Day 1, Lecture 2: Base R

Andie Creel

January, 2024

1 Goal

This is R Studio. The goal of this lecture is to see some basics about R so that we can dive into more exciting things tomorrow.

2 Overview of R Studio

Layout of RStudio:

- Script
  - Where you will be writing your own programs
- Environment/../Git
  - Mostly just environment and Git
  - Show which data objects you have loaded in memory
  - Eventually where you'll do version control with GitHub (last day)
- Files/Plots/Packages/Help/Viewer
  - Helps you load packages and other files to load
  - Default window when you're trying to get with a function from a package (we will get to this)
- Console/Terminal
  - where the code actually run
  - Scripts executes in console
  - Code disappear in the console, whereas a script saves your code
  - To run something in the console, type it in and hit "enter"
  - Important to know there is a "terminal" in R studio. Again, just know it's there.

3 Console

- We can type code straight into the console and run it.
- The console in R Studio knows your running R code.

```
# Type the following in the console, then press enter
2+3
```

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

```
# variable assignment happens with an arrow (on a mac can do option -)
a <- 2
b <- 3

a + b
```

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

4 How to write and execute a script

Writing Scripts and Running Code

- New Script: File > New File > R Script (or R Markdown, which is what I used to make the lecture notes), or just the New Document script in the upper left hand corner of the screen > R Script (or R Markdown)
- Keyboard shortcut to run a section of code: highlight or put your cursor on that line and hit Ctrl + Enter (Windows) or Command + Enter (Mac)
- Executing code: the run button at the top right corner of the script
- For R Markdowns, we also have the Knit button at the top of the script.
  - That will run all code and "knit" it together into a HTML or pdf or doc
  - File type is specified at the top of the R Markdown file
  - You can also run code in a R Markdown by highlighting it and hitting Ctrl + Enter (Windows) or Command + Enter (Mac)

R Markdown

- I wrote this pdf using an R Markdown (**show them quickly**).
- I like because I'm able to write lots of notes to myself while I'm coding
  - it produces a nice shareable file
  - can share my notes, code and results with others.

Problem Sets

- If you have no experience with R, start with a script.
- If you want to do a R Markdown and are used to them, that's fine.

You can "clean up" the Environment after you've executed code by clicking the broom icon. **This will delete everything in your environment**.

5 Comments

```
# This is a comment, you can use '#' to write notes to yourself in your code
# - Comments are what make or break good coders
# - Good comments also create coders who can collaborate with others.
# - If you ever think you're writing "too" many comments, you are not
# - The things you think are obvious in your code won't be to others
# - (nor yourself in a year when you get back to a project)

# Comments are written in the same spot as code
# But they are ignored by the computer
# Let's run some code
print("Hello, World!")
```

```
## [1] "Hello, World!"
```

6 Basic Data Types

```
# Numeric -- integer: no decimal points
myInt <- 1

# Numeric -- double: decimal points
myNum <- 2.4
```

```r
# character (string)
myChar_a <- "a"
myChar_b <- 'b'

# You can have a variable that stores a character value,
# but the character value is a number
trick_q <- "1"
trick_q
```

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

Notice that the '1' has quotation marks around it, as the 'a' and 'b' did. This indicates to us that is is a character data type. When you look at the output of myInt, it also returns 1 but without the tick marks because myInt is a numeric variable and trick_q is a character (or string) variable.

```r
# logical (Boolean/Indicator variable): a true/false statement. Use () to evaluate if something is true
myBool_1 <- (3 < 4)
myBool_2 <- (3 > 4)
```

7 Ways to store datatypes

```r
# vector: can only be a vector of one data type (numeric, logical, string)
myVec_n <- c(1, 2, 3, 4, 5)
myVec_s <- c(myChar_a, "b", "c")
myVec_string <- c(1, "b", "c")
myVec_string # notice the 1 has been make char bc of the "
```

```
## [1] "1" "b" "c"
```

Vectors are a useful way to store multiple observations. However, they only a single column of information. In EDS, we are interested in relating different variables to one another (i.e., How does temperature affect lion's hunting success rate?) Two relate multiple variables together, we need make collections of data. The most basic way to do this is a **matrix.**

```r
# we already have a vector of numeric data
myVec_n
```

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

```r
# matrix: should only be a matrix of one data type
myMat_n <- matrix(c(myVec_n,
                    6, 7, 8, 9, 10),
                  nrow = 2,
                  ncol = 5)
```

We can also collect objects in a "list". Lists are very powerful, but more advanced. For now, just know that they also exist.

```r
# Lists
myList <- list(2, "c", myMat_n)
myList[[1]] # returns numeric
```

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

```r
myList[[2]] # returns string
```

