Module 1.1

Getting started: working with objects

September 2018

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 2018 R bootcamps! Let's get started!!

Load script for module #1.1

1. Click here to download the script! Save the script to a convenient folder on your laptop. I recommend making a new folder to store all your scripts from this bootcamp!
2. Load your script in RStudio. To do this, open RStudio and click on the folder icon in the toolbar at the top and load your script. Your RStudio interface should be divided into four quadrants. On the top left is your **script**, on the bottom left is your **console**, on the top right is your **Environment**, and on the bottom left are some helper interfaces that let you get help, load packages, view plots, etc. Basically, RStudio provides all the functionality you need to develop, debug and execute R code!

Let's get started!!!

Explore the R console

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

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 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?
5. Now let's create our first **object** in R. Objects are assigned a value using R's **assignment operator**, which looks like a left arrow ("<-"). Type the following statement:

```
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 storing a value of "Alfred Pennyworth".

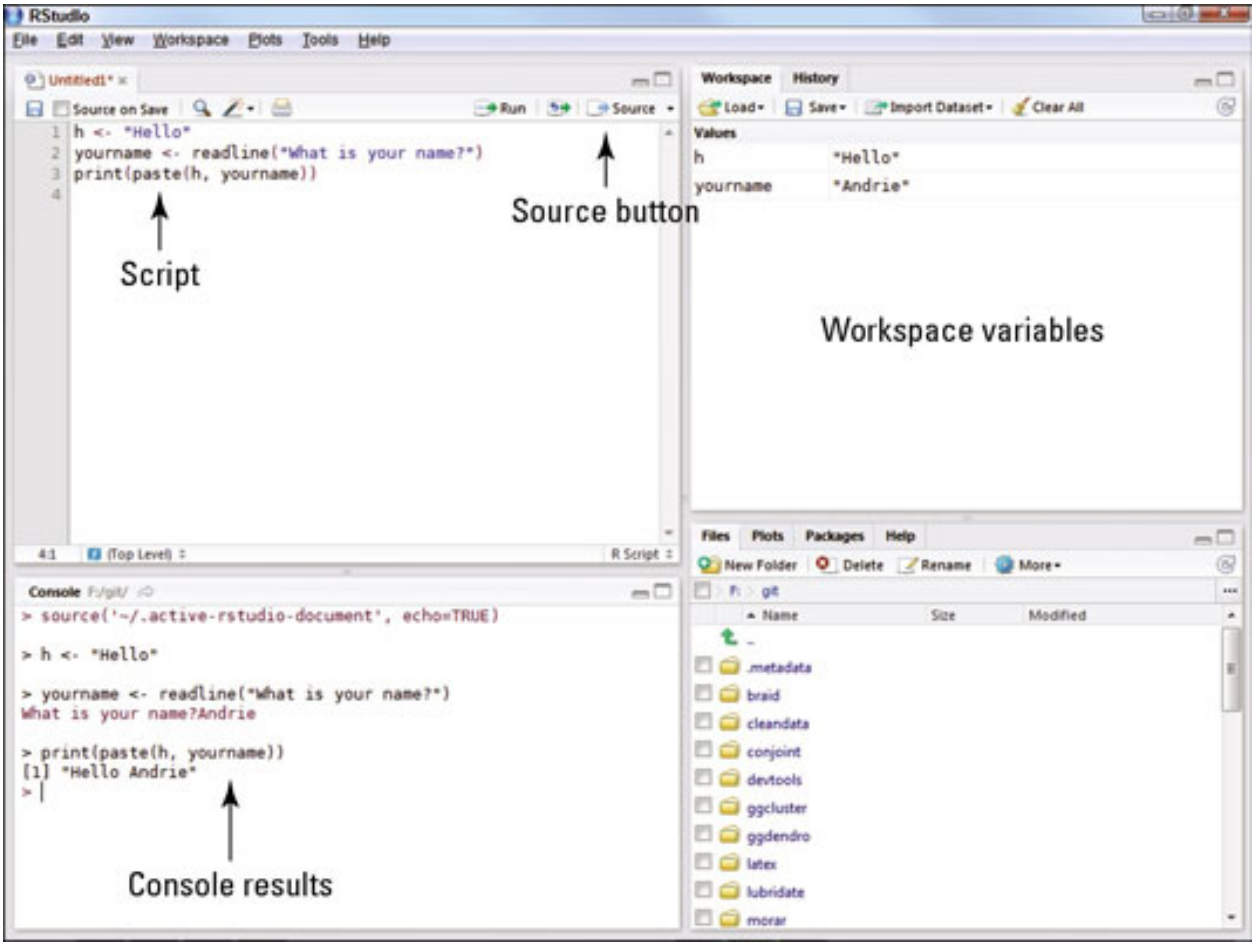
RStudio
File Edit View Workspace Plots Tools Help
Untitled1*
Source on Save
Run
Source
1 h <- "Hello"
2 yourname <- readline("What is your name?")
3 print(paste(h, yourname))
4
Source button
Script
Workspace
History
Load
Save
Import Dataset
Clear All
Values
h "Hello"
yourname "Andrie"
Workspace variables
4:1
(Top Level)
R Script
Console
> 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"
>
Console results
Files
Plots
Packages
Help
New Folder
Delete
Rename
More
git
Name
Size
Modified
_metadata
braid
cleandata
conjoint
devtools
ggcluster
ggdendro
latex
lubridate
morar

Figure 1:

6. Type "print(Batmans_butler)" into the R console and hit "Enter". What happens? RStudio has a useful auto-type feature, which can save you lots of time and headaches. After you've typed the first couple letters, RStudio will suggest "Batmans_butler" and you can just hit "Enter" to complete the object name!

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.

NOTE: Actually, 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.

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**. The top left quadrant of the RStudio interface is dedicated to developing R scripts. The file you downloaded at the beginning of this module is an R script.

The first few lines (commands) of the script 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.*

1. Click on the top of the script (first line of comments). To **run** a line of code, press the "Ctrl" key (or the "Apple" 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 a machine.
2. Click on a blank line in the script and type "2+2". Now hit "Ctrl+Enter" (or "Command+Enter") to run the line of code.

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

Demonstration: making up data in R!

NOTE: all R code here is also provided in the "Module_1.R" script that you've already loaded.

Clear the workspace

Let's start with a blank workspace! To do that, you can use the following command:

```
##############
# Clear the workspace

rm(list=ls())
```

Create R Objects

Let's create a bunch of new objects! Run the following lines:

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

```
a <- 1
b <- 2
c <- 3
```

Aside: R functions

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!

```
[name of function]([inputs])   # function syntax
```

For example...

```
###################
#### Try using an R Function

sum(1,2,3,10)     # "sum()" is an R function for computing the sum of a bunch of numbers
```

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

In this case, "sum" is the name of a function. The arguments are a set of values. The function takes these arguments, adds them up, and returns the sum of all the values.

To get help with a function, use the "help" function:

```
help(sum)    # get help for the function "sum()"

?sum    # alternative shortcut for getting help!
```

Manage your R workspace

One way to see what objects are in your workspace is to use an **R function** called "ls" (list):

```
###################
##### Manage your R workspace
###################

ls()                  # View contents of your workspace
```

```
## [1] "a" "b" "c"
```

```
?ls                   # View help pertaining to function â€~lsâ€
```

You can remove objects using another R function called "rm" (remove)

```
test <- 14+a      # make a new object
test
```

```
## [1] 15
```

```
ls()
```

```
## [1] "a"    "b"    "c"    "test"
```

```
rm(Batmans_butler,test)             # remove extraneous variables
```

```
## Warning in rm(Batmans_butler, test): object 'Batmans_butler' not found
```

```
ls()
```

```
## [1] "a" "b" "c"
```

Vector objects in R

Vector objects are just a bunch of objects grouped together in an orderly line. Each member of a vector is called an **element**.

Let's make our first vector:

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

d.vec <- c(a,b,c)            # d is now a â€~vectorâ€ in R
d.vec
```

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

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

```
d.vec <- c(1,2,3)             # another way to construct the vector "d.vec"
d.vec = c(1,2,3)            # the "equals" sign can also be an assignment operator
```

Now let's do some stuff with vectors!

