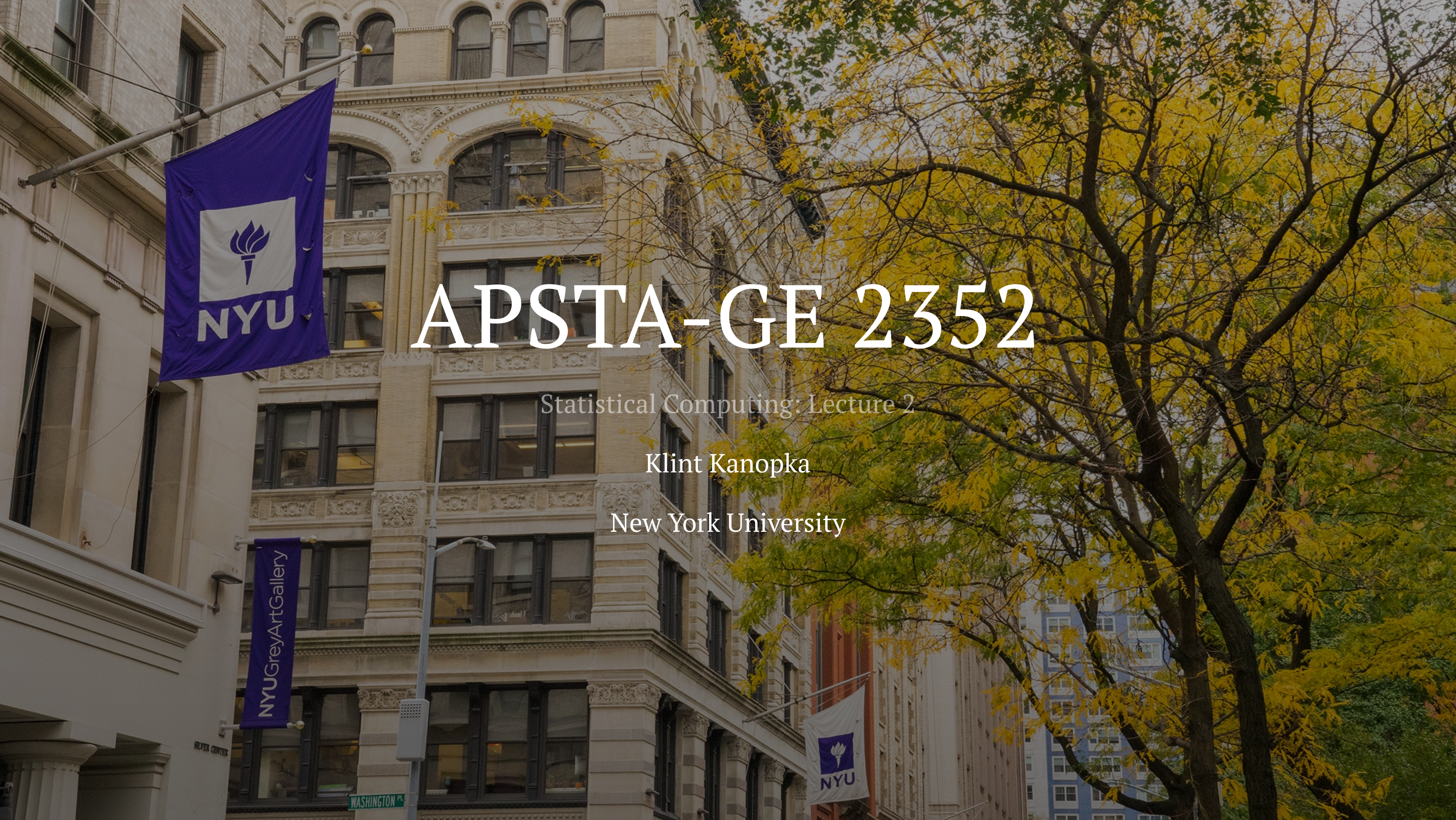

APSTA-GE 2352

Statistical Computing: Lecture 2

Klint Kanopka

New York University

Table of Contents

1. Statistical Computing - Week 2
2. Table of Contents
3. Front Matter
4. Vector and Matrix Arithmetic in `R`
5. Logicals
6. Functional and Object-Oriented Programming
7. Wrapping Up

