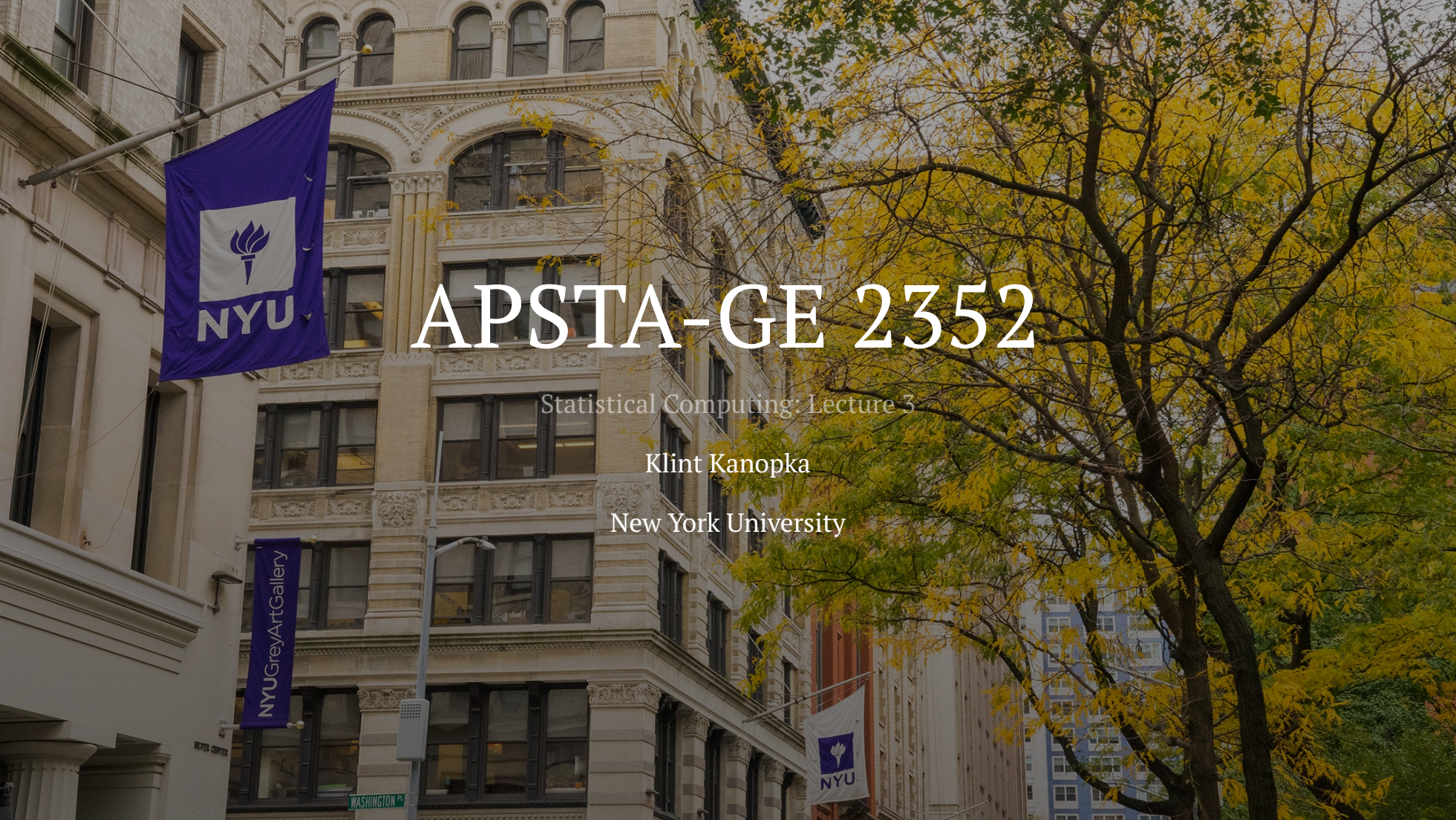

APSTA-GE 2352

Statistical Computing: Lecture 3

Klint Kanopka

New York University

Table of Contents

Announcements

- PS0 Grades have been released
  - I am really pleased with the grade distribution!
  - I do know that there were some growing pains and adjustments to the course (and my teaching style). If you have questions about your grades or your ability to stay on track in the course, please reach out.
- PS1 is now late
- PS2 is released. It’s due two weeks from today before class.

