An Introduction to R and Rstudio

Dr. Irene Vrbik

University of British Columbia Okanagan

irene.vrbik@ubc.ca

Term 2, 2018W

Background

- This lecture assumes you have downloaded R and Rstudio.
- If you haven't, please go back to the first days lecture to view instructions.
- Please stop me at any time if you have questions!

Background

- As mentioned last lecture, Rstudio has four panels:
    1. Console panel
    2. Source editor
    3. Environment panel
    4. Files, Plots, and Help panel
- Let's start by going through these in a little bit more detail.

Console panel

- The console panel is where R commands are executed.
- In this panel you see the input/output.
  - The input can either be typed in real time using the keyboard, or run from a R script (more on this soon)
  - The output is displayed to the screen.

Console panel

- To see how we can use the Console with keyboard input, simply navigate to the panel and start tying commands:
  - e.g. `2+ 4`
  - e.g. `print("Hello World")`
- Notice that R defaults functionality is to print the output of your command to the screen.

R scripts

- More commonly, we will be executing commands from an R script.
- An R script is a text document containing R commands.
- Apart from one-off calculations, it is always a good idea to save an R script (e.g. easy to share and reproduce you results)

R scripts

A very simple R script might look somthing like this:

```
# Simple addition
2 + 3
# Simple subtraction
2 - 3
# Simple multiplication
2*3
# Simple division
2/3
# Simple power
2^3
```

Comments

- Hashtags # indicate the start of a comment.
- Comments are not executed by R (i.e. they are ignored)
- Comments are used by the programmer to document and explain the code.
- Comments should give information about the code to the person reading it.
- A shortcut to commenting in Rstudio is shift + Command + C on a Mac and shift + Contrl + C on a PC

Commenting in Scripts

```r
# the following wont be executed
# 2 + 2
##### we can have # so many hashtags #somanyhashtags
2 + 2 # comments can appear after execute code
## [1] 4
```

Running code from scripts

- To run code from an R script, simply open the file (which will have a .R extension)
- Place your cursor on the line you wish to execute and press `../../Labs/01RIntro/run.png` or type SHIFT + ENTER
- If you want to run the entire script named myfile.R type `source("myfile.R")` into the console, or press `figure/RunAll.png` which appears in the drop-down menu of .)/../Labs/01RIntro

I will be using **knitr** to produce lecture and lab documentation.

```
# comments in purple italics
 2 + 3 # input
## [1] 5
```

- R input will be formated with highlighted text (colour will depend on the object), comments in purple italics, while the output is printed with two preceeding hashtags.
- The ## before the output provides an easy way for readers to copy and paste R source code (since output the output is masked as comments); you will not see the two hashtags on standard output in R and Rstudio.

- The [1] in the R output indicates that the adjacent element is the 1st element printed.
- For multiline outputs the number in the square brackets will indicate where the adjacent falls in term of the printed elements.

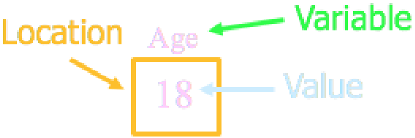
```r
longvector = c("this","is","a","long","vector","of",
    "words","that","will","print","on","multiple","lines")
longvector
```

```
##  [1] "this"     "is"       "a"        "long"     "vector"
##  [7] "words"    "that"     "will"     "print"    "on"
## [13] "lines"
```

For example [7]/[13] indicates that the first element of the second/third line is the 7th/13th element of `longvector`, repsectively.

Objects and Variables

- **Everything** you see or create in R is an object.
- This is an abstract term for anything that can be assignment to a varible.
- A variable is a name that refers to a location that stores a data



value.

IMPORTANT: The value at a location can change.

- Notice when you assign something to an object, R does not automatically print the result to the the screen.
- For that reason, when we save output to a variable, we will have to call the varaible name or use a print statment to see our result

```r
2 + 3

## [1] 5

x = 2+3
x

## [1] 5

# another alternative:
(x = 2+3)

## [1] 5
```

Data types

- Here's a list of some important data types used in R:
  1. Integer (Whole Numbers)
  2. Numeric (Real Numbers)
  3. Character
  4. Logical (True / False)
  5. Factor
  6. Complex (we don't focus on this type)

Numeric and Integer

- R will automatically assign the type to a variable once declared.
- Often the default storage of numbers in R is in as numeric.
- To force a variable to be an interger, we need to use the `as.integer()` function, or use the "L" suffix.

