BIOL 416 R Tutorial 1 - Introduction to R

Mairin Deith

Table of Contents

# Preface

These tutorials were developed to teach high-level undergraduate students the fundamentals of working in R and R Studio. Instructions have been modified from Dr. Steven Kembel’s [“Biodiversity analysis in R” walkthrough](http://kembellab.ca/r-workshop/biodivR/SK_Biodiversity_R.html) to reflect the learning objectives of BIOL 416 at the University of British Columbia.

# Tutorial 1

January 14 & 16, 2019

## Working in RStudio and R

*This section is primarily about showing how to do these things in R, no specific instructions have been included*.

Topics to demonstrate/cover:

* the console
* demonstrate a simple 1 + 4
* looking at the work space
* which parts do what (script space, console, environment, file panes)
* workflow between the script pane and the console
* commenting code in the script
* saving files and the R working environment + working directory/navigation
* the value in using code versus modifying the data file directly with analysis

## Objects

**Note: this does not currently include any introduction to lists, this may be necessary depending on how much understanding of lists is required to use the picante package and phylogenies in R.**

### 0-dimensional objects (single variables)

The simplest objects in R are single values, or variables.

You assign a value to an object using <- notation. For example, to create a variable called x that is equal to 5:

x <- 5

A 0-dimensional object contains a single datum - a number, string of text, logical TRUE or FALSE values, or other types. We’ll explore each of these later.

You can name these objects whatever you want, but be careful: capitalization matters and you can’t put spaces in the names.

It’s generally a good idea to give the objects names that mean something to you:

species.name <- "Homo sapiens"  
am.i.human <- TRUE  
number.of.legs <- 2

Once you create an object, you can see what value it holds by calling it:

x

## [1] 5

species.name

## [1] "Homo sapiens"

You can also re-write a variable by re-assigning a value to it:

x <- "new x"  
x

## [1] "new x"

### 1-dimensional objects (vectors)

It is also possible to make objects that have more than one value in them.

Vectors are objects that contain multiple single datum, all in a row. It’s the same as vectors in matrix mathematics - a one-dimensional row of values.

To make this vector: , we would write:

v <- c(0, 3, 5, 10, 2)  
v

## [1] 0 3 5 10 2

The c() here stands for combine.

There are a few ways to pull out a single value from your vector. First, you can use square brackets ([X], where X is a number indicating the position where the datum you want is stored, from 1 to however many objects are in your vector).

v[1]

## [1] 0

v[4]

## [1] 10

v[10]

## [1] NA

Because we don’t have 10 datum in the vector, trying to pull out the 10th item gives NA, not applicable. The spot is empty.

We can also pull out multiple datum using c(), as before, or by using :. The colon indicates to R that we want values across a range; e.g. 1:3 means all numbers between 1 and 3 (1,2,3).

# Using the combine function:  
v[c(1,2)]

## [1] 0 3

# Using a range of numbers:  
v[1:3]

## [1] 0 3 5

Another way to pull out datum from a vector is by naming each object in the vector, and referring to each datum by name rather than location:

names(v) <- c("One","Two","Three","Four","Five")  
# Now v has names attached to it  
v["One"]

## One   
## 0

### An aside on data classes

R uses three main types of classes:

* Numeric (numbers)
* Strings (text or individual characters, denoted by including " " around the text)
* Logical (also called Boolean, this class of data is a binary TRUE or FALSE value)

In a vector, all of the data have to be of the same type (numbers, text strings, TRUE/FALSE) or else R will convert each datum to be some universal type.

bad.v <- c(0, "twelve", TRUE)  
bad.v

## [1] "0" "twelve" "TRUE"

Notice that now R sees all of these as text strings - you can tell by the "".

Let’s try another example:

another.bad.v <- c(0, TRUE, FALSE)  
another.bad.v

## [1] 0 1 0

In this case, R transformed the TRUE and FALSE values into 0 and 1. This is because it is forcing the data to be of the same class.