Check-In

- PollEv.com/klintkanopka

Motivating Problem

Sorting

- Given a collection of objects and some way to compare them, how do you rearrange the objects in ascending order?
- Examples of things that may need sorting:
  - Lists of numbers
  - Rows in a dataframe
  - Words
- A key note is that we didn't specify *how* these things are getting compared, just getting compared in some way! Different objects may require different types of comparisons
- We need a few more tools to do sorting from scratch, however!

Tools

Indexing

Indexing

- When we deal with vectors, arrays, matrices, and dataframes in `R`, we often only care about some subset of the data - maybe a single row, or a single element, or a few columns
  - We access them using *indexing*
  - `R`, as a language, is 1-indexed, meaning the first position is labeled `1`
  - Contrast this with Python, C, and other languages which are 0-indexed
- We index into one dimensional objects (vectors and lists) using `[ ]`
  - You can supply single integers to get a single elements, or a vector to get multiple elements
    - `A[3]` gets the third element of `A`
    - `A[1:3]` gets the first three elements of `A`
    - `A[c(2, 4, 6)]` gets the second, fourth, and sixth elements of `A`
    - `A[-2]` gets everything in `A` *except* the second element
    - `A[-1:-3]` gets everything from `A` *except* the first three elements
  - Always returns objects of the same type - so single bracket indexing into a list returns another list

Indexing in Two Dimensions

- For matrices, we have slightly different behavior
  - `M <- matrix(1:9, ncol=3)`
  - `M[1]` returns just the first element
  - `M[3]` returns just the third element (which element is the third?)
- For better results, use two dimensional indexing!
  - `M[a,b]` returns the element in the `a` th row and the `b` th column (m_{ab})
  - `M[a,]` returns the entire `a` th row
  - `M[,b]` returns the entire `b` th column
  - The rules we use in one dimension also work here
    - `M[-1,]` will drop the first row of `M`
    - `M[1:2, 1:2]` will produce the 2×2 matrix made up of the first two rows and columns.
- You can always pass variables into indexing arguments, too.

Indexing into Named Lists

- If the elements of a list are named, you can index into them using the names!
  - `L['first']` provides a list with only the element of `L` named `first`
  - `L[c('first', 'second')]` provides a list with the element of `L` named `first` and `second`
  - `L[['first']]` provides the contents of element named `first` in `L`
  - `L[[c('first', 'second')]]` throws an error
  - `L$first` gets the element of `L` named `first`
  - Note that if the name of the element you want is stored in a variable `var`, you can't fish it out using `$`, but instead have to use `L[[var]]` or `L[var]`
- Remember that dataframes *are* lists, so most of this behavior holds, but they also look kind of like matrices, so some of that behavior holds

Indexing into Dataframes

- For a dataframe named `d` with columns named `x`, `y`, and `z`:
  - `d['x']` provides a dataframe with only column `x`
  - `d[c('x', 'y')]` provides a dataframe with only columns `x` and `y`
  - `d[['x']]` provides column `x` as a vector
  - `d[[c('x', 'y')]]` throws an error
  - `d$x` provides column `x` as a vector
  - Again: if the name of the element you want is stored in a variable `var`, you can't fish it out using `$` - `d$var` won't work
    - `d[[var]]` gives a vector
    - `d[var]` gives a dataframe
- Numbers also work
  - What do you think `d[2,]`, `d[,2]`, `d[2,2]`, and `d[2]` give you?
    - `d[2,]` gives you the second row (as a vector)

