Introduction to R

Business Analytics (WS 20/21)

What is …?

- R
    - a language and environment for statistical computing and graphics
    - Free and Open Source, developed by nonprofit R Foundation
    - Vast package ecosystem, most complete for statistics, competitive for ML.
    - *Version: >= 4.0.3*
    - Main package family we'll use: \`tidyverse\`, a coherent and opinionated system of packages for data analysis. (*Version >= 1.3.0)*
- R Studio
    - RStudio is an integrated development environment (IDE) for R.
    - Free (+ commercial enterprise features) and Open Source by company of same name
    - *Version: >= 1.3.1093*

- Focus in this course: Using R for data analysis
- *Not* teaching R as a programming language

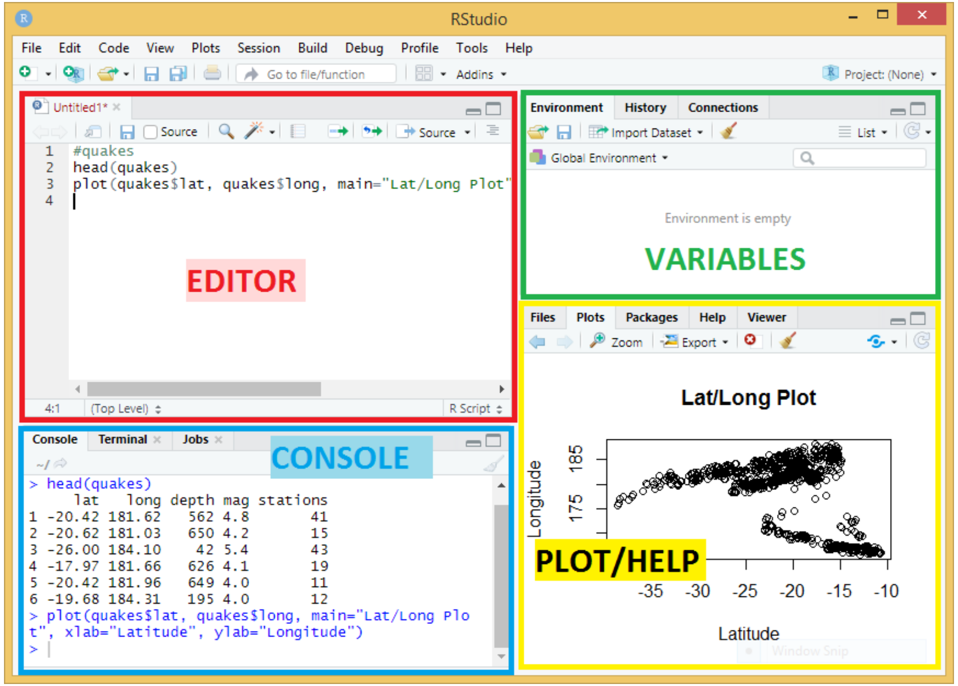
Further Resources

- Built-in tutorials in R/Rstudio (packages `learnr` or `swirl`)

- *R for Data Science* by Hadley Wickham et al. https://r4ds.had.co.nz/

R Studio Interface

- Editor / Source
  - This is the where you write code.
- Console
  - This is the console where you can run the commands and see the output.
- Environment
  - This window lists the variables in the current environment.
- Plots, Help
  - This window displays the plots or help pages.



Using the console

- One can run instructions directly from the console.
  - Type in a command and press “Enter” to see the result.
- Basic Mathematical Operations
  - You can type an arithmetic equation in the console and test it.
- Colon Operator (:)
  - This operation generates regular sequences.

```
> 1 + 4                        # commands can be run directly from the console.
[1] 5
> 2 + 6 * 5                    # comments are written using #
[1] 32
> 1:6                          # colon operator generates a sequence. In this case,
numbers from 1 to 6
[1] 1 2 3 4 5 6
```