### 2-dimensional objects (matrices and dataframes)

Most of the data that we’re used to looking at comes in the form of a table - the sort of data you would enter into a spreadsheet such that you have observations in rows and attributes of those observations in columns. Here’s an example of a 2D data-set that comes automatically with R - this is from the pre-loaded data-set called iris:

# Ignore this for now:  
  
# Load in the pre-loaded iris dataset  
data('iris')  
head(iris)

## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
## 1 5.1 3.5 1.4 0.2 setosa  
## 2 4.9 3.0 1.4 0.2 setosa  
## 3 4.7 3.2 1.3 0.2 setosa  
## 4 4.6 3.1 1.5 0.2 setosa  
## 5 5.0 3.6 1.4 0.2 setosa  
## 6 5.4 3.9 1.7 0.4 setosa

# Also demonstrate:  
# View(iris)

Notice that when we use data() to load in the iris data-set, the Environment has a new entry, iris. We’ll talk about what the data() command is in a minute - for now, just look at how the data are put together:

* each column is a variable - ALL THE SAME TYPE!
* each row is an observation

data.frames can hold any type of data: numbers, text, or factors. Each column contains the same type of data.

class(iris)

## [1] "data.frame"

Just like in 1-dimensional vectors, we can pull individual objects out of the iris data-set using matrix notation.

# To pull out the object in row 1, column 1:  
### Row first, then column  
iris[1,1] # Note the "," separating the row

## [1] 5.1

# To pull out a single column, leave the row spot empty...  
iris[,1]

## [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4  
## [18] 5.1 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5  
## [35] 4.9 5.0 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0  
## [52] 6.4 6.9 5.5 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8  
## [69] 6.2 5.6 5.9 6.1 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4  
## [86] 6.0 6.7 6.3 5.6 5.5 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8  
## [103] 7.1 6.3 6.5 7.6 4.9 7.3 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7  
## [120] 6.0 6.9 5.6 7.7 6.3 6.7 7.2 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7  
## [137] 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8 6.7 6.7 6.3 6.5 6.2 5.9

#...or use its name with a "$"  
iris$Sepal.Length

## [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4  
## [18] 5.1 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5  
## [35] 4.9 5.0 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0  
## [52] 6.4 6.9 5.5 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8  
## [69] 6.2 5.6 5.9 6.1 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4  
## [86] 6.0 6.7 6.3 5.6 5.5 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8  
## [103] 7.1 6.3 6.5 7.6 4.9 7.3 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7  
## [120] 6.0 6.9 5.6 7.7 6.3 6.7 7.2 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7  
## [137] 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8 6.7 6.7 6.3 6.5 6.2 5.9

# To pull out a single row, leave the column spot empty:  
iris[1,]

## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
## 1 5.1 3.5 1.4 0.2 setosa

# Don't forget that if you want to use a subset of the total data.frame later, you can save any of these as its own variable  
sep.length <- iris$Sepal.Length  
  
# Ask: any guesses about what type of object this will be?  
class(sep.length)

## [1] "numeric"

## Modifying variables

To modify numeric variables, you can re-save the variable with the same name…

x <- 5  
x <- x+3  
x

## [1] 8

…you can make the object larger by turning a single number into a list of numbers…

x <- c(x,5,12,15)  
x

## [1] 8 5 12 15

…you can make the object smaller by removing datum…

x <- x[-3]  
x

## [1] 8 5 15

…you can force R to read the object in a different way…

as.data.frame(x)

## x  
## 1 8  
## 2 5  
## 3 15

…and you can also change specific values within your data:

# Ask the class "How do you think I could change the first value, 8, into 10?"  
x[1] <- 10  
x

## [1] 10 5 15

You can do the same with data.frames - you can add to the data.frame to make it larger…

### Emphasize why this is important - do not overwrite the original data  
iris.2 <- iris  
# For example, this adds a column called "new.column" to the iris data.frame  
iris.2$new.column <- 0  
head(iris.2)

