Module 1.1

Getting started: working with objects

Fall 2019

NOTE: this module borrows heavily from an R short course developed by a team at Colorado State University.
- Thanks to Perry Williams for allowing us to use these materials!!

Welcome to the 2019 R bootcamps! Let's get started!!

Get materials for this module

- I recommend having a printed copy of a good base R "cheatsheet" as you get started. You will be introduced to a lot of new material very quickly in this workshop, and you are not expected to remember everything! 'Cheatsheets' like this can quickly and easily refresh your memory. So print out the cheatsheet and don't worry if you don't memorize everything we tell you in this workshop- the key is that you leave this workshop knowing what can be done in R (just about everything!) and how to learn how to do things in R. And most importantly, we hope the workshop gives you confidence to use R and learn new things in R!

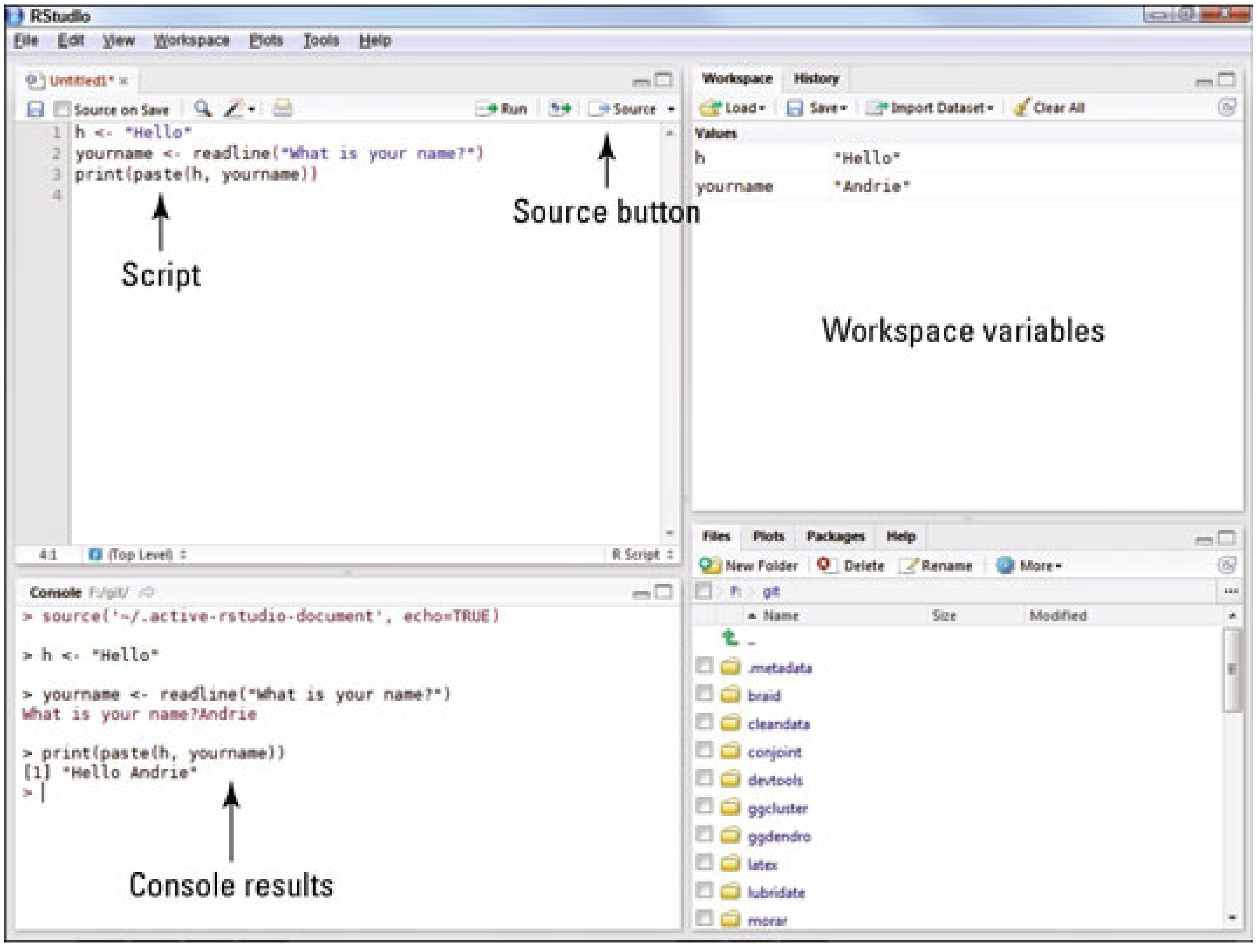
Here are links to good base R "cheatsheets":

Base R Cheat sheet
R reference card

Load script for module #1.1

1. Click here to download the script for this module! I recommend right-clicking (or control-clicking for Macs) on the link and selecting the "Save As" option from the context menu. Save the script to a convenient folder on your laptop, which will serve as your **working directory** for this bootcamp. I recommend creating a new folder to store all the scripts, data sets and other materials from these modules!
2. Start a new **RStudio Project**. To do this, open RStudio and from the File menu select File>>New Project. In the menu that follows, select "Existing Directory", and then navigate to the folder in which you saved the R script for this module (your working directory). The R project will automatically take the name of your working directory folder.
3. Load a blank script in RStudio. To do this, click on the "blank paper" icon at the far left of the RStudio toolbar (or File>>New File>>R script). Your RStudio interface should now be divided into four quadrants. In the Rstudio default configuration, the top left panel is your **script** (a set of written commands that R can recognize and execute), the bottom left panel is your **console** (your direct connection with R- prints errors and stores a record of commands that have already been run), the top right panel is your **Environment** (container for all the objects – data, variables, functions– that you have defined in your current **session**), and the bottom left panel is a set of tabbed interfaces that let you view helpfiles, load packages, view graphical plots, view files in your working directory, etc. Basically, RStudio provides all the functionality you need to develop, debug and execute R code!

Let's get started!!



```
h <- "Hello"
yourname <- readline("What is your name?")
print(paste(h, yourname))
```

h	"Hello"
yourname	"Andrie"

```
> source('~/.active-rstudio-document', echo=TRUE)

> h <- "Hello"

> yourname <- readline("What is your name?")
What is your name?Andrie

> print(paste(h, yourname))
[1] "Hello Andrie"
> |
```

Explore the R console

The R console (what you see if you opened R directly without going through RStudio) is a **command line** interface, and is *your direct connection with the R statistical programming language.* That means you give R a command to run in the console and R executes that command. This is the heart of R- all the rest of RStudio consist of wrappers to help us make effective use of the console! In fact, after this short demo you will rarely interact directly with the console (instead, you will use a script!)!

The R console is located in the left half or bottom left of the RStudio interface (see figure above).

1. Click on the console just to the right of one of the ">" prompts.
2. You can use the console like a calculator. For example, type `2+2` at the prompt and hit enter. What does R do?
3. Now hit the "up" arrow. The R console remembers all previous commands that it has executed in this **session**, which allows you to re-run commands relatively easily. However, if you accidentally hit the "up" arrow you lose everything you had been typing!
4. Any command that is preceded by a pound sign (#) is ignored by the R console. Try typing `# 2+2` and hitting "Enter". Nothing happens, right?

Let's create our first object

Objects are defined using R's **assignment operator**, which looks like a left arrow (`<-`). Type the following statement directly into the console:

```
Batmans_butler <- 'Alfred Pennyworth'
```

Then hit "Enter". What does R return?

