MAGIST

Geog 413

Module 1 Exercise 2

Objects in R

1. Objects within R are items that are stored in the workspace. Objects are most often variables, but they may also be functions as we will see. You will often create new variables and datasets in R as your work through materials in Geog 413.

2. R has a number of built-in variables, such as pi. You can find the value that pi represents directly from the R-console, by typing

```
> pi
```

[1] 3.141593 - and this is the result.

More formally, you can ask R to print the value of pi

```
> print(pi)
```

```
[1] 3.141593
```

3. You might use pi to find the area of a circle using the expression area = pi*r^2, where r represents the radius of the circle. Here we use the R-console rather like an interactive calculator. (Remember we could place all these commands in a script as well.) So here we define the object r and then combine the objects pi and r to find the area.

```
> r  <- 5

> area <- pi*r^2

> area
```

```
[1] 78.53982
```

Here notice that R uses the operator <- for assigning a value to a variable. We input the value of r = 5 first, and then use the assignment operator to define the value of the object labelled area.

4. R has a variety of objects for holding different kinds of variables, and collections of variables in the form of datasets, or more formally data frames. The object r (above) represents a scalar, which is a vector of size 1. Vectors are themselves one-dimensional arrays that can hold numeric variables, character variables, or logical variables. The combine (or concatenate) function can be used to form a vector. Here are examples of how to create numeric, character and logical vectors, with 7, 3 and 6 elements (individual values) respectively:

```
> a <- c(1, 2, 5, 3, 6, -2, 4)

> b <- c(“one”, “two”, “three”)

> c <- c(TRUE, TRUE, TRUE, FALSE, TRUE, FALSE)
```

You can extract individual values of groups of values from vectors. For example, to extract the sixth observation in the vector a, you could type

```
> a6 <- a[6]
```

and now the value -2 is stored as the object a6. Check this with the print command

```
> print (a6)
[1] -2
```

Similarly, we could extract the first three values from the vector c

```
> c <- c(TRUE, TRUE, TRUE, FALSE, TRUE, FALSE)
> c123 <- c[c(1, 2, 3)]
> print (c123)
[1] TRUE TRUE TRUE
```

We can easily generate a sequence of numbers in a vector. Say you wanted the vector containing the consecutive integers from 1 to 10, then use the colon operator

```
> d <- c(1:10)
> print (d)
 [1]  1  2  3  4  5  6  7  8  9 10
```

A slightly more sophisticated sequence is also readily generated, say every second integer between 0 and 20

```
> intby2 <- seq(from=0, to=20, by=2)
> print (intby2)
 [1]  0  2  4  6  8 10 12 14 16 18 20
```

And the output does not need to be restricted to integers

```
> seq2 <- seq(from=1.0, to=2.0, length.out=10)
> print(seq2)
 [1] 1.000000 1.111111 1.222222 1.333333 1.444444 1.555556 1.666667
 [8] 1.777778 1.888889 2.000000
```

Use the help function to understand the syntax of the seq function

```
> help(seq)
```

5. A matrix is a 2-dimensional array in which each element is of the same type (numeric, character or logical). Matrices can be generated using the matrix function. For many functions in R, the arguments used in the function can be found using the args command

```
> args(matrix)
```

function (data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL) NULL

The first argument data specifies the elements to populate the matrix with, nrow and ncol specify the number of rows and columns of the matrix, the option byrow specifies that the matrix should be filled in by row (byrow=TRUE) or by column (byrow=FALSE). The default is by column. The option dimnames specifies row and column labels.

So let us create a matrix of 5 rows and 4 columns containing the consecutive integers 1 to 20

```
> matrix1 <- matrix(1:20, nrow=5, ncol=4)

> print (matrix1)
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    6   11   16
[2,]    2    7   12   17
[3,]    3    8   13   18
[4,]    4    9   14   19
[5,]    5   10   15   20
```

Let us create a 2x2 matrix filled by rows. Here 2x2 refers to the dimension of the matrix, the number of rows is given first and then the number of columns. So matrix1 (above) has dimension 5x4.

```
> cells <- c(1,26, 24, 68)

> rnames <- c("R1", "R2")

> cnames <- c("C1”, "C2”)

> matrix2 <- matrix(cells, nrow=2, ncol=2, byrow=T, dimnames=list(rnames, cnames))

> print(matrix2)
```

```
   C1 C2
R1  1 26
R2 24 68
```

Try this again filling the matrix by columns with the option byrow=F.