## Sepal.Length Sepal.Width Petal.Length Petal.Width Species new.column  
## 1 5.1 3.5 1.4 0.2 setosa 0  
## 2 4.9 3.0 1.4 0.2 setosa 0  
## 3 4.7 3.2 1.3 0.2 setosa 0  
## 4 4.6 3.1 1.5 0.2 setosa 0  
## 5 5.0 3.6 1.4 0.2 setosa 0  
## 6 5.4 3.9 1.7 0.4 setosa 0

# Notice that the column is filled up with "0"s  
  
##### Use fake data for this?   
# We can also add a new row with all "0"s:  
# iris.2[nrow(iris)+1,] <- 0  
##### use rbind() for this

…you can get rid of values within your data frame…

iris.2 <- iris.2[,-6]  
iris.2 <- iris.2[-151,]

…and you can change values within the data.frame.

iris.2[1,"Sepal.Length"] <- 100  
# Or change it back to the original value in iris  
iris.2[1,"Sepal.Length"] <- iris[1,"Sepal.Length"]

You can also create columns to reflect mathematical relationships between different objects. For example, instead of our empty new.column, which we got rid of above when we wrote iris.2[,-6], let’s make a new column with <- that is the ratio of sepal length to petal length.

iris.2$length.ratio <- iris.2$Sepal.Length/iris.2$Petal.Length  
head(iris.2)

## Sepal.Length Sepal.Width Petal.Length Petal.Width Species length.ratio  
## 1 5.1 3.5 1.4 0.2 setosa 3.642857  
## 2 4.9 3.0 1.4 0.2 setosa 3.500000  
## 3 4.7 3.2 1.3 0.2 setosa 3.615385  
## 4 4.6 3.1 1.5 0.2 setosa 3.066667  
## 5 5.0 3.6 1.4 0.2 setosa 3.571429  
## 6 5.4 3.9 1.7 0.4 setosa 3.176471

summary(iris.2$length.ratio)

## Min. 1st Qu. Median Mean 3rd Qu. Max.   
## 1.050 1.230 1.411 2.018 3.176 4.833

## Writing and reading files into/out of R

Now that we’ve made some changes to the iris data, we’d probably like to save these changes.

There are two ways to save data in R:

1. Save all data in the environment as an .Rdata file
2. Save as a different file type, typically a .csv

To save your entire R session with all of the objects in your environment:

# Opens a dialogue that allows you to choose where to save  
save.image("test.Rdata")

We can now open this data-set and return to it any time we like - we don’t have to run the same commands above. To do this, we simply load() the .R data file into our new R session.

load("test.Rdata")

We can also save our modified iris data file as a csv, a common data type that is standard across multiple computers and statistical software.

# Data source first, then file name  
write.csv(iris.2, "iris2.csv", row.names=F)

Similar to load(), we can also read csv files (and many, many other file types) into R:

##### Mention that there are other file formats - including Excel spreadsheets, txt files  
iris2 <- read.csv("iris2.csv")  
head(iris2)

## Sepal.Length Sepal.Width Petal.Length Petal.Width Species length.ratio  
## 1 5.1 3.5 1.4 0.2 setosa 3.642857  
## 2 4.9 3.0 1.4 0.2 setosa 3.500000  
## 3 4.7 3.2 1.3 0.2 setosa 3.615385  
## 4 4.6 3.1 1.5 0.2 setosa 3.066667  
## 5 5.0 3.6 1.4 0.2 setosa 3.571429  
## 6 5.4 3.9 1.7 0.4 setosa 3.176471

# iris.2==iris2

Any time you have pre-existing data you want to use in R, you will probably end up using read.csv() or something similar. Most of the time we are interested in modifying and using our own data, not just the data already provided for us in R.