NOTHING!

BUT.... check the **Environment**, or **Workspace** window (top right of RStudio interface). You should see that you now have a new **object** in your environment, called "Batmans_butler", and this object contains a **text string** ("Alfred Pennyworth").

Now type `print(Batmans_butler)`[1] into the R console and hit "Enter". What happens? [2]

If you enter the name of an object at the command line, R automatically prints the value of the object- you don't need to use the `print()` function... Try it: type `Batmans_butler` into the R console and hit "Enter". R dutifully returns the name of Bruce Wayne's loyal and tireless butler, best friend and surrogate father figure. In fact, when you type the name of an object and hit "Enter", R will automatically print the contents of that object back to the console.

Graduating to R Scripts

You will quickly realize that although the command-line R console is nice for some things, there is a much better way to develop R code. That is by storing a sequential set of commands as a text file. This text file (the standard is to use the ".R" extension) is called an **R script**. Again, the top left quadrant of the RStudio interface (at least by default) is where you can view and edit your R scripts.

New script

If you haven't done it already, create a new R script by clicking on the "blank paper" icon at the far left of the RStudio toolbar (or File>>New File>>R script).

In your blank script, define a new object. For example:

```
my_obj <- 2+2
```

Now highlight the text you just wrote (or place the cursor somewhere within the text) and hit 'Ctrl+Enter' (or 'Command+Enter') to run that line of code!

Click on a new (blank) line in the script and type `2+2`. Now hit "Ctrl+Enter" (or "Command+Enter") to run the line of code.

Of course, it's always a good idea to save your scripts often – make sure to save your new script before you move on.

[1]NOTE: `print()` is an **R function** that comes with base R. It takes the name of an object as its **argument** and it responds by printing the contents of that object. We'll get more into R functions shortly.

[2]RStudio has a useful auto-type feature, which can save you lots of time and headaches. After you've typed the first couple letters (e.g., "bat"), RStudio will suggest "Batmans_butler" and you can just hit "Enter" to complete the object name! The auto-type feature, unlike R, is not case sensitive!

Load Existing script

Remember the file you downloaded at the beginning of this submodule? That file is an R script- Let's load it up now!

To do this, click on the "folder" icon on the Rstudio toolbar (or File>>Open File) and choose the script you downloaded earlier.

The first few lines (commands) of the script for this module are preceded by pound signs. R doesn't do anything with these commands- they are called **comments**, and they help to keep code clear, organized and understandable to human readers.

use comments early and often- they are tremendously valuable.

Click on the top of the script (first line of comments). To **run** a line of code, press the "Ctrl" key (or the "Command" key) and hit "Enter". R will then run that line through the console. But of course, nothing happens because it's a comment and not meant to be interpreted by your machine[3].

If you click on a line of code that is not followed by a pound sign, you should see that R interprets the text on that line of code and stores any newly defined objects in memory (see top right quadrant of the Rstudio interface – 'Environment' tab – for a list of objects stored in memory)

Now you know the basics of how to use R scripts.

A quick R demo

Before we get into the basics, let's just run through some of what R can do! Don't worry if you don't understand something- we will go over all of this in much greater detail later!

To get started quickly, we will first load a data set that is built into R!

```r
###############
# R DEMO:
###############

#  don't worry if you don't understand this just yet- this is just a taste of where we are going!

#########
# load a built-in dataset

data(trees)
```

Next, we can examine the data object in r. Again, we will get into this in much more detail later!

```r
#########
# explore the data object

summary(trees)
```

```
##      Girth           Height       Volume
##  Min.   : 8.30   Min.   :63   Min.   :10.20
##  1st Qu.:11.05   1st Qu.:72   1st Qu.:19.40
##  Median :12.90   Median :76   Median :24.20
##  Mean   :13.25   Mean   :76   Mean   :30.17
##  3rd Qu.:15.25   3rd Qu.:80   3rd Qu.:37.30
##  Max.   :20.60   Max.   :87   Max.   :77.00
```

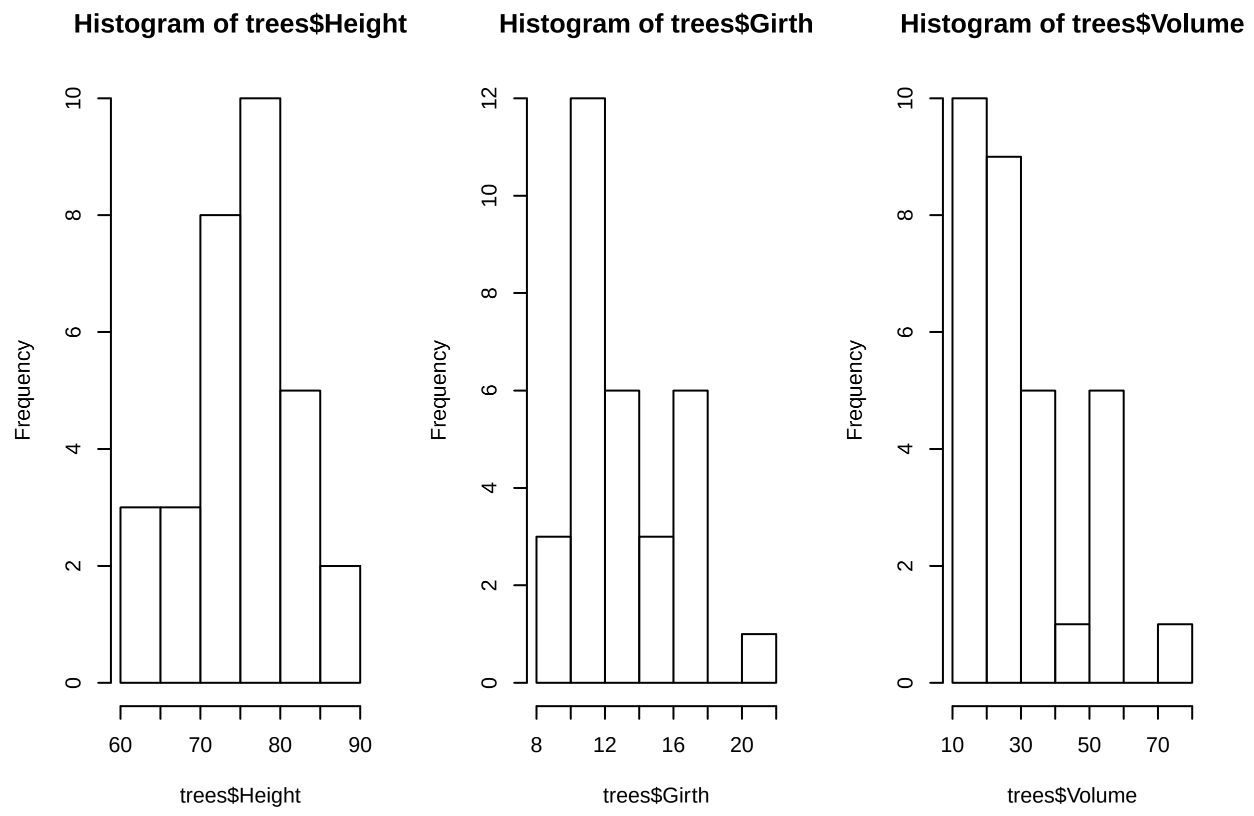
[3]RStudio has several helpful keyboard shortcuts to make running lines of code easier. Check this website for a comprehensive set of keyboard shortcuts

```r
str(trees)
```

