BIOST 561: basics of R

Lecture 2

Announcements

- I'm trying to learn all your names, so please say your name when you ask questions
- Homework 1 is due on Monday (April 8)
  - "Submit" your homework just by pushing your homework onto your GitHub
  - You do not need to do anything on Canvas
- From this lecture onwards, we'll be doing "Muddiest Point"

I forgot to mention

- There is another Kevin Lin who is a faculty at UW. (He's in the CS department.)
- Make sure you email the correct person: kzlin@uw.edu.

Agenda for today

- Quick demo of setting up the GitHub for Homework 1
- Going over the basics of R
  - vectors
  - matrices
  - factors
  - data frames
  - indexing
  - lists
- Going over basics of functions
  - writing your own functions
  - loops
  - `apply()` family of functions

Takeaways for today

- Refresher on basics of data structures in R and how to work with each
- Showcasing some bizarre quirky things in R
  - Because I've been coding in R since my undergrad, I know these quirky behaviors by heart
  - **You** might not know these by heart. But trust me, you will eventually.
  - Use ChatGPT as your friend to ask why some bizarre phenomenons are happening
  - Creating small, reproducible errors that result in the error message you're getting is itself an important skill
- As with any skill, you won't be good at it in the start, but you'll get better over time

Your expectations for today's lecture

- I can't possibly cover all the corner cases in R, or all the functions you'll need to know
- Instead, I'm giving you a taste of the main aspects I think are important, and it's up to you to learn the rest on your own for your own projects
- Your main reaction after today's lecture should be:
  - "My goodness, R is a bizarre language. I really need to learn how to test my code because some of these behaviors are really quite unpredictable."

Object classes

- Some useful object classes:
  - vector: one-dimensional, all data points have the same type
  - matrix: two-dimensional, all data points have the same type
  - data frame: two-dimensional, all data points in the same column have the same type
  - list: one-dimensional, each element is the same type, but can have different elements with different types

Vectors (Just good'ol numbers)

Vectors are your bread nd butter. They are declared wit hthe `c()` function. The most common types of data points contained in vectors are booleans (logical):

```
c(TRUE, FALSE, FALSE, TRUE, TRUE)
```

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

doubles (numerics):

```
c(0.1, 0.5, 2, 4)
```

```
## [1] 0.1 0.5 2.0 4.0
```

and string (character):

```
c("alice", "bob", "charlie")
```

```
## [1] "alice"   "bob"     "charlie"
```

Recyling of vectors

- You can add vectors of numerics

```
x <- c(-7, -8, -10, -45); y <- 1:4
x+y
```

```
## [1]  -6  -6  -7 -41
```

R will *recycle* values:

```
x <- 1:4; y <- c(-10,10)
x+y
```

```
## [1] -9 12 -7 14
```

If R cannot recycle a vector perfectly, it will throw a warning

```
x <- 1:5; y <- c(-10,10)
x+y
```

```
## Warning in x + y: longer object length is not a multiple
## length
```

```
## [1] -9 12 -7 14 -5
```

Unexpected recycling is the cause of many bugs.

Naming element of a vector

Names help us understand what we're working with

```
x <- 1:5
names(x) <- c("a", "b", "c", "d", "e")
x
```

```
## a b c d e
## 1 2 3 4 5
```

```
x["a"]
```

```
## a
## 1
```

```
x[c("b", "e")]
```

```
## b e
## 2 5
```

- I name a lot of the vectors I work with
  - A lot of data manipulation happens throughout your analysis
  - You never want to keep track of which element is where

```
people_names <- c("Alice", "Bob", "Charlie")
age_vec <- c(Charlie = 32, Bob = 28, Alice = 30)
age_vec[people_names]
```

```
##   Alice     Bob Charlie
##      30      28      32
```

Functions on vectors

Many functions can take vectors as arguments:

- `mean()`, `median()`, `sd()`, `var()`, `max()`, `min()`, `length()`, and `sum()` return single numbers
- `sort()` returns a new vector
- `hist()` takes a vector of numbers and produces a histogram, a highly structured object, with the side effect of making a plot
- `ecdf()` similarly produces a cumulative-density-function object
- `summary()` gives a five-number summary of numerical vectors
- `any()` and `all()` are useful on Boolean vectors

