Introduction to R
Part 3: Getting to Know R

Aram Balagyozyan

Department of Operations and Information Management
Kania School of Management
The
University of Scranton

January 27, 2022

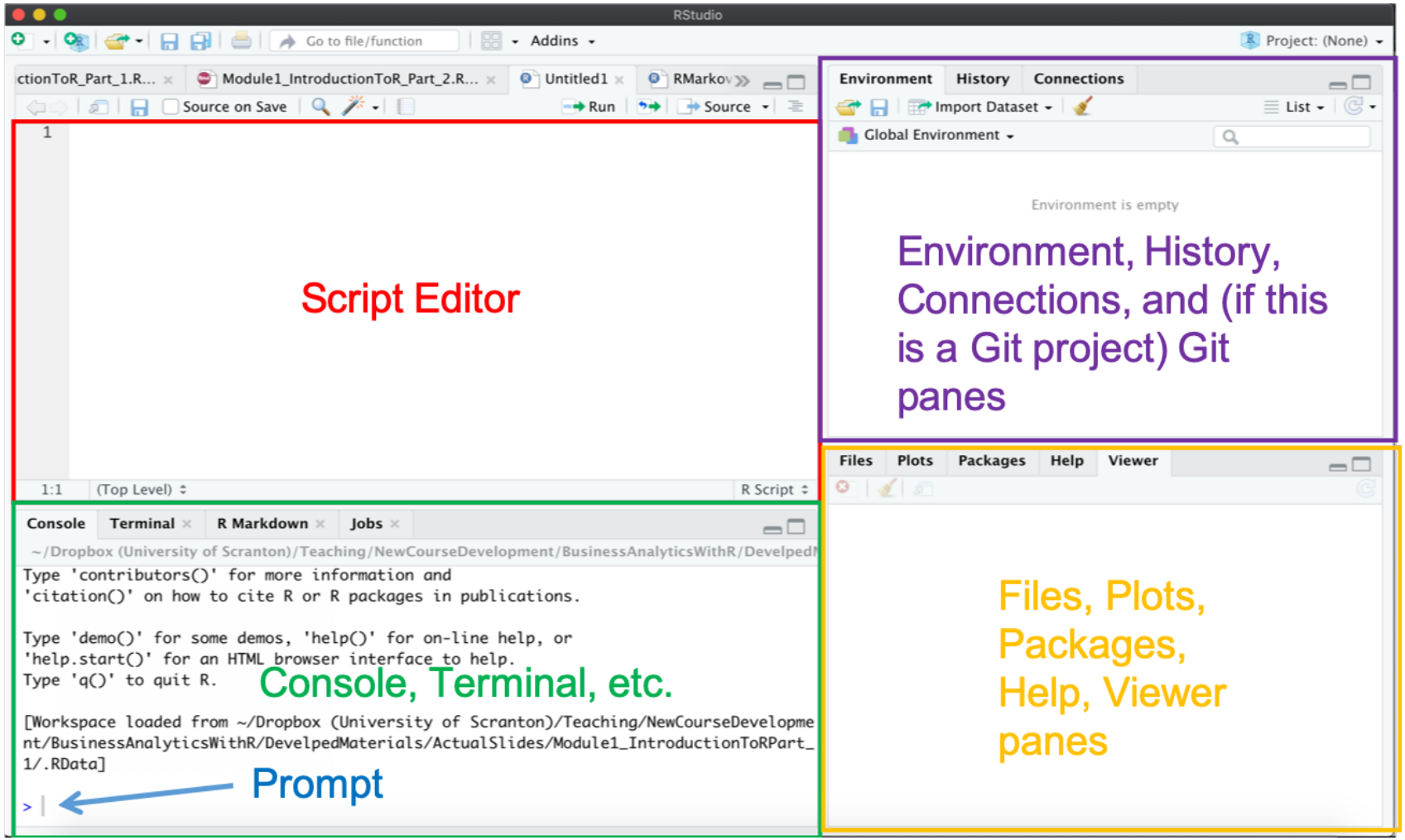
Outline

Outline

Basic Interaction with the R Console

- There are two basic ways in which one can interact with R.
  1. By entering commands directly in the R console where one carries out discreet calculations and interaction with R. This allows for easy interactive exploration of ideas.
  2. After the exploration phase, one tends to dump the sequence of R commands that lead to the solution into an R script file. These R script files then can be reused, for instance by asking R to execute all commands contained in the script file in a sequence at once.
- The interaction with the R console consists of typing some instructions followed by the ENTER key, and receiving back the result of this command.
- If you need to execute a selection of R code stored in a script file, select the code and hit CTRL + ENTER (or COMMAND + ENTER on a MAC).

Basic Interaction with the R Console



Basic Interaction with the R Console

- Try typing the following into the console and hit Enter

```
4 + 3 / 5 ^ 2
```

```
## [1] 4.12
```

- [1] in front of the output indicates the first element of the output object. This is particularly useful for results containing many values as these may spread over several lines of output. For now, simply ignore [1]

Basic Interaction with the R Console

- R has many useful base functions that can be utilized. For example

```
rnorm(4, mean = 10, sd=2)
```

```
## [1] 11.141417  8.894914  9.242263  9.139967
```

```
mean(sample(1:10,5))
```