There are a couple of useful functions with numeric data organized in the form of a matrix:

# rowSums(x) sums the values of matrix x by row. The result could be read into another vector v1 as

```
> v1 <- rowSums(matrix1)

> print (v1)

[1] 34 38 42 46 50
```

# colSums(x) sums the values of the matrix x by column

```
> c1 <- colSums(matrix1)

> print(c1)

[1] 15 40 65 90
```

6. You can also generate arrays in R. These are like matrices but of dimension > 2. You can also generate factor variables. A factor is a character variable that meets a certain criterion. You might have data on 10 individuals with each of those individuals identified by name. You could make a factor variable coded 1 or 2 representing the sex (female or male) of each individual.

7. We can also create a data frame in R. A data frame is like a matrix, but the types of variables in a data frame do not have to be the same. You might think of a data frame as simply a dataset. Let's create a data frame here comprising four variables – city name, longitude, latitude and population. Note that the first variable is a character, while the others are numeric and so you cannot store these variables in a single matrix.

```
msa_name <- c("New York", "Los Angeles", "Chicago", "Dallas", "Houston", "Washington DC",
"Philadelphia", "Miami", "Atlanta", "Boston")

> lon <- c(-74.567, -118.752, -88.331, -97.091, -95.248, -76.939, -75.25, -80.106, -84.271, -71.033)

> lat <- c(40.68, 34.14, 42.18, 32.672, 29.422, 38.855, 39.883, 26.41, 34.068, 42.366)

> pop2020 <- c(20140470, 13200989, 9618502, 7636387, 7122240, 6385162, 6245051, 6183333, 6089815,
5)
```

Note that the population value for Boston is wrong. Easy to correct this

```
> pop2020[10] <- 4941632

> print(pop2020)

 [1] 20140470 13200989  9618502  7636387  7122240  6385162  6245051

 [8]  6183333  6089815  4941632
```

Now combine the four variables in a single data frame and check the output

```
> mydata1 <- data.frame(msa_name, lon, lat, pop2020)

> str(mydata1)

'data.frame':	10 obs. of  4 variables:

 $ msa_name: chr  "New York" "Los Angeles" "Chicago" "Dallas" ...

 $ lon     : num  -74.6 -118.8 -88.3 -97.1 -95.2 ...

 $ lat     : num  40.7 34.1 42.2 32.7 29.4 ...

 $ pop2020 : num  20140470 13200989 9618502 7636387 7122240 ...
```

```
> head(mydata1)
      msa_name      lon    lat  pop2020
1     New York  -74.567 40.680 20140470
2  Los Angeles -118.752 34.140 13200989
3      Chicago  -88.331 42.180  9618502
4       Dallas  -97.091 32.672  7636387
5      Houston  -95.248 29.422  7122240
6 Washington DC  -76.939 38.855  6385162
```

Reading and Writing External Files

Now, after all that data entry, what if you want to save your data file? Let's get in the practice of saving all our data files externally as .csv files (text files with variables separated by a comma = .csv). These files are easily read by a large number of different kinds of software. So to save the data frame mydata1

```
> write.csv(mydata1, "C:/classes/MAGIST413/R413/mydata1.csv", row.names=F)
```

Here I have to specify the path to the folder where I want to save the file, along with the filename to be stored. If you don't set row.names=F, R will add row numbers to your file. You usually don't want this. You can use help to check the full syntax of the write.csv command. Much easier if I remember to set my working directory first

```
> setwd("C:/classes/MAGIST413/R413")
> write.csv(mydata1, "mydata1.csv", row.names=F)
```

Now, let us say that we have opened a new session of R and we want to work on the data frame mydata1.csv. Easy, just read in the file again, after setting the working directory

```
> mydata1 <- read.csv("mydata1.csv")
```

Check the data frame

```
> mydata1
   X      msa_name      lon    lat  pop2020
1  1      New York  -74.567 40.680 20140470
2  2   Los Angeles -118.752 34.140 13200989
3  3       Chicago  -88.331 42.180  9618502
4  4        Dallas  -97.091 32.672  7636387
5  5       Houston  -95.248 29.422  7122240
6  6 Washington DC  -76.939 38.855  6385162
7  7  Philadelphia  -75.250 39.883  6245051
```

```
8   8        Miami  -80.106 26.410  6183333
9   9      Atlanta  -84.271 34.068  6089815
10 10       Boston  -71.033 42.366  4941632
```

You can read a .csv files from the web following this example

```
> newfile <- read.csv("http://www.example.com/data.csv")
```

A variety of formats other than .csv are readily handled within R.

