R Programming: An Introduction

Stacey Borrego

10/27/2022

Helpful Links

- Data Carpentry: Data Analysis and Visualization in R for Ecologists
- Tidyverse: R packages for Data Science
- R for Data Science by Hadley Wickham and Garret Grolemund
- Advanced R by Hadley Wickham
- ggplot2: Elegant graphics for data analysis by Hadley Wickham
- R Studio Cheatsheets by RStudio
- The R Gallery by Kyle W. Brown

Simple math

In the **CONSOLE** section of Rstudio, type in the following expressions.
Hit `ENTER` after each expression.

```
3 + 5
12 / 7
3 * 4
8 ^ 2
log(8)
3 + 5 / 2
(3 + 5) / 2
```

Creating objects in R

Working with variables/objects

Assign values to a variable/object

```
weight_kg <- 55
```

- Assignment operator (<-)
- Cannot start with numbers
- Case sensitive
- Avoid function names

- Avoid dots (.)
- Be consistent in the styling of your code
  - R Style Guide
  - Styles can include "lower_snake", "UPPER_SNAKE", "lowerCamelCase", "UpperCamelCase", etc
  - We will use "lower_snake" for readibility during this workshop

Print the value of an object

```
weight_kg
```

Simple math with objects

```
2.2 * weight_kg
```

Reassign object's value

```
weight_kg

weight_kg <- 65
weight_kg

2.2 * weight_kg
```

Assign a new object a value containing another object

```
weight_lb <- 2.2 * weight_kg
weight_lb
```

Saving code in a script

- New script
  - File > New File > R Script (`Ctrl` + `Shift` + `N`)

- Save the file
  - File > Save (`Ctrl` + `S`)

Running code in a script

- Select `Run` button at the top of the script window
- Select code or move cursor to the line to be run and type `Ctrl` + `ENTER`

```
2 + 2
```

Add comments in a script

- The comment character in R is `#`. Anything to the right of a `#` in a script will be ignored.
- Comment out a section by selecting and pressing `Ctrl` + `Shift` + `C`
  - press the same keys again to remove the comment

```
# This is a comment.
# The sum of 2 + 2
2 + 2
```

Functions and arguments

- Functions are scripts that automate more commands
- Functions are available in base R, by importing packages, or by making them yourself
- Usually require one or more inputs called **arguments**
- Functions typically return a **value**

```
# This is a function call
sqrt(10)
weight_kg <- sqrt(10)
weight_kg
```

- Information about functions can be found in the `Help` window by typing `?` and the function name. Example: `?round`
- You can also use the function `args()` to see the arguments of a function.

```
round(3.14159)

?round
args(round)

round(3.14159, digits = 2)
```

Vectors and data types

- A vector is composed by a series of values, which can be either numbers or characters.
  - Quotes must surround characters.
  - In a vector, all of the elements are the same type of data.
- We can assign a series of values to a vector using the `c()` function.

```
# Numerical vector
weight_g <- c(50, 60, 65, 82)
weight_g

# Character vector
animals <- c("mouse", "rat", "dog")
animals
```

Explore the content of vector

- `length()` tells you how many elements are in a particular vector
- `class()` indicates what kind of object you are working with
- `str()` provides an overview of the structure of an object and its elements.

```
# Length
length(weight_g)
length(animals)

# Class
class(weight_g)
class(animals)

# Structure
str(weight_g)
str(animals)
```

Add elements to a vector

```
# Add to the end of the vector
weight_g <- c(weight_g, 90)

# Add to the beginning of the vector
weight_g <- c(30, weight_g)

# Inspect the modified vector
weight_g
```

Data Types

An atomic vector is the simplest R data type and is a linear vector of a single type.

There are four common types of atomic vectors: character, numeric or double, logical, and integer. There are two rare types that will not be discussed: complex and raw.

- `character`
- `numeric` or `double`
- `logical` for TRUE and FALSE (the boolean data type)
- `integer` for integer numbers (e.g., 2L, the L indicates to R that it's an integer)

- `complex` to represent complex numbers with real and imaginary parts (e.g., 1 + 4i)
- `raw` for bitstreams

```
typeof(weight_g)
typeof(animals)
```

Data Structures

Vectors are one of the many data structures that R uses.

- lists (`list`)
- matrices (`matrix`)
- data frames (`data.frame`)
- factors (`factor`)
- arrays (`array`)

Coercion

All elements of an atomic vector must be the same type, so when you attempt to combine different types they will be coerced to the most flexible type. Types from least to most flexible are: logical, integer, double, and character.

```
num_char <- c(1, 2, 3, "a")
typeof(num_char)
str(num_char)

num_logical <- c(1, 2, 3, TRUE, FALSE)
typeof(num_logical)
str(num_logical)

char_logical <- c("a", "b", "c", TRUE)
typeof(char_logical)
str(char_logical)

tricky <- c(1, 2, 3, "4")
typeof(tricky)
str(tricky)
```

Conditional subsetting

To extract one or several values from a vector, provide one or several indices in square brackets.

```
animals <- c("mouse", "rat", "dog", "cat")

animals[2]

animals[c(3, 1)]

animals[1:3]

more_animals <- animals[c(1, 4, 1, 3, 1, 2)]
more_animals
```

Subset by using a logical vector. TRUE will select the element with the same index, while FALSE will not.

```
weight_g <- c(21, 34, 55)

weight_g[c(TRUE, FALSE, TRUE)]

weight_g > 50

weight_g[weight_g > 50]
```

Combine multiple tests using `&` (both conditions are true, **AND**) or `|` (at least one of the conditions is true, **OR**).