Front Matter

Announcements

- Office hours update
  - Tons of people came, felt very productive
  - Some people just used that as time to work on their problem sets without specific questions, which is a good idea!
- PS0 is due 9/12 @ 11.59p
  - I'm generally pleased with the questions that I received and the work I saw people do!
  - This is mostly what my assignments look like
    1. Do some stuff
    2. Plot the stuff
    3. Tell me what you think about the stuff
- PS1 is out!
  - It is still due on 9/18 before class
  - I think it's more straightforward than PS0
  - It's all about writing functions to do things

Check-In

- PollEv.com/klintkanopka

Vector and Matrix Arithmetic in `R`

Adding Vectors

What do you think the results of each of these operations ought to be?

```
c(1, 2, 3) + c(4, 5, 6)
c(1, 2, 3) + c(4, 5, 6, 7)
c(1, 2, 3) + c(4, 5, 6, 7, 8, 9)
```

Adding Vectors

What do you think the results of each of these operations ought to be?

```
c(1, 2, 3) + c(4, 5, 6)
# [1] 5 7 9
c(1, 2, 3) + c(4, 5, 6, 7)
c(1, 2, 3) + c(4, 5, 6, 7, 8, 9)
```

Adding Vectors

What do you think the results of each of these operations ought to be?

```
c(1, 2, 3) + c(4, 5, 6)
# [1] 5 7 9
c(1, 2, 3) + c(4, 5, 6, 7)
# Warning message:
# In c(1, 2, 3) + c(4, 5, 6, 7) :
#   longer object length is not a multiple of shorter object length
# [1] 5 7 9 8
c(1, 2, 3) + c(4, 5, 6, 7, 8, 9)
```

Adding Vectors

What do you think the results of each of these operations ought to be?

```
c(1, 2, 3) + c(4, 5, 6)
# [1] 5 7 9
c(1, 2, 3) + c(4, 5, 6, 7)
# Warning message:
# In c(1, 2, 3) + c(4, 5, 6, 7) :
#   longer object length is not a multiple of shorter object length
# [1] 5 7 9 8
c(1, 2, 3) + c(4, 5, 6, 7, 8, 9)
# [1] 5 7 9 8 10 12
```

Vector Arithmetic

- Generally happens *elementwise*
- The first elements from each input are combined
- Then the second elements
- And so on…
- When vectors are the same size, this produces a vector the same length as the inputs
- What if they’re not the same length?

Recycling

- `R`’s general behavior when things aren’t the same length is to *recycle* the shorter object
- Behavior is the same regardless of order
- Length of the output is the *maximum* of the lengths of the inputs
- How does this work?
    - `R` will paste the shorter object to itself end-to-end until it matches the length of the longer object
    - If the longer object is an integer multiple of the length of the longer object, it does this silently
    - If the longer object is not, it throws a warning, **but still produces output according to the same rules!**

Vectors and Matrices

```r
v1 <- c(1,2,3)
v2 <- c(1,2,3,4)
v3 <- c(1,2)
mat <- matrix(1:9, nrow=3)

mat

v1 * mat
v2 * mat
v3 * mat
```

Vectors and Matrices

```r
v1 <- c(1,2,3)
v2 <- c(1,2,3,4)
v3 <- c(1,2)
mat <- matrix(1:9, nrow=3)

mat

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

v1 * mat
v2 * mat
v3 * mat
```

Vectors and Matrices

```
v1 <- c(1,2,3)
v2 <- c(1,2,3,4)
v3 <- c(1,2)
mat <- matrix(1:9, nrow=3)

mat

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

v1 * mat

#      [,1] [,2] [,3]
# [1,]    1    4    7
# [2,]    4   10   16
# [3,]    9   18   27

v2 * mat
v3 * mat
```

Vectors and Matrices

```
v1 <- c(1,2,3)
v2 <- c(1,2,3,4)
v3 <- c(1,2)
mat <- matrix(1:9, nrow=3)

mat

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

v1 * mat
v2 * mat

# Warning message:
# In v2 * mat :
#   longer object length is not a multiple of shorter object length
#      [,1] [,2] [,3]
# [1,]    1   16   21
# [2,]    4    5   32
# [3,]    9   12    9

v3 * mat
```