Conditional Statements

Conditional Statements

- Often we want to check if something is true, and then change the behavior of the code we write based upon that
- We call these *conditional statements*, and they're the key way we handle choice in control flow
- Three main conditional statements in `R`
  - `if` checks a condition and then executes some code if it evaluates to `TRUE`
  - `else` provides some code to execute if the previous `if` statement evaluates to `FALSE`
  - `else if` provides a secondary condition. Note that if you use `else if`, a final `else` statement will only execute if *all* of the previous `if` conditions evaluate to `FALSE`
- There are two ways to write these, single line and multiline
  - I prefer multiline in basically all situations (for readability)

Example Conditional 1

```
x <- 3

if (x > 0){
  print('positive')
}

# [1] "positive"

x <- -3

if (x > 0) print('positive')

#
```

Example Conditional 2

```
x <- 3

if (x > 0){
  print('positive')
} else {
  print('not positive')
}

# [1] "positive"

x <- -3

if (x > 0) print('positive') else print('not positive')

# [1] "not positive"
```

Example Conditional 3

```
x <- 'beans'

if (x > 0){
  print('positive')
} else if (x < 0){
  print('negative')
} else {
  print('neither positive nor negative')
}

# [1] "positive"
```

Conditional Tricks

- Conditional statements are not vectorized!
  - They check one value
  - Remember `&&` and `||` from last week?
  - What if you need to do vectorized if/else-ing?
- `R` has a vectorized conditional function, `ifelse()`
  - It takes three arguments:
    1. The condition
    2. The output if the condition is `TRUE`
    3. The output if the condition is `FALSE`
  - You can supply vectors to the condition, which is checked elementwise, and then replaced with the appropriate outputs
- `dplyr` has two related functions that are quite nice:
  - `if_else()` is like `ifelse()`, with better handling of missing data

`ifelse()` Example

```r
set.seed(8675309)
A <- rnorm(4)
A

# [1] -0.9965824  0.7218241 -0.6172088  2.0293916

ifelse(A>0, 'positive', 'not positive')

# [1] "not positive" "positive"     "not positive" "positive"
```

`for` Loops

`for` Loops

- Loops allow us to specify a chunk of code to be executed repeatedly
- The first type we'll look at is the `for` loop
  - This executes a pre-specified number of times
  - Contrast this with the `while` loop, which continues to execute *while* a condition is true
    - `while` loops can fall into conditions where they never terminate!
    - This means `for` loops are often considered "safer," but both are super useful

`for` Loops

- In `R`, a `for` loop has a few main components
  - The call: `for`
  - The conditions, specified in `( )` after the call and contain:
    - A variable name for the index: Common choices are i, j, k
    - The word `in`
    - A range: A vector of values you want the index to take on
  - The code, wrapped in `{ }`:
    - For each value in the range, the index variable is set to that value and all of the code within the loop is executed one time (called an *iteration*)
    - After this is done, the value of the index is updated to the next value in the range and the code is executed again
    - This continues for each value in the range
    - Reference the index and use its changing value to have each iteration do something slightly different

Example `for` Loop 1

What will be the result of executing this loop?

```
for (i in 1:5){
  print(i^2)
}

# [1] 1
# [1] 4
# [1] 9
# [1] 16
# [1] 25
```

Example `for` Loop 2

What will be the result of executing this loop?

```
for (i in 5:1){
  tmp <- i^2
  if (i%%2 != 0){
    print(tmp)
  }
}

# [1] 25
# [1] 9
 # [1] 1
```

Note: `%%` is the *modulo* operator. `A%%B` outputs the remainder of `A/B`

Example `for` Loop 3

What will be stored in `output` at the end of this loop?

```
input <- 1:10
output <- vector(mode='numeric', length=length(input))

for (i in 1:length(input)){
  output[i] <- sum(input[1:i])
}

output

 # [1]  1  3  6 10 15 21 28 36 45 55
```