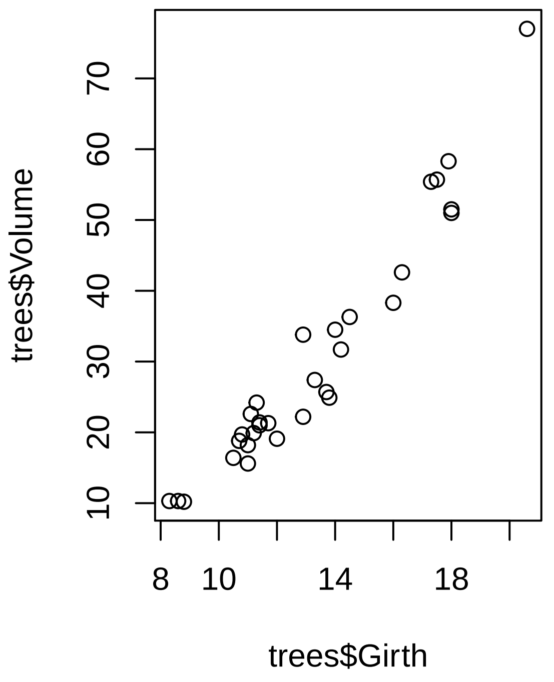
```
## 'data.frame':    31 obs. of  3 variables:
##  $ Girth : num  8.3 8.6 8.8 10.5 10.7 10.8 11 11 11.1 11.2 ...
##  $ Height: num  70 65 63 72 81 83 66 75 80 75 ...
##  $ Volume: num  10.3 10.3 10.2 16.4 18.8 19.7 15.6 18.2 22.6 19.9 ...
```

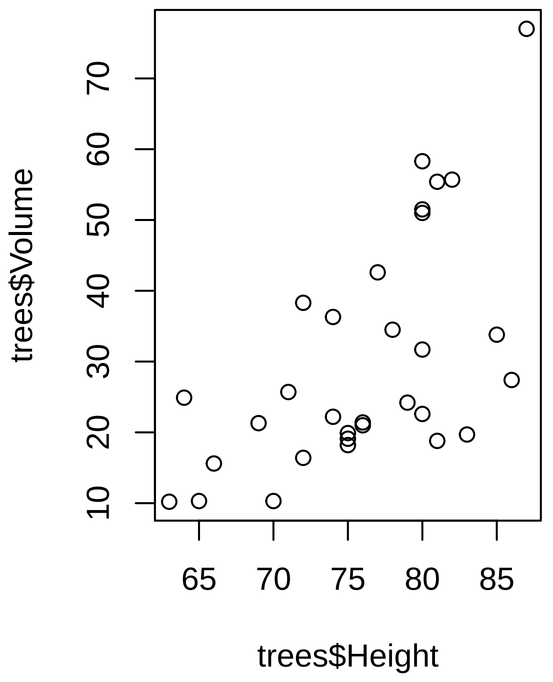
Now let's visualize the data and start to get a sense for what patterns we can detect.

```r
#########
# visualize the data

   # histograms:
layout(matrix(1:3,nrow=1,byrow = T))     # set up a multi-panel graphics device (three plots side by sid
hist(trees$Height)                        # visualize the distribution of height data
hist(trees$Girth)                         # visualize the distribution of girth data
hist(trees$Volume)                        # visualize the distribution of volume data
```

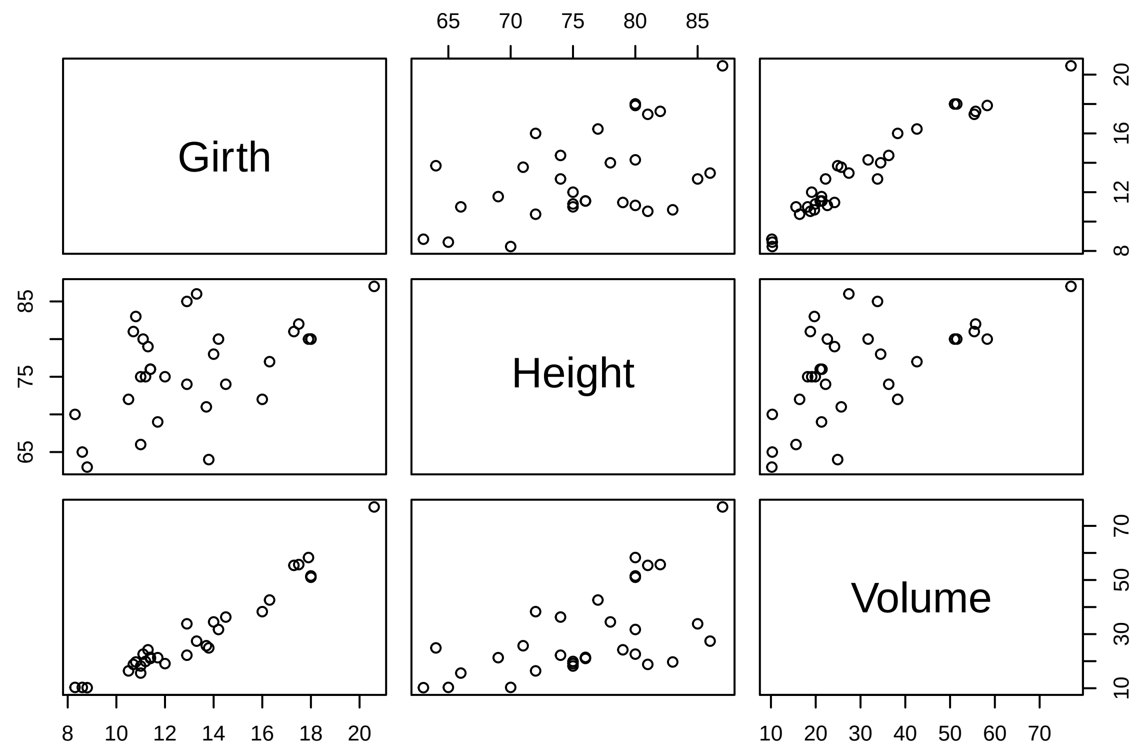


```r
   # scatterplots:

layout(matrix(1:2,nrow=1,byrow = T))       # set graphics device with 2 plots side by side
plot(trees$Volume~trees$Girth)             # scatterplot of volume against girth
plot(trees$Volume~trees$Height)            # scatterplot of volume against height
```





```r
pairs(trees)     # plots all scatterplots together as a scatterplot matrix!
```



