Stat 133 Review

Statistics Undergraduate Student Association

Fall 2018

0) R Markdown vs. RStudio vs. R vs. R Script

Remember we installed R, then RStudio in class.

- **R** is the programming language
- **RStudio** is the *Integrated Development Environment (IDE)* that helps you to develop R programs/documents
- **R Markdown** is what we have primarily used – it is a file that allows you to write, save, and execute code but also allows you to write lines which do not need to be executed. The formatting is conducive for generating detailed reports
- **R Script** is a lot like RMarkdown except instead of creating chunks when you want to execute code, the whole file is a chunk! So this is useful if you are not planning on writing lengthy explanations and are just trying to execute code.

1) R Markdown Syntax

Emphasis: * To make text bold or italic, wrap with asterisks `*` or underscores `_` * To prevent bold or italic, place a backslash in front `\*` or `\_`.

Paragraphs: * Create a line break using either a backslash `\` or two blank spaces at the end of the line.

Headings: * Use successive pound symbols `#` to create smaller and smaller headers.

Blockquotes: * Start a line using the greater than symbol `>` * A nested blockquote can use `> >`

Lists: * *Unordered lists* can are made using either asterisks `*`, plus signs `+`, or hyphens `-`. * Ordered lists use numbers followed by period `.` or right paren `)`.

Hyperlinks: * of the form `[text](URL)` * Example: Twitter

Images: * of the form `![image](URL)` or `![image](absolute path to image on computer)`

Create tables (manually):

- `|:------:|` to start a table
- Example:

|:————–|———:| | Ice Cream | Rating | | Chocolate | 9 | | Vanilla | 6 | | Cookies n Cream | 5 | | Mint Chip | 10 | | Strawberry | 9 |

LaTex:

- Use dollar sign notation (append with `$$` on each end) and LaTex syntax:
- Example: `$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$`

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

2) Data Structures and Data Types

- **Data Structures:** contain data – like vectors, matrices, etc.
- **Data Types**: **Integer**, **Double**, **Logical**, **Character**, **Complex**, **Raw**

3) Vectors

- Vectors are the most basic data structure in R. AKA "Contiguous cells containing data"
- The data types apply to vectors:

```
vec <- c('alpha', 'beta', 'lambda')
typeof(vec)
```

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

Atomicity:

- *vectors are atomic structures*
- This means all values in a vector must be of the same type
- If this is not the case, then R will attempt to coerce the data structure so that all elements it contains are the same type

Rules of coercion:

1) If a character is present, R will coerce all elements into characters (just add quotes, easy)
2) If a vector contains logicals and numbers, R will convert the logicals to numbers (TRUE = 1, FALSE = 0)

Example:

Suppose you enter the following command into a code chunk:

```
x <- c(1, 2, 3, TRUE, TRUE, FALSE)
x
```

What will the output be?

Vector Recycling

- R is vector-centric – you can perform arithmetic operations on vectors just like you would real numbers, but if you are working with vectors of different lengths, vector recycling occurs.

For example:

```
x <- c(3, 4, 9)
x + 3 # Adds 3 to each element of x, not just the first element.
```

```
## [1]  6  7 12
```

What is the output of the following chunk?

```
y <- c(2, 1/2, 1/3, 98)
z <- c(1, 2, 3)
```

Would it return an error message?

4) Arrays and Matrices

- You can create an array by giving a vector a dimension attribute. Arrays are not super important for this class. Matrices can be created in a similar way by specifying the number of rows and columns.

```
a <- 1:8
A <- matrix(a, nrow = 2, ncol = 4)
A
```

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

- R stores matrices as vectors (so they are also atomic). Matrices are stored *by columns* in R.

```
b <- 1:8
B <- matrix(b, nrow = 2, ncol = 4, byrow = TRUE)
B
```

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

- Matrices are the atomic version of dataframes, which are matrices that can include columns/rows of different datatypes.

5) Factors

Another data structure in R is called `factor`. Factors are used to manage *categorical data*.

For example, say I take a poll of UC Berkeley students

```
eye_color <- c("hazel", "blue", "brown", "green", "green", "green", "brown", "hazel", "blue", "green")
eye_color <- factor(eye_color)
eye_color
```

```
##  [1] hazel blue  brown green green green brown hazel blue  green
## Levels: blue brown green hazel
```

- Notice how it assigns levels to the vector. Factors are internally stored as *vectors of integers* and behave similarly to vectors.
- *Transform a variable into a factor:*

First, let's manipulate the vector `bluebirds` by setting all values less than or equal to 2 to 2, and all values greater than or equal to 6 to 6.

```
bluebirds <- c(1,0,3,5,4,6,9,2,1,5,2,3,8,0)
bluebirds[bluebirds <= 2] = 2
bluebirds[bluebirds >= 6] = 6
```

Now, we use factor to convert `bluebirds` to a factor vector.

```
bluebirds <- factor(bluebirds, 2:6, labels = c('2-', 3, 4, 5, '6+'))
```

6) Lists

A list is the most general data structure in R, and they can contain any other type of data strucure, even other lists (meta!).

Lists are a special type of vector, but they are NOT atomic structures.

7) Data Frames

The primary structure that R uses to handle tabular data. Use `data.frame()`. Let's use some of R's preloaded data to create a dataframe.

```
df <- data.frame(Species = iris$Species, Sepal.Length = iris$Sepal.Length, Petal.Width = iris$Petal.Wid
head(df, 5)
```

```
##   Species Sepal.Length Petal.Width
## 1  setosa          5.1         0.2
## 2  setosa          4.9         0.2
## 3  setosa          4.7         0.2
## 4  setosa          4.6         0.2
## 5  setosa          5.0         0.2
```

Data frames are stored as a list of vectors or factors in R. Columns are generally atomic structures, but column types can vary within a data frame.

Data frames are not matrices, although they behave similar to them.