Pop quiz (to test your understanding)

Just to make sure you're still with me

```
round(0.5)
round(1.5)
```

WARNING for numerics

Just kidding, that was a trick question

R does **VERY** weird things with rounding. This is why it's **SUPER** important to test your code rigorously, since these errors are (in some sense) unpredictable

```
round(0.5)
```

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

```
round(1.5)
```

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

```
round(2.5)
```

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

```
round(3.5)
```

```
## [1] 4
```

This is actually an intended standard. From the documentation of `round()`:

- Note that for rounding off a 5, the IEC 60559 standard (see also 'IEEE 754') is expected to be used, 'go to the even digit'. Therefore round(0.5) is 0 and round(-1.5) is -2. However, this is dependent on OS services and on representation error (since e.g. 0.15 is not represented exactly, the rounding rule applies to the represented number and not to the printed number, and so round(0.15, 1) could be either 0.1 or 0.2).

```r
sprintf("%.16f", c(6.65, 7 * 0.95))
```

```
## [1] "6.6500000000000004" "6.6499999999999995"
```

```r
sprintf("%.16f", 7 - 0.35)
```

```
## [1] "6.6500000000000004"
```

```r
1:7.999999
```

```
## [1] 1 2 3 4 5 6 7
```

```r
1:7.9999999
```

```
## [1] 1 2 3 4 5 6 7 8
```

See:

- https://stackoverflow.com/questions/35514929/rounding-in-r-returning-wrong-value
- https://stackoverflow.com/questions/39892999/is-there-an-error-in-round-function-in-r

Factors (Representing categorical data)

Factors are either loved or hated by R coders. (I personally think they're pretty useful.)

You can think of factors as *fancy* vector of strings that can only take one of a small number of values. It's mean for **categorical** data.

```
month_char <- c("May", "May", "Jun", "Dec", "Sep", "May")
month_levels <- c(
  "Jan", "Feb", "Mar", "Apr", "May", "Jun",
  "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
)
month_factor <- factor(month_char, levels = month_levels)
month_factor
```

```
## [1] May May Jun Dec Sep May
## Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

Okay... you're probably thinking I'm crazy

I defined a vector of characters `month_char` to make my factor `month_factor`. What did I gain?

Operationally: An easier way to flag typos. Nothing stops me from making typos

```
month_char <- c("May", "May", "Jun", "Deb", "Sep", "Mat")
```

It also helps me remember I'm working with a categorical variable

```
cluster_id
```

```
## [1] 1 1 2 5 2 1
```

```
mean(cluster_id) # oops! I forgot these are categories
```

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

However, R will catch me if I worked with factors

```
cluster_factor <- factor(cluster_id)
cluster_factor
```

```
## [1] 1 1 2 5 2 1
## Levels: 1 2 5
```

```
mean(cluster_factor)
# [1] NA
# Warning message:
# In mean.default(cluster_factor) :
#   argument is not numeric or logical: returning NA
```

You want design your code to have natural safeguards

- You would rather your code throw warnings and errors when something improper is done, instead of your code completing but it did something heinous

When to use a factor

- Scenarios where it's better of working with a character vector:
  - If almost every element in your vector is unique
  - Vector of Yelp review text
  - Vector of subject IDs
- Scenarios where you might want to consider a factor vector:
  - When you have a large vector of categories but a small number of possible categories
  - Which data points belong to which cluster
  - Recording which subject originated from which state (where the number of categories is the number of states)

Factor functions

- `levels()` to report the unique categories of a factor vector
- `table()` to count the number of unique instances of a vector (This function could've been used for character vectors as well).
- When we get to plotting next week, some plotting functions are expecting variables to be factors to know it's a categorical variable

WARNING for factors: Factors are really "fancy" numeric vectors

```
version_number
```

```
## [1] 3.1 3.1 5.2 2.0 3.1 4.1
## Levels: 2.0 3.1 4.1 5.2
```

What do you think will happen?