R Data Types

- **numeric**: 5, 25.5
- **integer**: 20L, 35L (L is a way to tell R to store it as an integer)
- **character**: “a”, ‘IN2028’, “single and double ticks are equivalent”
- **logical**: TRUE, FALSE
- **(factor:** holds categorical or ordinal data, internally stored as integers)
- **complex**: 3+4i, -21+22i (complex numbers)
- **raw**: to hold raw bytes

Variables

- R lets you save data by storing it inside a variable.
- Naming - Name cannot start with a number and cannot use some special symbols, like ^, !, $, @, +, -, /, or *

```
> s <- 42                   # assign value 42 to variable s
> b <- s * s                # calculate and assign the value to b
> b                         # print the value of variable b
[1] 1764
> s <- "Hello World!"       # assign a string to s
> s                         # previous instruction overwrites the value of 42,
                              let's print it
[1] "Hello World!"
> s = "Hello World!"        # can also use the = symbol
> f = 17.42                 # assign a float
> f * f                     # evaluating a term without assigning prints the
                              result to console
[1] 303.4564
```

Functions

- To know more about a function, you can use the following. They open the help page for the functions.
  - the ? operator or
  - help() function.

```
> help("round")             # displays the help page for the function “round”
> round(3.154)              # round it to an integer
[1] 3
> round(3.154, digits=2)    # round it to 2 decimal digits, 2 is an argument to
                              the function
[1] 3.15
> ?factorial                # this works similarly as the help() function
> factorial(4)
[1] 24
```

if-else Statement

- Logical expressions use a java-like syntax.

```
> num <- -4
> if (num < 0) {                # if value of num is less than 0
+     num <- num * -1           # do this
+ }
> num
[1] 4
> if (num %% 2) {               # check if num is odd or even
+     "ODD"                     # do this if odd
+ } else {                      # else run the below code
+     "EVEN"
+ }
[1] "EVEN"
```

Looping

- Loops can be used to iterate over code many times.
- Loops can be coded with *for* and *while* statements.

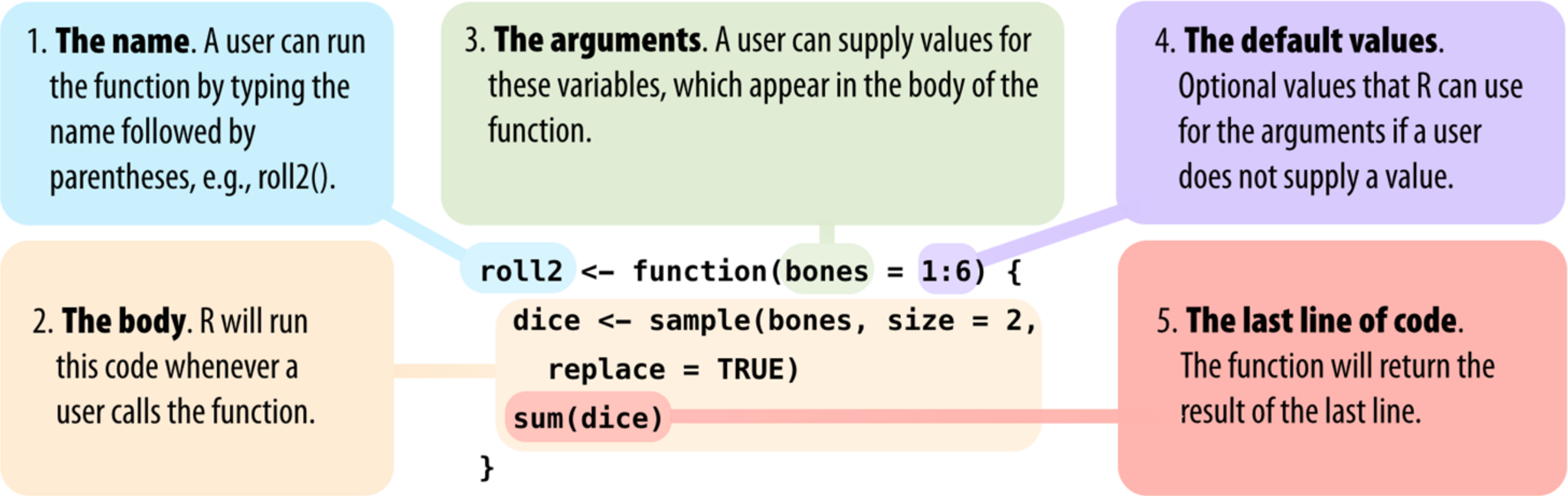
```
> for (i in 11:20) {                                           # for loop
+    print(i)
+ }


> i <- 1
> while (i < 10) {                                             # while loop
+    print(i)
+    i <- i+1
+ }
```