Functions for inspecting data frames:

- `str()` : structure
- `head()` : first rows
- `tail()` : last rows
- `summary()` : descriptive statistics
- `dim()` : dimensions
- `nrow()`, `ncol()`
- `names()`, `colnames()` : column names
- `rownames()`

8) Subsetting and Indexing

- *Don't forget that R indexing starts at 1* NOT *0*
- To extract values from R, use brackets and fill them with indices of what you want to return (indices need not be numbers).

```
y <- c("turquoise", "magenta", "chartreuse", "amber", "canary", "olive")
```

```
y[2:5]
```

```
## [1] "magenta"    "chartreuse" "amber"      "canary"
```

You can also return non-consecutive elements:

```
y[c(3,5)]
```

```
## [1] "chartreuse" "canary"
```

Extracting from lists:

```
list1 <- list(c(1,2,3), matrix(1:9, nrow = 3, ncol = 3), list(1:2, c(TRUE, FALSE), c("a", "b")))
list1[2] # a matrix, and the same as
```

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

```
list1[[2]]
```

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

Extracting from data frames:

```
df <- data.frame(Species = iris$Species, SepalLength = iris$Sepal.Length, PetalWidth = iris$Petal.Width)
head(df, 5)
```

```
##   Species SepalLength PetalWidth
## 1  setosa         5.1        0.2
## 2  setosa         4.9        0.2
## 3  setosa         4.7        0.2
## 4  setosa         4.6        0.2
## 5  setosa         5.0        0.2
```

```
df[1,2] # 1st row, 2nd column
```

```
## [1] 5.1
```

```
df[1:2, 2:3] # intersection of first 2 rows with 2nd and 3rd columns
```

```
##   SepalLength PetalWidth
## 1         5.1        0.2
## 2         4.9        0.2
```

```
df[c(1,50,100), c(1,3)] # intersection of 1st, 50th, and 100th rows and 1st and 3rd columns
```

```
##        Species PetalWidth
## 1       setosa        0.2
## 50      setosa        0.2
## 100 versicolor        1.3
```

```
head(df[ ,1],5) # first five rows of first column
```

```
## [1] setosa setosa setosa setosa setosa
## Levels: setosa versicolor virginica
```

```
head(df[ ,'SepalLength'],5) # first five rows of second column
```

```
## [1] 5.1 4.9 4.7 4.6 5.0
```

Dollar Notation

- You can index lists, vectors, dataframes, etc. using dollar notation.

```
list2 <- list(vec = c(1,2,3), matr = matrix(1:9, nrow = 3, ncol = 3), lst = list(1:2, c(TRUE, FALSE), c
list2$matr #prints the element in list that is named 'matr'
```

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

```
list2[[1]][2] # returns the same value as
```

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

```
list2$vec[2]
```

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

Example: How many flowers in `iris` dataset have sepal length of at least 7 cm?

```
sum(iris$Sepal.Length >= 7, na.rm = TRUE)
```

```
## [1] 13
```

- What is the median petal width of flowers of species versicolor?

```
median(iris$Petal.Width[iris$Species == "versicolor"])
```

```
## [1] 1.3
```

- Create a data frame with the sepal length, sepal width, petal length, and petal width of flowers of species setosa. Display this dataframe.

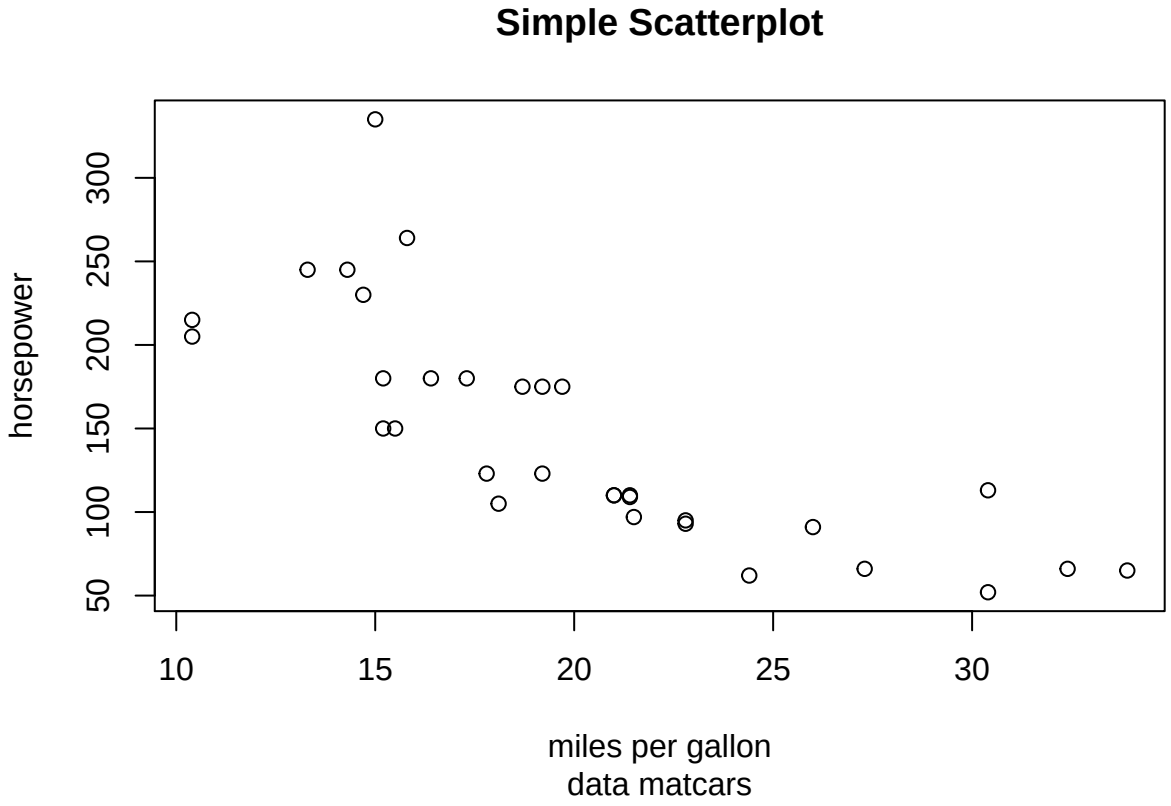
```
sl <- as.vector(iris$Sepal.Length[iris$Species == "setosa"])
sw <- as.vector(iris$Sepal.Width[iris$Species == "setosa"])
pl <- as.vector(iris$Petal.Length[iris$Species == "setosa"])
pw <- as.vector(iris$Petal.Width[iris$Species == "setosa"])
df <- data.frame("sepal length" = sl, "sepal width" = sw, "petal length" = pl, "petal width" = pw)
```

4) Perform

9) Plotting

- Let's use the pre-loaded dataset `mtcars` to create a scatterplot.