Vectors and Matrices

```
v1 <- c(1,2,3)
v2 <- c(1,2,3,4)
v3 <- c(1,2)
mat <- matrix(1:9, nrow=3)

mat

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

v1 * mat
v2 * mat
v3 * mat

# In v3 * mat :
#   longer object length is not a multiple of shorter object length
#       [,1] [,2] [,3]
# [1,]    1    8    7
# [2,]    4    5   16
# [3,]    3   12    9
```

Vectors and Matrices

- Here, recycling happens along the columns
  - For matrices in `R`, things are usually applied along columns first
- Under the hood:
  1. The matrix is unrolled into a vector of the form `c(col1, col2, ...)`
  2. Recycling happens as if two vectors were multiplied
  3. The output is reshaped back into the original dimensions of the matrix

Matrix and Matrix

```
mat1 <- matrix(1:9, ncol=3)
mat2 <- matrix(1:4, ncol=2)

mat1 + mat1
mat2 * mat2
mat1 + mat2
```

Matrix and Matrix

```
mat1 <- matrix(1:9, ncol=3)
mat2 <- matrix(1:4, ncol=2)

mat1 + mat1

#      [,1] [,2] [,3]
# [1,]    2    8   14
# [2,]    4   10   16
# [3,]    6   12   18

mat2 * mat2
mat1 + mat2
```

Matrix and Matrix

```
mat1 <- matrix(1:9, ncol=3)
mat2 <- matrix(1:4, ncol=2)

mat1 + mat1

#      [,1] [,2] [,3]
# [1,]    2    8   14
# [2,]    4   10   16
# [3,]    6   12   18

mat2 * mat2

#      [,1] [,2]
# [1,]    1    9
# [2,]    4   16

mat1 + mat2
```

Matrix and Matrix

```
mat1 <- matrix(1:9, ncol=3)
mat2 <- matrix(1:4, ncol=2)

mat1 + mat1

#      [,1] [,2] [,3]
# [1,]    2    8   14
# [2,]    4   10   16
# [3,]    6   12   18

mat2 * mat2

#      [,1] [,2]
# [1,]    1    9
# [2,]    4   16

mat1 + mat2

# Error in `mat1 + mat2`:
# ! non-conformable arrays
```

Matrix and Matrix

- For two matrix inputs, recycling does **not** happen!
- For standard arithmetic operators, everything is done elementwise
- If two matrices are not the same shape, `R` throws an error
  - `non-conformable arguments` or `non-conformable arrays`
  - No output is produced
  - Execution is halted

Matrix Multiplication

- There is a specific matrix multiplication operator, `%*%`
- Conducts matrix multiplication
  - Requires an $A \times B$ matrix and a $B \times C$ matrix
  - Produces $A \times C$ shaped output
- Works with vectors!
  - A vector of length N is treated as either an $N \times 1$ or $1 \times N$ matrix, depending on what is needed
  - The output is **always** as a matrix

Matrix Multiplication

```r
matrix(1:9, ncol=3) %*% matrix(1:9, ncol=3)
matrix(1:9, ncol=3) %*% c(1, 2, 3)
c(1, 2, 3) %*% matrix(1:9, ncol=3)
```

Matrix Multiplication

```
matrix(1:9, ncol=3) %*% matrix(1:9, ncol=3)

#      [,1] [,2] [,3]
# [1,]   30   66  102
# [2,]   36   81  126
# [3,]   42   96  150

matrix(1:9, ncol=3) %*% c(1, 2, 3)
c(1, 2, 3) %*% matrix(1:9, ncol=3)
```

Matrix Multiplication

```
matrix(1:9, ncol=3) %*% matrix(1:9, ncol=3)

#      [,1] [,2] [,3]
# [1,]   30   66  102
# [2,]   36   81  126
# [3,]   42   96  150

matrix(1:9, ncol=3) %*% c(1, 2, 3)

#      [,1]
# [1,]   30
# [2,]   36
# [3,]   42

c(1, 2, 3) %*% matrix(1:9, ncol=3)
```

