Introduction to
R and RStudio

RICE

Fondren Library
Digital Scholarship Services

WHAT IS R?

- Free, open-source
- Data handling and storage
- Programming language
- Huge number of free packages give extended modeling, visualization, manipulation, and more capabilities

WHAT IS RSTUDIO?

- Open-source, but not entirely free
- Graphical user interface providing easy access to R
- Not officially affiliated with R
- Provides amazing cheat sheets!

THE RICE CONNECTION

- Hadley Wickham
  - former Rice professor
  - Chief Scientist at Rstudio
  - Originator of the tidyverse
  - Author of “R for Data Science”

THE RSTUDIO INTERFACE

A walkthrough of important features

RSTUDIO INTERFACE

SIMPLE OPERATIONS

Arithmetic, assignment, matrix multiplication, converting between types

ARITHMETIC AND BASIC MATH

- Addition: `+ (5 + 3)`
- Subtraction: `- (5 - 3)`. Also, negation with `- (-3)`
- Multiplication: `* (5 * 3)`
- Division: `/ (5 / 3)`
- Power: `**` or `^ (5**3, 5^3)`
- Square Root: `sqrt` function (sqrt(5))

ASSIGNMENT

- Naming a value for future use
- Value can change!
- Variable names: must start with a letter, and then only letters, numbers, periods, and underscores

- `value <- 5, my.name <- “hello”, COOL_VALUE <-3 + 5i`
  - `NOT 1value <- 3, my-name <- 3, mass% <- 42`

TYPE CONVERSION

- `as.(TYPE)` functions
  - Example: `as.integer(3.3)`
  - Integer conversion always rounds down!
    - Use `round, floor, ceiling`
- Especially useful for converting between various numeric types

DATA TYPES

Scalars, lists, arrays, matrices, factors, data frames, and vectors

SCALARS

- One value
- Types:
  - Character: `'a'`
  - Numeric (decimal): `3.14`
  - Integer: `-1`
  - Logical: `TRUE` or `FALSE`
  - Complex: `3 + 5i`
- Special missing value: `NA`

ATOMIC VECTORS

- Fixed-length list of values of all the same type
- Created with the `c()` function (concatenate)
- Examples: `c(1, 2, 3)`, `c(0.4, 0.5)`, `c("hello", "hi")`. NOT `c(1, "hello")`!
- Access by subscript. If `a` is a vector, `a[1]` returns the 1st element.
- Ranges of integers with the colon operator: `1:14`
  - With step size: `seq(1, 14, 2)`

LISTS

- Variable list of values of any type
- Created with the `list()` function
- Examples: `list(1, 2, 3), c(0.4, 0.5), c("hello", "hi"), list(1, "hello")`
- Can always access by subscript. If `a` is a list, `a[[1]]` returns the 1st element
- Also, can access by giving custom names to values!
  - `ages <- list(16, 17); names(ages) <- c("steve", "tony")`
  - `ages$steve`

MATRICES

- Native support (uncommon in many programming languages!)
- Initialize: provide list of values and number of rows or columns
- Fills down a column first
  - `matrix(1:4, nrow = 2)` -> $\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$
- Arrays: just matrices with the option to have more dimensions

ACCESSING MATRICES

- Given matrix `mat`
  - Access one value:
    - `mat[3,4]:` element in 3rd row, 4th column
  - Access a column:
    - `mat[,3 ]:` 3rd column
  - Access a row:
    - `mat[3,]:` 3rd row
  - Access a range of rows or columns:
    - `mat[1:2, 3:5]`
      - First 2 rows, columns 3 to 5
- Can save to these as well using assignment operator
  - If using row or column, convert to matrix first!

FACTORS

- Categorical data
  - Only a few possible values
  - Example: US States
- Create a factor with `factor(c(...), levels=c(...))`
- First argument: atomic vector
- Levels: all possible values
  - If not provided, factor will assume you have provided all possibilities
- e.g. `factor(c("Grad Student", "Grad Student", "Staff"), levels=c("Grad Student", "Staff", "Faculty"))`

DATA FRAME

- The quintessential R data type
- Extremely flexible and powerful
- Structure
  - Column: a variable
  - Row: an observation
- Give data as columns
- Example: `data.frame(name=c("John", "Molly"), age=c(15,17))`
  - John is 15, Molly is 17

ACCESSING DATA FRAMES

- Extract a column:
  - `df$ages`
- Extract a row:
  - `df[1,]`
  - Note: this is the same syntax as matrices!
- Can also pass ranges of rows similarly

EXAMINING DATA FRAMES

- Data frames are frequently huge. Use the following commands to take a look without printing out thousands of lines:
  - `head(df)`
    - Take a look at the first few entries
  - `tail(df)`
    - Take a look at the last few entries
  - `colnames(df)`
    - Take a look at the available variables

A NOTE ABOUT FUNCTIONS

- Two types of arguments/parameters to functions:
  - Positional: `plot(data)`. Order matters!
  - Keyword: `plot(data, main="My Plot")`
- Functions in R often take optional keyword parameters. Check a function out in R with `?function` or `help(function)` and see if there are any additional arguments that might help make your life easier!