Helpful operators

- `>` greater than
- `<` less than
- `<=` less than or equal to
- `==` equal to, test for numerical equality between the left and right hand sides

```
weight_g

weight_g[weight_g > 20 & weight_g < 30]

weight_g[weight_g <= 30 | weight_g == 55]

weight_g[weight_g >= 30 & weight_g == 21]
```

Search for strings in a vector

```
animals <- c("dog", "rat", "cat", "cat")
animals

animals[animals == "cat" | animals == "rat"]

animals %in% c("rat", "cat", "duck", "fish")

animals[animals %in% c("rat", "cat", "duck", "fish")]
```

Missing data

- Missing data are represented in vectors as NA.
- You can add the argument `na.rm = TRUE` to calculate the result as if the missing values were removed (rm stands for **rem**ove)
- `is.na()` identifies missing elements
- `na.omit()` returns the object with incomplete cases removed
- `complete.cases()` returns a logical vector indicating which cases are complete

```r
heights <- c(1, 2, 3, NA, 5, NA)

# Performing an operation on a vector with NA will usually return NA
mean(heights)
max(heights)

# Using `na.rm = TRUE` will exclude NA values and perform the operation
mean(heights, na.rm = TRUE)
max(heights, na.rm = TRUE)
```

```r
# Returns vector of boolean values indicating if the value NA or not
is.na(heights)

# Sum of all the NAs present
sum(is.na(heights))

# Identifies the index containing the NA value
which(is.na(heights))

# Returns the TRUE values for `is.na()`
heights[is.na(heights)]

# Returns the FALSE values for `is.na()`
heights[!is.na(heights)]

# Returns values with incomplete cases (NA) removed
na.omit(heights)
mean(na.omit(heights))

# Returns values with complete cases only - NA removed
complete.cases(heights)
heights[complete.cases(heights)]
```

Data Frames

When we load data into R, it may be stored as an object of class data frame.

A data frame is the representation of data in the format of a table where the columns are vectors that all have the same length. Because columns are vectors, each column must contain a single type of data (e.g., characters, integers, factors, logical values).

For example, here is an example data frame comprising a numeric, character, and logical vector.

```
df <- data.frame(numeric = c(1, 3, 5, 7, 9, 11),
                 character = c("F", "O", "O", "B", "A", "R"),
                 logical = c(TRUE, FALSE, TRUE, FALSE, FALSE, FALSE))
df
```

```
##   numeric character logical
## 1       1         F    TRUE
## 2       3         O   FALSE
## 3       5         O    TRUE
## 4       7         B   FALSE
## 5       9         A   FALSE
## 6      11         R   FALSE
```

Data frames can also be subset using square brackets as demonstrated with the single vectors or using `$` to specifically name a column.

Using single square brackets will return a data frame containing the values of the vector, whereas using double square brackets or `$` will return a simplified vector of the type of class as the values contained with in it.

```
# Column 1 data frame
df[1]
```

```
##   numeric
## 1       1
## 2       3
## 3       5
## 4       7
## 5       9
## 6      11
```

```
# Column 1 simplified vector
df[[1]]
```

```
## [1]  1  3  5  7  9 11
```

```
df$numeric
```

```
## [1]  1  3  5  7  9 11
```

Data can be further subset by providing the indices for the rows and columns you want. The notation is to first name the data frame followed by single square brackets listing the rows and columns.

```
# Column 2
df[2]
```

```
##   character
## 1         F
## 2         O
## 3         O
## 4         B
## 5         A
## 6         R
```

```
# Rows 4 through 6 in column 2
df[4:6, 2]
```

```
## [1] "B" "A" "R"
```

```
# Rows 1, 5, and 4 in column 2
df[c(1, 5, 4), 2]
```

```
## [1] "F" "A" "B"
```

```
# Rows 1, 5, and 4 in all columns
df[c(1, 5, 4), ]
```

```
##   numeric character logical
## 1       1         F    TRUE
## 5       9         A   FALSE
## 4       7         B   FALSE
```

```
# Rows 1, 5, and 4 in columns 1 through 2
df[c(1, 5, 4), 1:2]
```

```
##   numeric character
## 1       1         F
## 5       9         A
## 4       7         B
```

```
# Rows 1, 5, and 4 in columns 1 and 3
df[c(1, 5, 4), c(1:3)]
```

```
##   numeric character logical
## 1       1         F    TRUE
## 5       9         A   FALSE
## 4       7         B   FALSE
```

The subset data frame can be further subset using single square brackets and the specific names of the columns. HERE is a pretty thorough article on subsetting.

```
df["character"]
```

```
##   character
## 1         F
## 2         O
## 3         O
## 4         B
## 5         A
## 6         R
```

```
df[1, "character"]
```