```
as.numeric(version_number)
as.character(version_number)
```

WARNING for factors: Factors are really "fancy" numeric vectors

```
version_number
```

```
## [1] 3.1 3.1 5.2 2.0 3.1 4.1
## Levels: 2.0 3.1 4.1 5.2
```

```
as.numeric(version_number)
```

```
## [1] 2 2 4 1 2 3
```

```
as.character(version_number)
```

```
## [1] "3.1" "3.1" "5.2" "2.0" "3.1" "4.1"
```

We wanted this one:

```
as.numeric(as.character(version_number))
```

```
## [1] 3.1 3.1 5.2 2.0 3.1 4.1
```

Matrices (Just a 2D vector of "numbers")

```
z_mat <- matrix(c(40,1,60,3), nrow=2)
z_mat
```

```
##      [,1] [,2]
## [1,]   40   60
## [2,]    1    3
```

```
is.matrix(z_mat)
```

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

Most of your experience with R will primarily be with matrices!

- Could also specify `ncol` for the number of columns
- To fill by rows, use `byrow=TRUE`
- Elementwise operations with the usual arithmetic and comparison operators (e.g., `z_mat/3`)

Matrix functions

- `%*%` for matrix multiplication
- `t()` for transpose
- `scale()` to normalize each column of a mtrix
- `rowSums()` and `colSums()` for row/column sums
- `rbind()` and `cbind()` to add rows/columns
- `eigen()` and `svd()` for eigen-decomposition or SVD
- `solve()` to compute the inverse of a function
- `dim()`, `nrow()`, `ncol()`, `head()`, `tail()`

Names in matrices

- We can name either rows or columns or both, with `rownames()` and `colnames()`
- These are just character vectors, and we use them just like we do `names()` for vectors

```
z_mat <- matrix(c(1:4), nrow = 2, ncol = 2)
colnames(z_mat) <- c("apple", "banana")
rownames(z_mat) <- c("day1", "day2")
z_mat
```

```
##      apple banana
## day1     1      3
## day2     2      4
```

```
z_mat["day1", "apple"]
```

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

WARNING for matrices: One string to rule them all

Once a matrix has a string it in, the whole matrix becomes strings

```
z_mat <- matrix(c(1:4), nrow = 2, ncol = 2)
z_mat[1,1] <- as.character(z_mat[1,1])
z_mat[1,1]
```

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

```
z_mat
```

```
##      [,1] [,2]
## [1,] "1"  "3"
## [2,] "2"  "4"
```

```
class(z_mat[2,2])
```

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

```
z_mat*2
# Error in z_mat * 2 : non-numeric argument to binary
# operator
```

WARNING for indexing for matrices (VERY common error)

My most common error is: when you index only one row or column of a matrix, it automatically becomes a vector

```
z_mat <- matrix(c(1:9), nrow = 3, ncol = 3)
z_mat
```

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

```
z_vec <- z_mat[,1]
z_vec
```

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

```
is.matrix(z_vec)
```

You need to use `drop = FALSE` to make sure it doesn't become a vector

```
z_mat <- matrix(c(1:9), nrow = 3, ncol = 3)
z_vec2 <- z_mat[,1,drop = FALSE]
z_vec2
```

```
##      [,1]
## [1,]    1
## [2,]    2
## [3,]    3
```

```
is.matrix(z_vec2)
```

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

```
z_vec[1,1]
# Error in z_vec[1, 1] : incorrect number of dimensions
```

```
z_vec2[1,1]
```

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

WARNING for indexing for matrices (Part 2)

You can even index matrices as if they were numbers

```
z_mat <- matrix(c(11:19), nrow = 3, ncol = 3)
z_mat
```

```
##      [,1] [,2] [,3]
## [1,]   11   14   17
## [2,]   12   15   18
## [3,]   13   16   19
```

```
z_mat[4]
```

```
## [1] 14
```

This can cause bugs in your code because you might not even realize you've indexed a matrix (instead of a vector).

Data frames (How most datasets is represented)

The format for the "classic" data table in statistics: **data frame**. Lots of the "really-statistical" parts of the R programming language presume data frames

- Think of each row as an observation/case
- Think of each columns as a variable/feature
- Not just a matrix because variables can have different types
- Both rows and columns can be assigned names

Creating a data frame from a matrix

Often times it's helpful to start with a matrix, and add columns (of different data types) to make it a data frame