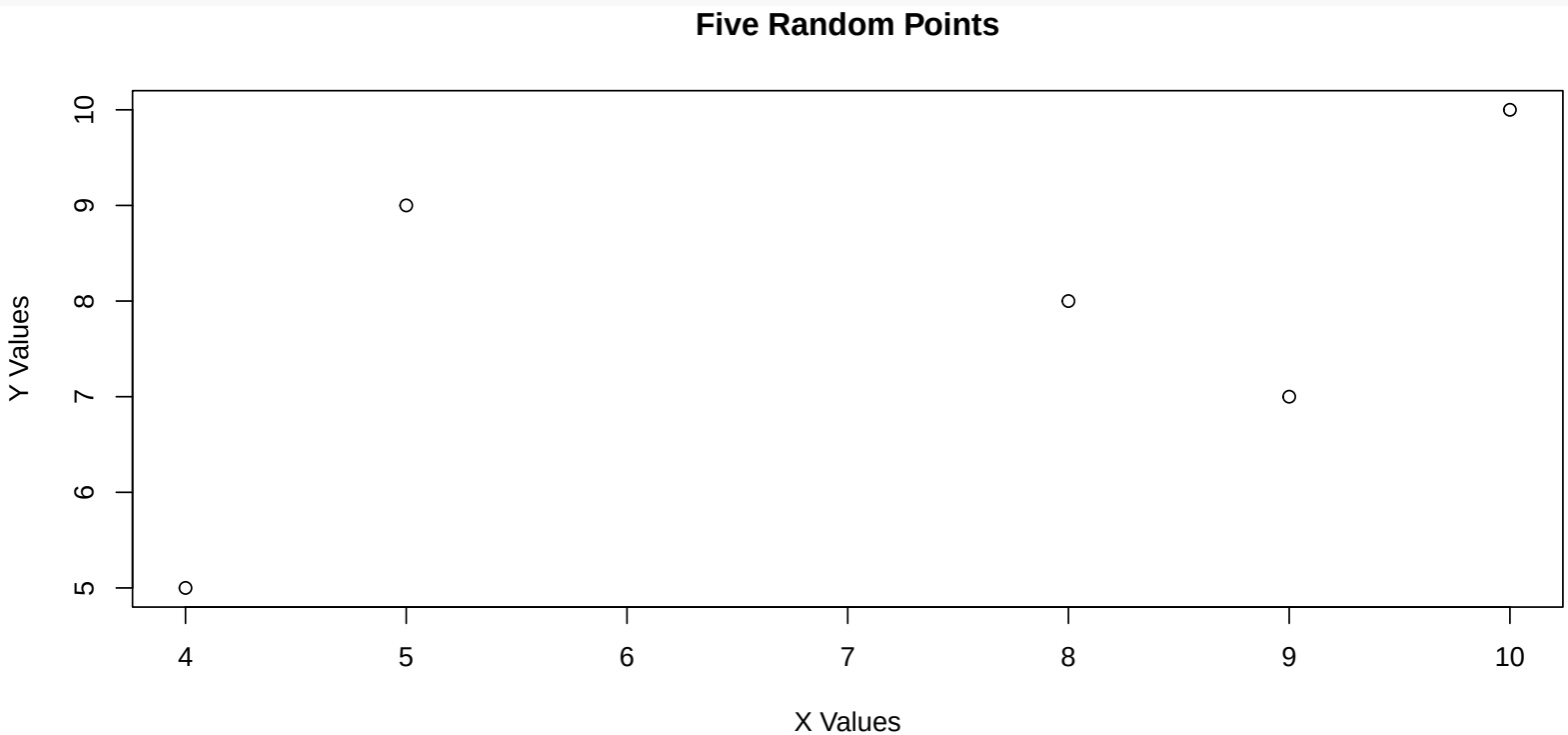
```
## [1] 6
```

- The first statement generates 4 random numbers from a normal distribution with mean 10 and standard deviation 2, while the second statement picks 5 random numbers between 1 and 10 and computes the average.

Basic Interaction with the R Console

- Another frequent task we often carry out at the R prompt is to generate some statistical graph of a dataset. For example:

```
plot(x=sample(1:10,5),y=sample(1:10,5),
main = "Five Random Points",
xlab = "X Values", ylab = "Y Values")
```



Outline

R Objects and Variables

- Everything is R is stored as an object that is most of the time associated with a variable name that allows us to refer to its content.
- Note that variable names in R are case sensitive.
- R objects may store diverse types of information. The simplest content is some value of one of R basic data types: numeric (numbers), character (text enclosed in single or double quotes), or logical (TURE/FALSE, in capital letters).
- Other, more complex data types may also be stored in R objects.
- Content (ie object) may be stored in a variable using the assignment operator <-

```
vat<-0.2
```

R Objects and Variables

- Assignment statements are destructive operation. By assigning some new content to an existing variable will overwrite the old content:

```
y <- 39
y
```

```
## [1] 39
```

```
y <- 43
y
```

```
## [1] 43
```

R Objects and Variables

- When a numerical expression is assigned to a variable, the variable stores the result of the evaluation of the expression, not the expression itself

```
z <- 5
w <- z^2
w
```

```
## [1] 25
```

- This means that the assignment operator evaluates whatever is given on the right hand side of the operator and assigns (stores) the result (an object of the same type) of this evaluation in the variable whose name is given on the left side.

R Objects and Variables

- You can assign an expression containing an object to self. In that case, the initial value of the object gets overwritten by the new value

```
z
```

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

```
z <- z^3
z
```

```
## [1] 125
```

- Every object you create will stay in the computer memory until you delete it (or exit R).

R Objects and Variables

- To list the objects currently in the memory issue the ls() function.

```
ls()
```

```
## [1] "w" "y" "z"
```

- To remove some objects from the memory use the rm() function.

```
rm(w,y)
```

- To remove all objects from the memory use both ls() and rm() functions.

```
rm(list=ls())
```

Outline

R Functions

- R functions are a special type of R object designed to carry out some operation. R functions, like mathematical functions, are applied to some set of arguments and produce a result.
- In R, both the arguments that we provide when we call the function and the result of the function execution are R objects whose type will depend on the function.
- R functions range from simple objects implementing some standard calculation, e.g. calculating square root of a number, to more complex functions that can obtain some model of a dataset, e.g. a neural network.
- The base R package already comes with an overwhelming set of functions available for us to use, but as we will see, the user can get more functions by downloading new packages or by creating new functions.

Base R Functions

- Below is an example of the `max()` function that as the name suggests returns the maximum value of the arguments supplied by the user when calling the function

```
x <-max(4,5,6,12,-4)
```

- In R, we frequently use function composition, a process of applying functions to the results of other functions. For example

```
max(sample(1:300,30))
```

```
## [1] 298
```

User-Defined Functions

- R also allows the user to create new functions. This is a useful feature, particularly when you want to automate certain tasks that you have to repeat over and over.
- R functions are objects that can be stored in a variable. The contents of these objects are the statements that, when executed, carry out the task for which the function was designed.
- The variable where we store the content of a function will act as the function name. Thus to create a new function we use the assignment operator to store the contents of the function in a variable (whose name will be the name of the function).