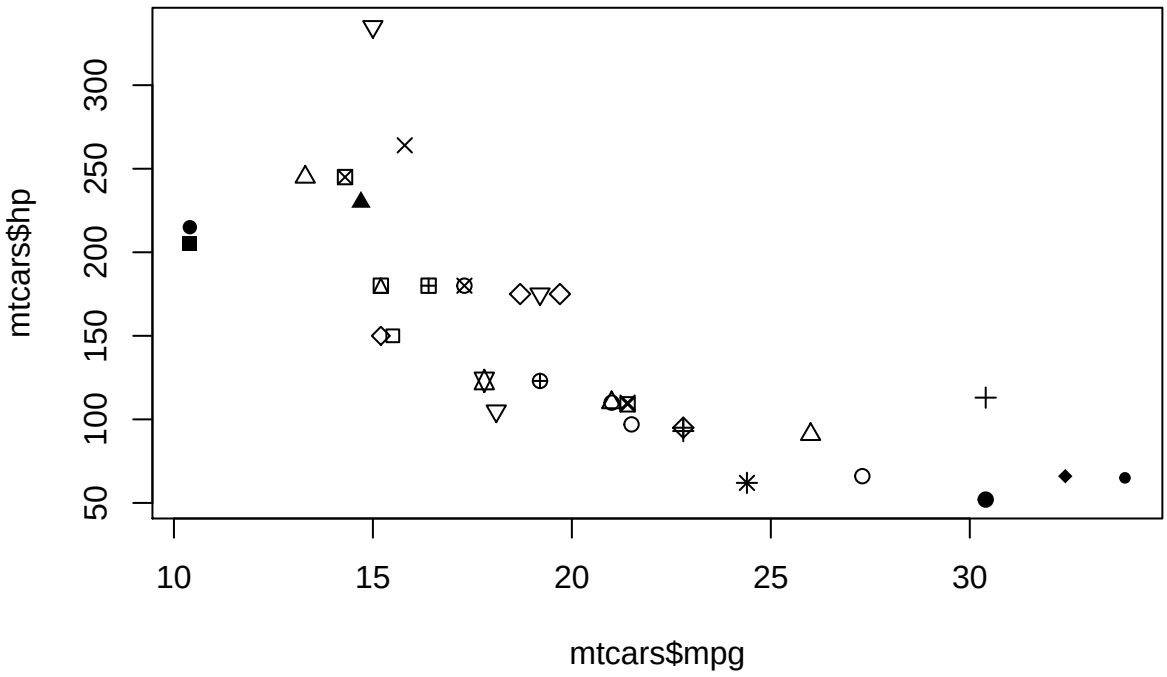
```
plot(mtcars$mpg, mtcars$hp, xlab = "miles per gallon",
     ylab = "horsepower", main = "Simple Scatterplot",
     sub = 'data matcars')
```



cex and pch

- `cex` scales the size of the plotting text and symbols (character expansion).
- `pch` changes the point shapes for a scatterplot. It ranges from 0 to 25, and also includes *, ., o, O, 0, +, -, |, %, #, @.
- You can also select a range of pch symbols, and these can be recycled (note how symbols are repeated here because there are more than 25 datapoints but only 25 symbols to choose from).

```
plot(mtcars$mpg, mtcars$hp, pch = 1:25)
```



- Use `abline()` to draw straight lines on your graph.
- Use `polygon()` to draw shapes in your graph.
- *Plot from Scratch with `plot.new()`*: make sure you understand how to use `plot.new()`

  Plot html graphs with `plotly` package: * `plot_ly(,)`

10) Data visualizations with `ggplot2`

- "Data Visualization is simply *mapping data* to *geometric objects* and their *visual attributes*."
- You should specify the dataset, the variables, the geometric objects, and the visual attributes to ggplot.
- **Geometric Objects** (primitives): points, lines, bars, polygons
- **Visual Attributes** position, shape, orientation, size, color, border, fill pattern

`ggplot2` terminology:

- `ggplot` - the main function where you specify the dataset and variables to plot
- `geoms` - geometric objects such as `geom_point()`, `geom_bar()`, `geom_density()`, `geom_line()`, `geom_area()`
- `aes` - aesthetics such as shape, transparency, color, fill, linetype
- `scales` - define how to plot your data (continuous, discrete, log)

`ggplot2` structure: uses +

Example:

We can color our points by the Species:

```
library(ggplot2)
ggplot(
  data = iris,
  aes(
    x = Sepal.Length,
    y = Sepal.Width,
```