Matrix Multiplication

```
matrix(1:9, ncol=3) %*% matrix(1:9, ncol=3)

#      [,1] [,2] [,3]
# [1,]   30   66  102
# [2,]   36   81  126
# [3,]   42   96  150

matrix(1:9, ncol=3) %*% c(1, 2, 3)

#      [,1]
# [1,]   30
# [2,]   36
# [3,]   42

c(1, 2, 3) %*% matrix(1:9, ncol=3)

#      [,1] [,2] [,3]
# [1,]   14   32   50
```

Matrix Multiplication

- We want to multiply two matrices, $AB = C$
  - Here, a_{ij} is the element of matrix A in the ith row and jth column
  - And matrix A is an $N \times K$ matrix and matrix B is a $K \times M$ matrix
- To construct the resultant $N \times M$ matrix, C:

$$c_{ij} = \sum_{k=1}^{K} a_{ik} b_{kj}$$

- Alternatively C can be constructed through dot products:
  - Where $\vec{a}_i$ is the ith *row* vector of A
  - And $\vec{b}_j$ is the jth *column* vector of B

$$c_{ij} = \vec{a}_i \cdot \vec{b}_j$$

- If this looks awful, a course in linear algebra could be useful (depending on your subplan and career goals)

Logicals

Logical Statements

- Sometimes we want to compare conditions and know if they’re `TRUE` or `FALSE`
  - Called *Boolean* after the work of George Boole
  - Only two possible values (dichotomous), and clear rules for evaluation

Logical Statements

- Sometimes we want to compare conditions and know if they’re `TRUE` or `FALSE`
  - Called *Boolean* after the work of George Boole
  - Only two possible values (dichotomous), and clear rules for evaluation

The primary comparison operators we use are:

Logical Statements

- Sometimes we want to compare conditions and know if they’re `TRUE` or `FALSE`
  - Called *Boolean* after the work of George Boole
  - Only two possible values (dichotomous), and clear rules for evaluation

The primary comparison operators we use are:

- `A == B` : returns `TRUE` if the value of `A` and `B` are equal, `FALSE` otherwise

Logical Statements

- Sometimes we want to compare conditions and know if they're `TRUE` or `FALSE`
  - Called *Boolean* after the work of George Boole
  - Only two possible values (dichotomous), and clear rules for evaluation

The primary comparison operators we use are:

- `A == B` : returns `TRUE` if the value of `A` and `B` are equal, `FALSE` otherwise
- `A != B` : returns `TRUE` if the value of `A` and `B` are **not** equal, `FALSE` otherwise

Logical Statements

- Sometimes we want to compare conditions and know if they're `TRUE` or `FALSE`
  - Called *Boolean* after the work of George Boole
  - Only two possible values (dichotomous), and clear rules for evaluation

The primary comparison operators we use are:

- `A == B` : returns `TRUE` if the value of `A` and `B` are equal, `FALSE` otherwise
- `A != B` : returns `TRUE` if the value of `A` and `B` are **not** equal, `FALSE` otherwise
- `A > B` : returns `TRUE` if the value of `A` is strictly greater than `B` , `FALSE` otherwise

Logical Statements

- Sometimes we want to compare conditions and know if they’re `TRUE` or `FALSE`
  - Called *Boolean* after the work of George Boole
  - Only two possible values (dichotomous), and clear rules for evaluation

The primary comparison operators we use are:

- `A == B` : returns `TRUE` if the value of `A` and `B` are equal, `FALSE` otherwise
- `A != B` : returns `TRUE` if the value of `A` and `B` are **not** equal, `FALSE` otherwise
- `A > B` : returns `TRUE` if the value of `A` is strictly greater than `B` , `FALSE` otherwise
- `A >= B` : returns `TRUE` if the value of `A` is greater than or equal to `B` , `FALSE` otherwise

Logical Statements

- Sometimes we want to compare conditions and know if they’re `TRUE` or `FALSE`
  - Called *Boolean* after the work of George Boole
  - Only two possible values (dichotomous), and clear rules for evaluation

The primary comparison operators we use are:

- `A == B` : returns `TRUE` if the value of `A` and `B` are equal, `FALSE` otherwise
- `A != B` : returns `TRUE` if the value of `A` and `B` are **not** equal, `FALSE` otherwise
- `A > B` : returns `TRUE` if the value of `A` is strictly greater than `B` , `FALSE` otherwise
- `A >= B` : returns `TRUE` if the value of `A` is greater than or equal to `B` , `FALSE` otherwise
- `A < B` : returns `TRUE` if the value of `A` is strictly less than `B` , `FALSE` otherwise

Logical Statements

- Sometimes we want to compare conditions and know if they're `TRUE` or `FALSE`
  - Called *Boolean* after the work of George Boole
  - Only two possible values (dichotomous), and clear rules for evaluation

The primary comparison operators we use are:

- `A == B` : returns `TRUE` if the value of `A` and `B` are equal, `FALSE` otherwise
- `A != B` : returns `TRUE` if the value of `A` and `B` are **not** equal, `FALSE` otherwise
- `A > B` : returns `TRUE` if the value of `A` is strictly greater than `B` , `FALSE` otherwise
- `A >= B` : returns `TRUE` if the value of `A` is greater than or equal to `B` , `FALSE` otherwise
- `A < B` : returns `TRUE` if the value of `A` is strictly less than `B` , `FALSE` otherwise
- `A <= B` : returns `TRUE` if the value of `A` is less than or equal to `B` , `FALSE` otherwise

Logical Statements

- Often we store `TRUE` or `FALSE` status in a variable and need to check multiple conditions, or need rules on how to combine them
- Arithmetic with Boolean variables is easy - `TRUE = 1` and `FALSE = 0`
  - This can be leveraged to do some really clever stuff!

Logical Statements

- Often we store `TRUE` or `FALSE` status in a variable and need to check multiple conditions, or need rules on how to combine them
- Arithmetic with Boolean variables is easy - `TRUE = 1` and `FALSE = 0`
  - This can be leveraged to do some really clever stuff!

Sometimes we have more complex conditions to check, and we get three primary logical operators:

Logical Statements

- Often we store `TRUE` or `FALSE` status in a variable and need to check multiple conditions, or need rules on how to combine them
- Arithmetic with Boolean variables is easy - `TRUE = 1` and `FALSE = 0`
  - This can be leveraged to do some really clever stuff!

Sometimes we have more complex conditions to check, and we get three primary logical operators:

- `&` is the logical AND - `A&B` is `TRUE` if both are `TRUE`, and `FALSE` otherwise

Logical Statements

- Often we store `TRUE` or `FALSE` status in a variable and need to check multiple conditions, or need rules on how to combine them
- Arithmetic with Boolean variables is easy - `TRUE = 1` and `FALSE = 0`
  - This can be leveraged to do some really clever stuff!

Sometimes we have more complex conditions to check, and we get three primary logical operators:

- `&` is the logical AND - `A&B` is `TRUE` if both are `TRUE`, and `FALSE` otherwise
- `|` is the logical OR - `A|B` is `TRUE` if either `A` or `B` are `TRUE`, and `FALSE` if both are `FALSE`

Logical Statements

- Often we store `TRUE` or `FALSE` status in a variable and need to check multiple conditions, or need rules on how to combine them
- Arithmetic with Boolean variables is easy - `TRUE = 1` and `FALSE = 0`
  - This can be leveraged to do some really clever stuff!

Sometimes we have more complex conditions to check, and we get three primary logical operators:

- `&` is the logical AND - `A&B` is `TRUE` if both are `TRUE`, and `FALSE` otherwise
- `|` is the logical OR - `A|B` is `TRUE` if either `A` or `B` are `TRUE`, and `FALSE` if both are `FALSE`
- `!` is the logical NOT - `!A` is `TRUE` if `A` is `FALSE`, and `FALSE` if `A` is `TRUE`

Logical Statements

- Often we store `TRUE` or `FALSE` status in a variable and need to check multiple conditions, or need rules on how to combine them
- Arithmetic with Boolean variables is easy - `TRUE = 1` and `FALSE = 0`
  - This can be leveraged to do some really clever stuff!