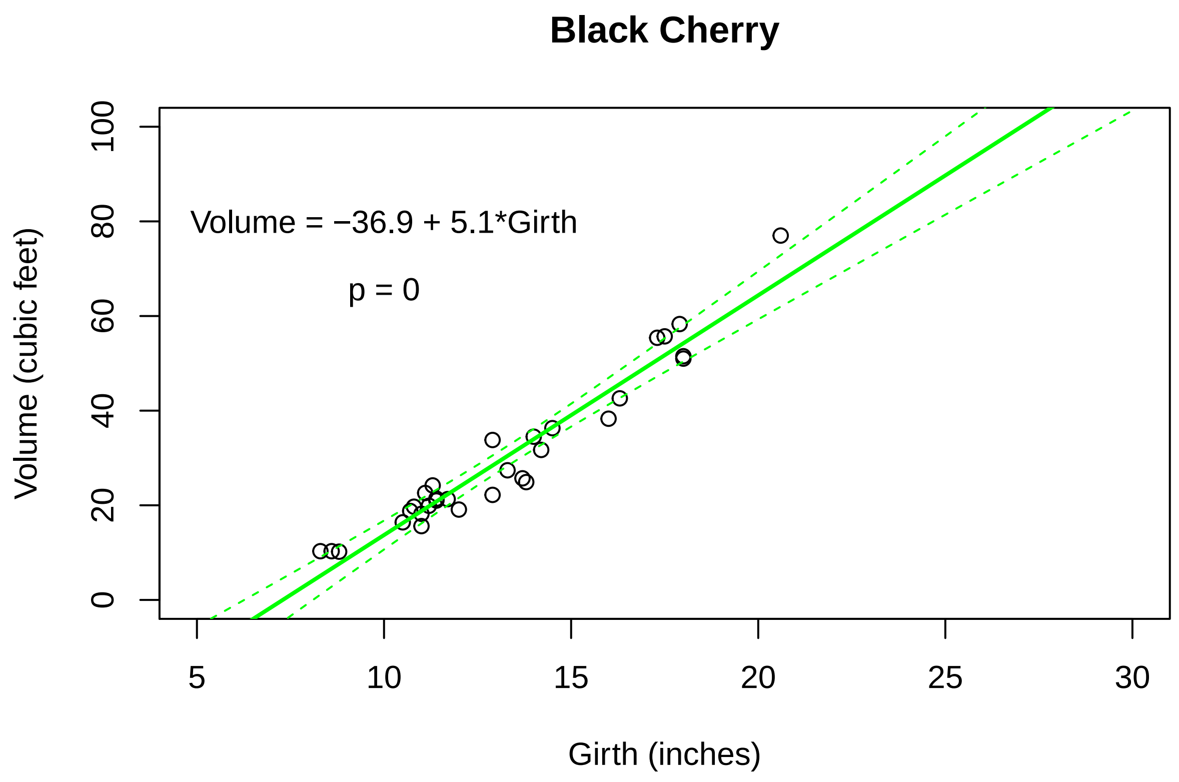
Now we can perform some basic statistics, like a regression analysis:

```
##########
# perform linear regression analysis

model1 <- lm(Volume~Girth,data=trees)         # regress Volume on Girth

summary(model1)     # examine the results
```

```
##
## Call:
## lm(formula = Volume ~ Girth, data = trees)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -8.065 -3.107  0.152  3.495  9.587
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -36.9435     3.3651  -10.98 7.62e-12 ***
## Girth         5.0659     0.2474   20.48  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 4.252 on 29 degrees of freedom
## Multiple R-squared:  0.9353, Adjusted R-squared:  0.9331
## F-statistic: 419.4 on 1 and 29 DF,  p-value: < 2.2e-16
```

And finally, let's visualize the results:

```
#########
# visualize the results!

xvals <- seq(5,30,0.5)              # set the range of "Girth" values over which you want to make predict
pred <- predict(model1,newdata=data.frame(Girth=xvals),interval = "confidence",level = 0.99)   # use th

plot(trees$Volume~trees$Girth,xlab="Girth (inches)",ylab="Volume (cubic feet)",main="Black Cherry",
     xlim=range(xvals),ylim=c(0,100))          # Make a pretty scatterplot

abline(model1,lwd=2,col="green")                  # Add the regression line
lines(xvals,pred[,"upr"],col="green",lty=2)       # Add the upper bound of the confidence interval
lines(xvals,pred[,"lwr"],col="green",lty=2)       #      ... and the lower bound
text(10,80,sprintf("Volume = %s + %s*Girth",round(coefficients(model1)[1],1),round(coefficients(model1)
text(10,65,sprintf("p = %s",round(summary(model1)$coefficients[,"Pr(>|t|)"][2],3)))     # Add the p-val
```



Workspace management

The workspace (environment) panel window in R can become cluttered as we create variables and assignments in R. As we work through the modules in this bootcamp it will be helpful to clear the workspace every once in a while to maintain our sanity. Earlier we assigned the value "Alfred Pennyworth" to the variable `Batmans_butler`. Let's remove that from the workspace. There are several ways to do this.

1. In the 'environment' panel in RStudio (upper right quadrant) there is a little broom. If you click that it will clear the entire workspace.

2. Use `rm(Batmans_butler)` to clear just the `Batmans_butler` variable.
3. Use this handy one-liner `rm(list=ls())` to clear the entire workspace.

Back to basics! R Objects

R has many different kinds of objects that you can define and store in memory. These are the building blocks of the language.

Object **classes** that enable storage of information and data are: *scalars*, *vectors*, *matrices*, *lists*, and *data frames*.

Objects that transform data and perform data operations/statistics/visualizations etc. are called *functions*.

Data types

It is important to note that all R data objects have a property called **type** (sometimes refered to as data types), which describes the type of data that is stored within the object. The most common data types in R are:

- "numeric" (numbers)
- "character" (text strings)
- "logical" (TRUE/FALSE)
- "factor" (categorical)

Data storage objects

Scalars

Scalars are the simplest objects in R. A scalar is just a single value. Remember when we assigned the text string 'Alfred Pennyworth' to the `Batmans_butler` object? When we did that we created a scalar data object that contained a single text string. Scalars can be a single value of any data type.

```
##################
####  Create R Objects
##################

#############
### scalars
#############

scalar1 <- 'this is a scalar'
scalar2 <- 104
scalar3 <- 5 + 6.5    # evaluates to the single value 11.5
scalar4 <- '4'
```

Scalars do have a specific type. In the example above, `scalar1` is a character, `scalar2` and `scalar3` are both numeric. What is the data type for `scalar4`?

If you are uncertain about the type of any R object the R function [4] `typeof()` will come in handy. Lets

[4]Functions are essentially machines that take inputs (objects, values) (also known as **arguments**) and produce something in return (objects, plots, tables, files). Functions have a common **syntax**: the name of the function is followed by parentheses-within which the arguments are entered. We will learn how to write our own R functions in a later module!

work through the code below.

```
typeof(scalar4)    # returns: character

## what is this type?
scalar5 <- TRUE
typeof(scalar5)    # returns: logical
```

Again, scalars can contain any of the most common data types:

- numeric (2, 8.2)
- character ('a', 'scalar')
- logical (`TRUE`, `FALSE`)
- factor (factor("a",levels="a")) [NOTE: we'll cover factors in more detail later. Scalar factors are not really useful!]

What happens when we try and add `scalar2` and `scalar4`? Think about the types of each scalar. Run the code below to find out.

```
## what happens when we run this line of code? Think about the types.
scalar_2 + scalar_4
```

This error can be common in R. Remember that `scalar4` is a character, R cannot perform mathematical operations on characters. Double check your types with the `typeof()` function.

Vectors

Vectors are combinations of scalars stored as a single object.

NOTE: strictly speaking, there are no true scalars in R- scalars are actually just vectors with 1 element!