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

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

```
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?
```

```
## Warning in d1 + c(1, 2): longer object length is not a multiple of shorter
## object length
```

```
d1
```

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

```
d2
```

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

```
d3
```

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

```
d4
```

```
## [1] 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 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!

NOTE: to form a matrix, the vectors must be of the same length and the same **class** (we'll get into what this means shortly). 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
d.mat
```

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

```
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
```

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
```

'Data frame' objects in R

Data frame objects are just a bunch of vectors grouped together! In this way they seem a lot like matrices! However, they are really much more general than matrices. In data frames, all the vectors must be the same length, but they no longer need to be the same **class**. A data frame, like a matrix, has two **dimensions**: rows and columns.

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 unit and each column represents a different

measurement taken on that observation unit.

NOTE: All R objects have a **property** called **class**, which describes the type of object it is. Common classes in R are "numeric" (numbers and vectors of numbers), "character" (text strings and vectors of text strings), "logical" (TRUE/FALSE data), and "factor" (categorical descriptors)

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 â€~data frameâ€ with two columns
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 variable names, or to change the variable 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 using the "rownames()" function:

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

'List' objects in R

List objects are just a bunch of objects grouped together! In this way they seem a lot like matrices and data frames! However, they are really much more general than either matrices or data frames. In lists, the objects don't even need to be the same length, never mind the same **class**. They don't even need to be vectors- they could be any type of R object. Each element of a list could be absolutely anything!

Let's make our first list:

```
#############
### LISTS
#############
d.list <- list()        # create empty list
d.list[[1]] <- c(1,2,3)
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"
```

Filling in vectors with numbers (making up data!)

One task that comes up a lot is making 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]  0.26285678  0.31171353 -2.82959063 -1.13954406  0.52445845
##  [6] -1.32344584 -0.07495631 -0.88928573 -0.11602894  1.43507171
```

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

```
##  [1]  -6.424339  -1.242659 -10.514327  -7.979934   4.874224   2.362126
##  [7]  -5.746620   4.290625  -1.308140  -5.343863
```

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

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

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

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

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

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

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

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

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

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

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

```
##  [1] 0.022919355 0.817908317 0.917509557 0.926777417 0.467131087
##  [6] 0.592509349 0.652660947 0.264929503 0.139455327 0.001982785
```

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

```
##  [1]  0.89087101 -0.04355084  0.68465463 -0.24101980  0.29597716
##  [6] -0.94667871 -0.42826993  0.91591796 -0.77516754  0.59248399
```

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

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

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

```
## [1]  8 10 10  7  3
```

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   : 3.50
##  3rd Qu.: 75.25   E:20      3rd Qu.:15.25   3rd Qu.: 5.25
##  Max.   :100.00             Max.   :20.00   Max.   :12.00
##    AvgHeight
##  Min.   : 7.155
##  1st Qu.:24.045
##  Median :29.854
##  Mean   :26.106
##  3rd Qu.:32.249
##  Max.   :36.563
```

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!

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 data frames and lists. However, now we can reference columns (variables) or 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


d.df$Germination      # Subsetting a data frame
```

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

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

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

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

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

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

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

```
d.df$AvgHeight
```

```
##  [1] 30.79969 28.69673 32.77038 29.65433 31.00650 32.81460 30.82770
##  [8] 32.88133 30.35020 29.09404
```

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

```
## [1] 32.77038
```

```
# Columns and rows of data frame
d.df$Treatment
```

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

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

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

```
d.df[,2:3]
cbind(ID=d.df$Obs.Id,Height=d.df$AvgHeight)    # make new data frame
```

```
##       ID   Height
##  [1,] 21 30.79969
##  [2,] 22 28.69673
##  [3,] 23 32.77038
##  [4,] 24 29.65433
##  [5,] 25 31.00650
##  [6,] 26 32.81460
##  [7,] 27 30.82770
##  [8,] 28 32.88133
##  [9,] 29 30.35020
## [10,] 30 29.09404
```

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

Here's a list of 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
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!)
```

We can structure information in more than two dimensions using objects of the "array" class.

```
############
### 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]
```

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

Basic mathematical operations

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

```
2+3             # addition
```

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

```
6-10            # subtraction
```

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

```
2.5*33          # multiplication
```

```
## [1] 82.5
```

```
4/5             # division
```

```
## [1] 0.8
```

```
2^3             # exponentiation
```

```
## [1] 8
```

```
sqrt(9)         # square root
```

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

```
8^(1/3)         # cube root
```

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

```
exp(3)          # antilog
```

```
## [1] 20.08554
```

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

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

```
log10(147.9108) # common logarithm
```

```
## [1] 2.17
```

```
log(147.9108, base=10)
```

```
## [1] 2.17
```

```
factorial(5)     # factorial
```

```
## [1] 120
```

```
21 %% 5          # modulus
```

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

Challenge exercises!

1. Create a 3 by 2 matrix equivalent to d.mat by binding rows, using the following as rows:

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

c(1,4)
```

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

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

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

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

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

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 includes 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!

```
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)
```

–go to next module–