Introduction to R - Basics

Israeli Geographical Association Conference

Alex Shtein
Geography and Environmental Development

18-12-2017

Ben-Gurion University of the Negev

Outline of R workshop - part 1 (R Basics)

- R studio components
- Expressions
- Assignment in r
- Mathematical operators in R
- Conditions
- Commenting in R
- Special values
- Functions
- Packages in R
- Data structures in R (Vectors, Martices, Data frames, Lists)

Installing R

- You can Download installation file from R- project site: http://www.r-project.org/
- Downloading Rstudio (Integrated Development Environment, IDE) from here: https://www.rstudio.com/products/rstudio/download/

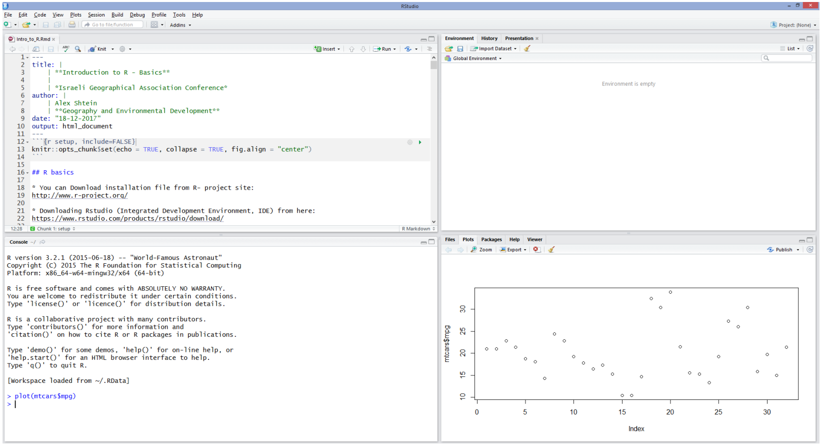
Open R studio

- RStudio is a free and open-source integrated development environment (IDE) for R, a programming language for statistical computing and graphics.

Start – All Programs – RStudio – RStudio

R studio components

In the first part of the workshop we will work mainly with the source (code editor) and the console components.



~~~
---
title: |
    | **Introduction to R - Basics**
    |
    | *Israeli Geographical Association Conference*
author: |
    | Alex Shtein
    | **Geography and Environmental Development**
date: "18-12-2017"
output: html_document
---
```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE, collapse = TRUE, fig.align = "center")
```


## R basics

* You can Download installation file from R- project site:
http://www.r-project.org/

* Downloading Rstudio (Integrated Development Environment, IDE) from here:
https://www.rstudio.com/products/rstudio/download/
~~~

```
R version 3.2.1 (2015-06-18) -- "World-Famous Astronaut"
Copyright (C) 2015 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Workspace loaded from ~/.RData]

> plot(mtcars$mpg)
> |
```

Figure 1: R studio components

Expressions

Lets type the following expression:

```
1+5+3+7
## [1] 16
```

The code was sent to the CPU and the result is presented in the following line

If we just print the following number or text we get constant values from different types: numeric and character (the simplest expressions):

```
600
## [1] 600

"Hello"
## [1] "Hello"
```

Assignment in r

- Elementary commands consist of either expressions or assignments. If an expression is given as a command, it is evaluated, printed (unless specifically made invisible), and the value is lost.
- An assignment also evaluates an expression and passes the value to a variable but the result is not automatically printed.

Assignment in r

$=$ or $<$- are assignment operators

```
# An example for assignment
x = 3
x <- 3
x
## [1] 3
```

Assignment in r

Note that **=** and $==$ are not the same!

```
# = is used for assignment
one = 1
two = 2
one = two
one
## [1] 2
two
## [1] 2

# == is used as a condition expression
one = 1
two = 2
one == two
## [1] FALSE
```