Let's create some vectors:

```
#############
### VECTORS
#############

vector1 <- c(1.1, 2.1, 3.1, 4)
vector2 <- c('a', 'b', 'c')
vector3 <- c(1, 'a', 2, 'b')
vector4 <- c(TRUE, 'a', 1)
vector5 <- c(TRUE, 1.2, FALSE)
```

There are many different ways to create a vector. You'll learn more about that later in the bootcamp. Check out some of the vectors you just created. Each vector is composed of several scalars. Each of these scalars is an **element** of the vector. You can still check the types of the objects with the `typeof()` function (try it yourself).

What happened with `vector2` and `vector4`? Looking at the script, these vectors appear to contain a mix of numeric and character types, but the resulting vector is all characters!

This is because *all elements of a vector must be the same type.* R will attempt to convert (coerce) all elements to a single type. Check the contents of `vector5`. The types of each element are logical, numeric, logical. In this case R has decided to convert the logical elements to numeric. `TRUE` maps to 1 and `FALSE` maps to 0.

In the code above we use another function called `c()`. You can think of it as 'c for combine'. It takes several objects, and combines them together into a vector. We can even use `c()` to combine scalar (or vector) objects, like below.

```
a <- 1
b <- 2
c <- c(3,4)

d.vec <- c(a, b, c)
d.vec
```

```
## [1] 1 2 3 4
```

`d.vec` is a vector with 3 elements: 1, 2, 3 and 4. Alternatively, we could create the vector "d.vec" with the following code:

```
d.vec <- c(1,2,3,4)             # another way to construct the vector "d.vec"
d.vec = c(1,2,3,4)              # the "equals" sign can also be an assignment operator
d.vec <- 1:4                    # we can use the colon operator as a shorthand for creating a sequence o
```

Now let's do some stuff with vectors!

```
length(d.vec)    # the "length()" function returns the number of elements in a vector

d1 <- d.vec           # copy the vector "d.vec"
d2 <- d.vec+3         # add 3 to all elements of the vector "d.vec"
d3 <- d1+d2           # elementwise addition
d4 <- d1+c(1,2)       # what does this do?

## inspect the objects by calling them in the console (or script window)
d1    # returns: 1 2 3
d2    # returns: 4 5 6
d3    # returns: 5 7 9
d4    # returns: 2 4 4
```

NOTE: the last command we ran, "d1+c(1,2)", produced a ***warning message****. Remember to take warning messages seriously- a warning message means "the command ran but the results might not make sense"! In this case, the warning was that we tried to add two vectors of different length. R's default is to repeat (recycle) the shorter vector until it matches the length of the longer vector.*

Matrix objects in R

Matrix objects are just a bunch of vectors grouped together! To form a matrix, the vectors must be of the same length and the same **type**. If different types are provided R will attempt to coerce the elements into a common type. A matrix has two **dimensions**: rows and columns.

Let's make our first matrix. One simple way to make a matrix is just by joining two vectors using the function `cbind()` (bind vectors or matrices together by column) or `rbind()` (bind vectors or matrices together by row)

```
#############
### MATRICES
#############

d.mat <- cbind(d1,d2)        # create a matrix by binding vectors, with vector d1 as column 1 and d2 as
d.mat
```

```
##      d1 d2
## [1,]  1  4
## [2,]  2  5
## [3,]  3  6
## [4,]   4   7
```

```
class(d.mat)    # confirm that the new object "d.mat" is a matrix!
```

```
## [1] "matrix"
```

And there are other ways to make a matrix in R. For instance, we can use the function “matrix”:

```
d.mat <- matrix(c(1,2,3,4,5,6),nrow=3,ncol=2)        # create matrix another way
d.mat
```

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

```
d.mat <- matrix(c(1,2,3,4,5,6),nrow=3,ncol=2,byrow=T)        # create matrix another way
d.mat
```

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

```
d.mat <- rbind(c(1,4),c(2,5),c(3,6))        # create matrix another way
d.mat
```

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

We can do math with matrices too:

```
d.mat + 2
```

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

```
d.mat/sum(d.mat)
```

```
##            [,1]      [,2]
## [1,] 0.04761905 0.1904762
## [2,] 0.09523810 0.2380952
## [3,] 0.14285714 0.2857143
```

Arrays

We can structure information in more than two dimensions using objects of the “array” class. Again, all elements of an array must be of the same data type.

```
############
### ARRAYS!
############

d.array=array(0,dim=c(3,2,4))       # create 3 by 2 by 4 array full of zeros
d.array                                   # see what it looks like
```

```
## , , 1
```

```
##
##      [,1] [,2]
## [1,]    0    0
## [2,]    0    0
## [3,]    0    0
##
## , , 2
##
##      [,1] [,2]
## [1,]    0    0
## [2,]    0    0
## [3,]    0    0
##
## , , 3
##
##      [,1] [,2]
## [1,]    0    0
## [2,]    0    0
## [3,]    0    0
##
## , , 4
##
##      [,1] [,2]
## [1,]    0    0
## [2,]    0    0
## [3,]    0    0
```

```
d.mat=matrix(1:6,nrow=3)
d.array[,,1]=d.mat                      # enter d as the first slice of the array
d.array[,,2]=d.mat*2                    # enter d*2 as the second slide...
d.array[,,3]=d.mat*3
d.array[,,4]=d.mat*4
d.array                                   # view the array
```

```
## , , 1
##
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
##
## , , 2
##
##      [,1] [,2]
## [1,]    2    8
## [2,]    4   10
## [3,]    6   12
##
## , , 3
##
##      [,1] [,2]
## [1,]    3   12
## [2,]    6   15
## [3,]    9   18
##
```

```
## , , 4
##
##      [,1] [,2]
## [1,]    4   16
## [2,]    8   20
## [3,]   12   24
```

```
d.array[1,2,4]  # view an element of the array
```

```
## [1] 16
```

```
d.array[1,,]    # view a slice of the array
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    4    8   12   16
```

Lists

List objects are far more general than either matrices or arrays. **List** objects are just a bunch of potentially unrelated objects grouped together! The elements of a list don't even need to be the same length or the same **type**. The elements of a list can be vectors, arrays, matrices, functions, or other lists - they could be literally any R object.

Let's make our first list:

```
#############
### LISTS
#############

d.list <- list()        # create empty list
d.list[[1]] <- c(1,2,3)     # note the double brackets- this is one way to reference a specified list e
d.list[[2]] <- c(4,5)
d.list[[3]] <- "Alfred Pennyworth"
d.list
```

```
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] 4 5
##
## [[3]]
## [1] "Alfred Pennyworth"
```

Data frames (a very special kind of list)

Data frame objects are a special type of **list** in which each element of the list is a *vector* object. Importantly, in a data frame object, *all of the component vectors must have the same number of elements.* The jth element (vector object) of a data frame is also known as the jth *column.* The *row* of a data frame refers to the ith element of each of the j vectors. Data frames superficially resemble matrices, since both types of objects have two **dimensions**: rows and columns. However, data frames are really much more general than matrices; in data frames, unlike matrices, the columns of a data frame can represent different **types** (i.e., character, logical, numeric, factor), and can thereby code for very different types of information!

Data frames are the basic data storage structure in R. You can think of a data frame like a spreadsheet. Each row of the the data frame represents a different observation, and each column represents a different measurement taken on that observation unit.

Let's make our first data frame. We will use the function "data.frame()".

```
#############
### DATA FRAMES
#############
d.df <- data.frame(d1=c(1,2,3),d2=c(4,5,6))        # create a 'data frame' with two columns. Each colum
d.df
```

Now we have a data frame with three observation units and two measurements (variables).