Loop Tricks

- Like `if`, you can skip the `{ }` if there's only a single line to execute:

```
for (i in 1:10){
  print(i^2)
}

for (i in 1:10) print(i^2)
```

- The range can also be a named vector:

```
A <- 1:10
for (i in A) print(i^2)
```

- The elements of the range don't have to be sequential (or even integers):

```
for (i in rnorm(10)) print(i^2)
```

Loop Tricks

- You can use `seq_along()` to generate an integer sequence the same length as some other object you care about using
  - The output of `seq_along(A)` is the same as `1:length(A)`

```r
A <- rnorm(5)
A

# [1]  1.14133758 -1.79670720 -0.01528379 -0.70880602  0.71905939

for (i in seq_along(A)){
  print(paste0(i, " : ", A[i]))
}

# [1] "1 : 1.14133758088966"
# [1] "2 : -1.79670719779003"
# [1] "3 : -0.0152837872326052"
# [1] "4 : -0.708806019727051"
# [1] "5 : 0.71905938753312"
```

Sorting

Our First Sorting Algorithm

- First, what even is an *algorithm?*
  - For us: *a precise set of steps to solve some problem*
- So, okay, let's come up with a sorting algorithm!
  - In a group of 3 ± 1, describe (in words) an algorithm that takes, as input, some vector of numbers `A` and then returns a new vector with the elements of `A` sorted in ascending order
  - Ascending is smallest to largest
  - Do this now

Selection Sort

Selection Sort

- This is kind of the most obvious to implement (to me, at least)
- Here's the plan:
  1. Find the smallest element in the vector
  2. Move it to the front
  3. Repeat 1-2 among the elements that are left until you're done
- Where do we need the tools we've developed so far in this class?
  - Last week: Logical statements, functions, and unit testing
  - This week: Indexing, conditionals, loops

Selection Sort

```r
SelectionSort <- function(A){
  # 3. repeat 1-2 among the elements that are left
  for (j in 1:length(A)){
    # 1. find the smallest element
    min <- j
    for (i in j:length(A)){
      if (A[i] < A[min]){
        min <- i
      }
    }
    # 2. move it to the front
    A[c(j, min)] <- A[c(min, j)]
  }
  # 4. return the sorted list
  return(A)
}
```

Selection Sort

- We care about two things with any algorithm:
  1. Is it correct?
  2. Is it efficient?
- Correctness is usually validated through proofs
- Efficiency is validated through runtime analysis
- Is `SelectionSort()` correct?
  - Does it always return a sorted version of the input?
  - Yes, but we won't prove it in this course
  - Instead, test your function until you're convinced it works

Selection Sort

- If `A` contains n elements, how many instructions (in terms of n) are run when you execute `SelectionSort(A)`?
  - The code in the outer loop executes n times
  - The code in the inner loop runs n, then $n-1$, then $n-2$, ..., 1 times
  - We then say that this runs *on the order of* n^2 instructions
  - Often this is written in "big-O notation" as $O(n^2)$
  - There's some technical stuff here that we'll get to later
- Each line of code takes time to run, and as we use larger datasets, efficiency starts to matter a lot!

Wrap Up

Recap

- Sorting presents a common problem in computer science
  - There are efficient ways to sort and there are horrible ways to sort
  - `bogosort` is an algorithm that generates random permutations of its input and then checks if it is sorted. This works, but is horrible!
  - You'll implement another sorting algorithm in PS2
- We now have the tools to do most programming tasks
  - Maybe not in an optimally efficient way, however!
  - Indexing lets us reach into objects and access the individual components in a consistent way
  - Conditionals allow us to use logical statements to make consistent decisions in code that we don't have to directly supervise
  - `for` loops allow us to repeat steps in a consistent way and do things slightly differently in each iteration
- Combining these tools allows you to build complex programs that can carry out complex tasks!
- Most of this course is built around expanding our toolkit to solve statistical problems
- You should start PS2 now. The last question, especially, is quite time consuming!

Final Thoughts

- PollEv.com/klintkanopka