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

```
df[1:3, "character"]
```

```
## [1] "F" "O" "O"
```

```
df[c(1, 5, 6), c("numeric", "character")]
```

```
##   numeric character
## 1       1         F
## 5       9         A
## 6      11         R
```

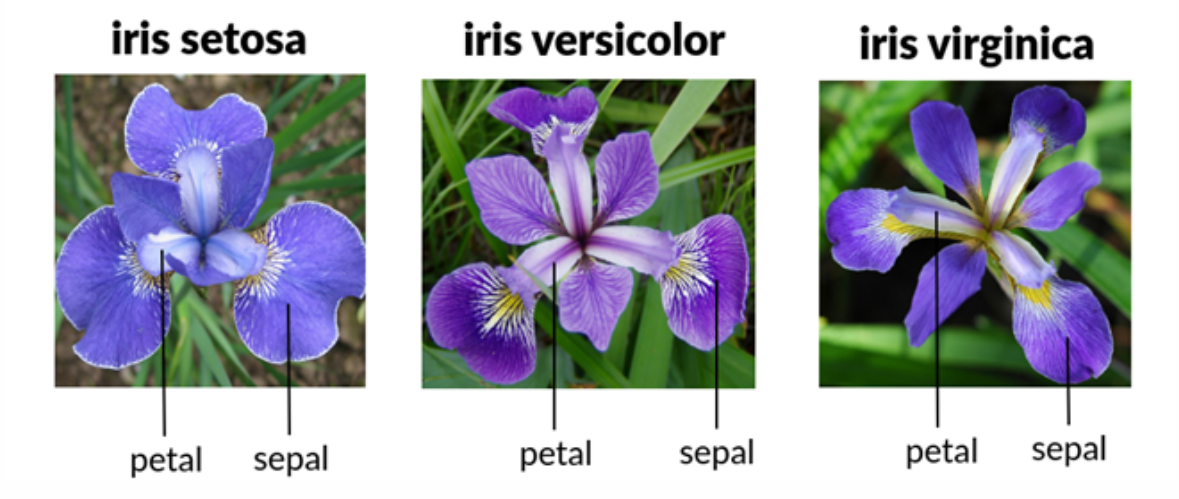


Figure 1: Understanding the Iris Dataset

Starting with Data

This is what the data will look like:

Column	Description
Sepal.Length	Sepal length in centimeters
Sepal.Width	Sepal width in centimeters
Petal.Length	Petal length in centimeters
Petal.Width	Petal width in centimeters
Species	Name of species name

```
# To see data sets available
data()

# Load the data set of choice
# use the data set name as the argument to `data()`
data(iris)
```

Open the dataset in RStudio's Data Viewer

```
# Check the data type of the data just loaded
class(iris)

# View the whole data set
View(iris)
```

Inspect the data

```
# See a few rows of the data
head(iris)
tail(iris)

# See a specific number of rows of the data
head(iris, n = 10)
tail(iris, n = 10)
```

- Size:
  - `dim(iris)` - returns a vector with the number of rows in the first element, and the number of columns as the second element (the dimensions of the object)
  - `nrow(iris)` - returns the number of rows
  - `ncol(iris)` - returns the number of columns
- Content:
  - `head(iris)` - shows the first 6 rows
  - `tail(iris)` - shows the last 6 rows
- Names:
  - `names(iris)` - returns the column names (synonym of colnames() for data.frame objects)
  - `rownames(iris)` - returns the row names
- Summary:
  - `str(iris)` - structure of the object and information about the class, length and content of each column
  - `summary(iris)` - summary statistics for each column

Indexing and subsetting data frames

A data frame has rows and columns in 2 dimensions. To extract specific data, specify the "coordinates" in `[ ]` indicating row numbers first followed by column numbers.

```
# dim() gives the output of the rows and columns of the object
dim(iris)

# Extract specific values by specifying row and column indices
# in the format:
# data_frame[row_index, column_index]

# Extract the first row and first column from iris
iris[1, 1]

# Extract the first row and sixth column
iris[1, 6]

# To select all columns, leave the column index blank
```

```r
# Extract the first row and all columns
iris[1, ]

# Extract all rows and the first column
iris[, 1]

# An even shorter way to select first column across all rows:
iris[1] # No comma!

# Select multiple rows or columns with vectors
# Extract the first three rows of the 1st and 2nd columns
iris[c(1, 2, 3), c(1, 2)]

# We can use the : operator to create a range and select all listed vectors
iris[1:3, 1:2]

# This is equivalent to head(iris)
iris[1:6, ]

# Subsetting with single square brackets ("[]") always returns a data frame.
# If you want a vector, use double square brackets ("[[]]")

# For instance, to get a column as a vector:
iris[[1]]
iris[["Species"]]

# To get the first value in our data frame:
iris[[1, 1]]
iris[1, "Species"]

# Data frames can be subset by calling their column names directly
# Use the $ operator with column names to return a vector
iris$Species
```

```r
# Exceptions: subsetting the whole data frame, except the first
# and second columns
iris[, -c(1, 2)]

# Exceptions: subsetting all columns of the data frame
# except for the first six rows
iris[-(7:nrow(iris)), ]
```

Plotting

Helpful links:

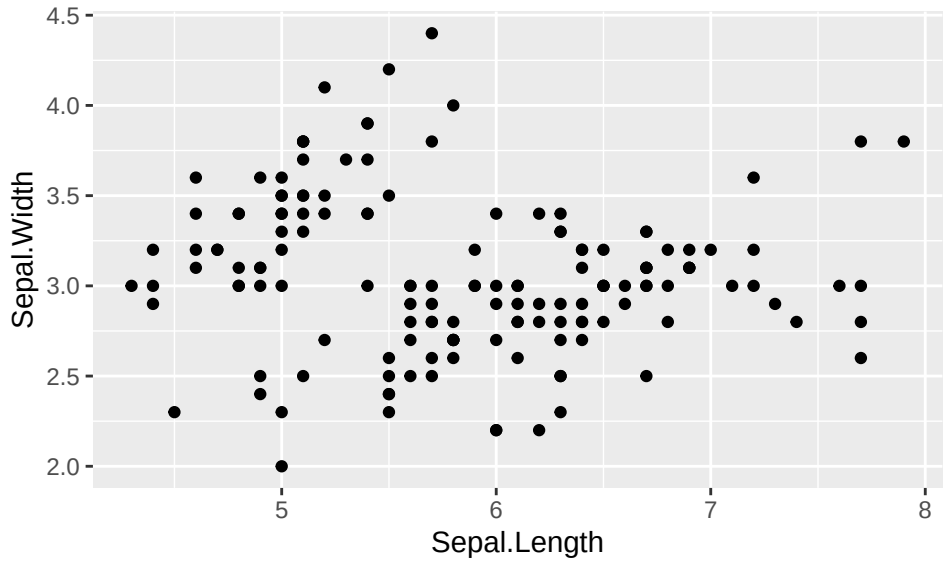
- Installation
- ggplot2: Elegant Graphics for Data Analysis

Plotting can be done in many different ways in R. Base R has several basic plotting functions which can be valuable for a quick peek at your data. However, a common and customizable option is to use the package `ggplot2`.

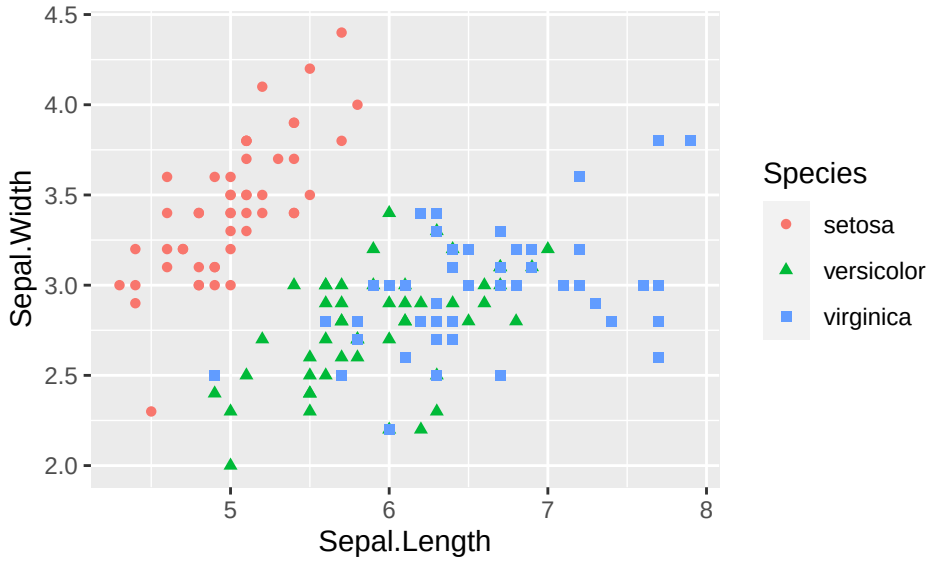
Every ggplot2 plot has three key components:

- **Data** (`data`)
- A set of **aesthetic mappings** between variables in the data and visual properties (`aes(x, y)`)
- At least one **layer** which describes how to render each observation. Layers are usually created with a geom function. It is important to note that layers are added with a `+`.

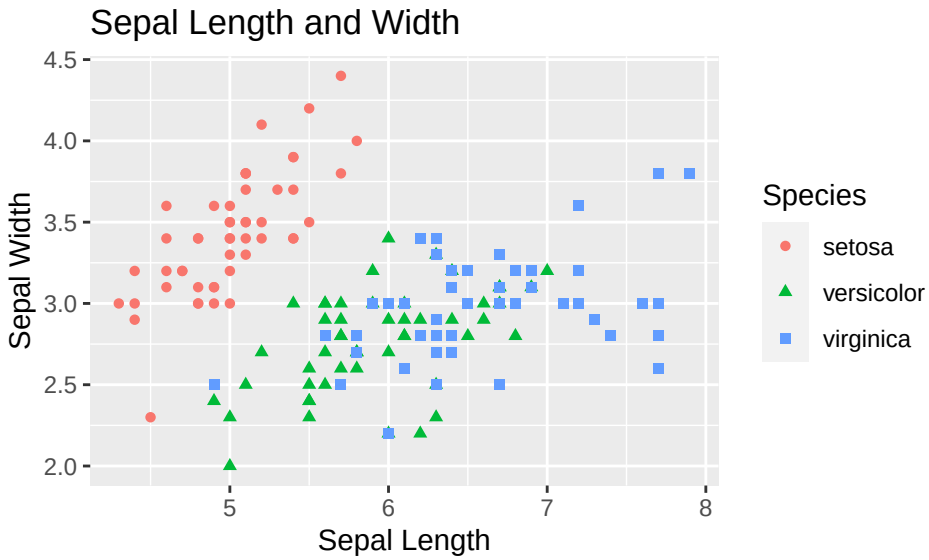
```
# Install the following library
# install.packages("ggplot2")

# Load the ggplot2 package to access its unique functions
library(ggplot2)



# Step 1: Make the plot object by indicating the input data
# and plot aesthetics
# Example of creating the plot object for a scatter plot
plot_1 <- ggplot(data = iris,
                 aes(x = Sepal.Length,
                     y = Sepal.Width))

# Step 2: Add layers to the plot object to plot the data. Here we add
# the points of the scatter plot
# Example of a basic scatter plot adding a `geom_point()` layer
plot_1 +
  geom_point()
```