READING AND WRITING DATA

Excel, Comma-Separated and Tab-Separated Values, and more!

COMMON FILE FORMATS

- Comma/Tab Separated Values: (.csv, .tsv)
  - Example:
    - name,age,occupation (first line: header)
    - tony,23,banker (all other lines: observations)
- Excel files (.xlsx, .xls)
- JSON (.json)
  - {“name”: “Tony”, “age”: 23, “occupation:” “banker”}
- XML (.xml)
  - <person> <name>Tony</name> <age>23</age> <occupation>banker</occupation> </person>

READING FILE FORMATS

- Comma/Tab Separated Values: (.csv, .tsv)
  - `read.csv("filename")`
- Excel files (.xlsx, .xls)
  - `library(openxlsx)`
  - `read.xlsx("filename")`
- JSON (.json)
  - `library(rjson)`
  - `fromJSON(file = "filename")`
- XML (.xml)
  - `library(XML); library("methods")`
  - `xmlParse(file = "filename")`

WRITING FILE FORMATS

- Comma/Tab Separated Values: (.csv, .tsv)
  - `write.csv(data, "filename")`
- Excel files (.xlsx, .xls)
  - `write.xlsx(data, "filename")`
- JSON (.json)
  - `write(toJSON(data), "filename")`
- XML (.xml)
  - `saveXML(xmlDoc, "filename")`

A WORD OF WARNING ON FACTORS

- R is quick to assume strings you pass in data frames are factors
- Example: `df <- data.frame(name=c("John", "Molly"), age=c(15,17))`
  - R assumes `name` is a factor!
  - Does this make sense?
- If this a problem (which it often is), change the type of the column with `as.character()`.
  - `df$name <- as.character(df$name)`
  - Or, pass `stringsAsFactors = FALSE` to your read command

DATA MANIPULATION

Subsetting and filtering

MAKING NEW COLUMNS FROM OLD

- Arithmetic operations can apply to the whole column with no extra work on your part!
  - Very useful for converting between units
  - `df$length.cm <- df$length.in * 2.54`

LOGICAL OPERATORS

- Test for Equality: `==` `(a == b)`
- Test for Inequality: `!=` `(a != b)`
  - Logical Negation: `!`
- Test for Less than: `<` `(a < b)`
- Test for Less than or equal to: `<=` `(a <= b)`
- Test for Greater than: `>` `(a > b)`
- Test for Greater than or equal to: `>=` `(a > b)`

FILTERING / SUBSETTING

- By Row:
  - `df[3:7]` selects the 3rd through 7th row
- By Column:
  - `subset(df, age > 10)` takes all observations where age is greater than 10. note that age is a column in the data frame, and you don't need to type `df$age`!

CLEANING

- Use subsetting to delete unusable rows
  - Common use case: remove rows which have a NA (erroneous results)
  - `df <- subset(df, !is.na(value))`
    - Note that `value != NA` does not work.

SUMMARY FUNCTIONS

- Mean: `mean(data)`
- Median: `median(data)`
- Range: `range(data)`. Gives lower and upper bounds
- Standard Deviation: `sd(data)`

- The `na.rm` parameter: ignore any missing values. Summaries don't work otherwise!
  - Example: `median(data, na.rm = TRUE)`