Sometimes we have more complex conditions to check, and we get three primary logical operators:

- `&` is the logical AND - `A&B` is `TRUE` if both are `TRUE`, and `FALSE` otherwise
- `|` is the logical OR - `A|B` is `TRUE` if either `A` or `B` are `TRUE`, and `FALSE` if both are `FALSE`
- `!` is the logical NOT - `!A` is `TRUE` if `A` is `FALSE`, and `FALSE` if `A` is `TRUE`

A confusing thing is that there are two other logical operators:

Logical Statements

- Often we store `TRUE` or `FALSE` status in a variable and need to check multiple conditions, or need rules on how to combine them
- Arithmetic with Boolean variables is easy - `TRUE = 1` and `FALSE = 0`
  - This can be leveraged to do some really clever stuff!

Sometimes we have more complex conditions to check, and we get three primary logical operators:

- `&` is the logical AND - `A&B` is `TRUE` if both are `TRUE`, and `FALSE` otherwise
- `|` is the logical OR - `A|B` is `TRUE` if either `A` or `B` are `TRUE`, and `FALSE` if both are `FALSE`
- `!` is the logical NOT - `!A` is `TRUE` if `A` is `FALSE`, and `FALSE` if `A` is `TRUE`

A confusing thing is that there are two other logical operators:

- `&&` is a logical AND that ONLY works on single values (not vectors)

Logical Statements

- Often we store `TRUE` or `FALSE` status in a variable and need to check multiple conditions, or need rules on how to combine them
- Arithmetic with Boolean variables is easy - `TRUE = 1` and `FALSE = 0`
    - This can be leveraged to do some really clever stuff!

Sometimes we have more complex conditions to check, and we get three primary logical operators:

- `&` is the logical AND - `A&B` is `TRUE` if both are `TRUE`, and `FALSE` otherwise
- `|` is the logical OR - `A|B` is `TRUE` if either `A` or `B` are `TRUE`, and `FALSE` if both are `FALSE`
- `!` is the logical NOT - `!A` is `TRUE` if `A` is `FALSE`, and `FALSE` if `A` is `TRUE`

A confusing thing is that there are two other logical operators:

- `&&` is a logical AND that ONLY works on single values (not vectors)
- `||` is a logical OR that ONLY works on single values (not vectors)

Logical Statements

- Often we store `TRUE` or `FALSE` status in a variable and need to check multiple conditions, or need rules on how to combine them
- Arithmetic with Boolean variables is easy - `TRUE = 1` and `FALSE = 0`
  - This can be leveraged to do some really clever stuff!

Sometimes we have more complex conditions to check, and we get three primary logical operators:

- `&` is the logical AND - `A&B` is `TRUE` if both are `TRUE`, and `FALSE` otherwise
- `|` is the logical OR - `A|B` is `TRUE` if either `A` or `B` are `TRUE`, and `FALSE` if both are `FALSE`
- `!` is the logical NOT - `!A` is `TRUE` if `A` is `FALSE`, and `FALSE` if `A` is `TRUE`

A confusing thing is that there are two other logical operators:

- `&&` is a logical AND that ONLY works on single values (not vectors)
- `||` is a logical OR that ONLY works on single values (not vectors)
- Both `&` and `|` are vectorized, and will do elementwise operations with normal recycling rules

Logical Statements

- Often we store `TRUE` or `FALSE` status in a variable and need to check multiple conditions, or need rules on how to combine them
- Arithmetic with Boolean variables is easy - `TRUE = 1` and `FALSE = 0`
  - This can be leveraged to do some really clever stuff!

Sometimes we have more complex conditions to check, and we get three primary logical operators:

- `&` is the logical AND - `A&B` is `TRUE` if both are `TRUE`, and `FALSE` otherwise
- `|` is the logical OR - `A|B` is `TRUE` if either `A` or `B` are `TRUE`, and `FALSE` if both are `FALSE`
- `!` is the logical NOT - `!A` is `TRUE` if `A` is `FALSE`, and `FALSE` if `A` is `TRUE`

A confusing thing is that there are two other logical operators:

- `&&` is a logical AND that ONLY works on single values (not vectors)
- `||` is a logical OR that ONLY works on single values (not vectors)
- Both `&` and `|` are vectorized, and will do elementwise operations with normal recycling rules
- Use `&&` and `||` for control flow!