Alternatively, we can convert a matrix to a data frame:

```
d.df=data.frame(d.mat)        # create data frame another way - directly from a matrix
d.df
```

We can use the "names()" function to see the names of each element of a list or data frame (column of a data frame), or to change these names!

```
names(d.df)          # view or change column names
```

```
## [1] "X1" "X2"
```

```
names(d.df)=c("meas_1","meas_2")        # provide new names for columns
d.df
```

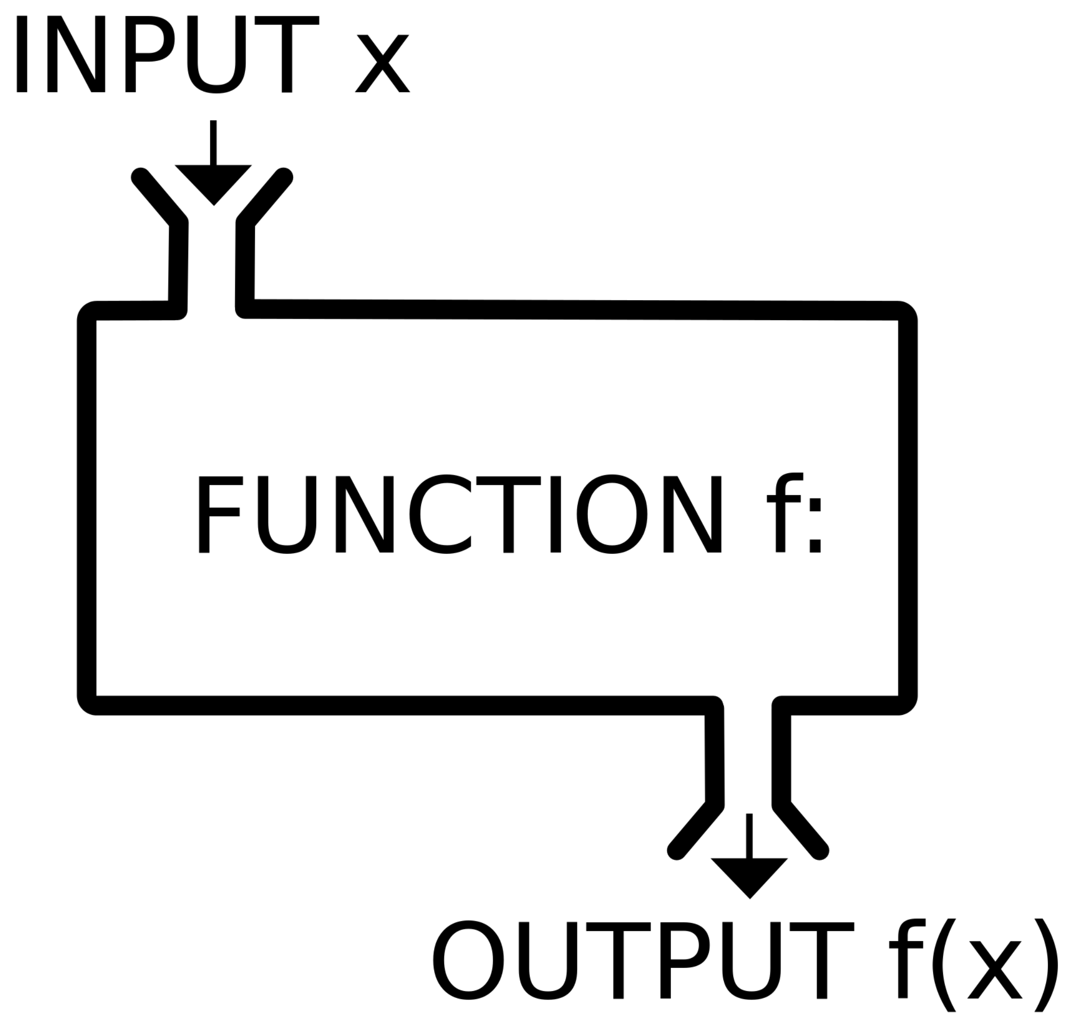
And we can view or change the row names of a data frame (or matrix) using the "rownames()" function:

```
rownames(d.df) <- c("obs1","obs2","obs3")
d.df
```

Functions

Functions are an essential building block in R. Without functions we would need to write out every bit of logic for everything we want to accomplish in R. We've already used a few functions (`typeof`, `c`) without fully understanding what they are. That is about to change!

Functions are essentially machines that take any inputs (objects, values) (also known as **arguments**) and produce something in return (objects, plots, tables, files). Functions have a common **syntax**: the name of the function is followed by parentheses- within which the arguments are entered. The basic syntax looks like the code snippet below.



```
## function syntax
functionName(input1=[user input #1], input2=[user input #2])
```

Let's use another function called `sum()`. This function takes a set of several values and computes the sum of those numbers.

```r
#############
### functions
#############

sum(1, 2, 3, 10)      # returns: 15

## sum can be used with one of the vectors we created
sum(vector1)          # returns: 10.3
```

If you ever forget the arguments of a function, or how it works there are several ways to get help. There is a built in function called `help()` that will takes a function name as an argument, and looks up the help documents for the given function. The `?` is an alternative to the `help` function.

```
help(sum)
?sum    # this is an alternative to 'help(sum)'!
```

We will revisit functions in more detail in the "Programming" submodule.

Making up data!

In this section, we will 'make up' a bunch of data objects. In the next submodule we'll practice working with real data!

Generating sequences of numbers

One task that comes up a lot is generating sequences of numbers. We certainly don't want to do this entirely by hand! Here are some shortcuts:

```
#############
### MAKING UP DATA!
#############

#######
# Sequences

1:10                        # sequence from 1 to 10
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
10:1                        # reverse the order
```

```
##  [1] 10  9  8  7  6  5  4  3  2  1
```

```
rev(1:10)                   # a different way to reverse the order
```

```
##  [1] 10  9  8  7  6  5  4  3  2  1
```

```
seq(from=1,to=10,by=1)      # equivalent to 1:10
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
seq(10,1,-1)                # equivalent to 10:1
```

```
##  [1] 10  9  8  7  6  5  4  3  2  1
```

```
seq(1,10,length=10)         # equivalent to 1:10
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
seq(0,1,length=10)          # sequence of length 10 between 0 and 1
```

```
##  [1] 0.0000000 0.1111111 0.2222222 0.3333333 0.4444444 0.5555556 0.6666667
##  [8] 0.7777778 0.8888889 1.0000000
```

Another task is to group regular recurring sequences together:

```
##############
# Repeating sequences

rep(0,times=3)                # repeat 0 three times
```

```
## [1] 0 0 0
```

```
rep(1:3,times=2)              # repeat 1:3 two times
```

```
## [1] 1 2 3 1 2 3
```

```
rep(1:3,each=2)               # repeat each element of 1:3 two times
```

```
## [1] 1 1 2 2 3 3
```