User-Defined Functions: An Example

- We often want to calculate the standard error of a mean associated with a set of values. By definition, the standard error of a sample mean is given by: standard error $= \sqrt{\frac{s^2}{n}}$ where s^2 is the sample variance and n is the sample size
- First we have to decide on the name of the function that hasn't yet been assigned to another object. Let's say, we decide to call the function `se`. In order to check whether an object named `se` exists, run the `exists()` function:

```
exists("se")
```

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

User-Defined Functions: An Example

- The fact that R returned `FALSE` means that we can use `se` for our function. Below is the syntax that one needs to use to create a function for standard error:

```
  se <- function(x) {
  v <- var(x)
  n <- length(x)
  return(sqrt(v/n))
  }
```

- Thus, to create a function object, you assign to its name something with the general form: `function(< set of parameters >) \{ <set of R instructions>\}`

```
mySample <- rnorm(100, mean=20, sd=4)
  se(mySample)
```

```
## [1] 0.3749389
```

Passing Arguments (Parameters) to a Function

- Sometimes you may want to create a function that carries default values for some parameters. E.g. suppose you create a function that takes in a value in meters and converts it to a user provided unit (say yards). However, if the user doesn't provide the conversion unit, you want the function to convert meters to inches by default. The following code chunk has an example of a function that accomplishes that:

```
convMeters <- function(val, to="inch") {
  mult <- switch(to,inch=39.3701,foot=3.28084,
                    yard=1.09361,mile=0.000621371,NA)
  if (is.na(mult))
     stop("Unknown target unit of length.")
  else return(val*mult)
}
```

Passing Arguments (Parameters) to a Function (continued)

- The above function is able to convert meters to inches, feet, yards, and miles. The user may omit the second argument as this has a default value ("inch"). This default value was established at the function creation by telling R not only the name of the parameter (`to`), but also a value that the parameter should take in case the user does not supply another value.

Passing Arguments (Parameters) to a Function (continued)

- There are few ways in which we can supply argument values in a function call (e.g. in calling the `convMeters`):
  a. by position (or by order):

```
convMeters(56.2,"yard")
```

```
## [1] 61.46088
```

  b. by name:

```
convMeters(val=56.2,to="yard")
```

```
## [1] 61.46088
```

  c. by a mix:

```
convMeters(56.2,to="yard")
```

```
## [1] 61.46088
```

Passing Arguments (Parameters) to a Function (continued)

- The benefit of calling by name has at least a couple of advantages:

1. When calling by name, the order in which parameters are supplied doesn't matter

```
convMeters(to="yard",val=56.2)
```

```
## [1] 61.46088
```

2. Say we have a function named `f` with 20 parameters, all but the first two having default values. Suppose we want to call the function but we want to supply a value different from the default for the tenth parameter named `tol`. With the possibility of calling by name we could do something like:

```
f(10,43.2,tol=0.25)
```

- This avoids having to supply all the values till the tenth argument in order to be able to use a value different from the default for this parameter.

Outline

Vectors

- The most basic object in R is vector. Even if you assign a number to a variable, you are creating a vector that contains a single element.
- A vector is an object that can store a set of values of the same base data type.
- The length of a vector object is the number of elements in it and can be obtained with the function `length()`.
- To create a vector in R, use the `c()` function:

```
v <- c(4, 7, 23.5, 76.2, 80)
v
```

```
## [1]  4.0  7.0 23.5 76.2 80.0
```

```
length(v)
```

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

```
mode(v)
```

```
## [1] "numeric"
```

Vectors

- All elements of a vector must belong to the same base data type. If that is not true, R will force it by type coercion.

```
v <- c(4, 7, 23.5, 76.2, 80, "rrt")
v
```

```
## [1] "4"    "7"    "23.5" "76.2" "80"   "rrt"
```

```
mode(v)
```

```
## [1] "character"
```

- All vectors may contain a special value called NA. This represents a missing value:

```
u <- c(4, 6, NA, 2)
u
```

```
## [1]  4  6 NA  2
```

Vectors

- One can access a particular element of a vector through an index between square brackets:

```
u[2]
```

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

- You can also change the value of one particular vector element by using the same indexing strategies:

```
u[3] <- 5
```

- R allows you to create empty vectors:

```
x <- vector()
```

Vectors

- The length of a vector can be changed by simply adding more elements to it using a previously non-existent index.

```
x[3] <- 45
x
```

```
## [1] NA NA 45
```

```
length(x)
```

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

```
x[10]
```

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

Vectors

- To shrink the size of a vector, you can take advantage of the fact that the assignment operation is destructive:

```
v <- c(45, 243, 78, 343, 445, 44, 56, 77)
v
```

```
## [1]  45 243  78 343 445  44  56  77
```

```
v <- c(v[5], v[7])
v
```

```
## [1] 445  56
```

Vectorization

- One of the most powerful aspects of the R language is the vectorization of several of its available functions. These functions can be applied directly to a vector of values producing an equal-sized vector of results:

```
v <- c(4, 7, 23.5, 76.2, 80)
v
```

```
## [1]  4.0  7.0 23.5 76.2 80.0
```

```
sqrt(v)
```

```
## [1] 2.000000 2.645751 4.847680 8.729261 8.944272
```

- You can also use this feature of R to carry out vector arithmetic:

```
v1 <- c(4, 6, 87)
v2 <- c(34, 32.4, 12)
v1 + v2
```

```
## [1] 38.0 38.4 99.0
```

Vectorization

- If the vectors do not have the same length, R will use a recycling rule by repeating the shorter vector until it reaches the size of the larger vector:

```
v1 <- c(4, 6, 8, 24)
v2 <- c(10, 2)
v1 + v2
```

```
## [1] 14  8 18 26
```

- It is just as if the vector `c(10,2)` was `c(10,2,10,2)`.
- If the lengths are not multiples, then a warning is issued, but the recycling still takes place:

```
v1 <- c(4, 6, 8, 24)
v2 <- c(10, 2,4)
v1 + v2
```

```
## Warning in v1 + v2: longer object length is not a multipl
## length
## [1] 14  8 12 34
```