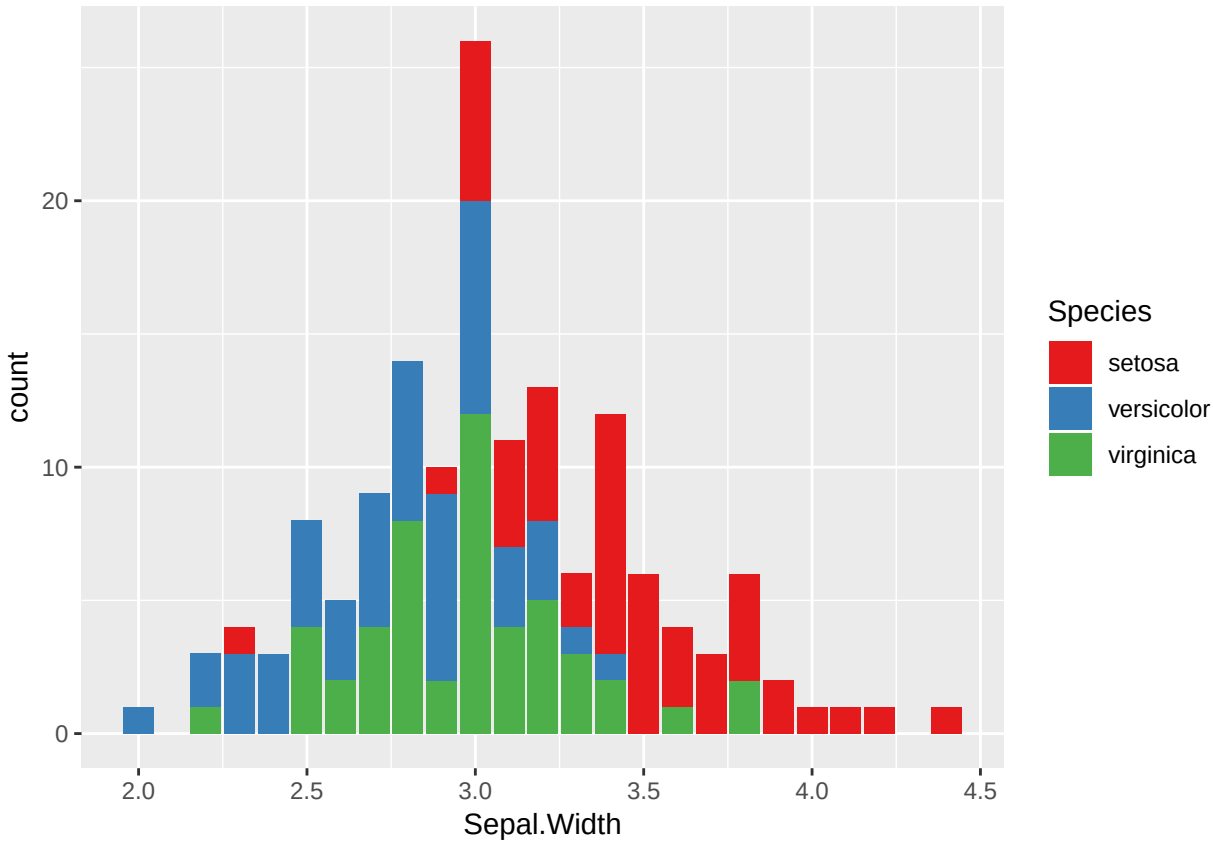
```
    color = Species)) +
  geom_point(size = 3)
```

We can differentiate points by shape:

```
 ggplot(iris, aes(Sepal.Length, Sepal.Width, color = Species)) +
geom_point(aes(shape = Species), size = 3)
```

Using `RColorBrewer` Package:

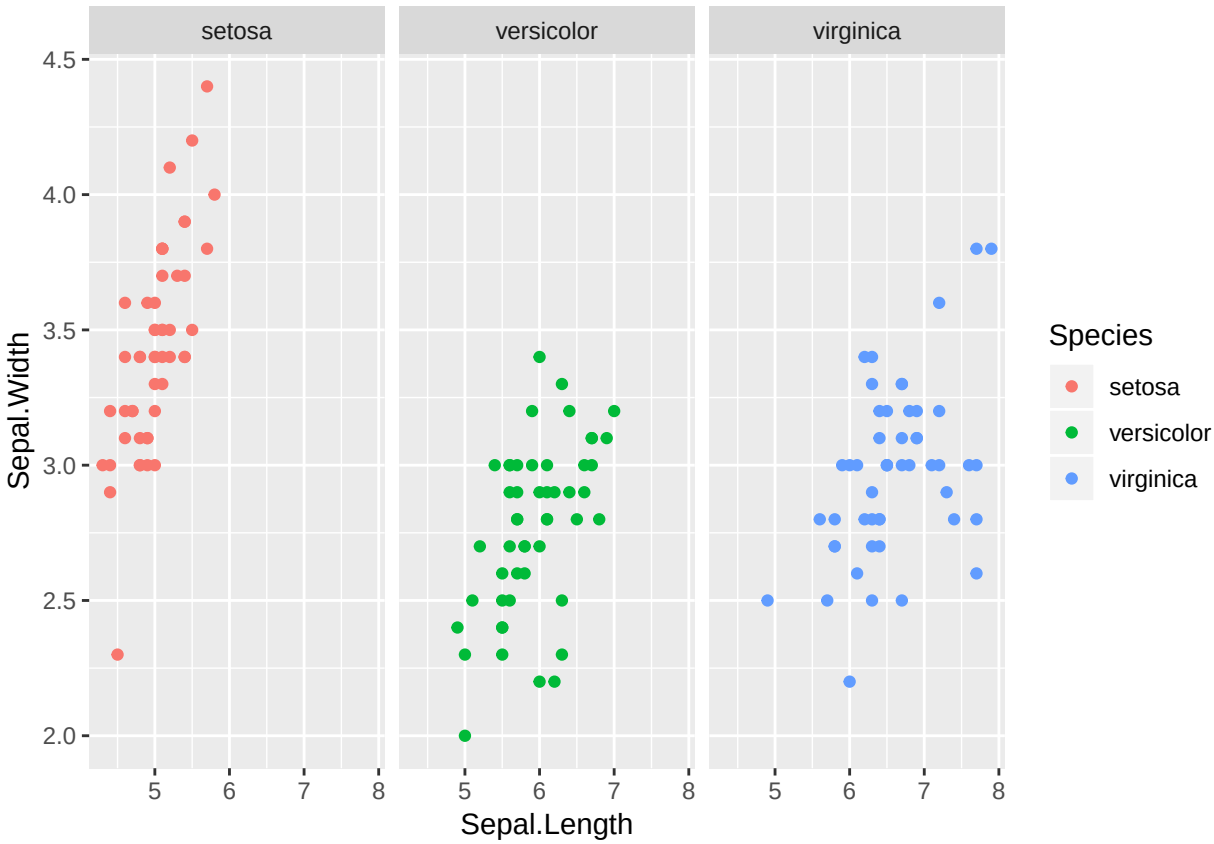
```
ggplot(iris, aes(Sepal.Width, fill = Species)) +
geom_bar() +
scale_fill_brewer(palette = "Set1")
```