## Functions

To summarize the last section, you can think of objects as collections of data. We can load existing data, modify those data, pull individual datum out of the object using names or numbered locations (like in matrix notation), and then save those changes.

We can use **functions** to modify, re-arrange, and do calculations on those data.

For example, if our data are numeric we can find mean values, sums, standard deviations, the range of values the data contain, etc.

The “stuff” that we can do with R’s objects are contained in functions. For example, the mean() function calculates the mean value from a series of numbers. So going back to our v vector, we could write out by hand how to calculate the mean:

*Ask class to come up with equation to calculate mean - should take 10 seconds or so, but will get them thinking about math*.

mean.of.v <- (v[1] + v[2] + v[3] + v[4] + v[5])/5  
mean.of.v

## One   
## 4

This is easier with functions:

mean.of.v.easier <- sum(v)/length(v)  
  
mean.of.v.easiest <- mean(v)

Functions are simply pre-written code chunks that take inputs, which we call **Arguments**.  
For example, when we used the mean(v) command, we told R to calculate the mean value of the data contained in the vector v.

You can learn more about functions by looking at their help files with ?mean() or by searching for the function name in lower right panel.

*Demonstrate looking at the help file and searching for help with ?*

# From the help file:  
mean(x, ...)  
  
## Default S3 method:  
mean(x, trim = 0, na.rm = FALSE, ...)

In the same way, we used a function called data() to load the iris data-set. data('iris') tells R that we want to load up a data-set with the name ‘iris’.

## Reading and writing data

* head()
* summary()
* View()

In R, we can use different functions to load up our own personal data-sets - but we’ll get into those later. For the moment, let’s return to the iris data set. Before working with a data set, it’s a good idea to examine it.

Here are a few ways you can examine objects in R, whether they be variables, vectors, data frames, or matrices.

##### Instead of reading in the data here, instead let's load the iris file from a .csv  
##### Brief demo with getwd()  
head(v)

## One Two Three Four Five   
## 0 3 5 10 2

head(iris)

## Sepal.Length Sepal.Width Petal.Length Petal.Width Species  
## 1 5.1 3.5 1.4 0.2 setosa  
## 2 4.9 3.0 1.4 0.2 setosa  
## 3 4.7 3.2 1.3 0.2 setosa  
## 4 4.6 3.1 1.5 0.2 setosa  
## 5 5.0 3.6 1.4 0.2 setosa  
## 6 5.4 3.9 1.7 0.4 setosa

# View(iris)  
summary(iris)

## Sepal.Length Sepal.Width Petal.Length Petal.Width   
## Min. :4.300 Min. :2.000 Min. :1.000 Min. :0.100   
## 1st Qu.:5.100 1st Qu.:2.800 1st Qu.:1.600 1st Qu.:0.300   
## Median :5.800 Median :3.000 Median :4.350 Median :1.300   
## Mean :5.843 Mean :3.057 Mean :3.758 Mean :1.199   
## 3rd Qu.:6.400 3rd Qu.:3.300 3rd Qu.:5.100 3rd Qu.:1.800   
## Max. :7.900 Max. :4.400 Max. :6.900 Max. :2.500   
## Species   
## setosa :50   
## versicolor:50   
## virginica :50   
##   
##   
##

## R’s true power: plotting and statistics

Now that we know what iris looks like, let’s use these data to make some figures and perform some simple statistics.

There are many ways to create plots - check out the [R Base Graphics Cheatsheet](http://publish.illinois.edu/johnrgallagher/files/2015/10/BaseGraphicsCheatsheet.pdf) for a full list of the types of plots that can be made with R.

### Histograms

Let’s start with something simple - a histogram of Sepal.Length?

We need to first tell R where to find the data we want to plot - we want to tell R that it can find a column of data called Sepal.Length inside of the iris data-frame. Take a minute to recall how we can do this - how do we tell R where to find our Sepal.Length data?