Vectorization

- Single numbers are represented in R as vectors of length 1. Together with the recycling rule this is very handy for operations like the one shown below:

```
v1 <- c(4, 6, 8, 24)
2 * v1
```

```
## [1]  8 12 16 48
```

Outline

Factors

- Sometimes your data may take only few number of different categorical values. For instance, data on the gender of your customers will contain only values “m” or “f”.
- Factors are a native R object that provide an easy and compact form of handling such categorical data.
- Factors have levels that are the possible values they can take. In the above example, “m” and “f” are the levels of your gender data.
- Several graphical and summarization functions that we will explore in the future take advantage of this type of information.
- Factors allow you to use and show the values of your categorical variables as they are, which is clearly more interpretable for the user, while internally R stores these values as numeric codes that are considerably more memory efficient

Factors (continued)

- Consider the following vector:

```
g <- c("f", "m", "m", "m", "f", "m",
       "f", "m", "f", "f")
g
```

```
##  [1] "f" "m" "m" "m" "f" "m" "f" "m" "f" "f"
```

- You can transform this vector into a factor by:

```
g <- factor(g)
g
```

```
##  [1] f m m m f m f m f f
## Levels: f m
```

Factors (continued)

- Suppose you have five extra individuals whose sex information you want to store in another factor object. Suppose that they are all males. If you still want the factor object to have the same two levels as object g, you must use the following:

```
other.g <- factor(c("m", "m", "m", "m", "m"),
                  levels = c("f","m"))
other.g
```

```
## [1] m m m m m
## Levels: f m
```

- Without the levels argument the factor `other.g` would have a single level ("m").

Factors (continued)

- One of the many things you can do with factors is to count the occurrence of each possible value. Try this:

```
table(g)
```

```
## g
## f m
## 5 5
```

```
table(other.g)
```

```
## other.g
## f m
## 0 5
```

Factors (continued)

- The `table()` function can also be used to obtain cross-tabulations of several factors.
- Suppose that we have in another vector the age category of the ten individuals stored in vector a.

```
a <- factor(c('adult','adult','juvenile','juvenile',
              'adult', 'adult','adult','juvenile',
              'adult','juvenile'))
```

- We could cross tabulate these two factors as follows:

```
table(a, g)
```

```
##           g
## a          f m
##   adult    4 2
##   juvenile 1 3
```

Factors (continued)

- Sometimes we wish to calculate the marginal frequencies for this type of contingency table. The following gives you the totals for both the sex and the age factors of this dataset:

```
t <- table(a, g)
margin.table(t,1)
```

```
## a
##    adult juvenile
##        6        4
```

```
margin.table(t,2)
```

```
## g
## f m
## 5 5
```

Factors (continued)

- For relative frequencies with respect to the total count, use the `prop.table()` function:

```
prop.table(t)
```

```
##            g
## a             f   m
##   adult     0.4 0.2
##   juvenile 0.1 0.3
```

Factors (continued)

- For relative frequencies with respect to each margin use the following commands:

```
prop.table(t,1)
```

```
##           g
## a                    f         m
##   adult    0.6666667 0.3333333
##   juvenile 0.2500000 0.7500000
```

```
prop.table(t,2)
```

```
##           g
## a            f   m
##   adult    0.8 0.4
##   juvenile 0.2 0.6
```

Outline

Generating Sequences

- R has several facilities to generate different types of sequences. For instance, if you want to create a vector containing the integers between 1 and 100, you can simply type

```
x<-1:100
```

  which creates a vector called x containing 100 elements, the integers from 1 to 100.
- You may also generate decreasing sequences such as the following:

```
5:0
```

```
## [1] 5 4 3 2 1 0
```

- To generate sequences of real numbers incremented by a given number, you can use the function `seq()`:

```
 seq(-2, 1, 0.5)
```

```
## [1] -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0
```

- The above positional call is equivalent to `seq(from=-2, to=1, by=0.5)`

Generating Sequences (continued)

- `seq()` can be used to generate sequence with a given number of intermittent digits as follows:

```
seq(from = 1, to = 5, length = 4)
```

```
## [1] 1.000000 2.333333 3.666667 5.000000
```

- Another useful way of generating sequences is by using the `rep()` function:

```
rep(3,6)
```

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

```
rep("hi",3)
```

```
## [1] "hi" "hi" "hi"
```

```
rep(1:2,each=3)
```

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

Generating Sequences (continued)

- The function `gl()` can be used to generate sequences involving factors.

```
gl(3,5)
```

```
##  [1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
## Levels: 1 2 3
```

```
gl(2,5, labels=c("f","m"))
```

```
##  [1] f f f f f m m m m m
## Levels: f m
```

- The `gl(k,n)`, generates k levels of the factor with n repetitions each.

Generating Sequences (continued)

- R can be used to generate random sequences according to certain probability distributions (density functions).
- The random number generators in R can be written out generically as `rdist(n, par1, par2,...)`, where *dist* is the name of the probability distribution, `n` is the number of random numbers to generate, and `par1, par2,...` are the values of some parameters of the density function that may be required.

Generating Sequences (continued)

- For example:

```
rnorm(5)
```

```
## [1]  0.0462509  0.2184953 -0.8243917 -0.5781866  0.526720
```

```
rnorm(4,mean = 10,sd=3)
```

```
## [1]  7.359267 13.069935  6.195814 12.277823
```

```
rt(3,df=100)
```

```
## [1] -0.5841644  1.1990238  0.3728751
```

- R has many more random number generators, as well as other functions for obtaining the probability densities, the cumulative probability densities, and quantiles of those distributions.

Outline

Sub-Setting: Selecting Certain Elements of a Vector

- We have already seen examples of how to get one element of a vector by indicating its position inside square brackets. R also allows you to use vectors within the brackets. There are several types of index vectors. Logical index vectors extract the elements corresponding to true values. Let us see a concrete example:

```
x <- c(0, -3, 4, -1, 45, 90, -5)
x>0
```

```
## [1] FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE
```

- If we use the produced vector of logical values to index x, we get as a result the positions of x that correspond to the true values:

```
x[x > 0]
```

```
## [1]  4 45 90
```

Sub-Setting: Selecting Certain Elements of a Vector (cont.)