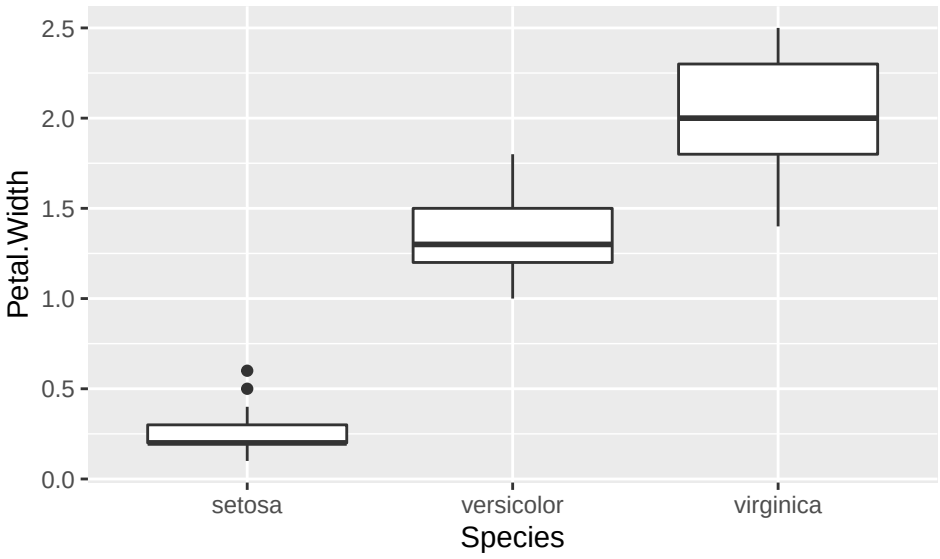

```
# Find out the options for `geom_point()`
?geom_point

# Step 3: Modify each layer using specific arguments and aesthetics options
# Example of a colorful scatter plot adding aesthetic mappings to the
# `geom_point()` layer
plot_1 +
  geom_point(aes(color = Species,
                 shape = Species))
```



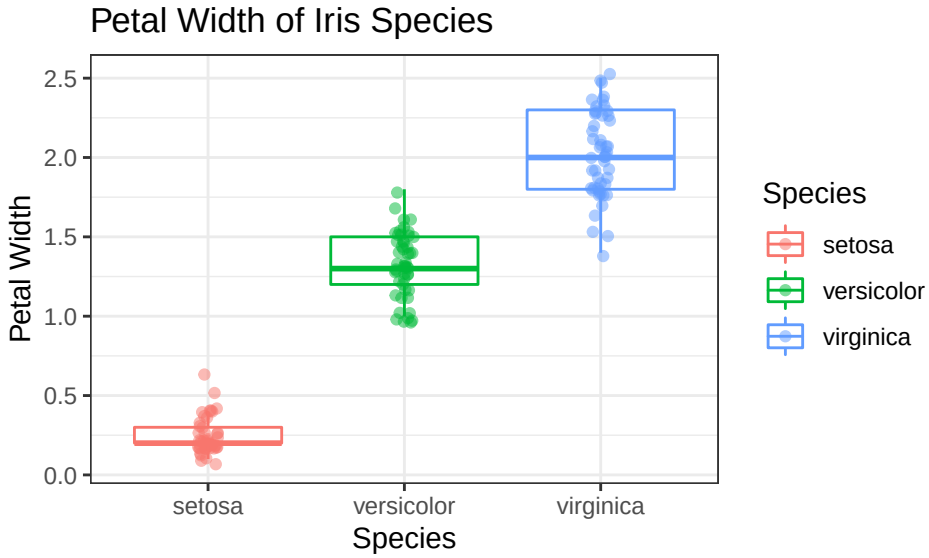
```
# Add labels using `labs()`
plot_1 +
  geom_point(aes(color = Species,
                 shape = Species)) +
  labs(x = "Sepal Length",
       y = "Sepal Width",
       title = "Sepal Length and Width")
```




```
# Boxplot
plot_2 <- ggplot(data = iris,
                 aes(x = Species,
                     y = Petal.Width))

# Basic boxplot adding a `geom_boxplot()` layer
# Important note: the x value must be factored or grouped to have individual
# boxplots returned
plot_2 +
  geom_boxplot()
```




```
# Colorful boxplot adding more layers and aesthetic mappings to the data
plot_2 +
  geom_boxplot(aes(color = Species),
               outlier.colour = NA,
               position = "dodge") +
  geom_jitter(aes(color = Species,
                  x = Species),
              position = position_jitter(width = .05),
              alpha = 0.5) +
  labs(x = "Species",
       y = "Petal Width",
       title = "Petal Width of Iris Species") +
  theme_bw()
```



Factors

This can be the cause of many issues when plotting data. Always check if the data is factored and determined whether is should or should not be to get the correct plot.