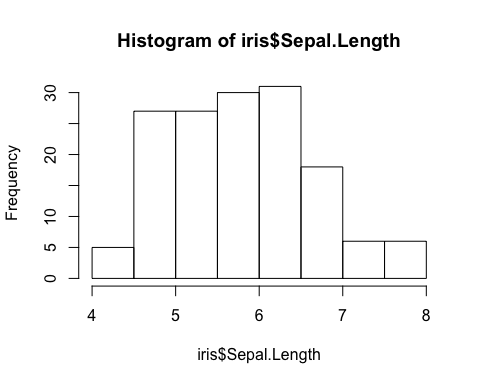
# There are two ways we can do this  
## Because the iris data has columns with names, we can use the name of the column we want   
iris$Sepal.Length

## [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4  
## [18] 5.1 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5  
## [35] 4.9 5.0 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0  
## [52] 6.4 6.9 5.5 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8  
## [69] 6.2 5.6 5.9 6.1 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4  
## [86] 6.0 6.7 6.3 5.6 5.5 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8  
## [103] 7.1 6.3 6.5 7.6 4.9 7.3 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7  
## [120] 6.0 6.9 5.6 7.7 6.3 6.7 7.2 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7  
## [137] 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8 6.7 6.7 6.3 6.5 6.2 5.9

## We can always use numbers-based indexing as well  
iris[,1]

## [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4  
## [18] 5.1 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5  
## [35] 4.9 5.0 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0  
## [52] 6.4 6.9 5.5 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8  
## [69] 6.2 5.6 5.9 6.1 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4  
## [86] 6.0 6.7 6.3 5.6 5.5 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8  
## [103] 7.1 6.3 6.5 7.6 4.9 7.3 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7  
## [120] 6.0 6.9 5.6 7.7 6.3 6.7 7.2 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7  
## [137] 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8 6.7 6.7 6.3 6.5 6.2 5.9

# To make the plot:  
hist(iris$Sepal.Length)

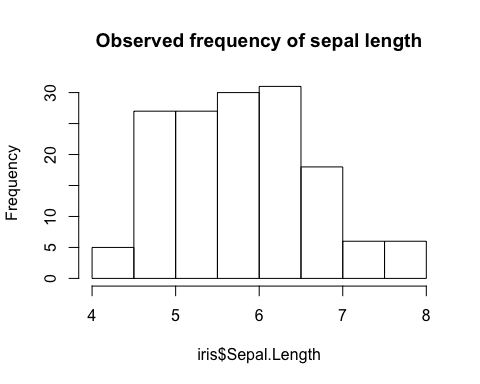


Easy, wasn’t it? But it isn’t very pretty - what could we change?

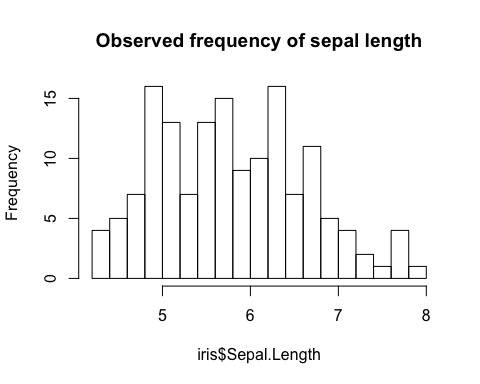
When we make modifications to our plot, we pass extra arguments. In the original command, hist(iris$Sepal.Length), the only argument we gave it was the data to plot.