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

```
myList [[3]] # returns matrix
```

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

7.1 Data Frames

- Like matrices
- Can have different data types in each column
- Reference specific columns using the "$" operator, followed by the name of the column
- For the most part, you'll be loading new data by reading a csv
- You might have to create one at some point.
- By looking at how they're created we can get a better sense of what goes into them

Let's convert our list to a data frame

```
# data frame: can have multiple data types
myDF <- as.data.frame(myMat_n)
colnames(myDF) # print column names
```

```
## [1] "V1" "V2" "V3" "V4" "V5"
```

These column names dont mean anything to me. Let's assign some column names. Each column is a variabel that you may have collected data on.

```
colnames(myDF) <- c("num_hunts", "temp", "num_adults", "num_cubs", "distance_from_road")

colnames(myDF)
```

```
## [1] "num_hunts"          "temp"               "num_adults"
## [4] "num_cubs"           "distance_from_road"
```

We can now look at a single column using the '$' operator.

```
# investigate one column
myDF$num_adults
```

```
## [1] 5 6
```

```
#create a new column
myDF$total_lions <- myDF$num_adults + myDF$num_cubs
myDF
```

```
##   num_hunts temp num_adults num_cubs distance_from_road total_lions
## 1         1    3          5        7                  9          12
## 2         2    4          6        8                 10          14
```

Let's build a dataframe with multiple datatypes (our lion data frame is all numeric).

```
# Create the data frame
myPpl <- data.frame(
  name = c("Andie", "Bridger", "Scott"),
  gender = c("Female", "non-binary", "Male"),
  male = c(TRUE, FALSE, FALSE),
  income_cat = c("middle", "poor", "rich"),
  park_dist_mi = c(1, 0.5, 0.1)
)
myPpl
```

```
##      name     gender  male income_cat park_dist_mi
## 1   Andie     Female  TRUE     middle          1.0
## 2 Bridger non-binary FALSE       poor          0.5
## 3   Scott       Male FALSE       rich          0.1
```

```
# Try referencing one column
myPpl$name # version 1: $ notation
```

```
## [1] "Andie"   "Bridger" "Scott"
```

```
myPpl[, 1] # version 2: [row, column] notation
```

```
## [1] "Andie"   "Bridger" "Scott"
```

```
# Try referencing one row
myPpl[1,]
```

```
##    name gender male income_cat park_dist_mi
## 1 Andie Female TRUE     middle            1
```

```
# Try referencing one cell
myPpl$name[2] # version 1: $ and bracket
```

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

```
myPpl[2, 1] # version 2: bracket only
```

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

8 A word of caution

- Make sure you don't over write your variables by accident.

```
# assigning new value to same variable (something to do carefully)
a <- 5
a <- a + 1 # If you run this line more than once, you will NOT get six
a
```

```
## [1] 6
```

```
# assigning new value to new variable
a <- 5
a_new <- a + 1 # If you run this line more than one, you WILL get six
a_new
```

```
## [1] 6
```

9 Functions

Functions: once you have initialized them, they take in an input, perform a set of operations on them, and then give you some return value.

Example on board

- consider the function: myF(x) { y <- x + 3; return(y)}
- what does myF(3) return? 6

Points:

- These are helpful when you have something that you do often

- Rule of thumb: if you're copying and pasting code 3 times or more, make function
- (I say if you are going to copy past ever, because even if you think it'll only be twice it'll probably be more)
- Recent example for me:
  - wrote a function to take a date and return the season
  - Wrote a function to get kelvin and return Fahrenheit

9.1 Example: park visitation

Let's say we want to write a function that models the relationship between the probability of someone visiting a national park and the temperature (F). You know that people don't visit the park when it's very cold, nor when it's very hot. You model the relationship using the following quadratic equation

$$v = F/100 - (F/100)^2$$

where v is visits and F is the temperature

```
get_visits <- function(Far){
  v <- Far/100 - (Far/100)^2
  return(v)
}

get_visits(0)
```

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

```
get_visits(25)
```

```
## [1] 0.1875
```

```
get_visits(50)
```

```
## [1] 0.25
```

```
get_visits(75)
```

```
## [1] 0.1875
```

```
get_visits(100)
```

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

So we can see that as the temperature increases from 0F to 50F, the probability of someone taking a trip increases up to 25%. However, after 50F, the probability of taking a trip begins to decrease.