- Taking advantage of the logical operators available in R, you can use more complex logical index vectors, as for instance,

```
x[x<= -2|x> 20]
```

```
## [1] -3 45 90 -5
```

- or

```
x[x > 40 & x < 100]
```

```
## [1] 45 90
```

Sub-Setting: Selecting Certain Elements of a Vector (cont.)

- R also allows you to use a vector of integers, sequences, variable names to extract several elements from a vector. The numbers in the vector of indexes indicate the positions in the original vector to be extracted:

```
x[c(3,5)]
```

```
## [1]  4 45
```

```
x[3:5]
```

```
## [1]  4 -1 45
```

```
y<-c(3,5)
x[y]
```

```
## [1]  4 45
```

Sub-Setting: Selecting Certain Elements of a Vector (cont.)

- Alternatively, you can use a vector with negative indexes to indicate which elements are to be excluded from the selection:

```
x[-1]
```

```
## [1] -3  4 -1 45 90 -5
```

```
x[-c(4, 6)]
```

```
## [1]  0 -3  4 45 -5
```

```
x[-(1:3)]
```

```
## [1] -1 45 90 -5
```

Sub-Setting: Selecting Certain Elements of a Vector (cont.)

- Indexes can also be formed by a vector of strings, taking advantage of the fact that R allows you to name the elements of a vector:

```
heights<-c(John=7.1, Paul=7.0, George=7.5,
           Ringo=5.9)
heights
```

```
##   John   Paul George  Ringo
##    7.1    7.0    7.5    5.9
```

- The same can be accomplished using the `names()` function in R:

```
weights<-c(180, 182, 190, 175)
names(weights)<-c("John", "Paul", "George", "Ringo")
weights
```

```
##   John   Paul George  Ringo
##    180    182    190    175
```

Sub-Setting: Selecting Certain Elements of a Vector (cont.)

- The vectors can now be indexed using the names shown above:

```
heights[c("John", "Ringo")]
```

```
##  John Ringo
##   7.1   5.9
```

```
weights[c("Paul", "George")]
```

```
##   Paul George
##    182    190
```

Sub-Setting: Selecting Certain Elements of a Vector (cont.)

- Finally, leaving square brackets empty selects all elements of the vector:

```
heights[]<-200
heights
```

```
##   John   Paul George  Ringo
##    200    200    200    200
```

- Note that if instead we had typed `heights<-200`, R would delete vector `height`, create a new element `height`, and assign a single value of 200 to it.

```
heights<-200
heights
```

```
## [1] 200
```

Outline

Matrices

- Storing data elements in objects with more than one dimension can be very useful.
- Arrays store data elements in several dimensions.
- Matrices are a special case of arrays with two single dimensions.
- Arrays and matrices in R are nothing more than vectors with a particular attribute that is the *dimension*.

```
m <- c(45, 23, 66, 77, 33, 44, 56, 12, 78, 23)
dim(m)<-c(2, 5)
m
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   45   66   33   56   78
## [2,]   23   77   44   12   23
```

Matrices (continued)

- You could accomplish the above more simply by using the `matrix()` function:

```
m <- matrix(c(45, 23, 66, 77, 33, 44, 56, 12,
              78, 23), 2, 5)
m
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   45   66   33   56   78
## [2,]   23   77   44   12   23
```

- You could make the numbers spread by rows instead by typing

```
m <- matrix(c(45, 23, 66, 77, 33, 44, 56, 12,
              78, 23), 2, 5, byrow = TRUE)
m
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   45   23   66   77   33
## [2,]   44   56   12   78   23
```

Matrices (continued)

- You can access the elements of a matrix through a similar indexing scheme as in vectors, but this time with two indexes, with the first index indicating the row and the second index the column of the selected element :

```
m[2,3]
```

```
## [1] 12
```

- If you omit any dimension, R will return full columns or rows of the matrix:

```
m[1,]
```

```
## [1] 45 23 66 77 33
```

```
m[,4]
```

```
## [1] 77 78
```

Matrices (continued)

- Notice that, as a result of sub-setting, you may end up with a vector, as in the two examples above. If you still want the result to be a matrix, even though it is a matrix formed by a single line or column, you can use the following instead:

```
m[1, , drop = FALSE]
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   45   23   66   77   33
```

```
m[,4, drop = FALSE]
```

```
##      [,1]
## [1,]   77
## [2,]   78
```

Matrices (continued)

- Functions `cbind()` and `rbind()` may be used to join together two or more vectors or matrices, by columns or by rows, respectively.

```
m1 <- matrix(c(45, 23, 66, 77, 33, 44,
               56, 12, 78, 23), 2, 5)
cbind(c(4, 76), m1[, 4])
```

```
##      [,1] [,2]
## [1,]    4   56
## [2,]   76   12
```

```
m2<-matrix(rep(10, 20), 4, 5)
rbind(m1[1, ], m2[3, ])
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   45   66   33   56   78
## [2,]   10   10   10   10   10
```

Matrices (continued)

- You can also give names to the columns and rows of matrices, using the functions `colnames()` and `rownames()`, respectively. This facilitates memorizing the data positions and accessing certain elements of a matrix.

```
results <- matrix(c(10, 30, 40, 50,
                    43, 56, 21, 30),
                  2, 4, byrow = TRUE)
colnames(results) <- c("1qrt", "2qrt", "3qrt", "4qrt")
rownames(results) <- c("store1", "store2")
results["store1", ]
```

```
## 1qrt 2qrt 3qrt 4qrt
##   10   30   40   50
```

```
results["store2", c("1qrt", "4qrt")]
```

```
## 1qrt 4qrt
##   43   30
```

Arrays

- Arrays are extensions of matrices to more than two dimensions. This means that they have more than two indexes.Thus, whereas matrices are data "planes," arrays may be data "cubes," or have even more than 3 dimensions.
- Apart from this, they are similar to matrices and can be used in the same way.
- Similar to the `matrix()` function, there is an `array()` function to facilitate the creation of arrays.