Assignment in r

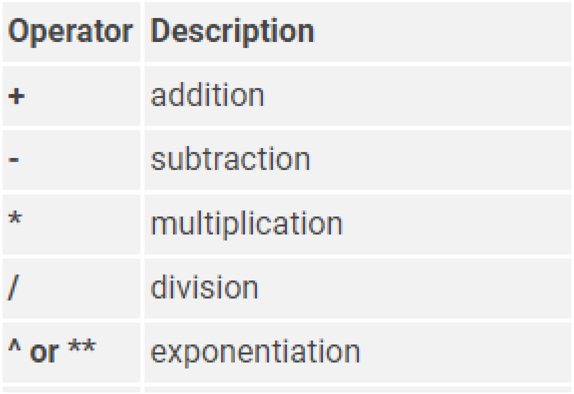
If you assign a different value to an existing variable, the variable is updated and the new value is assigned to this variable.

```
x = 55
x
## [1] 55
x = "Hello"
x
## [1] "Hello"
```

- You can use ls() function to get the list of objects that are saved in the temporary memory.

```
ls()
## [1] "one" "two" "x"
```

Mathematical operators in R



Operator	Description
+	addition
-	subtraction
*	multiplication
/	division
^ or **	exponentiation

Figure 2: Basic math operators

Example for basic Mathematical operators in R

```
5+3
## [1] 8
4-5
## [1] -1
1*10
## [1] 10
1/10
## [1] 0.1
10^2
## [1] 100
```

Mathematical operators in R

Additional math functions: `abs`, sign acos, asin, atan, atan2 sin, cos, tan ceiling, floor, round, trunc, signif exp, log, log10, log2, sqrt

max, min, prod, sum cummax, cummin, cumprod, cumsum, diff pmax, pmin range mean, median, cor, sd, var rle

Example for math functions in R

```
mean(10,5,20)
## [1] 10
min(1,2,3)
## [1] 1
sum(4,4,4)
## [1] 12
```

Expressions

- If you insert space the system ignores it:

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

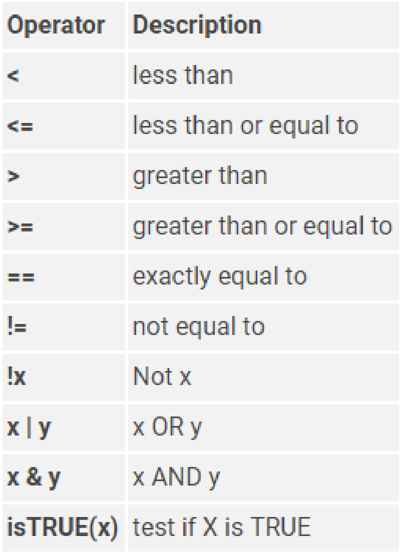
- If you press enter in the middle of your expression the system ignores it:

```
2*
5
## [1] 10
```

You can press **Esc** in order to quit

- You can go back to your previous actions using the the up and down arrows in your keyboard
- You can use **Ctrl+L** to clean the Console

Conditions



Operator	Description
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	exactly equal to
!=	not equal to
!x	Not x
x | y	x OR y
x & y	x AND y
isTRUE(x)	test if X is TRUE

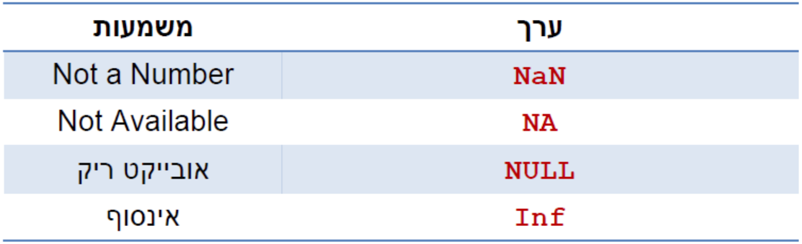
Figure 3: Operators Description

Conditions

```
1<2
## [1] TRUE
2>=2
## [1] TRUE
2!=2
## [1] FALSE
(1<10) & (10<100)
## [1] TRUE
(1<10) | (10<100)
## [1] TRUE

1 == 1
## [1] TRUE
1 != 1
## [1] FALSE
!(1 == 1)
## [1] FALSE
```

Special values



משמעות	ערך
Not a Number	NaN
Not Available	NA
אובייקט ריק	NULL
אינסוף	Inf

Figure 4: Table of special values in R

```
0/0
## [1] NaN
NA + 3
## [1] NA
NULL
## NULL
NULL
```

Commenting in R

If you want to add a comment next to your code use # sign and write your comments after it. The system ignores all text that follows the # sign.

```
1+1 # This is comment
## [1] 2
```

Case sensetivity in R

Technically R is an expression language with a very simple syntax. It is case sensitive as are most UNIX based packages, so A and a are different symbols and would refer to different variables.