Take a look at what other arguments are available with ?hist():

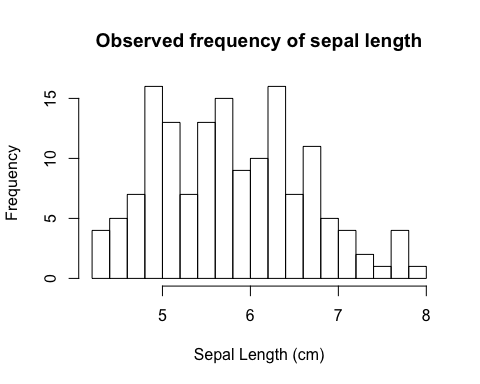
?hist()  
  
# Change the title, NOTICE THAT WE USE COMMAS AND ARGUMENT NAMES!!!  
hist(x = iris$Sepal.Length, main = "Observed frequency of sepal length")



hist(x = iris$Sepal.Length, main = "Observed frequency of sepal length", breaks = 20)



hist(iris$Sepal.Length,   
 main = "Observed frequency of sepal length",   
 breaks = 20,   
 xlab = "Sepal Length (cm)")

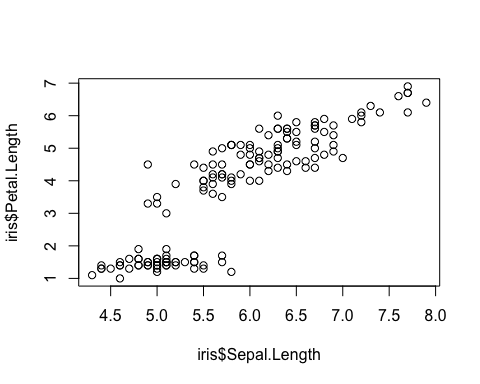


### Scatterplot

What about a more complicated type of plot next?

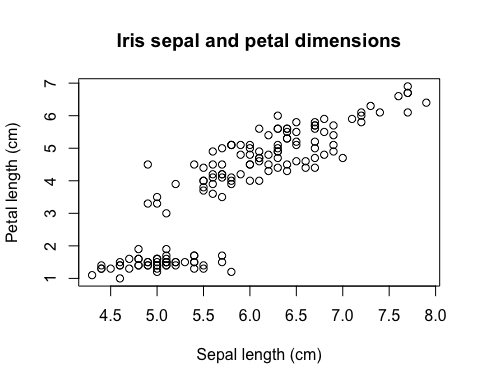
Let’s try an x-y/scatter plot:

?plot()  
  
plot(x = iris$Sepal.Length,   
 y = iris$Petal.Length)



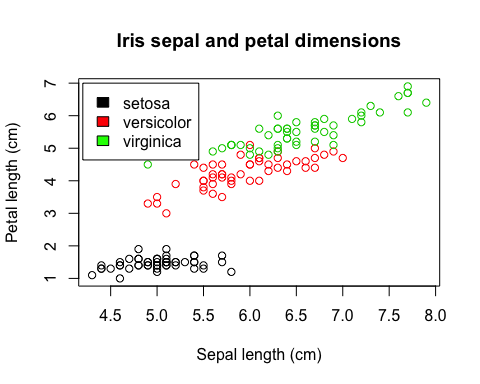
Take five minutes to change the main title and the X and Y axes’ titles.

# Output should look something like this  
plot(x = iris$Sepal.Length,   
 y = iris$Petal.Length,  
 main = "Iris sepal and petal dimensions",  
 ylab = "Petal length (cm)",  
 xlab = "Sepal length (cm)")



We can also goof around with colors and point types:

plot(x = iris$Sepal.Length,   
 y = iris$Petal.Length,  
 main = "Iris sepal and petal dimensions",  
 ylab = "Petal length (cm)",  
 xlab = "Sepal length (cm)",  
 col = iris$Species)  
# pch = 19)  
  
# And we can add a legend:  
legend(x = 4.2,  
 y = 7,   
 legend = unique(iris$Species),  
 fill = c('black','red','green'))



### Saving plots

Plots that you create in R can be saved in one of two ways:

1. Saving in the Plots tab in the lower right panel
2. Saving directly in the code - most useful if you’re going to be recreating the plots a bunch

*Quickly demonstrate how to save plots using the GUI with the Plots tab*.