```
class(state.x77) # Built-in matrix of states data
```

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

```
head(state.x77[,1:3])
```

```
##            Population Income Illiteracy
## Alabama          3615   3624        2.1
## Alaska            365   6315        1.5
## Arizona          2212   4530        1.8
## Arkansas         2110   3378        1.9
## California      21198   5114        1.1
## Colorado         2541   4884        0.7
```

```r
# Combine these into a data frame with 50 rows
state_df <- data.frame(state.x77[,1:3])
class(state_df)
```

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

```r
head(state_df)
```

```
##            Population Income Illiteracy
## Alabama          3615   3624        2.1
## Alaska            365   6315        1.5
## Arizona          2212   4530        1.8
## Arkansas         2110   3378        1.9
## California      21198   5114        1.1
## Colorado         2541   4884        0.7
```

Deleting columns from a data frame

To delete columns: we can either use negative integer indexing, or set a column to NULL

```
# First way: use negative integer indexing
state_df2 <- state_df[,-ncol(state_df)]
head(state_df2, 3)
```

```
##            Population Income
## Alabama          3615   3624
## Alaska            365   6315
## Arizona          2212   4530
```

```r
# Second way: just directly set a column to NULL
state_df2$Illiteracy <- NULL
head(state_df2, 3)
```

```
##            Population Income
## Alabama          3615   3624
## Alaska            365   6315
## Arizona          2212   4530
```

Functions on data frames

- `summary()` to print out a summary of each column in the data frame
- `subset()` to create a new data frame with only a portion of the rows
- `merge()` to merge two data frames
- `split()` to split a data frame into two
- `aggregate()` to apply a function to different portions of a data frame
- `dim()`, `nrow()`, `ncol()`, `head()`, `tail()`

(Honestly, `summary()` is the only data-frame specific function I use. But this is very useful when you want a quick sense of what's in your data frame)

Matrices vs. data frame

Time for some hot takes. What's the difference between matrices and data frames?

Technically speaking, a data frame is a list (more on this in a bit) where the list has the same length but not necessarily same type. A matrix is a two-dimensional array where all the elements are of the same type.

Fact: Matrices take up less memory on your laptop than a data frame of the same size

Personal opinion: Use matrices if and only if your data/results are entirely composed of ordinal or numerical data

- There are functions that specific work (or work better) with data frames or matrices. Your experience will let you know what is the best tool that suits your needs
- If your data is less than 50 Mb, then it honestly doesn't make too big of a difference either way
- If your data is 100 Mb or larger, you need to start seriously think about what you're using

More on indexing (The non-tidyverse way)

The most common usage of indices is when you have two vectors of the same length

```
# technically, this could be better as a data frame,
# but let's just roll with this example
animal_vec
```

```
## [1] "cat" "cat" "dog" "dog" "cat"
```

```
weight_vec
```

```
## [1]  8 11 35 72  9
```

How do we grab only the cat weights?

```r
idx_vec <- which(animal_vec == "cat")
idx_vec
```

```
## [1] 1 2 5
```

```r
weight_vec[idx_vec]
```

```
## [1]  8 11  9
```

Other useful functions

The `which()` function is a very useful function for "grabbing" the correct indicies.

If you specifically want the index of either the largest or smallest numeric value in a vector, you can use `which.max()` and `which.min()`.

```
weight_vec
```

```
## [1]  8 11 35 72  9
```

```
which.max(weight_vec)
```

```
## [1] 4
```

Boolean indexing

With matrices or data frames, we'll often want to access a subset of the rows corresponding to some condition. You already know how to do this, with Boolean indexing

```
# Compare the income average between states with
# large vs. small population
mean(state_df[state_df$Population > 10000, "Income"])
```

```
## [1] 4720.333
```

```
mean(state_df[state_df$Population < 10000, "Income"])
```

```
## [1] 4397
```

Lists (The catch-all "anything goes" data structure)

Lists are "fancier", more flexible objects. They can hold a ton of different objects (vectors, data frames, other lists) inside a *single* object