Functions

- A function is a piece of code written to carry out a specified task; it can or can not accept arguments or parameters and it can or can not return one or more values.
- Functions are created using the function() directive and are stored as R objects just like anything else. In particular, they are R objects of class “function”.

Functions

An Example for the general structure of function:

f <- function() { ## Do something interesting }

(1) input arguments **(2)** the code in between the curly braces is the body of the function

Examples for function in R:

- The math functions that we mentioned earlier.

```r
# dim(x) - x    an R object,
# for example a matrix, array or data frame.
dim(iris)
## [1] 150   5
```

More about function in R

Useful article for studying more about function in R:
https://www.datacamp.com/community/tutorials/functions-in-r-a-tutorial#functions

Help option in R

Use **?** for to get information and help about functions in R. for example if you want to know more about R function called str, paste **?str** in the Console and get more details in the Help viewer.

```
?str
```

Packages in R

- Packages are collections of R functions, data, and compiled code in a well-defined format.
- The directory where packages are stored is called the **library**.
- R comes with a standard set of packages. Others are available for download and installation. Once installed, they have to be loaded into the session to be used.

Classification of R packages and functions useful for certain disciplines and methodologies:

- http://www.maths.lancs.ac.uk/~rowlings/R/TaskViews/
- https://cran.r-project.org/web/views/

Example for package installation in R

install.packages("sp")

library(sp)

learn more about the difference between R packages and libraries in DataCamp's Beginner's Guide to R Packages: https://www.datacamp.com/community/tutorials/r-packages-guide

Data structures in R

There are many types of R-objects. The frequently used ones are:

- Vectors
- Matrices
- Data Frames
- Lists

R objects in R

d	Homogeneous	Heterogeneous
1d	atomic vectors	list
2d	matrix	data.frame
3d	array	-

Vectors

The basic data structure in R is the vector. There are two types of Vectors: atomic vectors and lists. They have three common properties:

- Type, typeof(), what it is.
- Length, length(), how many elements it contains.
- Attributes, attributes(), additional arbitrary metadata.

Atomic vectors

- The atomic vector is the simplest R data type.
- Atomic vectors are linear vectors of a single type.
- There are four common types of atomic vectors that we will discuss in detail: logical, integer, double (often called numeric), and character.

Examples for each Vector type:

```r
dbl_var <- c(1, 2.5, 4.5) # Double
# With the L suffix, you get an integer
# rather than a double
int_var <- c(1L, 6L, 10L)
# Use TRUE and FALSE (or T and F)
# to create logical vectors
log_var <- c(TRUE, FALSE, T, F)
# Character vectors
chr_var <- c("these are", "some strings")
```

Creating a vector

You can create a vector using the **c()** function that combines the arguments by their order:

```
x= c(1, 2, 3)
c(x, 5)
## [1] 1 2 3 5
y = c("cat", "dog", "mouse", "apple")
y
## [1] "cat"   "dog"   "mouse" "apple"
```

Additional ways to create vectors

(1) Using the **:** operator

```
1:10
##  [1]  1  2  3  4  5  6  7  8  9 10
55:43
##  [1] 55 54 53 52 51 50 49 48 47 46 45 44 43
```

(2) Using the **seq** function for creating vector of sequential numbers This function has three parameters: from (from which number to start), to (at which number to end), and by (increment of the sequence)

```
seq(from = 100, to = 150, by = 10)
## [1] 100 110 120 130 140 150
seq(from = 190, to = 150, by = -10)
## [1] 190 180 170 160 150
```