10 Loops

- for loops: iterates through a task for a set number of times
- Consider these loops (psuedo code):
  - For (i in 1 through 4) { print i }
  - For (i in 1 through 4) { print i / 4}
- Can be helpful when
  - Iterating through a column of data and do something to each row
  - Construct a new column and want to construct each row by scratch
  - Simulation models

```
# Complicated code that is simplified by the loop
print(1)
```

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

```
print(2)
```

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

```
print(3)
```

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

```
print(4)
```

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

```
# The following loop does the exact same thing
for (i in 1:4){
  print(i)
}
```

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

```
# more involved
for (i in 1:4){
  print(i/4)
}
```

```
## [1] 0.25
## [1] 0.5
## [1] 0.75
## [1] 1
```

10.1 Let's combine the loop with our get_visits function

```
# combining loop and function
for (i in 0:4){

  y = get_visits(i*25)

  print(y)
}
```

```
## [1] 0
## [1] 0.1875
## [1] 0.25
## [1] 0.1875
## [1] 0
```

What if you want more temperatures than just 0, 25, 50, 75, 100?

```
# Create a vector of the max_temps we want to get visits for
max_temps <- seq(0, 100, by = 10) # could sequence by steps of 1 (n = 100)
max_temps
```

```
##  [1]   0  10  20  30  40  50  60  70  80  90 100
```

```
# combining loop and our function
for (i in max_temps){
```

```
y <- get_visits(i)
print(y)
}
```

```
## [1] 0
## [1] 0.09
## [1] 0.16
## [1] 0.21
## [1] 0.24
## [1] 0.25
## [1] 0.24
## [1] 0.21
## [1] 0.16
## [1] 0.09
## [1] 0
```

10.2 Iterating over a dataframe

Let's recall our data from `myPpl`. We can use a for loop to iteratively change each cell in a column. Let's say all three of our people, Andie, Bridger and Scott, move one mile away from their nearest park (bummer). We could do this one line at at time, like this:

```
# the [i] here is indicating which row we are editing
myPpl$new_park_dist_a[1] <- myPpl$park_dist[1] + 1 # edit for Andie
myPpl$new_park_dist_a[2] <- myPpl$park_dist[2] + 1 # edit for Bridger
myPpl$new_park_dist_a[3] <- myPpl$park_dist[3] + 1 # edit for Scott

# print
myPpl
```

```
##      name     gender  male income_cat park_dist_mi new_park_dist_a
## 1   Andie     Female  TRUE     middle          1.0             2.0
## 2 Bridger non-binary FALSE       poor          0.5             1.5
## 3   Scott       Male FALSE       rich          0.1             1.1
```

This worked, but there is an easier way to do it with a loop. The loop will help minimize the chance of making an error and shorten the amount of code we need to write to achieve our goal.

```
for(i in seq_along(myPpl$park_dist)){
  # change all people's distance
  myPpl$new_park_dist_b[i] <- myPpl$park_dist[i] + 1
}

myPpl
```

```
##      name     gender  male income_cat park_dist_mi new_park_dist_a
## 1   Andie     Female  TRUE     middle          1.0             2.0
## 2 Bridger non-binary FALSE       poor          0.5             1.5
## 3   Scott       Male FALSE       rich          0.1             1.1
##   new_park_dist_b
## 1             2.0
## 2             1.5
## 3             1.1
```

We can see that both versions (a versus b) worked the same. However, the for loop simplifies the code and makes it easier to read.

11 Loop for Simulation

Let's create a simulation model if we know the temperature is going to increase by 0.02 degrees C each year for the next 50 years. Let's make a data frame of these temperature increases (if we have time).