```
my_results <- list(method = "lm",
                   data_file = "Bios561_dataset.csv",
                   variable_idx = c(1,4,17),
                   coefficients = c(0.5, 0.5, 0.1))
my_results
```

```
## $method
## [1] "lm"
##
## $data_file
## [1] "Bios561_dataset.csv"
##
## $variable_idx
## [1]  1  4 17
##
## $coefficients
```

Indexing elements of a list

Double square brackets pull out elements of a list, single square brackets pull out subset of lists

```
my_results$method
```

```
## [1] "lm"
```

```
my_results[[3]]
```

```
## [1]  1  4 17
```

```
my_results[3:4]
```

```
## $variable_idx
## [1]  1  4 17
##
## $coefficients
## [1] 0.5 0.5 0.1
```

Why should you care about lists?

I use lists when I want to return lots of things from a single function (which we'll discuss in just a couple minutes!) For example, in my `tiltedCCA` package:

```
.create_cca_obj <- function(cca_obj,
                            score_1,
                            score_2){
  structure(list(cca_obj = cca_obj,
                 score_1 = score_1,
                 score_2 = score_2),
            class = "cca")
  # oooh, what is this structure thing?
  # we'll discuss this around lecture 4!
}
```

WARNING for lists: With great power comes great responsibility

You can change anything you want in a list, so be careful with what you change

```
head(state_df, 2)
```

```
##            Population Income Illiteracy
## Alabama          3615   3624        2.1
## Alaska            365   6315        1.5
```

```
lm_res <- stats::lm(Income ~ Population, data = state_df)
is.list(lm_res)
```

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

```r
names(lm_res)
```

```
##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.resi
##  [9] "xlevels"       "call"          "terms"         "model"
```

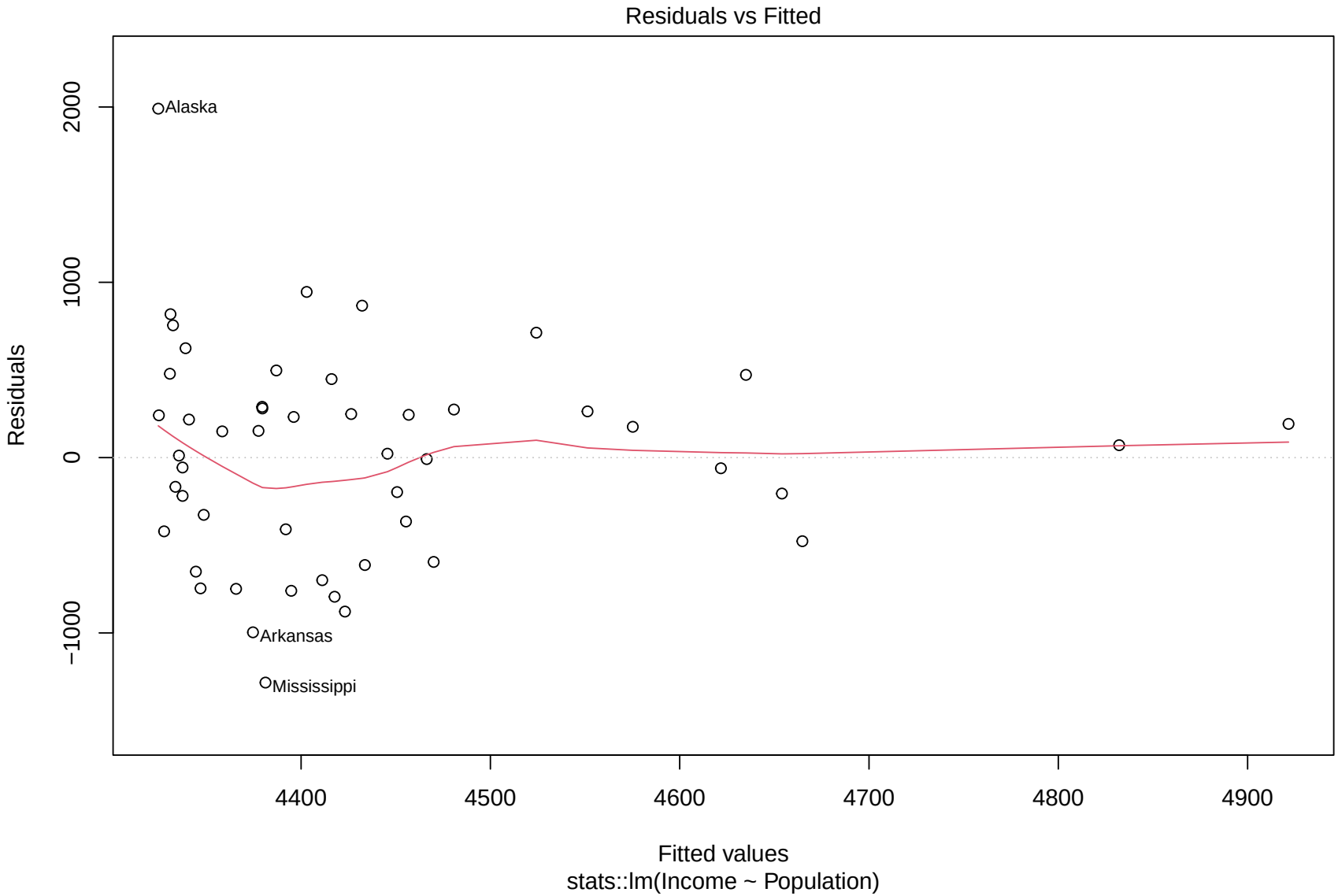
```r
lm_res # we'll talk a future week how this print function works
```