Arrays (continued)

- The following is an example:

```
a <- array(1:24, dim = c(4, 3, 2))
a
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]   13   17   21
## [2,]   14   18   22
## [3,]   15   19   23
## [4,]   16   20   24
```

Arrays (continued)

- You can use the same indexing schemes to access elements of an array.

```
a[,2:3,1]
```

```
##      [,1] [,2]
## [1,]    5    9
## [2,]    6   10
## [3,]    7   11
## [4,]    8   12
```

```
a[c(2, 3), , -2]
```

```
##      [,1] [,2] [,3]
## [1,]    2    6   10
## [2,]    3    7   11
```

Arrays and Matrices (continued)

- The recycling and arithmetic rules also apply to matrices and arrays, although they are tricky to understand at times. Below are a few examples:

```
m1<- matrix(c(15, 7, 11, 13, 14, 20), 2, 3)
m2<- matrix(c(12, 65, 32, 7, 4, 78), 2, 3)
m1*3
```

```
##      [,1] [,2] [,3]
## [1,]   45   33   42
## [2,]   21   39   60
```

```
m1+m2
```

```
##      [,1] [,2] [,3]
## [1,]   27   43   18
## [2,]   72   20   98
```

- R also includes operators and functions for standard matrix algebra that have different rules. You may obtain more information on this by looking at Section 5 of the document "An Introduction to R" that comes with R.

Outline

Lists

- R lists consist of an ordered collection of objects known as their components.
- Unlike the elements of vectors, list components do not need to be of the same data type or length.
- The components of a list are always numbered and may also have a name attached to them.

```
 my.lst <- list(stud.id=34453,
                stud.name="John",
                stud.marks=c(14.3,12,15,19))
my.lst
```

```
## $stud.id
## [1] 34453
##
## $stud.name
## [1] "John"
##
## $stud.marks
## [1] 14.3 12.0 15.0 19.0
```

Lists (continued)

- You can extract the components of a list by using one of the few available ways. You can use either single square brackets [ ] or double square brackets, [[ ]], or a dollar sign, $.

```
my.lst[3]
```

```
## $stud.marks
## [1] 14.3 12.0 15.0 19.0
```

```
my.lst[[3]]
```

```
## [1] 14.3 12.0 15.0 19.0
```

```
my.lst$stud.marks
```

```
## [1] 14.3 12.0 15.0 19.0
```

- Notice the difference. `my.lst[3]` extracts a sub-list formed by the third component of my.lst. On the contrary, `my.lst[[3]]` or `my.lst$stud.marks` extracts the values of the first component (in this case, numbers).

Lists (continued)

- To confirm the above, run the following code:

```
mode(my.lst[3])
```

```
## [1] "list"
```

```
mode(my.lst[[3]])
```

```
## [1] "numeric"
```

```
mode(my.lst$stud.marks)
```

```
## [1] "numeric"
```

Lists (continued)

- The names of the components of a list can be manipulated as we did with the names of elements of vectors:

```
names(my.lst)
```

```
## [1] "stud.id"    "stud.name"  "stud.marks"
```

```
names(my.lst)<-c("id", "name", "marks")
my.lst
```

```
## $id
## [1] 34453
##
## $name
## [1] "John"
##
## $marks
## [1] 14.3 12.0 15.0 19.0
```

Lists (continued)

- Lists can be extended by adding new components to them:

```
my.lst$parents<-c("Anna","Patrick")
my.lst
```

```
## $id
## [1] 34453
##
## $name
## [1] "John"
##
## $marks
## [1] 14.3 12.0 15.0 19.0
##
## $parents
## [1] "Anna"    "Patrick"
```

Lists (continued)

- You can check the number of components of a list using the function `length()`:

```
length(my.lst)
```

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

- You can remove components of a list using the same approach as for vectors:

```
my.lst<-my.lst[-4]
```

Lists (continued)

- You can join (concatenate) two lists using the `c()` function

```
other <- list(age = 19, sex = "male")
lst<-c(my.lst,other)
lst
```

```
## $id
## [1] 34453
##
## $name
## [1] "John"
##
## $marks
## [1] 14.3 12.0 15.0 19.0
##
## $age
## [1] 19
##
## $sex
## [1] "male"
```

Lists (continued)

- Finally, you can convert list into a conventional vector using the `unlist()` function:

```
unlist(my.lst)
```

```
##      id    name  marks1  marks2  marks3  marks4
## "34453"  "John"  "14.3"    "12"    "15"    "19"
```

- Notice that this creates a vector with as many elements as there are data objects in a list.
- This will coerce different data types to a common data type, which means that most of the time you will end up with everything being character strings. Moreover, each element of this vector will have a name generated from the name of the list component that originated it.

Outline

Data Frames

- Data frames are the recommended and most commonly used data structure for storing data in R.
- They are similar to matrices in structure as they are also bi-dimensional. However, contrary to matrices, data frames may store data of a different type in each column. In this sense they are more similar to lists.
- We can think of each row of a data frame as an observation (or case), being described by a set of variables (the named columns or *fields* of the data frame).

Data Frames (continued)

- At some point, we will learn how to import large amount of data into the R environment and store it in a data frame. For the sake of an example, however, let's create a simple data frame using the `data.frame()` command.

```
my.df <- data.frame(student=c('John','Paul',
                              'Paul','George'),
                    subject=c('Physics','Chemistry',
                              'Physics','Chemistry'),
                    mark=c(82,37,52,100))
my.df
```

```
##   student   subject mark
## 1    John   Physics   82
## 2    Paul Chemistry   37
## 3    Paul   Physics   52
## 4  George Chemistry  100
```

Data Frames (continued)

- Elements of data frames can be accessed like a matrix:

```
my.df[3,2]
```

```
## [1] "Physics"
```

- Note that upon conversion, the `data.frame()` function by default coherces data stored in character strings into a factor format.