Numeric and Integer

```
numobj = 2 # numeric (not integer)
as.integer(numobj) # integer

## [1] 2

intobj = 2L # integer
```

Character

- A single letter or string of characters will be recorded as a `character` object.
- You can use double quotes or single quotes for all of these specifications (just be consistent)

```
# note our usage of quotes
charobj = "s"
strobj = 'string'
# badobj = "str' # unmatched quotes = bad
```

Aside

Auto-complete

Notice that R trys to help us out by autocompleting our quotes (i.e. when we type one quotation/parenthesis the closing one will appear automatically. If you don't like this feature, you can turn it off (but I suggest you keep it!)

Character

We can also treat numbers as characters:

```
x = c(1,2,3)
x
## [1] 1 2 3
x = as.character(x)
x # notice the quotes in the output
## [1] "1" "2" "3"
# PS - To change back to numeric use:
x = as.numeric(x)
x # no more quotes
## [1] 1 2 3
```

Logical

In R, `T` and `F` are reserved for logicals (i.e. True/False objects) and R reads `T` the same as `TRUE` or `F` the same as `FALSE`.

```
x = c(T, FALSE,TRUE, F) # notice we don't use quotes
x
```

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

Logical

- Oftentimes, True and False conditions are coded as 0 and 1s. To covert them to logical, use `as.logical()`.

```
x = c(0,1,1,0,0,0,1,0,0,1,1)
class(x)

## [1] "numeric"

x = as.logical(x)
class(x)

## [1] "logical"
```

Factor

- If you want a variable to be a factor type (so that different symbols are regarded as a level), use `var=factor(var)`.

```
notfactor = c("H","M", "M", "L", "H")
notfactor
```

```
## [1] "H" "M" "M" "L" "H"
```

```
fvec = factor(notfactor)
fvec
```

```
## [1] H M M L H
## Levels: H L M
```

Factor

- We also have the option to specify the labels to something more meaningful:

```
fvec = factor(x=notfactor, labels=c("High","Medium","Low"))
fvec

## [1] High   Low    Low    Medium High
## Levels: High Medium Low
```

Aside

Notation

- We say that `factor()` is a **function** for which we have specified two **arguments** (having argument names: `from`, `x` and `labels`).
- You can access the help file to see the details of this function using `?factor`
- Notice that is has other (optional) arguments that we haven't specified.

Factor

- The case my also arise that we need to specify levels not seen in the data set. In that case we can specify the `levels` argument:
- Using `letters` a built in vector containing the 26 letters of the alphabet;

```
x = c("o","q","h","n","s","b","u","d","p","r")
xlist = factor(x, levels=letters)
table(xlist)

## xlist
## a b c d e f g h i j k l m n o p q r s t u v w x y z
## 0 1 0 1 0 0 0 1 0 0 0 0 0 1 1 1 1 1 1 0 1 0 0 0 0 0
```

Aside

Built-in Constants

- On the previous slide we used the built-in constant `letters`
- Other build in constants include:
  - `LETTERS` (capital letters)
  - `month.name` (eg. "January", "February", etc..)
  - `month.abb` (eg. "Jan", "Feb", etc..)
  - `pi` 3.141593

Data structures

- R has a wide variety of data structures including:
  - scalars
  - vectors (numerical, character, logical)
  - matrices
  - data frames, and
  - lists.

Vectors

- A vector is the most basic data structure in R and can be of two types:
  - atomic vectors
  - lists
- As we did in the examples above, you can create a vector using `c` (see `?c` for more details on this *combine* function)

Vectors

Atomic Vectors can be vectors characters, logical, integers or numeric.

```
# example of a numeric vector:
y=c(2,3,0,3,1,0,0,1)
# example of a character vector:
letters<-c('A','B','C')
# example of a logical vector:
lvec <- c(TRUE, FALSE, FALSE)
```

Vectors

Numeric vectors can be created a number of different ways.

```
# Ceate a vector of size 10, where each element is a 2:
y=rep(2,10)
# This specifies a sequence of integers:
y=3:12
#This specifies a sequence of real numbers:
z=seq(from=1,to=2,by=0.1)
#This specifies a sequence of real numbers:
z=seq(from=2,to=1,by=-0.1)
```