Writing your own functions

- Every function in R has three basic parts: a name, a body of code, and a set of arguments.
- Function returns the result of the last line of code. If the last line of code doesn't return a value, neither will the function.

```
> my_function <- function() {        # my_function is the function name
+     s <- 5
+     s * s                          # last line of the function body
+ }
> my_function()                      # calling the function
[1] 25
> my_square <- function(a) { a*a }   # a is the argument for this function
> my_square(5)                       # 5 is the value passed as argument
[1] 25
```

Function Anatomy



Source: https://rstudio-education.github.io/hopr/basics.html#write-functions

Vectors

- are comparable to arrays or lists in Java.
- can easily be addressed and extended.
- Indices in R start at 1, not at 0!

```
> v1 = c(3, 5, 2)                 # create vector, c: combine
> v1[1]                           # address elements of vector
[1] 3
> v1 = c(v1, 8, 3, 7, 5)          # extend vector
> v1[1:3]                         # address several elements
[1] 3 5 2
> v1 = v1 * 2                     # multiply by a scalar
> v1[2] = 14                      # set second value
> v2 = seq(from=0, to=12, by=2)   # sequence from 0 to 12 in steps of 2
> v2
[1] 0 2 4 6 8 10 12
> v3 = v1 + v2                    # vector addition
> v3
[1]  6 16  8 22 14 24 22
```

Matrices

- Matrices store values in a two-dimensional array.
- Can be easily addressed.

```
> m <- matrix(c(9,2,5,3,1,8), ncol=3, nrow=2)  # create matrix
> m                                            # print matrix
     [,1] [,2] [,3]
[1,]    9    5    1
[2,]    2    3    8
> m[1,2]                                       # address element in first row
and second column
[1] 5
> m[2,]                                        # address whole row
[1] 2 3 8
> m[2,3] = 14                                  # set specific value
> dim(m)                                       # dimensions of matrix m
[1] 2 3
```

Factors

- Store categorical information like gender or eye color.
- Can only take certain values, which may have their own idiosyncratic order.
- Stored as a vector of integers with a corresponding map of character names (“levels”).
- Historically widely used to save RAM, less important today
  (But you will still encounter and use them sometimes!)

```
> eye_color <- factor(c("blue", "brown", "black", "green", "brown", "blue"))
# create a factor
> eye_color
[1] blue  brown black green brown blue
Levels: black blue brown green
> unclass(eye_color)                       # show how R stores this factor
[1] 2 3 1 4 3 2
attr(,"levels")
[1] "black" "blue"  "brown" "green"
```

Data Frames (1/2)

- Data frames group vectors together into a two-dimensional table. Each vector becomes a column in the table.
- Data stored in a data frame can be of numeric, factor or character type.

```
> n <- c("Alice", "Bob", "Charlie")
> a <- c(25, 27, 23)
> u <- factor(c("TUM", "LMU", "TUM"))
> df <- data.frame(names=n, age=a, uni=u)   # create a data frame with column
                              # names - names, age and uni and corresponding data.
> df
    names age uni
1   Alice  25 TUM
2     Bob  27 LMU
3 Charlie  23 TUM
> df[3]                                          # address third column
> df$uni                                         # address column by name
```