Data Frames (continued)

- Columns of a data frame can be accessed like both a matrix or list:

```
my.df[,2]
```

```
## [1] "Physics"   "Chemistry" "Physics"   "Chemistry"
```

```
my.df$subject
```

```
## [1] "Physics"   "Chemistry" "Physics"   "Chemistry"
```

```
my.df[[2]]
```

```
## [1] "Physics"   "Chemistry" "Physics"   "Chemistry"
```

Data Frames (continued)

- Again, note the difference between `my.df[[2]]` and `my.df[2]`

```
my.df[[2]]
```

```
## [1] "Physics"   "Chemistry" "Physics"   "Chemistry"
```

```
  mode(my.df[[2]])
```

```
## [1] "character"
```

```
my.df[2]
```

```
##     subject
## 1   Physics
## 2 Chemistry
## 3   Physics
## 4 Chemistry
```

```
 mode(my.df[2])
```

```
## [1] "list"
```

Data Frames (continued)

- You can perform some simple querying of the data in data frame as follows:

```
my.df[my.df$mark>70,]
```

```
##   student   subject mark
## 1    John   Physics   82
## 4  George Chemistry  100
```

```
my.df[my.df$student=="Paul", "subject"]
```

```
## [1] "Chemistry" "Physics"
```

```
my.df[my.df$student=="Paul", c("subject","mark")]
```

```
##     subject mark
## 2 Chemistry   37
## 3   Physics   52
```

Data Frames (continued)

- Another way of accessing certain rows of a data frame is to use the `subset()` function:

```
subset(my.df, mark>70)
```

```
##   student   subject mark
## 1    John   Physics   82
## 4  George Chemistry  100
```

```
subset(my.df, student=="Paul", c("subject","mark"))
```

```
##     subject mark
## 2 Chemistry   37
## 3   Physics   52
```

- Note, however, that if you wanted to select a subset in a dataframe and assign a certain set of values to it, the only way you could do it is by using the method on the previous slide.

```
my.df[my.df$student=="Paul", "mark"]<-
  my.df[my.df$student=="Paul", "mark"]+1
```

Data Frames (continued)

- You can add new columns to a data frame in the same way you did with lists. First check how many rows of data you have by running either `nrow()` or `dim()`, then add the column with appropriate number of rows:

```
   dim(my.df)
```

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

```
my.df$attendance<-c(100,90,95,100)
my.df
```

```
##   student   subject mark attendance
## 1    John   Physics   82        100
## 2    Paul Chemistry   38         90
## 3    Paul   Physics   53         95
## 4  George Chemistry  100        100
```

Data Frames (continued)

- Usually you will be reading your datasets into a data frame either from some file or from a database. You will seldom type the data using the `data.frame()` function as above. You will learn how to import external data into a dataframe in the next module.
- However, R has a simple spreadsheet-like interface that allows you to edit or create small data frames:

```
my.df<-edit(my.df)
new.data<-edit(data.frame())
```

Data Frames (continued)

- You can use `names()` function to check and/or change the name of the column of a data frame

```
names(my.df)
```

```
## [1] "student"    "subject"    "mark"       "attendance"
```

```
names(my.df)<-c('stu','sub','mrk','attend')
names(my.df)[4]<-'att'
my.df
```

```
##      stu       sub mrk att
## 1   John   Physics  82 100
## 2   Paul Chemistry  38  90
## 3   Paul   Physics  53  95
## 4 George Chemistry 100 100
```

Outline

Useful Extensions to Data Frames

- The packages **tibble** (Wickham et al., 2016) and **dplyr** (Wickham and Francois, 2015a) provide some useful extensions to data frames that are very convenient for many data manipulation tasks.
- The package **tibble** defines tibbles that can be regarded as "special" data frames.
- Tibbles change some of the behavior of "standard" data frames.
- The more important differences between a tibble and a data frame are:
  1. tibbles never change character columns into factors as data frames do by default;
  2. tibbles are more relaxed in terms of naming of the columns; and
  3. the printing methods of tibbles are more convenient particularly with large datasets.

Tibbles

- Tibbles can be created with the function `tibble()`.

```
library(tibble,warn.conflicts=FALSE)
dat <- tibble(TempCels = sample(-10:40, size=100,
                                replace=TRUE),
              TempFahr = TempCels*9/5 + 32,
              Location = rep(letters[1:2], each=50))
head(dat,5)
```

```
## # A tibble: 5 x 3
##   TempCels TempFahr Location
##      <int>    <dbl> <chr>
## 1        7     44.6 a
## 2       -2     28.4 a
## 3        3     37.4 a
## 4        3     37.4 a
## 5        1     33.8 a
```

Tibbles (continued)

- The `tibble()` function works in a similar way as the standard `data.frame()` function but, as you can observe, each column is calculated sequentially allowing you to use the values of previous columns, and character vectors are not converted into factors.
- Any standard data frame can be converted into a tibble. Consider the famous *Iris* dataset that is available directly in R and contains 150 rows with information (5 attributes) on variants of Iris plants

```
data(iris)
dim(iris)
```

```
## [1] 150   5
```

```
class(iris)
```

```
## [1] "data.frame"
```

Tibbles (continued)

- To convert the *Iris* data frame into a *tibble* use the following code:

```
library(tibble)
ir<-as_tibble(iris)
class(ir)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

```
head(ir,3)
```

```
## # A tibble: 3 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##          <dbl>       <dbl>        <dbl>       <dbl> <fct>
## 1          5.1         3.5          1.4         0.2 setosa
## 2          4.9         3            1.4         0.2 setosa
## 3          4.7         3.2          1.3         0.2 setosa
```

Tibbles (continued)

- Note that the new *ir* object is still of class **data.frame** but it also belongs to two new classes: **tbl_df** and **tbl**.
- The first (**tbl_df**) is the class of a *tibble*, while the second is a further generalization that allows us to look at a dataset independently of the data source used to store it.
- As you might have observed, the printing method of tibbles is more interesting than that of standard data frames as it provides more information and avoids over cluttering your screen. You can also get better control over what rows and columns of a *tibble* printed by using the `print()` method of objects of class **tbl_df**.