Faceting

By column:

```
ggplot(iris,
       aes(Sepal.Length,
           Sepal.Width,
           color = Species)) +
  geom_point() +
  facet_grid( ~ Species)
```



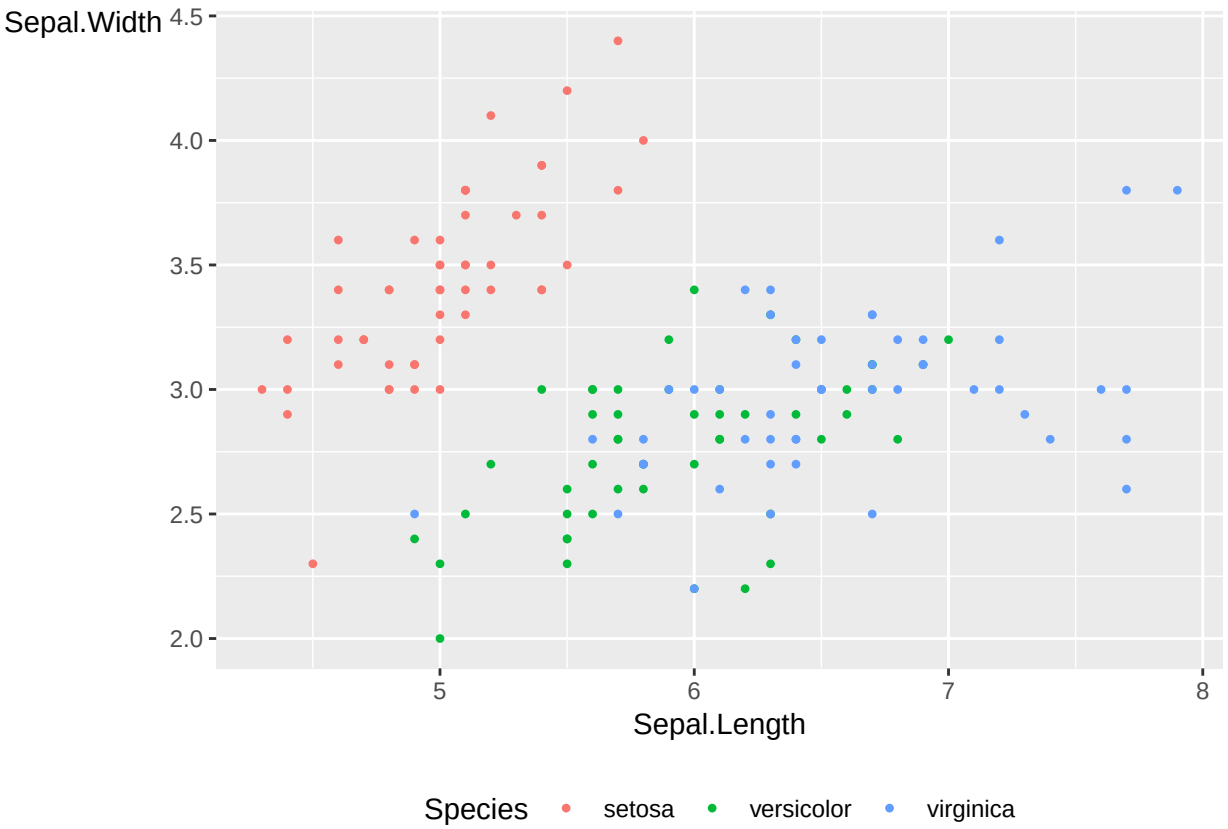
- To facet by columns: just write `facet_grid(Species ~ .)`
- By rows: just write `facet_grid(. ~ Species)`

Themes

- Type ?theme() to see all the modifications.
- Themes are useful for customizing plots.

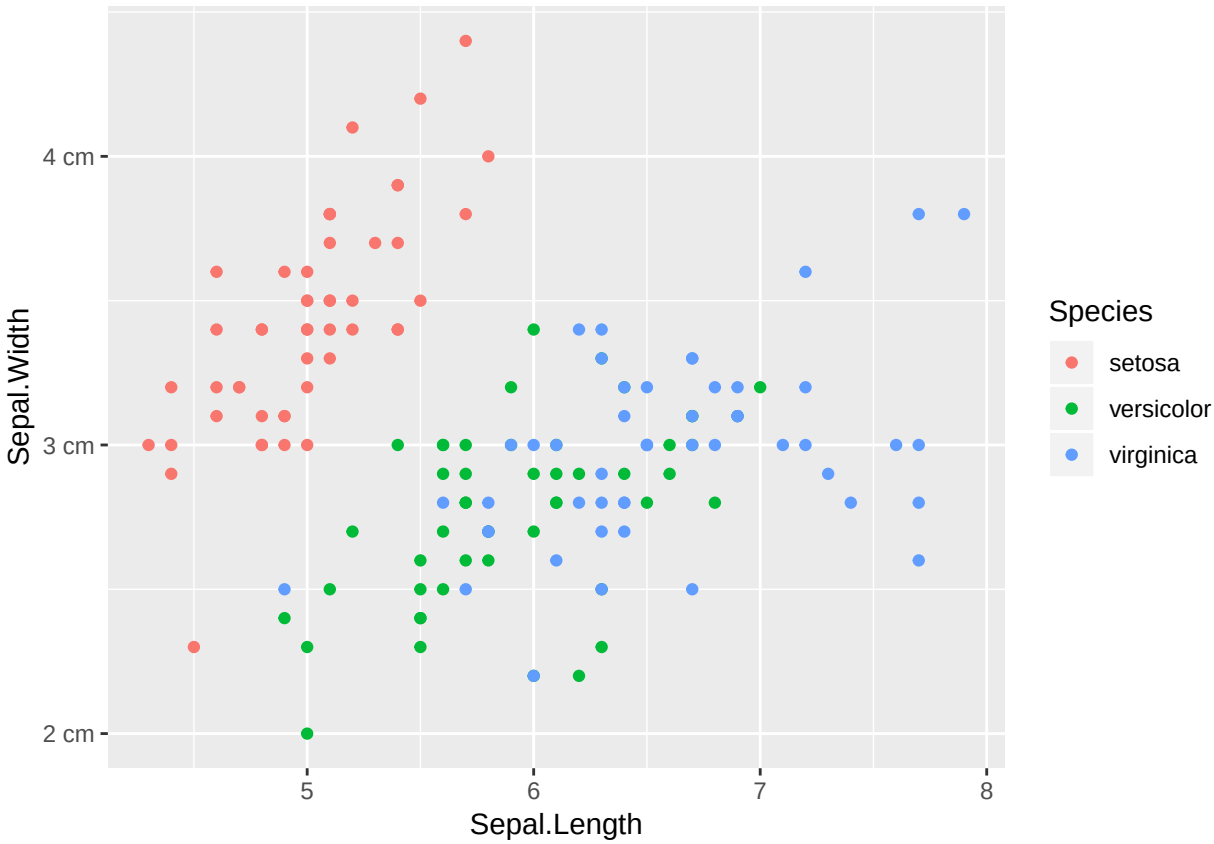
Example:

```
ggplot(iris, aes(Sepal.Length, Sepal.Width, color = Species)) +
geom_point(size = 1.2, shape = 16) +
theme(legend.key = element_rect(fill = NA),
legend.position = "bottom",
strip.background = element_rect(fill = NA),
axis.title.y = element_text(angle = 0))
```