THE APPLY FAMILY

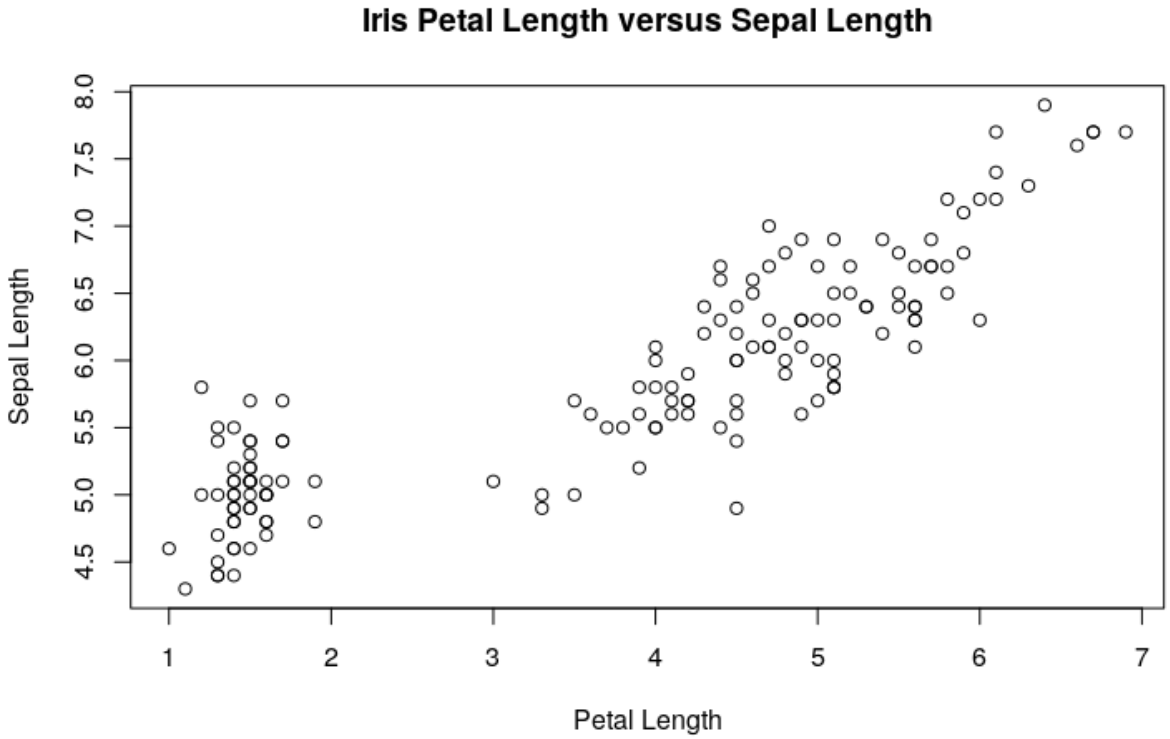
- `sapply(column, function)`
  - Apply function to every element of a vector, and use simplest type of output (e.g. vector if all elements of list are the same)
  - Can pass a function as an argument to another function!
    - Ex: `sapply(names, tolower)`
- `tapply(column, groups, function)`
  - Apply a summary function to a column for each of the groups
  - Ex: Say `gpadf` is a dataframe of students. `tapply(gpadf$gpa, gpadf$major, mean)` will give you the average GPA for each major!