Additional ways to create vectors

(3) Using the **rep** function to create a vector of repeating values
This function has three main parameters: x- which value to repeat, times- how many times to repeat it, each- how many times to repeat each of x elements

```
# Example 1
rep(x = 22, times = 10)
##  [1] 22 22 22 22 22 22 22 22 22 22
# Example 2
x = c(18, 0, 9)
rep(x = x, times = 3)
## [1] 18  0  9 18  0  9 18  0  9
# Example 3
x = c(18, 0, 9)
rep(x = x, each = 3)
## [1] 18 18 18  0  0  0  9  9  9
```

Missing values

Missing values are specified with NA, which is a logical vector of length 1. NA will always be coerced to the correct type if used inside c(), or you can create NAs of a specific type with NA_real_ (a double vector), NA_integer_ and NA_character_.

Example for NA whithin a vector:

```
dbl_var <- c(1, 2.5, 4.5,NA)
dbl_var <- c(1, 2.5, 4.5,NA_real_)
```

Missing values

- You can identify missing values in a vector using **is.na()** function
- This function receives a vector and returns a logical vector where TRUE is in locations where there is NA and FALSE where there is no NA.

```r
# Example
x = c(28, 58, NA, 31, 39, NA, 9)
is.na(x)
## [1] FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE
```

Missing values

- Different functions that receive vectors have an **na.rm** argument
- The default of this argument is FALSE, meaning that the NA values would not be excluded from the calculation of the function.

```
x = c(28, 58, NA, 31, 39, NA, 9)
mean(x)
## [1] NA
mean(x, na.rm = TRUE)
## [1] 33
```

all and any fucntions

- Sometimes we would like to know if a logical vector includes at least one TRUE value or if all its elements are true.
- all and any functions can be used for that

```
# The any() function accepts a logical vector
# and returns TRUE if at least one of the vector's
# elements is TRUE, otherwise it returns FALSE
x = 1:10
x > 8
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TR
any(x > 8)
## [1] TRUE
any(x > 88)
## [1] FALSE
```

all and any fucntions

```
# The all() function accepts logical vector
# and returns TRUE if all of the vector's elements
#  are TRUE, otherwise it returns FALSE
x = 1:10
x
##  [1]  1  2  3  4  5  6  7  8  9 10
all(x > 8)
## [1] FALSE
all(x > 0)
## [1] TRUE
```

Types and tests

Given a vector, you can determine its type with typeof(), or check if it's a specific type with an "is" function: is.character(), is.double(), is.integer(), is.logical(), or, more generally, is.atomic().

```
typeof(dbl_var)
## [1] "double"
is.character(dbl_var)
## [1] FALSE
is.double(dbl_var)
## [1] TRUE
is.numeric(dbl_var)
## [1] TRUE
```

is.numeric() is a general test for the "numberliness" of a vector and returns TRUE for both integer and double vectors. It is not a specific test for double vectors, which are often called numeric.

Coercion

All elements of an atomic vector must be the same type, so when you attempt to combine different types they will be coerced to the most flexible type. Types from least to most flexible are: logical, integer, double, and character.

For example, combining a character and an integer yields a character:

```r
str(c("a", 1))
##  chr [1:2] "a" "1"
```

Subsetting with numbers and names

With a vector:

```
# A sample vector
v <- c(1,4,4,3,2,2,3)

v[c(2,3,4)]
## [1] 4 4 3
v[2:4]
## [1] 4 4 3
v[c(2,4,3)]
## [1] 4 3 4
```

Recycling of Vectors in R

- A very important, concept: R likes to operate on vectors of the same length, so if it encounters two vectors of different lengths in a binary operation, it merely replicates (recycles) the smaller vector until it is the same length as the longest vector, then it does the operation.
- If the recycled smaller vector has to be "chopped off" to make it the length of the longer vector, you will get a warning, but it will still return a result.