**Note that when you exit RStudio (and R), you are asked if you want to save your workspace image. If you choose yes, your current workspace objects will be available to you the next time you open RStudio.

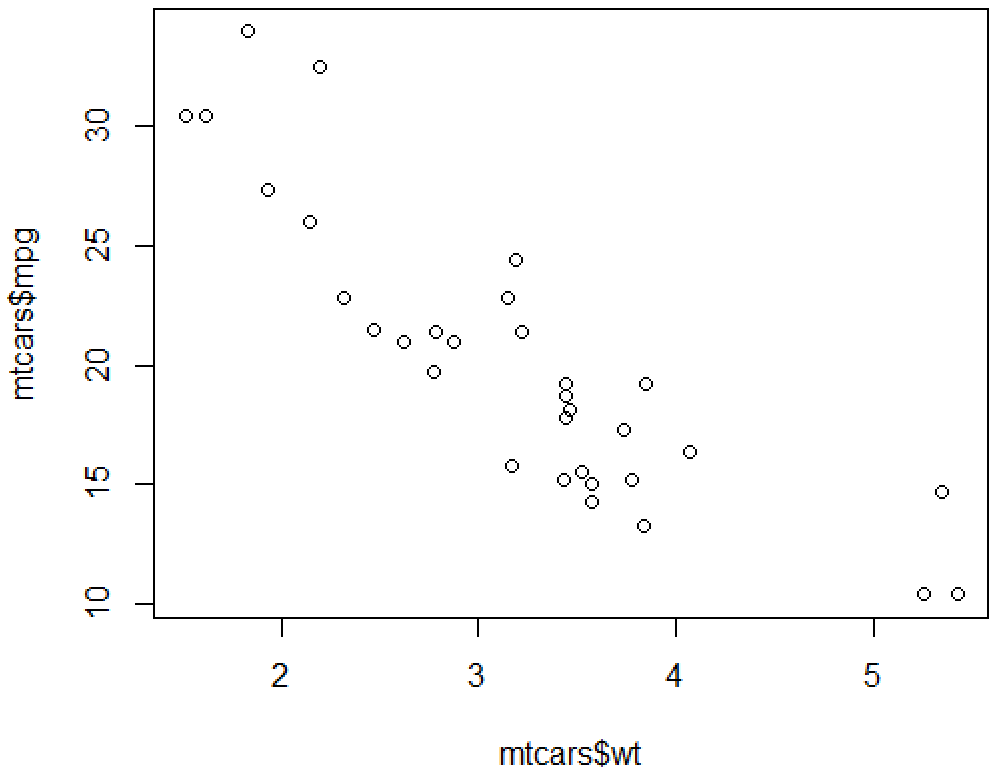
Some other Useful Functions

1. Using the attach function, can save a lot of typing. The data frame "mtcars" is preloaded in R. Look at the structure of this data frame

```
> str(mtcars)
'data.frame':     32 obs. of  11 variables:
 $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
 $ disp: num  160 160 108 258 360 ...
 $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
 $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
 $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
 $ qsec: num  16.5 17 18.6 19.4 17 ...
 $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
 $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
 $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
 $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

You might be interesting in exploring the relationship between the fuel efficiency of cars and their weight. We can look at this relationship in the form of a plot

```
> plot(mtcars$wt, mtcars$mpg)
```



What a nuisance always having to type the filename before the variable. You can get around this issue by attaching the mtvars data frame to the search path of R. Do this as

```
> attach(mtcars)
```

Then to plot, you can simply use the variable names

```
> plot(wt, mpg)
```

You can detach a data frame using the detach function

```
> detach(mtcars)
```

At any time, if you are unsure what objects you have defined in your workspace use the ls function

```
> ls()
```

You can delete an object using the remove command

```
> rm(object name)
```

You might want to create a new object by combining columns

```
> newobject1 <- cbind(lon, lat)
```

or by combining rows

```
> newobject2 < rbind(lon[1], lat[1])
```

Check out these new objects.

That’s it for now!