Functional and Object-Oriented Programming

Functions

- Functions are objects in R that package code
- Functions take named *arguments*
- Executing a function creates a new environment with the arguments assigned to their names
  - Then they execute their code
  - When a function is done running, its environment is destroyed/lost
- In general, we do not write functions that modify the global variables (this is super dangerous)!
- If you need information that’s computed within a function, you need to return it
- This lets you maintain whatever object is returned for future use outside of the function’s environment

Function Anatomy

Let’s write a function called `RollDice()` that rolls an arbitrarily sized die an arbitrary number of times and returns the individual results.

Function Anatomy

Let’s write a function called `RollDice()` that rolls an arbitrarily sized die an arbitrary number of times and returns the individual results.

```r
RollDice <- function(){}
```

Function Anatomy

Let’s write a function called `RollDice()` that rolls an arbitrarily sized die an arbitrary number of times and returns the individual results.

```r
RollDice <- function(){
  results <- sample()
}
```

Function Anatomy

Let’s write a function called `RollDice()` that rolls an arbitrarily sized die an arbitrary number of times and returns the individual results.

```r
RollDice <- function(){
  results <- sample(x, size, replace = FALSE, prob = NULL)
}
```

Function Anatomy

Let’s write a function called `RollDice()` that rolls an arbitrarily sized die an arbitrary number of times and returns the individual results.

```r
RollDice <- function(){
  result <- sample(x = 1:N_sides, size, replace = FALSE, prob = NULL)
}
```

Function Anatomy

Let’s write a function called `RollDice()` that rolls an arbitrarily sized die an arbitrary number of times and returns the individual results.

```r
RollDice <- function(N_sides){
  result <- sample(x = 1:N_sides, size, replace = FALSE, prob = NULL)
}
```

Function Anatomy

Let’s write a function called `RollDice()` that rolls an arbitrarily sized die an arbitrary number of times and returns the individual results.

```r
RollDice <- function(N_sides){
  result <- sample(x = 1:N_sides, size = N_dice, replace = FALSE, prob = NULL)
}
```

Function Anatomy

Let’s write a function called `RollDice()` that rolls an arbitrarily sized die an arbitrary number of times and returns the individual results.

```r
RollDice <- function(N_sides, N_dice){
  result <- sample(x = 1:N_sides, size = N_dice, replace = FALSE, prob = NULL)
}
```

Function Anatomy

Let’s write a function called `RollDice()` that rolls an arbitrarily sized die an arbitrary number of times and returns the individual results.

```r
RollDice <- function(N_sides, N_dice){
  result <- sample(x = 1:N_sides, size = N_dice, replace = TRUE, prob = NULL)
}
```

Function Anatomy

Let’s write a function called `RollDice()` that rolls an arbitrarily sized die an arbitrary number of times and returns the individual results.

```r
RollDice <- function(N_sides, N_dice){
  result <- sample(x = 1:N_sides, size = N_dice, replace = TRUE, prob = NULL)
  return(result)
}
```

Function Anatomy

Let’s write a function called `RollDice()` that rolls an arbitrarily sized die an arbitrary number of times and returns the individual results.

```r
RollDice <- function(N_sides = 6, N_dice = 1){
  result <- sample(x = 1:N_sides, size = N_dice, replace = TRUE, prob = NULL)
  return(result)
}
```

Function Anatomy

Let’s write a function called `RollDice()` that rolls an arbitrarily sized die an arbitrary number of times and returns the individual results.

```r
RollDice <- function(N_sides = 6, N_dice = 1){
  result <- sample(x = 1:N_sides, size = N_dice, replace = TRUE, prob = NULL)
  return(result)
}
```

Function Anatomy

Let’s write a function called `RollDice()` that rolls an arbitrarily sized die an arbitrary number of times and returns the individual results.

```r
RollDice <- function(N_sides = 6, N_dice = 1){
  result <- sample(x = 1:N_sides, size = N_dice, replace = TRUE, prob = NULL)
  return(result)
}
```

Function Anatomy

Let’s write a function called `RollDice()` that rolls an arbitrarily sized die an arbitrary number of times and returns the individual results.

```
RollDice <- function(N_sides = 6, N_dice = 1){
  result <- sample(x = 1:N_sides, size = N_dice, replace = TRUE, prob = NULL)
  return(result)
}
```