# Saving plots with code:  
## For example, to save a .png:  
png("iris\_scatterplot.png")  
plot(x = iris$Sepal.Length,   
 y = iris$Petal.Length,  
 main = "Iris sepal and petal dimensions",  
 ylab = "Petal length (cm)",  
 xlab = "Sepal length (cm)",  
 col = iris$Species,  
 pch = 19)  
dev.off()

## quartz\_off\_screen   
## 2

This uses two new functions:

* png() tells R to prepare a .png file with the name you provide
* then, running iris.scatterplot prints out the plot within that file
* finally, dev.off() closes the file and saves it

You can also save to other formats:

pdf("iris\_scatterplot.pdf")  
plot(x = iris$Sepal.Length,   
 y = iris$Petal.Length,  
 main = "Iris sepal and petal dimensions",  
 ylab = "Petal length (cm)",  
 xlab = "Sepal length (cm)",  
 col = iris$Species,  
 pch = 19)  
dev.off()

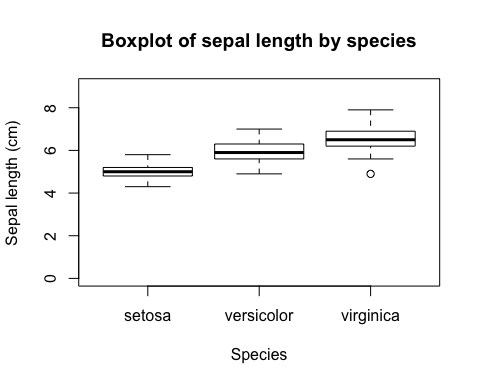
## quartz\_off\_screen   
## 2

# Or pass arguments about the plot itself, like the size, resolution, etc.  
png("iris\_scatterplot\_smaller.png", width = 200, height = 200)  
plot(x = iris$Sepal.Length,   
 y = iris$Petal.Length,  
 main = "Iris sepal and petal dimensions",  
 ylab = "Petal length (cm)",  
 xlab = "Sepal length (cm)",  
 col = iris$Species,  
 pch = 19)  
dev.off()

## quartz\_off\_screen   
## 2

*Get the students to make a totally new type of plot - boxplot() - and save it to their computer. Provide only a sample plot and the name of the boxplot() function*.

?boxplot()  
  
boxplot(iris$Sepal.Length~iris$Species, fill=iris$Species,  
 main = "Boxplot of sepal length by species",  
 ylab = "Sepal length (cm)",  
 xlab = "Species",  
 ylim = c(0,9)  
)



### Intro to statistics - linear regression

The final thing that R is particularly good at besides plotting data performing statistics on those data.

Let’s start with something really simple - what if we want to know how correlated sepal length and petal length are in these three species of iris?

*Ask class if anyone knows what statistical test could be used here*.

The tool that we want is called linear regression using a new function, lm().

lm() stands for linear model - this function fits a linear regression between two variables. Essentially, linear regression is used to calculate how closely correlated one variable is to another.

The linear part refers to the fact that we are calculating a straight-line relationship between these two, just like in the classic linear equation:

If our is sepal length and our is petal length, we want to create an equation that we can use to predict petal length based on some observed sepal length.

In other words, we’re asking “when observed sepal length increases, how much of an increase in petal length should we expect?” We probably also want to know if there is a statistically significant relationship between the two, or if the trend we see in this plot could arise from random noise in our data.

Linear regression finds the values for , the slope of the relationship, and , the y-intercept of the line, that best predict petal length from sepal length.

We’re not going to go too much further into the statistical basis for this, so let’s just dive into the code.

# Regress petal length against sepal length  
  
?lm()  
  
# lm() looks for a formula; in other words, we need to provide an equation for what variables we expect to be dependent on the other.   
# In this case, let's assume that we are testing the petal length as a function of sepal length.   
  