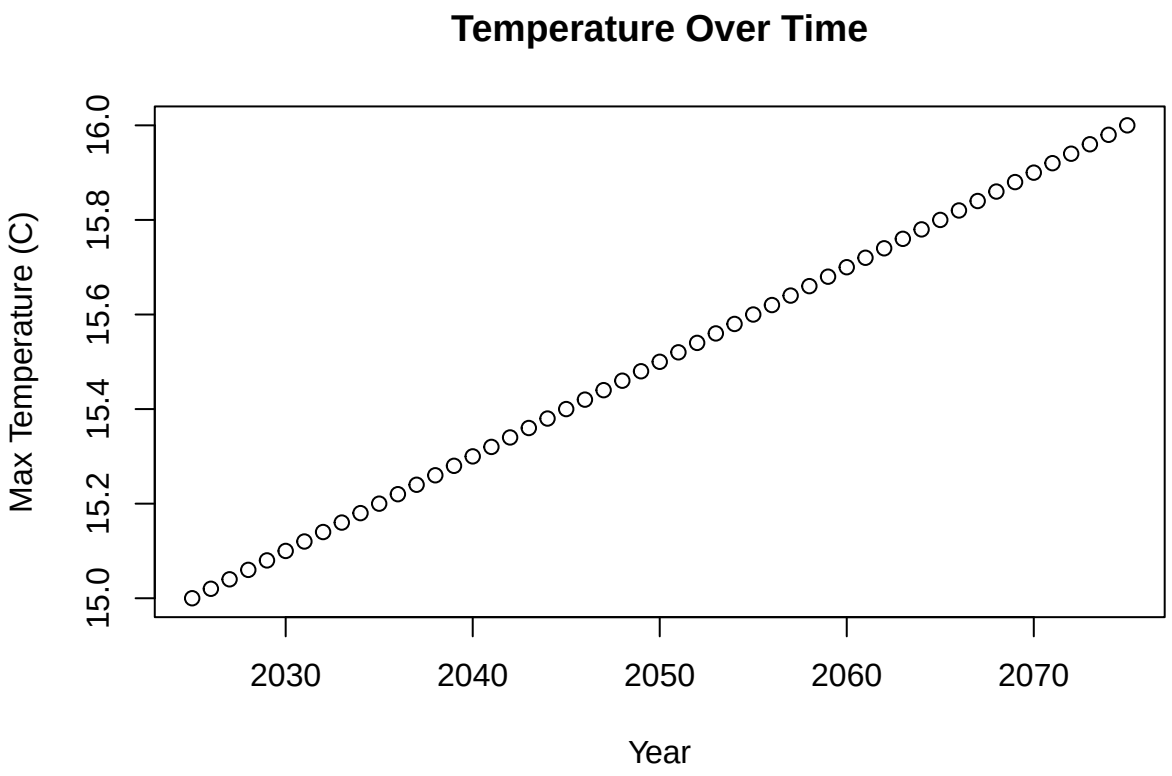
```
# Initialize parameters
initial_temp <- 15 # Initial temperature in degrees Celsius
n_year <- 51 # Number of n_year to simulate
temp_increase_per_year <- 0.02 # Temperature increase per year in degrees Celsius

# Create a numeric vector for the 50 years we will simulate
years <- numeric(n_year)
years[1] <- 2025 # initialize the first year

# Create a numeric vector for the 50 years of temperatures we will simulate
max_temps <- numeric(n_year)
max_temps[1] <- initial_temp # initialize temp in first year

# Simulate the temperature change over the n_year
# notice this will start in year TWO
for (yr in 2:n_year) {
  years[yr] <- years[yr - 1] + 1 # fill in the years
  max_temps[yr] <- max_temps[yr - 1] + temp_increase_per_year # fill in the max_temps
}

# Create a data frame to store the results
simulation_results <- data.frame(
  Year = years,
  Temperature = max_temps
)

# Print the results
head(simulation_results) # show the first few rows
```

```
##   Year Temperature
## 1 2025       15.00
## 2 2026       15.02
## 3 2027       15.04
## 4 2028       15.06
## 5 2029       15.08
## 6 2030       15.10
```

```
tail(simulation_results) # show the last few rows
```

```
##    Year Temperature
## 46 2070       15.90
## 47 2071       15.92
## 48 2072       15.94
## 49 2073       15.96
## 50 2074       15.98
## 51 2075       16.00
```

```
# Plot the results
plot(simulation_results$Year, simulation_results$Temperature,
     xlab = "Year", ylab = "Max Temperature (C)",
     main = "Temperature Over Time")
```



12 If statements

- sometimes you want to execute a task ONLY if a certain condition is met.
- Let's return to our myPpl dataset one last time (for today):
  - Our RA did not record women and non-binary's original distances from parks correctly
  - All women and non-binary people are actually 0.25 miles closer to parks than thought
- What would the correct DF look like?
  - If statements let you fix a mistake like this
  - Also demonstrates why the Boolean (true/false or indicator) variable is so powerful

```
# goes through each row and changes distance if someone is not male
for (i in seq_along(myPpl$male)) {
  if (myPpl$male[i] == FALSE) { # check if someone is "not male"
    myPpl$park_dist_correct[i] <- myPpl$park_dist_mi[i] - 0.25 # adjust
  }else{
    myPpl$park_dist_correct[i] <- myPpl$park_dist_mi[i]
  }
}

myPpl
```

```
##      name     gender  male income_cat park_dist_mi new_park_dist_a
## 1   Andie     Female  TRUE     middle          1.0             2.0
## 2 Bridger non-binary FALSE       poor          0.5             1.5
## 3   Scott       Male FALSE       rich          0.1             1.1
##   new_park_dist_b park_dist_correct
## 1             2.0              1.00
## 2             1.5              0.25
## 3             1.1             -0.15
```

13 Other R Tutorials

UCLA Getting Started with R

git lab intro

14 Some specific packages

We haven't covered packages yet, but a few good resources for the future

ggplot

dplyr and tidyr