Aside

Notation

- We say that `seq()` is a **function** for which we have specified three **arguments** (having argument names: `from`, `to` and `by`).
- Assuming we are putting the values in the correct order, you do not need to specify the argument names explicitly.

```
seq(1,2,0.1) # same as seq(from=1,to=2,by=0.1)
##  [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0
# if we want to specify arguments out of order,
# we HAVE to include argument names:
seq(by=0.1, to=2, from=1) # arguments in green
##  [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0
```

Vectors

Numeric vectors can be created a number of different ways.

```
# Ceates a vector of size 10, of random numbers between 0 and 1
y=runif(10)
# Creates a vector of size 3 sampled from x without replacement
set.seed(4758)
(y=sample(x=c(1,2,3), size=3))

## [1] 3 2 1

# Creates a vector of size 3 sampled from x with replacement
(y=sample(x=c(1,2,3), size=3, replace=TRUE))

## [1] 3 2 2
```

Vectors

We can also create empty vectors

```
x <- vector() # defaults to logical (FALSE)
x <- vector(length = 10) # defaults to logical vec of FALSES
x <- vector("character", length = 10)
x <- vector("numeric",   length = 10)
x <- vector("integer",   length = 10)
x <- vector("logical",   length = 10)
```

Seed

- Notice that I use `set.seed(4758)` on slide 34.
- This was created so that when you run the same exact code on our computer, you will obtain the same 'random' numbers.
- A random seed is a number used to initialize a pseudorandom number generator.
- Without going into to much detail, setting a seed using `set.seed` will yield same 'random' results every time.

```r
set.seed(4444)
runif(1)
## [1] 0.9849711
set.seed(4444)
runif(1)
## [1] 0.9849711
runif(1)
## [1] 0.1085671
```

Scalars

Scalars are just vectors of length 1. To check the length of any vector use `length()`

```
x = 4
length(x)
## [1] 1
y=c(2,3,0,3,1,0,0,1)
length(y)
## [1] 8
```

Indexing Vectors

To index an element from a vector, use single square brackets

```
z = c("apples","bananas","oranges", "pineapples")
z[1]

## [1] "apples"

z[2:3]

## [1] "bananas" "oranges"

z[c(4,2)]

## [1] "pineapples" "bananas"

z[-1]

## [1] "bananas"    "oranges"    "pineapples"
```

We have the option of including names for our vector elements.

```
z = c(123456, 25, 2019); names(z) <- c("studentID","age","year")
z["studentID"] # same as z[1]

## studentID
##    123456

z[c("age, year")] # wont work like z[c(2,3)]

## <NA>
##   NA

z[c(FALSE, TRUE, TRUE)] # will work

##  age year
##   25 2019

z[-"year"] # wont work like z[-3]

## Error in -"year":  invalid argument to unary operator
```

To add a new element/delete/replace an element:

```
z = c("first","second","third", "fourth")
# adds a fith element and replaces the second
z[5] = "fifth";  z[2] = "2nd"
z
## [1] "first"  "2nd"    "third"  "fourth" "fifth"
#removes the second element
(z = z[-2])
## [1] "first"  "third"  "fourth" "fifth"
```

Notice what happens if we add a 10th element to z (without specifying elements 6–9):

```
z[10] = "tenth"
z
```

```
##  [1] "first"   "third"   "fourth"  "fifth"   NA        NA        NA
##  [8] NA        NA        "tenth"
```

We can also add new named elements

```r
z = c(123456, 25, 2019); names(z) <- c("studentID","age","year")
# adds a forth element with the students major
z["major"] = "COSC"
z
```

```
## studentID       age      year     major
##  "123456"      "25"    "2019"    "COSC"
```

Notice how the addition of this forth element cuase the vector to change to a vector of characters! More on this later

Basic Operations for Vectors

We can also perform basic operations on vectors:

Operation	Command
natural log	`log(x)`
Exponent	`exp(x)`
Log base 10	`log10(x)`
Absolute value	`abs(x)`
Square root	`sqrt(x)`
Sum	`sum(x)`
Number of elements in x	`length(x)`