Data Frames (2/2)

```
> df[c(1,3)]                  # select columns 1 and 3
> df$by = 2020 - df$age       # create new column for birth year
> df$by
[1] 1995 1993 1997
> df$age = NULL               # remove column age
> df[df$by < 1994, ]          # filter rows
  names uni   by
2   Bob LMU 1992
> df[order(df$by),]           # sort in order with column “by”
    names uni   by
2     Bob LMU 1993
1   Alice TUM 1994
3 Charlie TUM 1997
```

Tibbles

- Part of *tidyverse* package, can be considered as enhanced data frames
- Some differences to base R: Printing, subsetting, no partial matching (+more)

```
> tib_data <- as_tibble(df)          # convert dataframe df to tibble
> tib_data                           # view the tibble
> df$b                               # partial matching in dataframe
[1] 1994 1992 1996
> tib_data$b                         # no partial matching
NULL
Warning message:
Unknown or uninitialised column: 'b'.
> tib_data[["uni"]]                  # similar to tib_data$uni
> tib_data[[2]]                      # similar to tib_data$uni
```

Data Import & Export

- *readr*, a *tidyverse* package, can be used to import and export data.
- *read_csv()* and *read_delim()* are some of the basic data import functions provided by *readr*.
- *readr* functions guess the types of each column and convert types when appropriate and return tibbles automatically.
- To be able to use *tidyverse* packages, load the library using *library(tidyverse).*

```
> getwd()                                     # get working directory path
[1] "/Users/YourUserName”
> setwd("~/Documents/")                       # set path to required directory

> testdata <- read_csv("input_data.csv")     # read csv
> write_csv(testdata, "/Users/YourUserName/output_data.csv")   # write data to
csv file
```

Sample Data - iris

```
> data <- as_tibble(iris) # convert iris data into a tibble
> data
# A tibble: 150 x 5
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
          <dbl>       <dbl>        <dbl>       <dbl> <fct>
 1          5.1         3.5          1.4         0.2 setosa
 2          4.9         3            1.4         0.2 setosa
                        3.2          1.3         0.2 setosa
                                 ... ...
```

Image Source: http://de.wikipedia.org/wiki/Portal:Statistik/Datensaetze#mediaviewer/File:IMG_7911-Iris_virginica.jpg

Tidy Data

When working with tabular data in R, we will always follow the following format:

- **Each variable forms a column.** *(Sepal.Length etc. each are variables describing iris)*
- **Each observation forms a row.** (*Every row refers to one specimen of an iris flower.)*
- **Each type of observational unit forms a table.** *(No row describes anything other than an iris.)*

See ‘Tidy Data’ by Hadley Wickham (2013) for more information https://vita.had.co.nz/papers/tidy-data.pdf

Data Exploration Overview (1/2)

- *glimpse()* function shows the details of the dataset.
- *summary()* function shows statistics per column of the dataset.

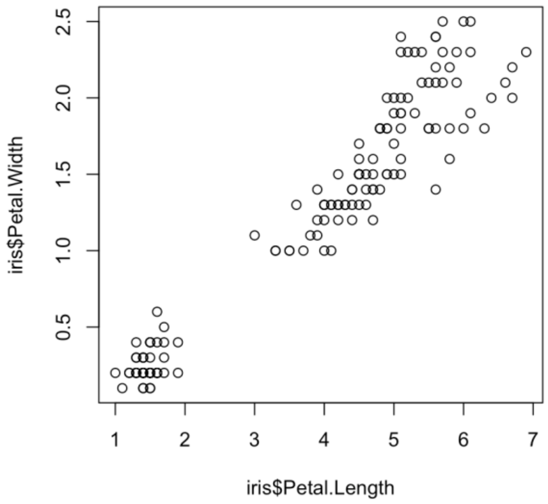
```
> glimpse(data)                # structure of data set
Observations: 150
Variables: 5
$ Sepal.Length <dbl> 5.1, 4.9, 4.7, 4.6, 5.0, 5.4, 4.6, 5.0, 4.4, 4.9, ...
$ Sepal.Width  <dbl> 3.5, 3.0, 3.2, 3.1, 3.6, 3.9, 3.4, 3.4, 2.9, 3.1, ...
$ Petal.Length <dbl> 1.4, 1.4, 1.3, 1.5, 1.4, 1.7, 1.4, 1.5, 1.4, 1.5, ...
$ Petal.Width  <dbl> 0.2, 0.2, 0.2, 0.2, 0.2, 0.4, 0.3, 0.2, 0.2, 0.1, ...
$ Species      <fct> setosa, setosa, setosa, setosa, setosa, setosa, set…
> summary(iris)                # summary of data
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width          Species
 Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100   setosa    :50
                                                             ...
```