R has a special class for working with categorical data, called factors. Once created, factors can only contain a pre-defined set of values, known as levels. Factors are stored as integers associated with labels and they can be ordered or unordered. While factors look (and often behave) like character vectors, they are actually treated as integer vectors by R. So you need to be very careful when treating them as strings.

When working with data in a data frame, the columns that contain text are not automatically coerced (= converted) into the factor data type, but once we have loaded the data we can do the conversion using the factor() function

```
head(iris)
iris$Species <- factor(iris$Species)
head(iris)

summary(iris$Species)
levels(iris$Species)
nlevels(iris$Species)
```

By default, R always sorts levels in alphabetical order. In the example below, R will assign 1 to the level "experimental" and 2 to the level "wild type" (because "e" comes before "w", even though the first element in this vector is "wild type").

```
sample <- factor(c("wild type", "experimental", "wild type", "experimental"))

# Different ways to see the levels of factored data
sample
levels(sample)
nlevels(sample)
```

Reorder factors

```
sample <- factor(sample,
                 levels = c("wild type", "experimental"))

sample
levels(sample)
nlevels(sample)
```

Converting Factors

Sometimes it is necessary to convert data from one type to another

```
sample <- factor(sample,
                 levels = c("wild type", "experimental"))

as.character(sample)
sample_char <- as.character(sample)
sample_char

as.numeric(sample)
```

```
year_fct <- factor(c(1990, 1983, 1977, 1998, 1990))
levels(year_fct)

as.numeric(year_fct)
as.character(year_fct)

as.numeric(as.character(year_fct))
as.numeric(levels(year_fct))[year_fct]
```

Notice that in the `levels()` approach, three important steps occur:

- Obtain all the factor levels using `levels(year_fct)`
- Convert these levels to numeric values using `as.numeric(levels(year_fct))`
- Access these numeric values using the underlying integers of the vector `year_fct` inside the square brackets

Renaming Factors

```
# Inspect the contents of the column
levels(iris$Species)

# Replace level names with a new vector of names
levels(iris$Species) <- c("Set", "Ver", "Virg")
levels(iris$Species)

# Check names of factors have been replaced within data frame
iris$Species
```

Manipulating, analyzing and exporting data with tidyverse

Data manipulation using dplyr and tidyr

dplyr is a package for helping with tabular data manipulation. It pairs nicely with tidyr which enables you to convert between different data formats for plotting and analysis.

The tidyverse package is an "umbrella-package" that installs tidyr, dplyr, and several other useful packages for data analysis, such as ggplot2, tibble, etc.

dplyr functions

- `select()` subset columns
- `filter()` subset rows on conditions
- `mutate()` create new columns by using information from other columns
- `group_by()` and `summarize()` create summary statistics on grouped data
- `arrange()` sort results
- `count()` count discrete values

Selecting columns and filtering rows

- `select()` subset columns
  - The first argument is the data frame (iris), and the subsequent arguments are the columns to keep.
- `filter()` subset rows on conditions

```
# Install Tidyverse
# install.packages("tidyverse")

# Load the umbrella package Tidyverse
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --
```

```
## v tibble  3.1.1     v dplyr   1.0.7
## v tidyr   1.1.4     v stringr 1.4.0
## v readr   2.0.2     v forcats 0.5.1
## v purrr   0.3.4
```

```
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
# Reloading data to ensure nothing has been manipulated
data(iris)
```

```
# Select specific columns
select(iris,
```

```
      Species, Petal.Length, Petal.Width)
```

Select all columns except certain ones by using a "-" in front of the variable to exclude it

```
select(iris,
       -Sepal.Length, -Sepal.Width)
```

Choose rows based on specific criterion with `filter()`

```
filter(iris, Species == "virginica")
```

Pipes

What if you want to select and filter at the same time? There are three ways to do this: use intermediate steps, nested functions, or pipes.