Unique elements of x	`unique(x)`
Mean	`mean(x)`
Variance	`var(x)`
Standard Deviation	`sd(x)`
Minimum value	`min(x)`
Maximum value	`max(x)`
Smallest and largest values	`range(x)`
median	`median(x)`
Basic statistical summary	`summary(x)`
Sort	`sort(x)`

```r
(y= sample(1:20, 9)) # samples 9 random numbers from 1 to 20
## [1]  6 15  8 16 17  1 18 12  7
# multiplies each element by 2
2*y
## [1] 12 30 16 32 34  2 36 24 14
sort(y)
## [1]  1  6  7  8 12 15 16 17 18
sort(y, decreasing = TRUE)
## [1] 18 17 16 15 12  8  7  6  1
summary(y)
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##    1.00    7.00   12.00   11.11   16.00   18.00
```

coercion

- By *atomic vectors* we mean they can only take on one data type.
- If we specify more than one type in a single vector, R will convert the mixed types to a single type which it deems most appropriate.
- The coercion will move towards the one that's easiest to coerce to.
- You can coerce vectors explicitly using the `as.`<*class*> (eg. `as.numeric`, `as.character`, etc.)

```r
(x = c("apple", 2, TRUE)) # converts all to character

## [1] "apple" "2"     "TRUE"

(x = c(TRUE, 4)) # converts all to numeric

## [1] 1 4

(x = c(2L, -1.3)) # converts to numeric

## [1]  2.0 -1.3

(x = c(TRUE, 0, 1, FALSE)) # converts all to numeric

## [1] 1 0 1 0

as.logical(x = c(TRUE, 0, 1, FALSE)) # converts to logical

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

Matrices

- While we can view matrices as column vectors stacked side by side or row vectors stack one on top of another, matrices are really just a vector in R. item The main difference between matrices and the vectors discussed in the previous slides is that a matrix has a **dimension**.
- The dimenions can be found using the `dim()` function.

Matrices

```r
(m <- matrix(1:8, nrow = 2, ncol = 4))
```

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

```r
dim(m) # gives the number or rows and columns
```

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

```r
class(m)
```

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

Matrices

- By default, matrices are constructed columnwise.
- You can always change to row-wise by specifying `byrow=TRUE`

```
(m <- matrix(1:6, nrow=2, ncol =3)) # fill columnwise

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

(m <- matrix(1:6, nrow=2, ncol =3, byrow=TRUE)) #fill row-wise

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

Matrices

An alternative way of constructing matrices is to use `array()`

```
array(1:6, dim=c(2,3))
```

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

Named Matrices

We may also choose to name our rows and columns

```
rownames(m) = c("row1", "row2")
colnames(m) = c("col1", "col2", "col3")
m
```

```
##      col1 col2 col3
## row1    1    2    3
## row2    4    5    6
```

Indexing

- To index an element from a matrix, we can again use square brackets, but we need to specify TWO numbers `[row, col]`.
- If we want to extract an entire row, we'll leave the `col` field blank (indicating we want all colums).
- If we want to extract an entire column, we'll leave the `row` field blank (indicating we want all rows).

Indexing

The following extracts a column, row, and cell from matrix `m`,

```
# extract the third column of m
m[,3]

## row1 row2
##    3    6

# extract the first row of m
m[1,]

## col1 col2 col3
##    1    2    3

# extract the element in the 1st row and 3rd column
m[1,3]

## [1] 3
```

Alternatively we could call them by name

```r
# extract the third column of m
m[,"col3"] # same as m[,3]
## row1 row2
##    3    6
# extract the first row of m
m["row1",] # same as m[1,]
## col1 col2 col3
##    1    2    3
# extract the element in the 1st row and 3rd column
m["row1","col3"] # same as m[1,3]
## [1] 3
```

Indexing

Note that we can also use one number (tricker this way)

```
(m = matrix(11:4, 2,4))

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

m[8] # extracts the 8th element in m

## [1] 4

(mVector = as.numeric(m))

## [1] 11 10  9  8  7  6  5  4

mVector[8]

## [1] 4
```