Function Anatomy

Let’s write a function called `RollDice()` that rolls an arbitrarily sized die an arbitrary number of times and returns the individual results.

```r
RollDice <- function(N_sides = 6, N_dice = 1){
  result <- sample(x = 1:N_sides, size = N_dice, replace = TRUE, prob = NULL)
  return(result)
}
```

Vectorization

- We’ve already seen this, but let’s be explicit!
- Some functions are *vectorized*, meaning they can operate independently on all elements of a vector
- Vectorized functions take in vectors, arrays, or matrices and return objects of the same size with consistent behavior across all elements

```r
x <- 0:3
exp(x)
x^2
x == 2
```

Vectorization

- We’ve already seen this, but let’s be explicit!
- Some functions are *vectorized*, meaning they can operate independently on all elements of a vector
- Vectorized functions take in vectors, arrays, or matrices and return objects of the same size with consistent behavior across all elements

```r
x <- 0:3
exp(x)

# [1] 1.000000 2.718282 7.389056 20.085537

x^2
x == 2
```

Vectorization

- We’ve already seen this, but let’s be explicit!
- Some functions are *vectorized*, meaning they can operate independently on all elements of a vector
- Vectorized functions take in vectors, arrays, or matrices and return objects of the same size with consistent behavior across all elements

```r
x <- 0:3
exp(x)

# [1] 1.000000 2.718282 7.389056 20.085537

x^2

# [1] 0 1 4 9

x == 2
```

Vectorization

- We’ve already seen this, but let’s be explicit!
- Some functions are *vectorized*, meaning they can operate independently on all elements of a vector
- Vectorized functions take in vectors, arrays, or matrices and return objects of the same size with consistent behavior across all elements

```
x <- 0:3
exp(x)

# [1] 1.000000 2.718282 7.389056 20.085537

x^2

# [1] 0 1 4 9

x == 2

# [1] FALSE FALSE TRUE FALSE
```

Object Oriented Programming

- There are lots of different types of objects in `R`
- These different types of objects are identified internally with “classes”
  - You can use the `class()` function on an object to see what class it is
  - Things without classes are often called “base objects”
- We want objects that keep our data, code and results neatly organized
- We want functions that do predictable things to these objects
- Object Oriented Programming (OOP) is centered around *objects*
  - Objects contain data
  - Objects contain code (called *methods*)
  - Methods are specifically designed to operate on the data in the object
- `R` has a few ways to implement this (S3 and S4 being most common)

Generic Functions (aka Generics)

- Functions that are designed to operate on many different types of objects with a common call
  - `print()`, `summary()`, `coef()`, `plot()`, etc
- Generics look at the type of object they are called on and then use the *method* associated with that type of object
- `print()` just prints an object out
  - What that means depends on what the object is!
- `summary()`
  - Prints out summary statistics for data frames and vectors
  - Prints out whole tables and descriptions for different types of model objects!
- In Part 3 of PS1, you'll start to construct your first model object!

Unit Testing

- Unit testing is an idea we'll introduce now, but it's a *practice* we should always engage in when writing code!
- The basic idea is that we want to write our code in chunks (often in the form of functions)
  - If we write our code in chunks, we can also test our code in chunks
  - This makes it *much* easier to pinpoint where things may be going wrong
  - This will become much more important very soon once we start to include control flow and loops!
- How do you do this?
  - First, make sure your code gives the correct output under a variety of conditions!
  - Second, see what your code does in unexpected situations
    - How does it handle inputs of the wrong type?
    - How does it handle inputs of the wrong size?
    - How does it handle missing (or `NA`) inputs?
  - Third, once you validate each individual piece works, make sure they work *together*

Wrapping Up

Wrapping up

- When you’re combining `R` objects arithmetically, be aware of how things are handled and what conditions do (and do not) trigger warnings and errors!
- Logical statements will help you count objects that satisfy certain conditions and control program behavior in the future
- Writing functions allow you to stop copy-pasting big chunks of code when carrying out repetitive tasks!
- S3 and S4 objects contain both sub objects and code that controls how generic functions act on them!
- Make sure to thoroughly test the different components of your code so that you can pinpoint where problems are coming from

Wrapping up

- PollEv.com/klintkanopka