function repeatedly but that needs to be fed a list

```
##bind into a list
listacs <- list(acs1, acs2, acs3, acs4)
```

```
## Error in eval(expr, envir, enclos): object 'acs1' not found
```

```
##use do.call with the list to bind into a data.frame
acsallyears <- do.call(rbind.data.frame, listacs)
```

```
## Error in do.call(rbind.data.frame, listacs): object 'listacs' not found
```

```
head(acsallyears, 3); tail(acsallyears, 3)
```

```
## Error in head(acsallyears, 3): object 'acsallyears' not found
```

```
## Error in tail(acsallyears, 3): object 'acsallyears' not found
```

```
dim(acsallyears)
```

```
## Error in eval(expr, envir, enclos): object 'acsallyears' not found
```

Bonus, can you hand code a function for vector cross products?

```
# code a function for vector cross products
# Compute generalized cross product by taking the determinant of sub-matricies
xprod <- function(x, y) {
  args <- list(x, y)
  len <- unique(sapply(args, FUN=length))
  m <- do.call(rbind, args)

  sapply(seq(len),
         FUN=function(i) {
           det(m[,-i,drop=FALSE]) * (-1)^(i+1)
         })
}
x <- c(2,3, 5)
y <- c(4, 10, 10)
xprod(x, y)
```

```
## [1] -20   0   8
```

```
#check that it's done correctly b/c [sum of (cross product * vector) should be 0]
crossprod <- xprod(x, y)
sum(x * crossprod) # should be 0 or close to 0 due to rounding
```

```
## [1] 0
```

```
sum(y * crossprod)
```