And finally, we can fill up a vector with random numbers using one of R's built in **random number generators**:

```
###########
# Random numbers

z <- rnorm(10)                # 10 realizations from std. normal
z
```

```
##  [1]  1.4795197 -1.7838162  0.1315966  0.5138481  1.0085172  0.6331399
##  [7]  0.5216227  1.3700235 -0.3705176 -1.3403783
```

```
y <- rnorm(10,mean=-2,sd=4)           # 10 realizations from N(-2,4^2)
y
```

```
##  [1] -4.0611974 -2.1935246 -4.3780492 -7.3999831 -3.0583962 -0.4632406
##  [7] -5.5560278  0.8435765  0.6901333 -8.4543477
```

```
rbinom(5,size=3,prob=.5)                # 5 realizations from Binom(3,0.5)
```

```
## [1] 2 1 1 0 2
```

```
rbinom(5,3,.1)                # 5 realizations from Binom(3,0.1)
```

```
## [1] 0 1 2 0 1
```

```
rbinom(5,3,.8)                # 5 realizations from Binom(3,0.8)
```

```
## [1] 2 3 2 3 2
```

```
rbinom(5,1:5,0.5)        # simulations from binomial w/ diff number of trials
```

```
## [1] 1 0 2 2 3
```

```
rbinom(5,1:5,seq(.1,.9,length=5))    # simulations from diff number of trials and probs
```

```
## [1] 1 1 2 3 4
```

```
runif(10)                 # 10 standard uniform random variates
```

```
##  [1] 0.15556279 0.57328189 0.68111016 0.17062397 0.09333731 0.71890979
##  [7] 0.72510000 0.38278121 0.87282089 0.40241357
```

```
runif(10,min=-1,max=1)        # 10 uniform random variates on [-1,1]
```

```
##  [1]  0.84440635 -0.55410388 -0.29910667 -0.03945426 -0.17479479
##  [6] -0.98847877  0.75835911 -0.65356953  0.21197508  0.09870945
```

```
sample(1:10,size=5,replace=TRUE)         # 5 rvs from discrete unif. on 1:10.
```

```
## [1]  7  4  2  5 10
```

```
sample(1:10,size=5,replace=TRUE,prob=(1:10)/sum(1:10)) # 5 rvs from discrete pmf w/ varying probs.
```

```
## [1]  4  8 10  7 10
```

And finally, we can make up a fake data frame using some of the tricks we just learned!

```
############
# Make up an entire data frame!


my.data <- data.frame(
  Obs.Id = 1:100,
  Treatment = rep(c("A","B","C","D","E"),each=20),
  Block = rep(1:20,times=5),
  Germination = rpois(100,lambda=rep(c(1,5,4,7,1),each=20)),
  AvgHeight = rnorm(100,mean=rep(c(10,30,31,25,35,7),each=20))
)
head(my.data)

summary(my.data)    # Use the "summary()" function to summarize each column in the data frame.
```

```
##      Obs.Id        Treatment     Block        Germination
##  Min.   :  1.00   A:20      Min.   : 1.00   Min.   : 0.00
##  1st Qu.: 25.75   B:20      1st Qu.: 5.75   1st Qu.: 1.00
##  Median : 50.50   C:20      Median :10.50   Median : 3.00
##  Mean   : 50.50   D:20      Mean   :10.50   Mean   : 3.87
##  3rd Qu.: 75.25   E:20      3rd Qu.:15.25   3rd Qu.: 6.00
##  Max.   :100.00             Max.   :20.00   Max.   :12.00
##    AvgHeight
##  Min.   : 8.466
##  1st Qu.:24.253
##  Median :29.798
##  Mean   :26.119
##  3rd Qu.:31.684
##  Max.   :36.841
```

NOTE: the “head()” function displays only the first few rows of a data frame, making it easier to look at. Displaying all 100 rows is a little distracting, and never mind if you had thousands of observations!

Accessing, indexing and subsetting data

In any data analysis project, we will need to manage, manipulate and process your data objects. To do this, we need to be able to access elements and subsets of our data objects.

To access a particular element of a vector, we just enclose the element(s) we want in brackets:

```
############
### Accessing, indexing and subsetting data
############

# X[i]          access the ith element of X

d.vec <- 2:10
d.vec
```

```
## [1]  2  3  4  5  6  7  8  9 10
```

```
d.vec[3]
```

```
## [1] 4
```

```
d.vec[c(1,5)]
```

```
## [1] 2 6
```

```
d.vec[-3]
```

```
## [1]  2  3  5  6  7  8  9 10
```

It is even more intuitive to access elements of a **named vector**:

```
d.vec <- 1:4
names(d.vec) <- c("fred","sally","mimi","terrence")
d.vec
```

```
##      fred     sally      mimi terrence
##         1         2         3        4
```

```
d.vec["terrence"]
```

```
## terrence
##        4
```

```
d.vec[c("sally","fred")]
```

```
## sally  fred
##     2     1
```

We can similarly subset matrices using brackets, but we need to specify the rows AND columns we're interested in extracting!

```
# X[a,b]       access row a, column b element of matrix/data frame X
# X[,b]        access column b of matrix/data frame X
# X[a,]        access row a of matrix/data frame X

d <- matrix(1:6,3,2)
d
```

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

```
d[,2]                # 2nd column of d
```

```
## [1] 4 5 6
```

```
d[2,]                # 2nd row of d
```

```
## [1] 2 5
```

```
d[2:3,]        # 2nd and 3rd rows of d in a matrix
```

```
##      [,1] [,2]
## [1,]    2    5
## [2,]    3    6
```

Subsetting works much the same for lists and data frames. However, now we can reference data frame columns (variables) and other list elements using the dollar sign operator:

```
# $            access component of an object (data frame or list)
# Z_list[[i]]  access the ith element of list Z

d.df=my.data[21:30,]  # only take 10 observations
d.df

## many ways of accessing a particular column of a data frame
d.df$Germination           # Subsetting a data frame
```

```
##  [1] 11  7  5  5  5  5  5  4  5  4
```

```
d.df[["Germination"]]      # same thing!
```

```
##  [1] 11  7  5  5  5  5  5  4  5  4
```

```
d.df[,4]                   # same thing!
```

```
##  [1] 11  7  5  5  5  5  5  4  5  4
```

```
d.df[[4]]                  # same thing!
```

```
##  [1] 11  7  5  5  5  5  5  4  5  4
```

```
d.df[,"Germination"]       # same thing!
```

```
##  [1] 11  7  5  5  5  5  5  4  5  4
```

```
d.df["Germination"]        # same thing!  (or is it...)


d.df$AvgHeight
```

```
##  [1] 31.85986 29.44906 29.97378 31.04416 30.06897 30.33944 29.97128
##  [8] 30.53673 30.83508 29.45402
```

```
d.df$AvgHeight[3]  # subset an element of a data frame
```

```
## [1] 29.97378
```

```
# Data frame can be indexed just like a matrix
d.df[,2]
```

```
##  [1] B B B B B B B B B B
## Levels: A B C D E
```

```
d.df[2,]
d.df[,2:3]
d.df[2,2:3]
```

Here are some useful R functions for exploring data objects:

```
###############
# Other data exploration tricks in R

length(d2)         # Obtain length (# elements) of vector d2
dim(d.mat)         # Obtain dimensions of matrix or array
summary(my.data)   # summarize columns in a data frame.
names(my.data)     # get names of variables in a data frame (or names of elements in a named vector)
nrow(my.data)      # get number of rows/observations in a data frame
```

```r
ncol(my.data)      # get number of columns/variables in a data frame
str(my.data)       # look at the "internals" of an object (useful for making sense of complex objects!)
```

Basic mathematical operations

Here are some basic mathematical operations you can do in R:

```r
2+3               # addition

6-10              # subtraction

2.5*33            # multiplication

4/5               # division

2^3               # exponentiation

sqrt(9)           # square root

8^(1/3)           # cube root

exp(3)            # antilog

log(20.08554)     # natural logarithm (but it's possible to change the base)

log10(147.9108)   # common logarithm

log(147.9108, base=10)

factorial(5)      # factorial

21 %% 5           # modulus
```

Challenge exercises!