VISUALIZATION

Scatterplots, boxplots, and histograms

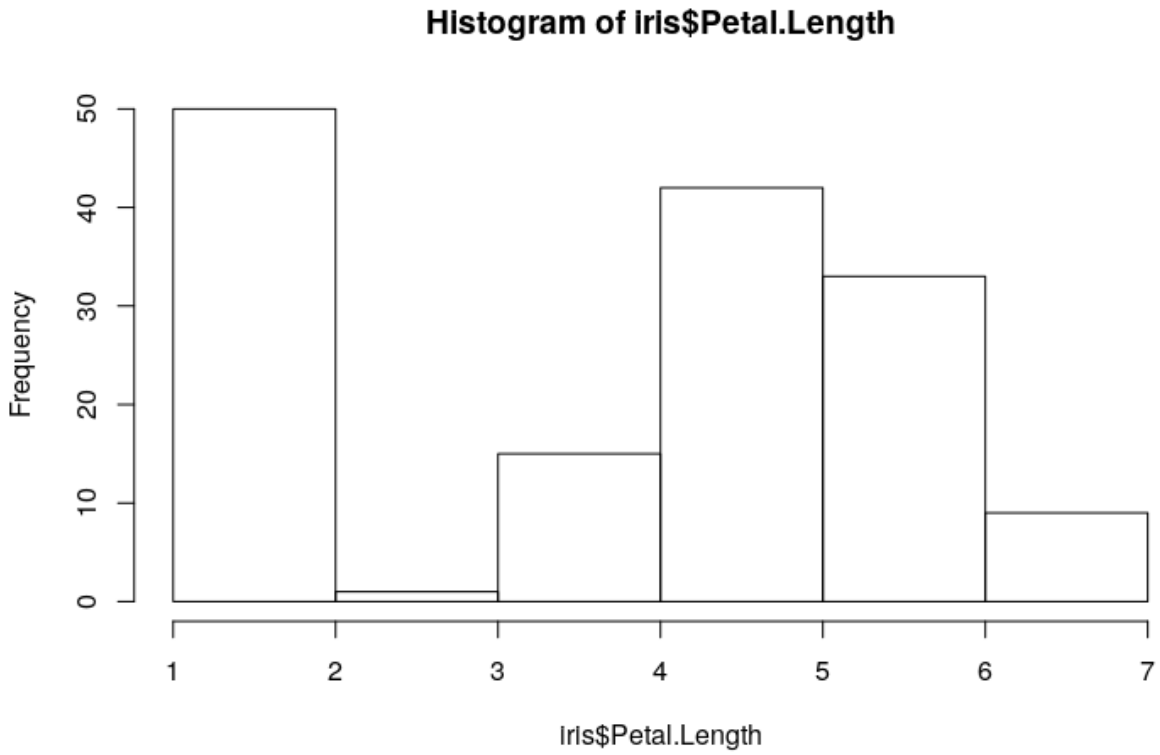
SCATTERPLOTS

- `plot(x_axis, y_axis)` for vectors x_axis and y_axis
- `plot(dataframe)` if the data has two columns, will plot the first variable on the x-axis
- Pass keyword arguments `xlab="x-axis title"`, `ylab="y-axis title"`, and `main="Main Title"` to add titles!



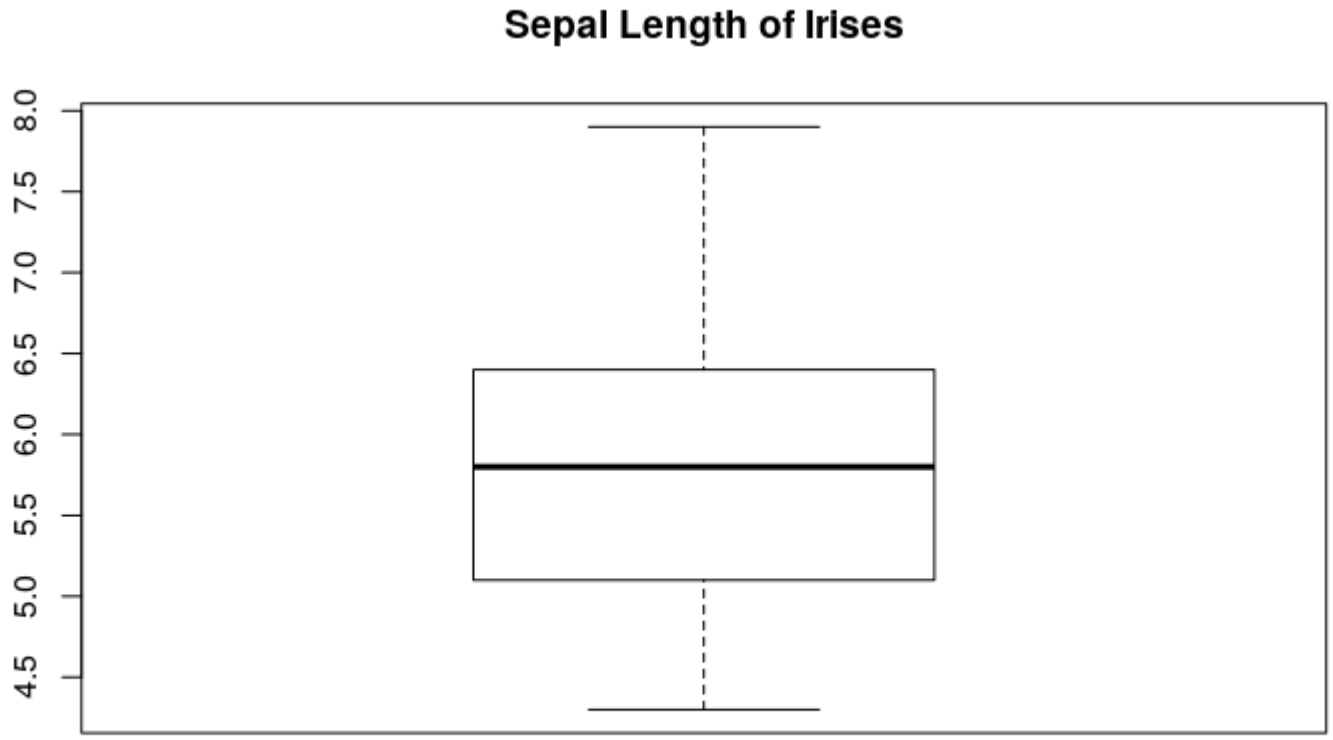
HISTOGRAMS

- `hist(values)`
  - Plots how many values fit into each of a number of equal-sized sections
  - Mildly interesting: you can pass `breaks="scott"` to employ an algorithm to automatically determine a bin size. This is named in honor of a Rice professor's work!