Scaling

```
 # Assign the plot to an object
ggplot(iris,
       aes(Sepal.Length,
           Sepal.Width,
           color = Species)) +
  geom_point() +
  scale_y_continuous(breaks = seq(2, 8, by = 1),
                     labels = paste0(2:8, " cm"))
```



11) Package `dplyr`

`dplyr` is often implemented for package manipulation.

- `filter()` : keep rows matching criteria
- `select()` : pick columns by name
- `slice()` : select rows by position
- `arrange()` : reorder rows
- `mutate()` : add new variable
- `summarise()`: reduce variables to values

Find all people from the following dataframe that are older than 20 years with salaries of at least 50,000 dollars.

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
df <- data.frame(name = c('April Applegate', 'Bridget Boone', 'Christina Castles', 'Devonte Denison', '
filter(df, age > 20 & salary_in_thousands >= 50)
```

```
##                name age               job salary_in_thousands
## 1 Christina Castles  46 software engineer                  100
## 2   Devonte Denison  45           teacher                   55
## 3        Gina Green  34  marine biologist                   65
## 4 Herbert Hitchcock  22 software engineer                   95
```

Find the salaries of all clerks

```
select(filter(df, job == "clerk"), salary_in_thousands)
```

```
##   salary_in_thousands
## 1                  40
## 2                  43
## 3                  49
```

Order the people by age in reverse order

```
arrange(df, desc(age))
```

```
##                name age               job salary_in_thousands
## 1 Christina Castles  46 software engineer                  100
## 2   Devonte Denison  45           teacher                   55
## 3      Ivan Iverson  44             clerk                   49
## 4        Gina Green  34  marine biologist                   65
## 5      Freddy Fugle  31             clerk                   43
## 6     Bridget Boone  23           teacher                   45
## 7 Herbert Hitchcock  22 software engineer                   95
## 8        Edna Epsom  19             clerk                   40
## 9   April Applegate  18      telemarketer                   30
```

Print the second, third and fourth rows in the dataframe.

```
slice(df,2:4)
```

```
##                name age               job salary_in_thousands
## 1     Bridget Boone  23           teacher                   45
## 2 Christina Castles  46 software engineer                  100
## 3   Devonte Denison  45           teacher                   55
```

Add a column to the dataframe called `salary` that takes `salary_in_thousands` and multiplies it by 1000.

```
df <- mutate(df, salary = salary_in_thousands*1000)
df
```

```
##                name age               job salary_in_thousands salary
## 1   April Applegate  18      telemarketer                   30  30000
## 2     Bridget Boone  23           teacher                   45  45000
## 3 Christina Castles  46 software engineer                  100 100000
## 4   Devonte Denison  45           teacher                   55  55000
## 5        Edna Epsom  19             clerk                   40  40000
## 6      Freddy Fugle  31             clerk                   43  43000
## 7        Gina Green  34  marine biologist                   65  65000
## 8 Herbert Hitchcock  22 software engineer                   95  95000
## 9      Ivan Iverson  44             clerk                   49  49000
```

Print the min, median, average, and max salaries using `summarise()`

```
summarise(
  df,
  min_salary = min(salary),
```

```
  median_salary = median(salary),
  avg = mean(salary),
  max = max(salary))
```

```
##   min_salary median_salary   avg   max
## 1      30000         49000 58000 1e+05
```

Calculate average salary by job

```
summarise(
  group_by(df, job),
  avg_salary = mean(salary)
)
```

```
## # A tibble: 5 x 2
##   job               avg_salary
##   <fct>                  <dbl>
## 1 clerk                  44000
## 2 marine biologist       65000
## 3 software engineer      97500
## 4 teacher                50000
## 5 telemarketer           30000
```

12) Data Pipelines from `magrittr`

%>% is the pipe operator from `magrittr`

Print the means of the salaries of people grouped by their jobs, on the condition that their salaries are at least 50,000 a year.