1. Create a 3 by 2 matrix equivalent to d.mat by binding rows (HINT: use the "rbind()" function), using the following as rows:

```r
############
### CHALLENGE EXERCISES
############

### Challenge 1: Create a 3 by 2 matrix equivalent to d.mat by binding rows

# (HINT: use the "rbind()" function), using the following as rows:

c(1,4)
```

```
## [1] 1 4
```

```r
c(2,5)
```

```
## [1] 2 5
```

```r
c(3,6)
```

```
## [1] 3 6
```

```r
### Challenge 2: Is d.mat[-c(1,2),] a matrix or a vector? HINT: you can use the "class()" function to h

### Challenge 3: Create a new matrix 'd.mat2' by converting directly from data frame d.df, and using on

### Challenge 4: Create a 3 by 1 (3 rows, 1 column) matrix called 'd.mat3' with elements c(1,2,3). How

### Challenge 5: Take your matrix 'd.mat2' and convert it to a vector containing all elements in the ma

### Challenge 6: Create a list named 'd.list' that is composed of a vector: 1:3, a matrix: matrix(1:6,n

### Challenge 7: Extract (subset) the 2nd row of the 3rd matrix in the array in 'd.list' (the list crea

### Challenge 8: (extra challenging!) Create a data frame named 'df_spatial' that contains 25 locations

###     HINT: you don't need to type each location in by hand!
```

2. Is d.mat[-c(1,2),] a matrix or a vector? HINT: you can use the "class()" function to help out.
3. Create a new matrix 'd.mat2' by converting directly from data frame d.df, and using only columns 3 to 5 of the object 'd.df'.
4. Create a 3 by 1 (3 rows, 1 column) matrix called 'd.mat3' with elements c(1,2,3). How is this matrix different from a vector (e.g., created by running the command '1:3')? HINT: use the "dim()" function, which returns the dimensionality of an object.
5. Take your matrix 'd.mat2' and convert it to a vector containing all elements in the matrix. HINT: use the "as.vector()" function.

6: Create a list named 'd.list' that is composed of a vector: 1:3, a matrix: matrix(1:6,nrow=3,ncol=2), and an array: array(1:24,dim=c(3,2,4)).

7: Extract (subset) the 2nd row of the 3rd matrix in the array in 'd.list' (the list created in challenge 6 above).

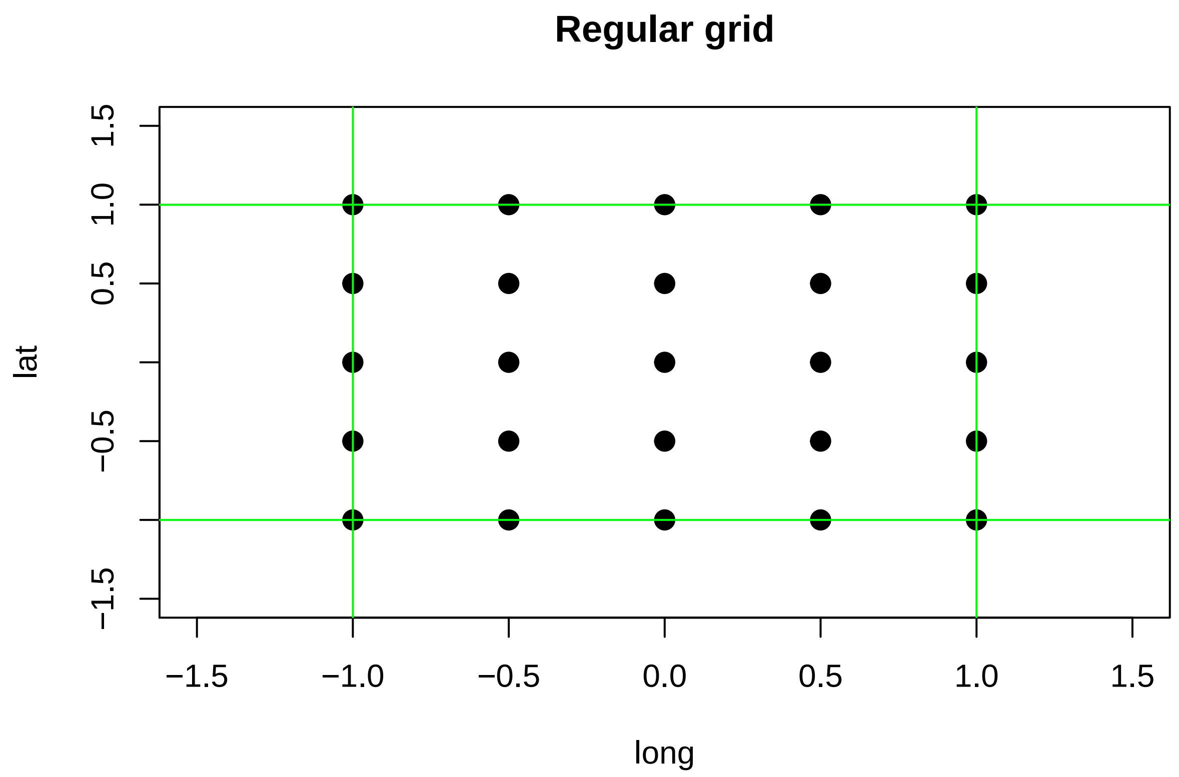
8: (extra challenging!) Create a data frame named 'df_spatial' that contains 25 locations, with 'long' and 'lat' as the column names (25 rows/observations, 2 columns/variables). These locations should describe a 5 by 5 regular grid with extent long: [-1,1] and lat: [-1,1].

HINT: you don't need to type each location in by hand!

To make sure you got it right, use the following commands in R:

Don't worry if you don't understand the code- we'll talk about plotting soon enough!

```r
########
# Code for visualizing the results from challenge problem 8

plot(x=df_spatial$long, y=df_spatial$lat, main="Regular grid",xlab="long",ylab="lat",xlim=c(-1.5,1.5),y
abline(v=c(-1,1),h=c(-1,1),col="green",lwd=1)
```



It should look something like this (above)!

Finally: some more advanced exploration of functions

Here is some code to help you get more familiar with some of the less intuitive aspects of R functions:

```
#################
# More advanced exploration of functions!

########
## first class functions

## assign functions to values
sum2 <- sum
sum2(1, 2, 3, 10)     # use the value sum2 as a function!

## use functions as arguments in other functions
## lets compute the average length of some of the vectors we've created
## this should return 3.66
mean(c(length(vector1), length(vector2), length(vector3)))
```

Let's walk through this last line of code together. We are use 3 different functions: `c`, `length`, `mean`. `length` is calculating the length of each vector. We are then taking the results of length and creating a vector with the `c` functions. The result of this is then used as the argument for the `mean` function.

Throughout your R journey you will see these properties used many times. Maybe not so much when first starting. But they are important concepts. Especially the fact that the R has first-class functions. Functions are often used as arguments to other functions as a shortcut. It saves the intermediate step of saving the results to another variable, then providing that variable as an argument. It also tends to save memory, as we are no longer storing that variable in memory.

–go to next module–