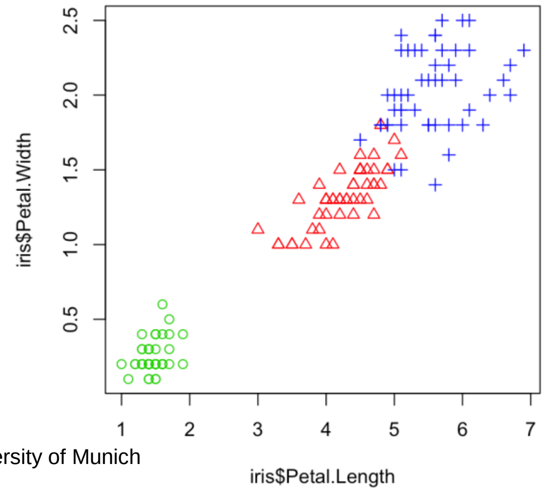
Data Exploration Overview (2/2)

```
> names(data)                 # attribute/column names
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species”
> ncol(data)                  # number of columns(attributes)
[1] 5
> nrow(data)                  # number of rows observations)
[1] 150
> dim(data)                   # dimensions (#rows and #columns)
[1] 150   5
> head(data)                  # returns the first few observations from the data
> tail(data)                  # returns the last few observations from the data
```

Plotting (Base R) (1/3)

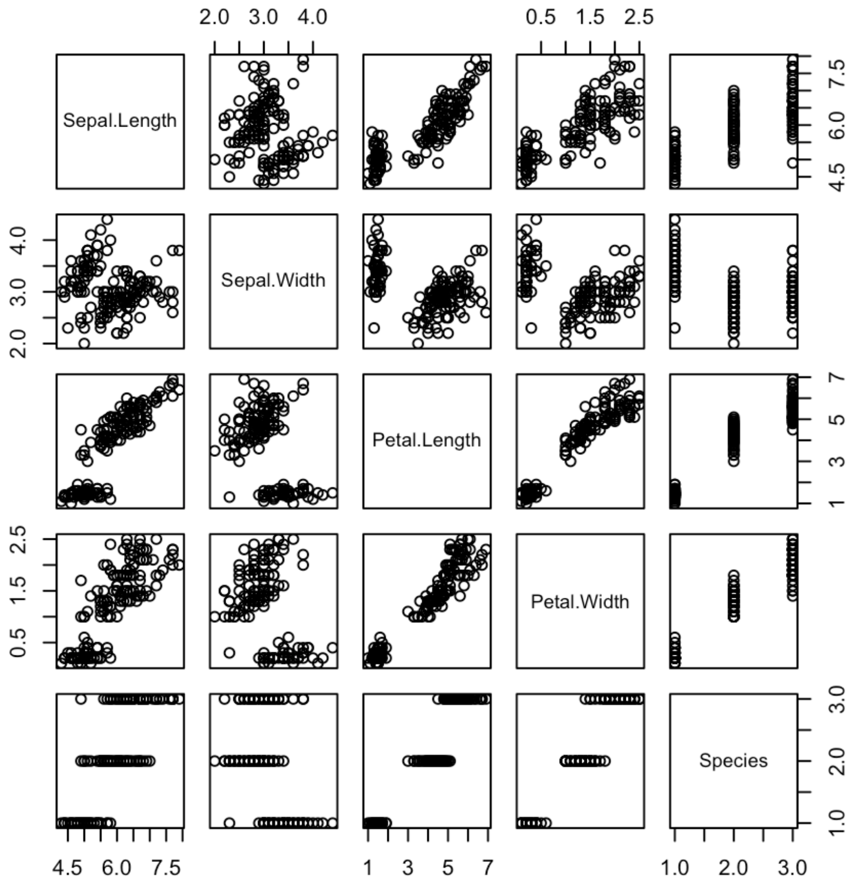
```
# Plot the relationship between petal length
and width
> plot(iris$Petal.Length, iris$Petal.Width)

# Plot with different colors for species
> plot(iris$Petal.Length, iris$Petal.Width,
       pch = as.numeric(iris$Species),
       col=c("green3", "red", "blue"
             )[as.numeric(iris$Species)]
       )

> cor(iris$Petal.Length, iris$Petal.Width)
# [1] 0.9628654
```