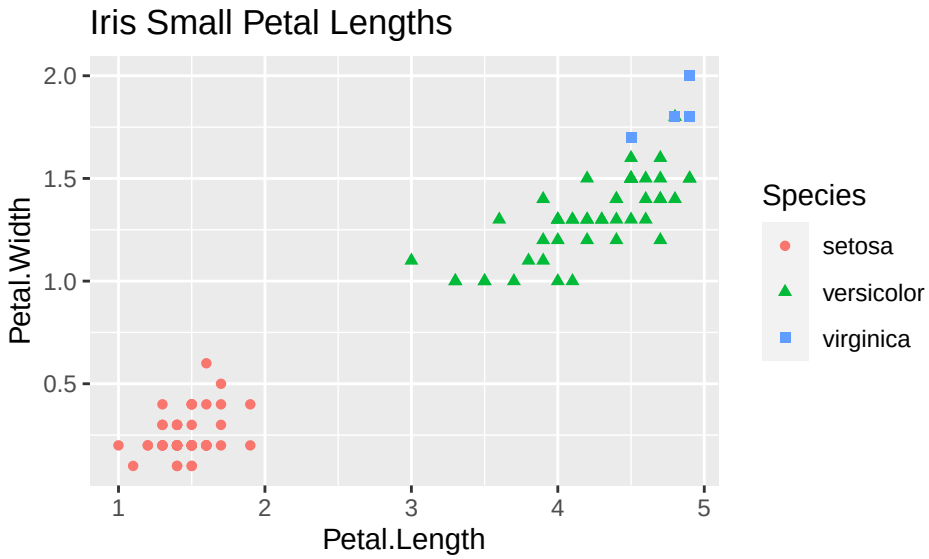
Pipes let you take the output of one function and send it directly to the next, which is useful when you need to do many things to the same data set. Pipes in R look like `%>%` and are made available via the magrittr package, installed automatically with dplyr. If you use RStudio, you can type the pipe with `Ctrl` + `Shift` + `M` if you have a PC or `Cmd` + `Shift` + `M` if you have a Mac.

```
# Filter rows and save as an object
small <- filter(iris,
                Petal.Length < 5)

# Select columns and save as another object
small_x <- select(small,
                  Species, Petal.Length, Petal.Width)
small_x

# Combine the two operations in one piece of code using pipes
iris %>%
  filter(Petal.Length < 5) %>%
  select(Species, Petal.Length, Petal.Width)
```

Create a plot using the subset data

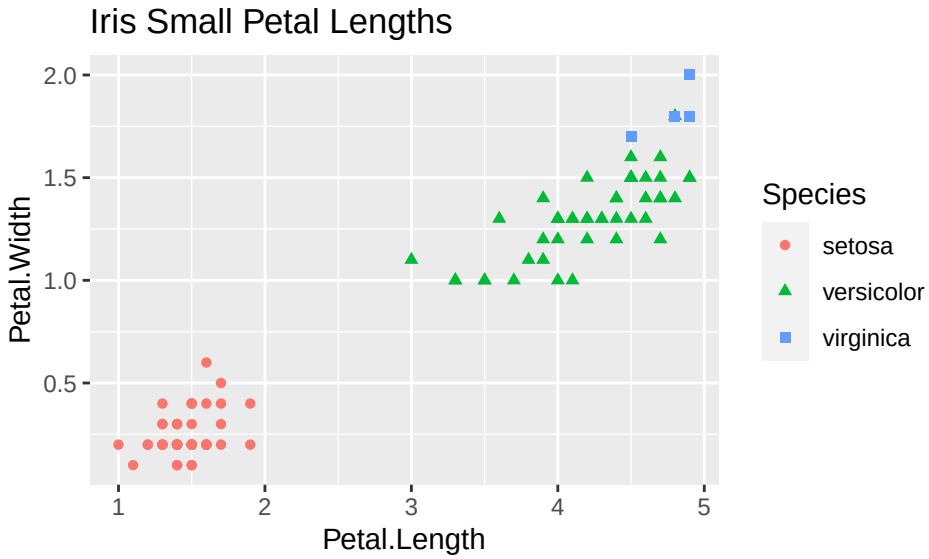
```
# Create a new object with the subset data
iris_petals <- iris %>%
  filter(Petal.Length < 5) %>%
  select(Species, Petal.Length, Petal.Width)

# Use new data to create a plot object
petal_plot <- ggplot(iris_petals,
                 aes(x = Petal.Length,
                     y = Petal.Width))

# Add layers to plot object
petal_plot +
```

```
  geom_point(aes(shape = Species,
                 color = Species)) +
  labs(title = "Iris Small Petal Lengths")
```



Write the subset data and plot in one piece of code

```
iris %>%
  filter(Petal.Length < 5) %>%
  select(Species, Petal.Length, Petal.Width) %>%
  ggplot(aes(x = Petal.Length,
             y = Petal.Width)) +
  geom_point(aes(shape = Species,
                 color = Species)) +
  labs(title = "Iris Small Petal Lengths")
```