```
  summarise(
    group_by(
      filter(
        df, salary_in_thousands >= 50
      ),
      job
    ),
    mean_salary = mean(salary_in_thousands)
  )
```

```
## # A tibble: 3 x 2
##   job               mean_salary
##   <fct>                   <dbl>
## 1 marine biologist         65
## 2 software engineer        97.5
## 3 teacher                  55
```

This is simpler to do using piping:

```
df %>% filter(salary_in_thousands >= 50) %>% group_by(job) %>% summarise(mean_salary = mean(salary_in_th
```

```
## # A tibble: 3 x 2
##   job               mean_salary
##   <fct>                   <dbl>
## 1 marine biologist         65
## 2 software engineer        97.5
## 3 teacher                  55
```

13) Functions

Roxygen Comments: Include them before your function.

'@title #' @description

'@param x description #' @return

- *Example*: Write a function called `convert` that converts Fahrenheit temperatures to Kelvin. Then call that function.

```
#' @title Fahrenheit to Kelvin
#' @description converts temperature in Fahrenheit to Kelvin
#' @param x temperature in degrees Fahrenheit
#' @return temperature in degrees Kelvin
convert <- function(x) {
  y <- ((x - 32) * (5 / 9)) + 273.15
  return(y)
}
convert(32)
```

```
## [1] 273.15
```

- Use `stop("")` inside your function to indicate that a condition returns an error message.
- use if...else statements when you want to test a condition only if a separate condition fails

Loops

For loops: When applying a function many times, consider using a for loop to save yourself time.

- *Example*: Write a for loop that triples each element of the vector `vec` and stores these values as a vector vec2.

```
vec <- c(3, 1, 4, 1, 5)
vec2 <- rep(0, length(vec))

for (i in c(1:length(vec))) {
  vec2[i] <- vec[i]*3
}
vec2
```

```
## [1]  9  3 12  3 15
```

This is shorter than telling R to multiply each element in `vec` by 3 and then store each element.

- *Example*: Write a for loop that sums the first n non-positive powers of 2.

```
n = 1000 #can place any nonnegative integer here
total = 0

for (k in c(0:n)) {
  total <- total + 2^-k
}
```

Repeat loops:

- *Example*: Divide a number by 2 until it becomes odd.

```
val_rep <- 898128000 # Change this value!

repeat {
  print(val_rep)
  if (val_rep %% 2 == 1) {
    break
  }
  val_rep <- val_rep / 2
}
```

```
## [1] 898128000
## [1] 449064000
## [1] 224532000
## [1] 112266000
## [1] 56133000
## [1] 28066500
## [1] 14033250
## [1] 7016625
```

While loops

- *Example*: Same as previous.

```
val_while <- 898128000 # Change this value!

while (val_while %% 2 == 0) { # Continue the loop as long as val_while is even.
  print(val_while)
  val_while <- val_while / 2
}
```

```
## [1] 898128000
## [1] 449064000
## [1] 224532000
## [1] 112266000
## [1] 56133000
## [1] 28066500
## [1] 14033250
```

14) Tests

- It is important to test your functions to make sure they operate the way you want them to.
- *Example*: The typical structure of the tests has the following form:

```
# load the source code of the functions to be tested
source("functions.R")

# context with one test that groups expectations
context("Test for range value")

test_that("range works as expected", {
  x <- c(1, 2, 3, 4, 5)

  expect_equal(stat_range(x), 4)
  expect_length(stat_range(x), 1)
```

```
  expect_type(stat_range(x), 'double')
})
```

- use `context()` to describe what the test are about
- use `test_that()` to group expectations
- to run the tests from the R console, use the function `test_file()` by passing the path of the file `tests.R`
- File structure should include a folder called `tests` that includes R scripts of the tests for your functions

15) String Manipulation & Regex

- `nchar()` : counts the number of characters in a string
- `paste()` : allows you to append character vectors separated by a blank space (by default) or any other separation
- `paste0()` : is the same as `paste` except the default separation is "" (no separation)
- `substr()` : extracts substrings in a character vector

```
substr(
  'Alphabet',
  2, # the index of the character to start with
  4) # the indec of the character to end with
```

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

- *Example*: How can you generate a character vector with the names file1.csv, file2.csv, ... , file10.csv in R? Come up with at least three different ways to get such a vector:

```
# vector of file names
noquote(paste(paste('file', 1:10, sep  = ''), ".csv", sep = ''))
```

```
##  [1] file1.csv  file2.csv  file3.csv  file4.csv  file5.csv  file6.csv
##  [7] file7.csv  file8.csv  file9.csv  file10.csv
```

```
noquote(paste0(paste0('file', 1:10), ".csv"))
```

```
##  [1] file1.csv  file2.csv  file3.csv  file4.csv  file5.csv  file6.csv
##  [7] file7.csv  file8.csv  file9.csv  file10.csv
```

```
noquote(sprintf("file%d.csv", 1:10))
```

```
##  [1] file1.csv  file2.csv  file3.csv  file4.csv  file5.csv  file6.csv
##  [7] file7.csv  file8.csv  file9.csv  file10.csv
```

- `gsub()` : replaces all matches of a string

  `gsub()` takes arguments: * pattern: string to be matched * replacement: string for replacement * x: string or string vector * ignore.case: if TRUE, ignore case

- `cat()` function: converts its arguments to character strings, concatenates them, separating them by the given sep= string, and then prints them.

  The following functions are from the package `stringr` * `str_sub()` : used to extract certain characters from a string * `str_replace()` : used to replace certain patterns from strings * `str_split()` : used to split a string based on a certain pattern

- *Example*: Subset the vector `times` by AM/PM and by actual hour using `stringr` functions

```
library(stringr)
times <- c('12PM', '10AM', '9AM', '8AM', '2PM')
str_sub(times, start = 1, end = nchar(times) - 2)
```

```
## [1] "12" "10" "9"  "8"  "2"
```

```
str_sub(times, start = nchar(times) - 1, end = nchar(times))
```

```
## [1] "PM" "AM" "AM" "AM" "PM"
```

- Note that **metacharacters** need special treatment. For example, `.`, `[`, `]`, `(`, `)`, `^`, `$`, `/`, `*`, `+` need to be considered when we are searching for them appearing in a string.
- *Example* : Take the following vector `locs` and create a list containing the latitudes and longitudes of `locs`.