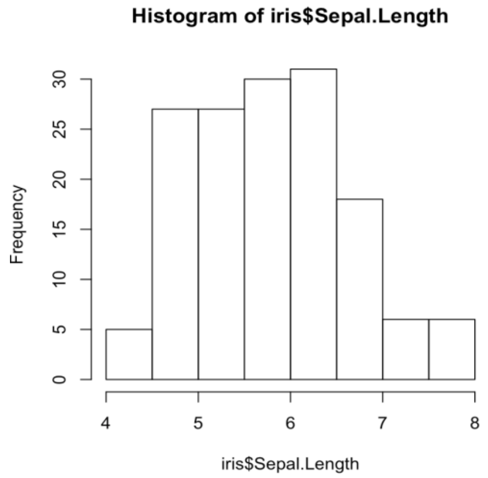
Plotting (Base R) (2/3)

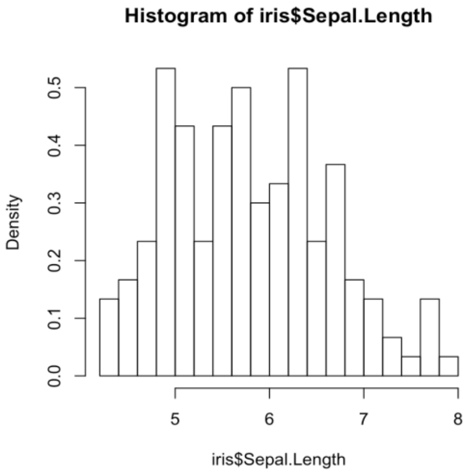
```
> pairs(iris)

# compare to the correlation matrix
> cor(iris[1:4])
```



Plotting (Base R) (3/3)

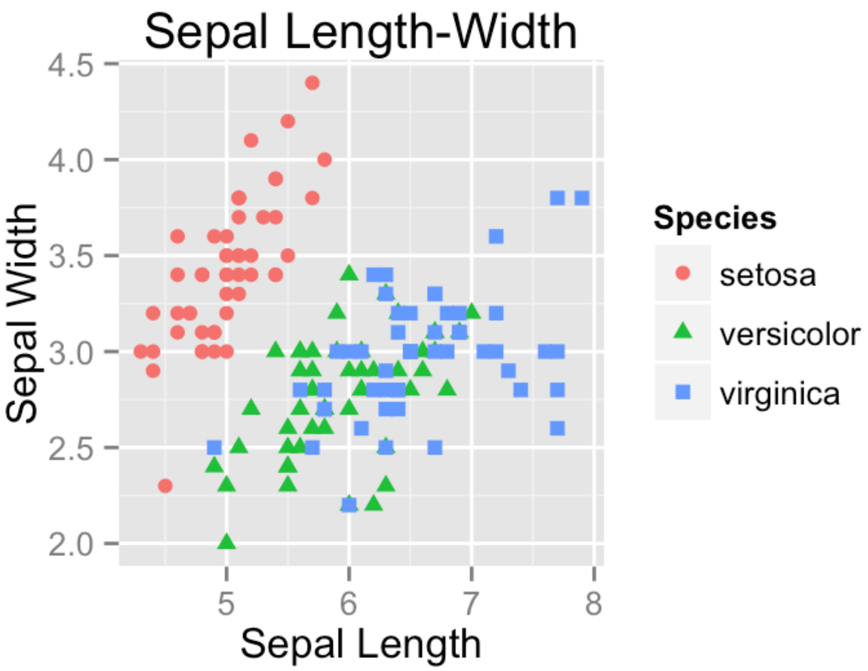
```
> hist(iris$Sepal.Length)  # histogram

> hist(iris$Sepal.Length, breaks=20,
freq=FALSE)
# histogram with 20 breaks and density
```