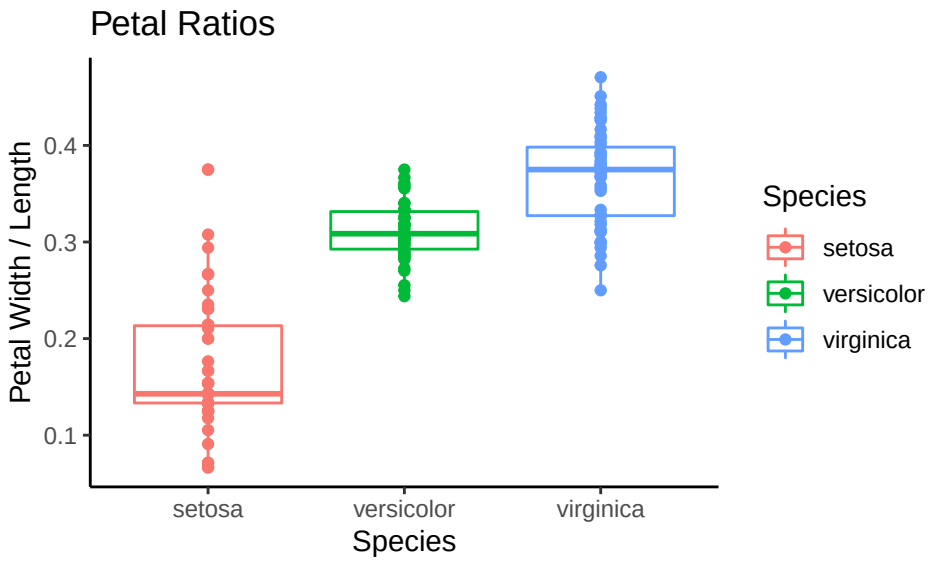
Mutate

Create new columns based on the values in existing columns using `mutate()`

```
iris %>%
  mutate(petal_ratio = Petal.Width / Petal.Length,
         petal_percent = petal_ratio * 100)
```

Add a plot in one piece of code

```
iris %>%
  mutate(petal_ratio = Petal.Width / Petal.Length,
         petal_percent = petal_ratio * 100) %>%
  ggplot(aes(x = Species,
             y = petal_ratio)) +
  geom_boxplot(aes(color = Species)) +
  geom_point(aes(color = Species)) +
  labs(x = "Species",
       y = "Petal Width / Length",
       title = "Petal Ratios") +
  theme_classic()
```



Pretend the first few rows of the output contain NAs. To remove those we can insert a filter() in the chain of functions combined with `is.na()` to identify all NAs and the `!` operator to negate the result

```
iris %>%
  filter(!is.na(Petal.Length)) %>%
  mutate(petal_ratio = Petal.Width / Petal.Length) %>%
  tail()
```

Split-apply-combine data analysis and the `summarize()` function

Many data analysis tasks can be approached using the **split-apply-combine paradigm**

- Split the data into groups
- Apply some analysis to each group
- Combine the results

Key functions of dplyr for this workflow are `group_by()` and `summarize()`. `group_by()` is often used together with `summarize()`, which collapses each group into a single-row summary of that group.

```
iris %>%
  group_by(Species) %>%
  summarize(petal_width_mean = mean(Petal.Width, na.rm = TRUE))
```

Once the data are grouped, you can also summarize multiple variables at the same time. Add a column indicating the minimum and maximum width for each species.

```
iris %>%
  filter(!is.na(Petal.Width)) %>%
  group_by(Species) %>%
  summarize(petal_width_mean = mean(Petal.Width),
            petal_width_min = min(Petal.Width),
            petal_width_max = max(Petal.Width))
```

Exporting plots

`ggsave()` allows you easily change the dimension and resolution of your plot by adjusting the appropriate arguments (width, height, and dpi)

```
my_plot <- ggplot(data = iris,
                  aes(x = Sepal.Length,
                      y = Sepal.Width,
                      color = Species)) +
    geom_point() +
    facet_wrap(vars(Species)) +
    labs(title = "Petal width by species",
         x = "Sepal Length",
         y = "Sepal Width") +
    theme_bw() +
    theme(axis.text.x = element_text(colour = "grey20", size = 12, angle = 90,
                                     hjust = 0.5, vjust = 0.5),
          axis.text.y = element_text(colour = "grey20", size = 12),
          text = element_text(size = 16))

ggsave("path/name_of_file.png", my_plot, width = 15, height = 10)
```

Review

Objects

- Variables: placeholders for assigned values and objects.
- Vectors: contain a single type of value (character, numeric, logical, or integer)
- Data frames: are multiple vectors of the same length, each vector can contain a different data type.

Subsetting

- Subsetting: extracts values from a vector or larger objects
  - Brackets indicate the values to be subset
  - Logical vectors determine the values to be selected
  - Conditional operators: select values based on criterion provided
    - or (`|`), and (`&`)
    - less than (`<`, `<=`), greater than (`>`, `>=`), equal to (`==`)

Data

- The most common object you'll see are data frames, which are a collection of vectors of the same length.
- Missing data: missing data can cause issues with some functions but can be easily circumvented using the argument `na.rm = TRUE`, making a logical vector with `is.na()` or `!is.na()`, or using the functions `na.omit()` or `complete.cases()`
- Data sets are available in R for you to use, simply choose one from a list `data()`

Tidyverse

- It is common to use tools and packages available in the Tidyverse collection for data analysis and visualization.
- `Ggplot2`: this package is for data visualization. It requires a ggplot for the base plot which contains the data, then layers can be added to generate a personalized plot.
- Factors are critical for generating the correct plot (`levels()`)
- `Dplyr`: is helpful for data frame manipulation.
  - `%>%` is used as a pipe, which sends the result from one process to the next. Results from data manipulation can be sent directly to make a ggplot and layers can be added from there.
  - `select()` selects columns
  - `filter()` filters rows based on conditions provided
- Save
  - Save your script in a `.R` plain text file to save your code
  - Save your data in CSV format using `write.csv()`
  - `ggsave` save your images to any type of file