```
##
## Call:
## stats::lm(formula = Income ~ Population, data = state_df)
##
## Coefficients:
## (Intercept)   Population
##   4.314e+03    2.866e-02
```

```r
head(lm_res$residuals, 4)
```

```
##   Alabama    Alaska   Arizona  Arkansas
## -793.7039 1990.4391  152.5052 -996.5715
```

```r
plot(lm_res, which = 1)
```



```r
head(lm_res$residuals, 4)
```

```
##   Alabama    Alaska   Arizona  Arkansas
## -793.7039 1990.4391  152.5052 -996.5715
```

```r
# uh-oh! We're tampering with the results of an analysis!
lm_res$residuals <- "apple"
```

```r
plot(lm_res, which = 1)
# Error in plot.lm(lm_res, which = 1) :
# 'id.n' must be in {1,..,1}

# uh-oh! We've completely broke all the functions
# that expect our lm_res to be formatted
# a specific way!
```

Break

- Questions?
- I'll pass around paper for Muddiest Point
- You can write on it at the end of class: What was the most confusing aspect of today's lecture?

Creating your own functions

You need functions in your life! If you ever plan on repeating a statistical task, writing your procedure as a function can save you loads of time. The classic (somewhat boring) example of a function:

```
# please give your functions
# more informative names than my_func()...
my_func <- function(){
  print("Hello world!")
}
my_func()
```

```
## [1] "Hello world!"
```

Note that (as above) functions don't *necessarily* need inputs. But in most cases, you'll want lost of inputs *and* a specified output (not just a print statement).

We will spend time in Lecture 4 more about the "software-engineering" perspective on how to design a function.

- This will integrate with our discussion on how to test your functions

WARNING: Be careful for what you wish for

Please please please do **NOT** name any of your variables or custom functions after functions you're likely to use.

For example:

- `T` is (by default) defined to be `TRUE`
- `t()` is (by default) defined to be the tranpose function
- `mean()` is (by default) defined to be the function that computes the mean

```
# a pathological example
isFALSE(T)
```

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

```
T <- FALSE
isFALSE(T)
```

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

```
mat_a
```

```
##      [,1] [,2]
## [1,]    1    1
## [2,]    2    4
## [3,]    1    6
```

```
t(mat_a)
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    1
## [2,]    1    4    6
```

```
t <- sum # R has no guardrails
# you can literally do whatever you want
t(mat_a)
```

```
## [1] 15
```

```r
mean = mean(c(1,3,5))
# trust me, we've all been here before
mean
```

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

```r
# this is a great way to give yourself regret
# in a couple weeks
mean(c(2,4,6))
```