Indexing

Like the vectors discussed earlier, matices fall under the atomic vector category and therefore can only store data of one type:

```
m = matrix(c("bananas", 2, TRUE, 0),  nrow=2, ncol=2)
# notice how all coerce to characters:
m
```

```
##      [,1]      [,2]
## [1,] "bananas" "TRUE"
## [2,] "2"       "0"
```

lists

- If we want our vector to contain objects of different data types we need to create a list.
- We can think of lists as a special type of vector that can mix data types and structures

```r
(x = list("apple", "banana", 2, TRUE, 4, "four"))
## [[1]]
## [1] "apple"
##
## [[2]]
## [1] "banana"
##
## [[3]]
## [1] 2
##
## [[4]]
## [1] TRUE
##
## [[5]]
## [1] 4
##
## [[6]]
## [1] "four"
```

lists

Unlike vectors, the member of a list are accessed using double square brackets:

```
x[[1]] # first member

## [1] "apple"

x[[2]] # second member, and so on ...

## [1] "banana"
```

lists

- While I have specified each member to be a single word/number/logical, we could also have specified them to be vectors or matrices.
- To index the elements within the members of a list we use single square brackets (usually nested after double square brackets)

lists

```
y = list(c("apples", "banana", "four"), c(2,4), TRUE)
y
## [[1]]
## [1] "apples" "banana" "four"
##
## [[2]]
## [1] 2 4
##
## [[3]]
## [1] TRUE
```

lists

```
y[[1]][3] # the third element of the first member of y
## [1] "four"
y[[2]][1] # the first element of the second member of y
## [1] 2
y[[1]][4] # calling something that does not exist
## [1] NA
```

Named lists

Alternatively we could have names for our list members

```
numbers= c(2,4) # can define the member outside first
y = list(words=c("apples", "banana", "four"),
         numbers, logicals=TRUE)
y$words # same this as y[[1]]

## [1] "apples" "banana" "four"

y$numbers # same this as y[[2]]

## NULL

y$logicals # same this as y[[3]]

## [1] TRUE
```

We could also name the list members after they have been assigned

```
y = list(c("apples", "banana", "four"),
         c(2,4), TRUE)
names(y) <- c("words","numbers","logicals")
y
## $words
## [1] "apples" "banana" "four"
##
## $numbers
## [1] 2 4
##
## $logicals
## [1] TRUE
```

lists

We can index members of lists using the integer index numbers OR their member names (in quotations)

```
y$words

## [1] "apples" "banana" "four"

y[["words"]]

## [1] "apples" "banana" "four"

y[[4]]

## Error in y[[4]]:  subscript out of bounds

# the following are not valid:
# y$4
# y[[words]]
```

lists

We can modify its content/add new content directly either using `[[]]` with numbers or `$` with member names.

```
y$words[3] = "oranges"#replace the 3rd element of the 1st member
y[[1]][4] = "pineapples"#add a 4th element to the 1st member
y[[4]] = c(0,1,1,0) # add a forth member to y
y$"fifth" = 3:12 # add a fifth member to y (named "fifth")
```

Data Frames

- Data frames are perhaps the most used data structure used in statistics.
- Like list, they can store different data types.
- Futhermore, they can contain additional attributes such as column and row names.
- When combining vectors in a data frame all must have the same size (i.e. length).
- Missing observations will be recorded as `NA`.

Data Frames

Here is an example of how to create a data frame and add to it:

```r
Person=c('John', 'Jill', 'Jack')
Grade=c('45','92','91')
(Lab=data.frame(Person, Grade))
```

```
##   Person Grade
## 1   John    45
## 2   Jill    92
## 3   Jack    91
```

Data Frames

Like matrices, we can create column and row names

```r
colnames(Lab) # adopted from the variable names inputed
```

```
## [1] "Person" "Grade"
```

```r
rownames(Lab) # default to increasing integers
```

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

```r
# we could also add row names if we choose
rownames(Lab) = c("Student1", "Student2", "Student3")
Lab
```

```
##          Person Grade
## Student1   John    45
## Student2   Jill    92
## Student3   Jack    91
```

Data Frames

Unlike matrices, we can extract a column of this data.frame we use the $ notation:

```
Lab$Person

## [1] John Jill Jack

## Levels: Jack Jill John

Lab$Grade

## [1] 45 92 91

## Levels: 45 91 92

# N.B. This does not work for rows:

Lab$Student1

## NULL
```

Aside

Smart Autofill

- In some situations, R will try to autocomplete commands for us. When you begin to type `Lab$` for instance, you will see the options for the column names appear.
- You can use the arrow keys on your keyboard to navigate through the options.
- Press ENTER when you land on the option you want.