BOXPLOTS

- `boxplot(column)`
  - Similar visualization to a histogram
  - Shows the minimum, 25% percentile, median, 75% percentile, and maximum
  - Also will detect and show outliers



MODELING

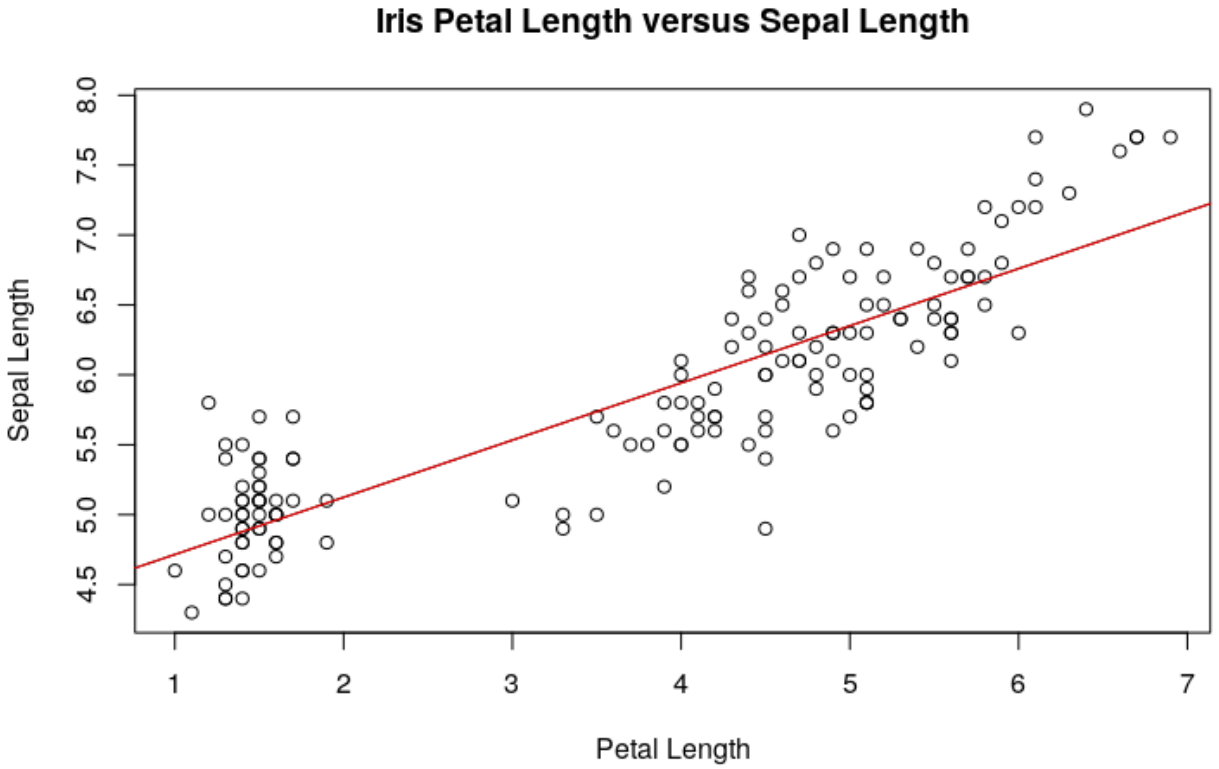
Simple Linear Regression and plotting

LINEAR MODELING

- R uses the `lm` command (linear model) to produce statistical models very quickly
- The syntax can be confusing. Type `model <- lm(y ~ x)` to model an expression of the form $y = mx + b$.
  - Expressions like `y ~ x` are called **formulas** in R.
- Model is now a variable containing a lot of useful information!
  - `model$fitted.values`
  - `model$coefficients`

VISUALIZING LINEAR MODELS

- Use `abline(lm)` to add a line to a graph that already exists
  - `lm` is a linear model object
  - Use `col="red"` (or any other common color name) to change the color of the line and make it easier to see!



ACKNOWLEDGMENTS

- The R logo (https://www.r-project.org/logo/) is used under the terms of the Creative Commons Attribution-ShareAlike 4.0 International license.