```
## [1] 4
```

```r
mean + mean(c(2,4,6))
```

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

```r
# technically still works...
```

Loops (for-loop)

The gist of it: you'll need this as you write more complicated procedures (but avoid them when you can use a vectorized function!)

Basic structure

```
n <- 10
for (i in 1:n){
  # starting with i = 1,...
  print(i)
  # execute something indexed by i
  # (here, we're just printing)
} # continue with i = 2,..
```

Sometimes it's nice to add a `print(i)` inside your loop to see how quickly your loop is progressing, but it does slow down your loop, so be aware of that

Loops (while-loop)

Most of the times, you'll be using `for()` loops, but occassionally a `while()` loop will be useful.

Just don't get infinitely stuck...

```
# DO NOT RUN, IT WILL NEVER END!
samp <- 0.5
while (samp < 2){
  samp <- runif(n = 1, min = 0, max = 1)
  # we'll never get higher than 1,
  # let alone 2
}
```

Some examples in biostats where you need to loop

- A loop is commonly used when you need to follow a specific "recipe" many many times
- Name 5 different examples where you might need to a loop

Some examples in biostats where you need to loop

- A loop is commonly used when you need to follow a specific "recipe" many many times
- Name 5 different examples where you might need to a loop
  - Analyzing each gene, one gene at a time
  - Improving an optimization problem's solution, one iteration at a time
  - Looping over a vector strings to process the string
  - Analyzing many datasets, where each dataset is "in the same format"
  - Creating many plots of the same type, where each plot is of a result that uses a slightly different method

The apply family

`apply()` applies a function to the margins of a matrix and returns a vector, matrix, or list

```
# Example: get the median of each column of a matrix
mat_a
```

```
##      [,1] [,2]
## [1,]    1    1
## [2,]    2    4
## [3,]    1    6
```

```
apply(mat_a, 2, median) # 2 is column, 1 is row
```

```
## [1] 1 4
```

You can also write your own functions to put into `apply()`

```
# a manual way to rescale columns
mat_a
```

```
##      [,1] [,2]
## [1,]    1    1
## [2,]    2    4
## [3,]    1    6
```

```
apply(mat_a, 2, function(x){
  (x-mean(x))/sd(x)
})
```

```
##             [,1]       [,2]
## [1,] -0.5773503 -1.0596259
## [2,]  1.1547005  0.1324532
## [3,] -0.5773503  0.9271726
```

You can also define the function outside of the `apply()` function

```
custom_square <- function(x){
  x^2
}
apply(mat_a, 2, custom_square)
```

```
##      [,1] [,2]
## [1,]    1    1
## [2,]    4   16
## [3,]    1   36
```

You can also pass additional parameters into `apply()`

```
custom_power <- function(x, p){
  x^p
}
power <- 3
apply(mat_a, 2, custom_power, p = power)
```

```
##      [,1] [,2]
## [1,]    1    1
## [2,]    8   64
## [3,]    1  216
```

The `apply` family

There are other types of `apply()` functions:

- `sapply()`
- `lapply()`
- `mapply()`
- `vapply()`

I personally use `sapply()` and `lapply()` a lot. Both iterate over a vector of indices, and outputs either a vector/matrix or list respectively.

Simple example with `sapply()`

For example, you'll likely get data like this at some point in your life

```
income_vec
```

```
## [1] "$60,000"  "$120,000" "$45,152"
```

```
is.numeric(income_vec)
```

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

```
# we can't take the average of this!
```

```r
vec <- sapply(income_vec, function(val){
  as.numeric(gsub("[^0-9.-]", "", val))
})
names(vec) <- NULL
vec
```

```
## [1]  60000 120000  45152
```

```r
mean(vec)
```

```
## [1] 75050.67
```

With the remaining time. . .

- Answer any questions about Homework 1
- Ask me about a function in base R that you like that I didn't talk about
- Ask me how I remember all these tricks in R
- Live coding to demonstrate some of the aspects we talked about today?
- Next week: Basics of tidyverse (specifically, tibbles and ggplot2)

Acknowledgments

- Amy Willis
- Ryan Tibshirani: https://www.stat.cmu.edu/~ryantibs/statcomp-F18/