```
print(ir,n=2,width=Inf)
```

```
## # A tibble: 150 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##          <dbl>       <dbl>        <dbl>       <dbl> <fct>
## 1          5.1         3.5          1.4         0.2 setosa
## 2          4.9         3            1.4         0.2 setosa
## # ... with 148 more rows
```

Tibble Subsetting

- Another crucial difference between standard data frames and tibbles is subsetting. In standard data frames subsetting may sometimes lead to some puzzling results, particularly for newcomers to the R language. Check the following example

```
iris[1:3,"Petal.Length"]
```

```
## [1] 1.4 1.4 1.3
```

```
class(iris[1:3,"Petal.Length"])
```

```
## [1] "numeric"
```

- We sub-setted a data frame and we obtained as result a different data structure. With tibbles this never happens.

Tibble Subsetting

- Subsetting a tibble always results in a tibble.

```
ir[1:3,"Petal.Length"]
```

```
## # A tibble: 3 x 1
##   Petal.Length
##          <dbl>
## 1          1.4
## 2          1.4
## 3          1.3
```

```
class(ir[1:3,"Petal.Length"])
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

The **dplyr** Package

- Using **dplyr** you can create **tbl** objects connected with some data source, and once you do this you can mostly ignore this source as all data manipulation statements the package provides work the same way independently of the source.
- These *tbl* objects encapsulate the data source and provide a set of uniform data manipulation verbs irrespective of these sources.
- The package currently covers several data sources like standard data frames (in the form of tibbles), several database management systems, and several other sources.
- The main advantages of this package are:
  1. the encapsulation of the data source;
  2. providing a set of uniform data manipulation functions;
  3. the computational efficiency of the provided functions.

Querying Data Using the **dplyr** Package

- The **dplyr** package has many functions that allow easy manipulation of *tbl* objects.
- It provides two main functions for querying data stored in *tbl* objects.
- Function `select()` can be used to select a subset of columns of the dataset.
- Function `filter()` is used to select a subset of the rows.
- Suppose you wish to inspect the petal lengths and widths of the plants of the species Setosa. You could achieve that as follows:

```
library(dplyr,warn.conflicts=FALSE)
select(filter(ir,Species=="setosa"),
          Petal.Width,Petal.Length)
```

```
## # A tibble: 50 x 2
##    Petal.Width Petal.Length
##          <dbl>        <dbl>
##  1         0.2          1.4
##  2         0.2          1.4
##  3         0.2          1.3
```

Querying Data Using the **dplyr** Package

- In the above code, the == operator is a logical R operator that in this case checks if each element of *Species* is exactly equal to "Setosa" or not. It returns a vector of the same length as *Species* with each element being either TRUE or FALSE.
- For more logical operators in R see for example https://www.statmethods.net/management/operators.html.
- Note also that you may refer to the column names directly (without referencing the table name).
- The above code uses the traditional function composition by applying the `select` function to the output of the `filter` function.
- Function composition is a very nice concept but it often gets too difficult to understand the code if a lot of nesting composition is taking place.
- The **dplyr** package provides the pipe operator %>% to make our life easier in those cases.

Pipe Operator

- The `select` statement above could have been written as:

```
 filter(ir,Species == "setosa")
%>% select(Petal.Width,Petal.Length)
```

- The pipe operator is in effect a simple re-writing operator that can be applied to any R function, not only in this context (just need to activate the **dplyr** package prior to using).
- The idea is that the left-hand side of the operator is passed as the first argument of the function on the right-side of the operator.
- So `x %>% f(y)` is translated into `f(x,y)`.

Pipe Operator

- Statements that use the pipe operator are often significantly easier to "read" than with the standard function composition strategies.
- The above code can be read as: "filter the dataset *ir* by the setosa species and then show the respective petal width and length".
- This reading is more natural and more related with the way our brain thinks about these querying tasks.

Outline

Managing your Session

- When you are using R for complex tasks, the command line typing style of interaction becomes a bit limited and hard to replicate. In these situations it is more practical to write all your code in a text (script) file and then ask R to execute it.
- To produce such a file, you can use your favorite text editor (like Notepad, Emacs, etc.) or, in case you are using RStudio, you can use the script editor available in the File menu.
- After creating and saving the file (by naming it for example as mycode.R), you can either issue the following command at R prompt to execute **all** commands in the file

```
source('mycode.R')
```

or opening the file in the Rstudio script editor and executing a certain part of it (or a specific line) by selecting the part and hitting `Ctrl+Enter` (`Cmd+Enter` on a Mac)

Managing your Session

- When calling a source file (such as mycode.R) using the method on the previous slide, the file should either be located under the current working directory of the R session or the `source()` command should specify the full (or relative) path of the file.
- In the Windows versions of R, the easiest way to change the current working directory is through the option "Change directory" of the "File" menu.
- In Mac OS X versions there is an equivalent option under menu "Misc."
- In RStudio, you may use the option "Set working directory" of the "Session" menu.
- Regardless of the operating system, you may programatically check and change the current working directory by using the functions `getwd()` and `setwd()` respectively.

Managing your Session

- Sometimes, you may wish to save some of the existing objects for later use (such as some function you have typed in). The following example saves the objects named `f` and `my.dataset` in a file named "mysession.RData":

```
save(f,my.dataset,file='mysession.RData')
```

- Later, for instance in a new R session, you can load these objects by issuing

```
load('mysession.RData')
```

- You can also save all objects currently in R workspace by issuing

```
save.image()
```

  This command will save the workspace in a file named ".RData" in the current working directory. This file is automatically loaded when you run R again from this directory. This kind of effect can also be achieved by answering Yes when quitting R.

Outline

Common Problems with R Programming

- As you start to run R code, you are likely to run into problems. Don't worry, it happens to everyone. I have been writing R code for years, and every day I still write code that doesn't work!
- R is extremely picky:
  a. R code is case sensitive.
  b. R doesn't have an "intuition" for and doesn't understand misspelled words.
  c. Missplaced or unpaired characters such as quotation marks or parentheses can make all the difference.
- If you run the code and nothing happens, check the left-hand side of the console. If it displays a plus sign (+), it means that R doesn't think you've typed a complete expression and it is waiting for you to finish it. In those instances, it is usually easier to start from scratch by pressing `Esc` to abort processing the current command.

Common Problems with R Programming

- If you run the code and get an error message, make sure to carefully read it. Sometimes the answer to the problem will be buried there.
- If you don't understand the error message, try googling the error message, as it is likely someone else has had the same problem and has received help online.