```
locs <- c(
  "(37.7651967350509,-122.416451692902)",
  "(37.7907890558203,-122.402273431333)",
  "(37.7111991003088,-122.394693339395)",
  "(37.7773000262759,-122.394812784799)",
  NA
)
lat_lon <- str_split(str_replace_all(locs, pattern = '\\(|\\)', replacement = ''), pattern = ',')
```

16) Packages

Writing an R Package from scratch:

1) Install `devtools` and `roxygen2` packages
2) File > New Project > New Directory > R Package
3) Name your package

   Minimal Filestructure: * `DESCRIPTION` is a text file (with no extension) that has metadata for your package. Simply put, this file is like the business card of your package. * `NAMESPACE` is a text file (also with no extension) that is used to list the functions that will be available to be called by the user. * The `R/` directory which is where you store all the R script files with the functions of the package. * The `man/` directory which is the folder containing the Rd (R documentation) files, that is, the text files with the technical help documentation of the functions in your package. * `nameofyourpackage.Rproj` is an RStudio project file that is designed to make your package easy to use with RStudio. * `.Rbuildignore` is a hidden text file used to specify files to be ignored by R when building the tar-ball or bundle of the package.

- *Vignettes*:
- If your package includes vignettes (i.e. there's a `vignettes/` subdirectory) written with .Rmd files, then `DESCRIPTION` needs a field and value `VignetteBuilder: knitr`.
- A vignette is basically a tutorial, created in order to show users how to execute the functions in the package.

13) Directories and Paths

Think of directories like folders. R is always situated at a directory in your computer.

- getwd()
- setwd()

Can be used to manipulate which directory you are working in.

The current directory is the *working directory.*

When you open up terminal, the working directory is the *home directory.*

- / is the root (top level) directory
- ~ is the home directory
- . is the current directory
- .. is the parent directory

Each file and directory has a unique name in the filesystem called a path.

Absolute Paths

An absolute path name starts with the root directory and separates the subsequent subdirectories with forward slashes.

- Ex: /Users/YourName/Documents/file.Rmd

Relative Paths

Relative pathnames begin at some working directory and separates the subsequent subdirectories or containing directories with forward slashes.

- Use ../ to indicate that you are exiting your current folder and retreating to the folder that contains your current folder.

14) Unix and Bash Basics

- *Graphical User Interface (GUI)* : how Mac and Windows interact with the Operating System
- we use a *Command-Line Interface (CLI)*: no mouse involved, just keyboard. Type commands into the CLI.
- Unix: Operating System
- versions of Unix: Mac OS X, Linux, etc.

Kernel*: core of Unix, determines how time and memory are allocated to programs* **Shell**: the outer layer of Unix, what the user interacts with (what you see in Terminal) i.e. bash *__Terminal__: program that opens a window and lets you interact with the shell

BASH: Bourne Again Shell – the most common type of shell

- use `touch` in terminal to create a file
- use `mkdir` to create a directory
- use `ls` to list the contents of a directory
- use `cd` to change to home directory
- use `cd ..` to change to parent directory
- use `cp` to copy a file to a different location
- use `mv` to move a file
- use `rm` to remove a rile

15) Git

- Git is a *Version Control System (VCS)*
- Git records the changes made on a project's files via "snapshots"

- `git init` to initialize a repository

Run the following to tell Git to track recent changes:

- `git add`
- `git status`
- `git commit -m "add file"`

Create a new repository:

- You can do this in github manually

Establish a connection with repository

- `git remote add origin _URL_`

Push commits

- `git push origin master`

Pull commits

- `git pull -u origin master`

Clone a remote

- `git clone _URL_`

16) Miscellaneous

- `set.seed()` helps avoid randomness when sampling
- **`Sink()`** : used to export an output to a desired file

Here's an example of what the code should look like.

```
sink(file = '../output/summary-height-weight.txt')
summary(dat[ ,c('height', 'weight')])
sink()
```

`sample()` naturally samples without replacement.

> *SHINY WEB APP*: * To create one – 1) Open RStudio. 2) Go to the File option from the menu bar. 3) Select New File and choose Shiny Web App. 4) Give a name to your App, choose a location for it, and click the Create button. * By default, shiny creates a basic template with a histogram of the variable waiting from the data set faithful. You can try running the app by clicking on the Run App button (see buttons at the top of the source pane). * Review the shiny tutorial from Gaston Sanchez's slides

Special Data Values

- NULL = null object
- NA = Not Available (missing value)
- Inf = positive infinite
- -Inf = negative infinite
- NaN = Not a Number (different from NA)

RStudio Basics

Source: this is where you write your code. Your code is not "ran" until you select "run" so that they are put into the console for evaluation.

Console: this is where your source code is ran by R. You can also import packages here and do quick computations that you don't want to save.

Environment, History, etc: This is where you can view (in an organized manner) your command history (History) and what objects and variables are in use by R (Environment).

Files, Plots, Packages, Help, Viewer: this is where you look for help/assistance with R, where you can view your file directories, where you can see which packages are available and which you are running, and also where you can view graphs and plots.

Packages (and Syntax)

- To insert packages:
- install.packages("")
- library()
- Packages we have used:
- **knitr**
- **readr** : `read_table()`, `read_csv()`
- `read_table()` is stricter than read.table() and requires that lines be of the same length
- Example: read_csv("nba2018.csv", col_names = TRUE, col_types = "ccccciccccdiiiiiiiidiidiiddiidiiiiiiiiii")
- **dplyr** : `filter()`, `select()`, `slice()`, `mutate()`, `arrange()`, `summarise()`, `group_by()`
- **ggplot2** : used to generate detailed plots
- **stringr** : for working with strings as easily as possible
- **plotly** : produces graphics in html form
- **graphics** : used to generate plots, `plot()` function
- **grid** : used to build graphics packages like ggplot2

Help functions

- ?FunctionName
- help.search("Topic you want to know about")
- rm(list = ls()) clears your environment