Data Frames

Adding a column to the data frame is as easy as typing:

```
Lab$Passed=c(FALSE,TRUE,TRUE)
Lab
```

```
##          Person Grade Passed
## Student1   John    45  FALSE
## Student2   Jill    92   TRUE
## Student3   Jack    91   TRUE
```

Attributes

- Each of these objects have so-called attributes.
- Here are some examples of attributes:
  1. names, dimension names
  2. dimensions
  3. class
  4. length

Attributes

```
attributes(Lab) # data vector defined in previous example

## $names
## [1] "Person" "Grade"  "Passed"
##
## $row.names
## [1] "Student1" "Student2" "Student3"
##
## $class
## [1] "data.frame"

attributes(m) # matrix defined previously

## $dim
## [1] 2 2
```

Data Entry

- Data entry can be either done by hand (impractical), or loaded from an existing file.
- `read.table` is a convenient way of reading data into R and storing it to a data frame.
- The data should be formated such that each rows contains information regarding one subject with data separated by white space/commas/tabs/.
- The first line may (or may not) contain a `header` with the names of the variables in each column.

Data Entry

Let's take the data `intake.txt` data available on CANVAS which looks like this:

"pre"	"post"
5260	3910
5470	4220
5640	3885
6180	5160
6390	5645
6515	4680
6805	5265

Data Entry

To read the data in:

```
read.table("intake.txt", header=TRUE)
## Warning in file(file, "rt"):  cannot open file 'intake.txt':
No such file or directory
## Error in file(file, "rt"):  cannot open the connection
```

Data Entry

- `header=TRUE` indicates that the first line of file contains the names of the variables (rather than data values).
  - `header=TRUE` allows `pre` and `post` to be read in as column names.
- (`header=FALSE` by default)

Data Entry

Since we didn't save it to a variable, it simple prints it to the screen. This wont be very useful in practice, so save it to a variable with a name that makes sense.

```
intake = read.table("intake.txt", header=TRUE)
## Warning in file(file, "rt"):  cannot open file 'intake.txt':
No such file or directory
## Error in file(file, "rt"):  cannot open the connection
```

Aside

Name Rules

There are rules on the types of names we can give variables:

- No spaces
- No special characters (apart from periods and underscores)
- Can't begin with a number

Data Entry

If we instead want to import the `data.csv` file avialble on CANVAS, we simply specify that our separator is a comma.

```
data = read.table("data.csv", sep=",", header=TRUE)
## Warning in file(file, "rt"):  cannot open file 'data.csv':
No such file or directory
## Error in file(file, "rt"):  cannot open the connection
```

Alternatively, we could have used the `read.csv()` function

```
data = read.csv("data.csv") # header=TRUE by default in read.csv
## Warning in file(file, "rt"):  cannot open file 'data.csv':
No such file or directory
## Error in file(file, "rt"):  cannot open the connection
```

Data Entry

If you want to specify a file that is outside of your working directory, simply specify the path name preceeding the filename. For example:

```
data <- read.csv("/Users/ivrbik/DATA101/data.csv", header=T)
```

Working Directory

- Since the data was stored in my local directory, I did not did to specify a path.
- To check what your working directory is, type `getwd()`
- To change your working directory, type `setwd(<path to your folder>)` or do the following on a PC or MAC:

  Session $\rightarrow$ Set Working Directory $\rightarrow$ Choose Directory...

Data Exporting

There is also a function that will allow you to write the contents of a data frame to a file. For instance, we could save our simple `Lab` data frame to a textfile called `Lab.txt` or `Lab.csv` using the following:

```
write.table(Lab, file="Lab.txt")
write.csv(Lab, file="Lab.csv")
```

Like the `read` functions, `write` will, by default, save these files to your working direction. To specify otherwise use:

```
write.table(Lab, file="<path>/Lab.txt")
```

Tip of the Day

To obtain a previous calculation in R, navigate to the console, and press the 'Up/Down Arrow' key on your keyboard.