Plotting with ggplot2

```
# An advanced plotting tidyverse package
# that allows you to create (almost) any
# plot you can think of
> library(ggplot2)
> scatter <- ggplot(
    data=data,
    aes(x = Sepal.Length,
        y = Sepal.Width))
> scatter +
    geom_point(aes(color=Species,
                   shape=Species)) +
    xlab("Sepal Length") +
    ylab("Sepal Width") +
    ggtitle("Sepal Length-Width")
```



You are ***not*** expected to learn how to read or write ggplot2 code in this course, but it's a very useful tool for data exploration. Details are taught in Module IN2339. More sample ggplot2 code at https://www.mailman.columbia.edu/sites/default/files/media/fdawg_ggplot2.html

Descriptive Statistics

```
> mean(data$Sepal.Length)                          # calculate mean of the column
[1] 5.843333
> var(data$Sepal.Length)                           # calculate variance
[1] 0.6856935
> sd(data$Sepal.Length)                            # calculate standard deviation
[1] 0.8280661
> cov(data$Petal.Length, data$Petal.Width)         # calculate covariance
[1] 1.295609
> cor(data$Petal.Length, data$Petal.Width)         # calculate correlation
[1] 0.9628654
> cor(data[1:4])                                   # calculate correlation matrix
             Sepal.Length Sepal.Width Petal.Length Petal.Width
Sepal.Length    1.0000000  -0.1175698    0.8717538   0.8179411
Sepal.Width    -0.1175698   1.0000000   -0.4284401  -0.3661259
Petal.Length    0.8717538  -0.4284401    1.0000000   0.9628654
Petal.Width     0.8179411  -0.3661259    0.9628654   1.0000000
```

dplyr

- *dplyr* is a *tidyverse* package which provides tools for data manipulation. Some of the functions are
    - *filter()* to select rows based on their values.
    - *arrange()* to reorder the rows.
    - *select()* to select columns.
    - *mutate()* to add new variables that are functions of existing variables.
    - *summarise()* to condense multiple values to a single value.
    - *group_by()* can be used to apply the above on specified groups of dataset.
- Each of the above functions are similar
    - The first argument is a data frame.
    - Subsequent arguments describe what to do with the data frame.
    - The result is a new data frame.

dplyr - filter()

- *filter()* allows you to subset observations based on their values.

```
> data <- as_tibble(quakes)          # we will work with quakes dataset
> ?quakes                            # this will show the details of the dataset
> nrow(data)                         # number of observations in quakes
[1] 1000
> d <- filter(data, mag > 5, stations > 20)   # filter all earthquakes with
                                     # magnitude greater than 5 and
                                     # reported by more than 20 stations
> nrow(d)                            # number of filtered observations
[1] 149
> d <- filter(data, mag > 6 | stations > 60)  # magnitude greater than 6 or
                                     #stations reporting is greater than 60
> nrow(d)
[1] 124
```

dplyr - arrange()

- *arrange()* takes a data frame and a set of column names (or more complicated expressions) to order by.
- If you provide more than one column name, each additional column will be used to break ties in the values of preceding columns.

```
> d <- quakes[1:4,]                    # let us work with a subset of the data
> d                                    # view the subset
> arrange(d, desc(mag), stations)      # arrange the rows in descending order of
                                       # “mag” and, in case of ties, ascending
                                       # order of “stations”
```

dplyr - select()