# NOTE: new argument, data =, allows us to use just the variable names directly  
  
linear.model <- lm(Petal.Length~Sepal.Length,  
 data = iris)  
  
summary(linear.model)

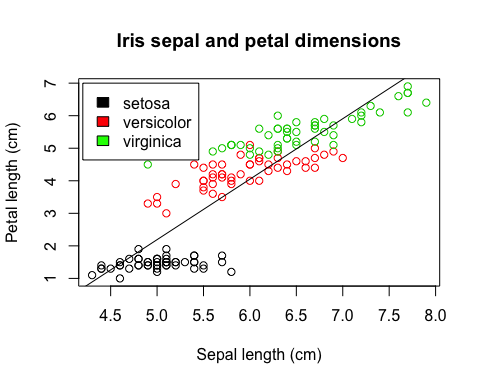
##   
## Call:  
## lm(formula = Petal.Length ~ Sepal.Length, data = iris)  
##   
## Residuals:  
## Min 1Q Median 3Q Max   
## -2.47747 -0.59072 -0.00668 0.60484 2.49512   
##   
## Coefficients:  
## Estimate Std. Error t value Pr(>|t|)   
## (Intercept) -7.10144 0.50666 -14.02 <2e-16 \*\*\*  
## Sepal.Length 1.85843 0.08586 21.65 <2e-16 \*\*\*  
## ---  
## Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1  
##   
## Residual standard error: 0.8678 on 148 degrees of freedom  
## Multiple R-squared: 0.76, Adjusted R-squared: 0.7583   
## F-statistic: 468.6 on 1 and 148 DF, p-value: < 2.2e-16

Using the summary() command, R will spit out a bunch of information about the results of our linear regression.

For example, the coefficient for Sepal.Length is 1.85843 - this is the value in our classical linear equation. Meanwhile, the estimate for our y-intercept is -7.10144. Putting these into our equation, we can say that the best linear equation to describe the relationship between petal length and sepal length is:

Now that we have this line, we can put it on top of our scatter plot!

# the abline() function can be used to directly put linear model objects onto a plot  
plot(x = iris$Sepal.Length,   
 y = iris$Petal.Length,  
 main = "Iris sepal and petal dimensions",  
 ylab = "Petal length (cm)",  
 xlab = "Sepal length (cm)",  
 col = iris$Species)# ,  
 #pch = 19)  
abline(linear.model)  
legend(x = 4.2,  
 y = 7,   
 legend = unique(iris$Species),  
 fill = c('black','red','green'))



*Brief conversation about whether the regression actually means anything given the species level differences*.

##### END HERE

Finally, we probably also want to know if the relationship between these two variables is statistically significant.

lm.anova <- anova(linear.model)  
lm.anova

## Analysis of Variance Table  
##   
## Response: Petal.Length  
## Df Sum Sq Mean Sq F value Pr(>F)   
## Sepal.Length 1 352.87 352.87 468.55 < 2.2e-16 \*\*\*  
## Residuals 148 111.46 0.75   
## ---  
## Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

Finally, just like with other objects, we can pull out individual datum from this object.

For example, to save the p-value of the Sepal.Length coefficient returned by the model fitting:

lm.anova$`Pr(>F)`

## [1] 1.038667e-47 NA

# Note that the NA is because there is no P value for the Residuals in the model - residuals measure the extra noise that is not captured by the linear model.

Don’t forget to save your work, and see you next week for more about R and how to use it to manipulate and work with phylogenies.

## Summary

* objects - 0d, 1d, 2d
* indexing objects:
* with $ if the components of the object are named or in **lists**
* with [] in all cases; if 2d, use multiple indexes with [,]
* operations - math, comparisons
* functions - more complex operations that are often pre-compiled
* plotting - many different types, generally you pass on graphical parameters within the plot() call itself or, in some cases, after plot() has been called
* statistics - R can perform many, many statistical analyses - many simple statistical tools are available right away in R, and many others are available through libraries (which we’ll discuss next week)

Summary with why to follow these steps:

* species bounded regressions, explain why is this important?
* does the relationship hold for the species individually?