- *select()* allows you to choose columns.
- *rename()* allows you to rename columns.

```
> data <- as_tibble(quakes)                     # we will work with quakes dataset
> glimpse(data)                                 # display the structure of quakes
> glimpse(select(data, lat, long))              # select columns lat and long
> glimpse(select(data, lat:depth))              # select columns from lat until depth
> glimpse(select(data, -(lat:depth)))           # select columns other than columns
                                                #from lat until depth
> glimpse(select(data, starts_with("l")))        # select all columns starting
                                                # with "l". Other helpers can be found
                                                # in ?select
> glimpse(rename(data, latitude=lat, longitude=long)) # rename the columns
```

dplyr - mutate()

- *mutate()* adds new columns to your dataset or modifies values in existing ones.
- *transmute()* is similar to *mutate()* but it keeps only the new columns.

```
> d <- quakes[1:4,]                # let us use a subset of the dataset
> d                                # view the data
     lat   long depth mag stations
1 -20.42 181.62   562 4.8       41
2 -20.62 181.03   650 4.2       15
3 -26.00 184.10    42 5.4       43
4 -17.97 181.66   626 4.1       19
> mutate(d, height = 0-depth)   # add a new column height
     lat   long depth mag stations height
1 -20.42 181.62   562 4.8       41   -562
2 -20.62 181.03   650 4.2       15   -650
3 -26.00 184.10    42 5.4       43    -42
4 -17.97 181.66   626 4.1       19   -626
```

dplyr - summarise()

- *summarise()* collapses the data to a single row. (summary statistics)
- Frequently used with *group_by()* (which we will look into next)

```
> summarise(quakes, mean_mag=mean(mag), median_stations=median(stations))
# compute the mean and median of mag and stations columns respectively
  mean_mag median_stations
1   4.6204              27
> summarise(quakes, sd_mag=sd(mag), min_stations=min(stations))
# compute sd and min of mag and stations columns respectively
    sd_mag min_stations
1 0.402773           10
# works with any function that takes a vector and returns a scalar.
# some possible functions like mean, min etc are listed at ?summarise
# To summarise many columns at once, see ?summarise_at()
```

dplyr - group_by()

- Converts data to grouped data on which operations can be performed on each group.

```
> grouped <- group_by(quakes, mag)                          # group the data by mag
> summarise(grouped, mean_stations=mean(stations))# compute the mean of
                                                      # stations column for every group
# A tibble: 22 x 2
      mag mean_stations
    <dbl>         <dbl>
 1    4            14.9
 2    4.1          15.7
 3    4.2          18.4
 4    4.3          19.3
 5    4.4          22.3
# ... with 17 more rows
```

dplyr - piping (1/2)

- Piping makes the code more readable
- For a large set of consecutive transformations to the data, the code can become confusing since we have to either
  - Assign intermediate results to a variable
  - Nest the different functions
- Usage of the pipe is using the symbol ***%>%*** *(Hotkey in RStudio: **Ctrl + Shift + M**)*

```
  x %>% f(y)       is the same as   f(x, y)
y %>% f(x, ., z)   is the same as   f(x, y, z )
```

Source: RStudio Data Wrangling with dplyr and tidyr Cheat Sheet
https://rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf

dplyr - piping (2/2)

```
> # Method 1 - Storing intermediate values.
> temp <- filter(quakes, mag > 5)
> result <- summarise(temp, mean_stations=mean(stations))
> result
  mean_stations
1      72.23179
```

```
> # Method 2 - Nesting functions.
> summarise(filter(quakes, mag > 5), mean_stations=mean(stations))
  mean_stations
1      72.23179
```

```
> # Method 3 - Using pipe operator
> quakes %>% filter(mag > 5) %>% summarise(mean_stations=mean(stations))
  mean_stations
1      72.23179
```