Recycling of Vectors in R

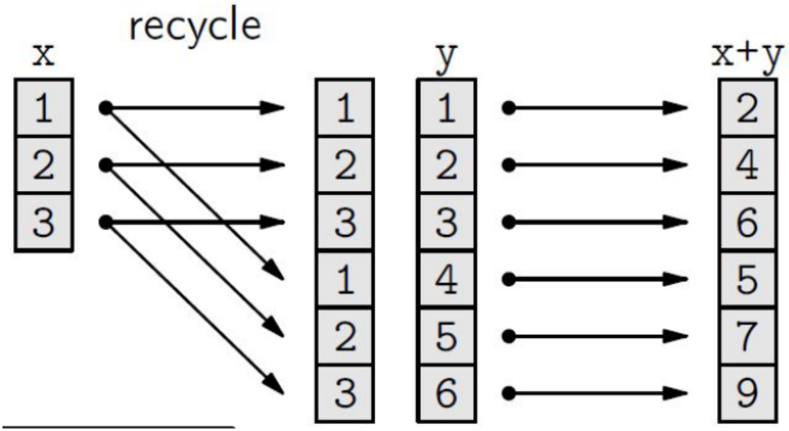


Figure 5: Recycle concept

Recycling of Vectors in R

```
# Example for recycling
x <- c(1,2,3)
y <- c(1,10)
x * y
## Warning in x * y: longer object length is not a multiple
## length
## [1]  1 20  3
```

Matrices

- A matrix is a collection of data elements arranged in a two-dimensional rectangular layout.
- Matrix has two dimensions and are used commonly as part of the mathematical machinery of statistics.
- Matrices are created with matrix() , or by using the assignment form of dim():

Matrices

- Use the **matrix** function to create a matrix.
- This function accepts the following parameters:

1. data - A vector the contains the matrix data
2. nrow - number of rows
3. ncol - number of columns
4. byrow - How to fill the data in the matrix (byrow\bycolumn), the default is by column.

Matrices

Code examples:

```r
# Create a matrix - specify rows and columns
a <- matrix(1:6, ncol = 3, nrow = 2)

# You can also modify an object in place by setting dim()
c <- 1:6
dim(c) <- c(3, 2)
c
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

Matrices

Usefull functions

```
a <- matrix(1:6, ncol = 3, nrow = 2)

length(a)
## [1] 6
nrow(a)
## [1] 2
ncol(a)
## [1] 3
rownames(a) <- c("A", "B")
colnames(a) <- c("a", "b", "c")
```

Matrices

- c() generalises to cbind() and rbind() for matrices.
- You can transpose a matrix with t()
- You can test if an object is a matrix is.matrix() or by looking at the length of the dim().
- as.matrix() make it easy to turn an existing vector into a matrix.

Matrices

Vectors are not the only 1-dimensional data structure. You can have matrices with a single row or single column. They may print similarly, but will behave differently. As always, use str() to reveal the differences.

```
str(1:3)
##  int [1:3] 1 2 3
str(matrix(1:3, ncol = 1))
##  int [1:3, 1] 1 2 3
str(matrix(1:3, nrow = 1))
##  int [1, 1:3] 1 2 3
```

Subsetting in Matrices

- use of [x,] or [,x] to access to a whole column or line
- The result is a vecotor

```
x = matrix(1:6, ncol = 3)
x
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
x[2, ] # Show the vlaues of the second row
## [1] 2 4 6
x[ ,2] # Show the vlaues of the second column
## [1] 3 4
```

Subsetting in Matrices

```
# If you want to avoid the conversion of the result
# to vector use drop= FALSE
x[2, , drop = FALSE]
##      [,1] [,2] [,3]
## [1,]    2    4    6
x[, 2, drop = FALSE]
##      [,1]
## [1,]    3
## [2,]    4
```

Data frames

- A data frame is the most common way of storing data in R, and if used systematically makes data analysis easier.
- A data frame is used to represent table in r.
- This makes it a 2-dimensional structure, so it shares properties of both the matrix and the list. This means that a data frame has names(), colnames(), and rownames(), although names() and colnames() are the same thing.

Data frames

- The length() of a data frame is the length of the underlying list and so is the same as ncol(); nrow() gives the number of rows.
- As described in subsetting, you can subset a data frame like a 1d structure (where it behaves like a list), or a 2d structure (where it behaves like a matrix).

Creating a data.frame

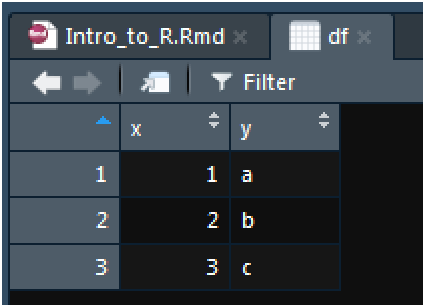
You create a data frame using data.frame(), which takes named vectors as input:

```
df <- data.frame(x = 1:3, y = c("a", "b", "c"))
str(df)
## 'data.frame':    3 obs. of  2 variables:
##  $ x: int  1 2 3
##  $ y: Factor w/ 3 levels "a","b","c": 1 2 3
class(df)
## [1] "data.frame"
is.data.frame(df)
## [1] TRUE
df
##   x y
## 1 1 a
## 2 2 b
## 3 3 c
```

Viewing the data.frame

You can use this to sort and classify the table

View(df)



	x	y
1	1	a
2	2	b
3	3	c

Figure 6: View the data frame

Data Frames

You can coerce an object to a data frame with as.data.frame():

- A vector will create a one-column data frame.
- A matrix will create a data frame with the same number of columns and rows as the matrix.

Subsetting in data.frame

```r
# Create a sample data frame
data <- read.table(header=T, text='
 subject sex size
       1   M    7
       2   F    6
       3   F    9
       4   M   11
 ')
```

Subsetting in data.frame

```
# Get the element at row 1, column 3
data[1,3]
## [1] 7
data[1,"size"]
## [1] 7
```

Subsetting in data.frame

```
# Get rows 1 and 2, and all columns
data[1:2, ]
##   subject sex size
## 1       1   M    7
## 2       2   F    6
data[c(1,2), ]
##   subject sex size
## 1       1   M    7
## 2       2   F    6
```

Subsetting in data.frame

```
# Get rows 1 and 2, and only column 2
data[1:2, 2]
## [1] M F
## Levels: F M
data[c(1,2), 2]
## [1] M F
## Levels: F M

# Get rows 1 and 2, and only the columns
# named "sex" and "size"
data[1:2, c("sex","size")]
##   sex size
## 1   M    7
## 2   F    6
data[c(1,2), c(2,3)]
##   sex size
## 1   M    7
## 2   F    6
```

Combining data frames

You can combine data frames using cbind() and rbind():

```
cbind(df, data.frame(z = 3:1))
##   x y z
## 1 1 a 3
## 2 2 b 2
## 3 3 c 1
rbind(df, data.frame(x = 10, y = "z"))
##    x y
## 1  1 a
## 2  2 b
## 3  3 c
## 4 10 z
```

When combining column-wise, the number of rows must match, but row names are ignored. When combining row-wise, both the number and names of columns must match.

Data frame - calculatig new columns

A new column can be added using the **$** sign.

For example:

```
df = data.frame(a=c(1:3),b=c("A","B","C"),c=c(2,2,2))
df$new = df$a*df$c
df
##   a b c new
## 1 1 A 2   2
## 2 2 B 2   4
## 3 3 C 2   6
```

Loading data files to R

Often you would like to load your data to R. There are different functions for different file formats.

An example for loading a csv file

```
dat <- read.csv("D:/Users/shtien/Dropbox/R_workshop/alex/df
```

Useful functions

These functions will provide you some information about your data.frame:

```
head(dat)
##          Name Value
## 1   Tel-aviv   100
## 2   Jeruslem    80
## 3      Haifa    60
## 4 Beer-Sheva    65
tail(dat)
##          Name Value
## 1   Tel-aviv   100
## 2   Jeruslem    80
## 3      Haifa    60
## 4 Beer-Sheva    65
dim(dat)
## [1] 4 2
```

Useful functions

These functions will provide you some information about your data.frame:

```
summary(dat)
##          Name        Value       
##  Beer-Sheva:1   Min.   : 60.00  
##  Haifa     :1   1st Qu.: 63.75  
##  Jeruslem  :1   Median : 72.50  
##  Tel-aviv  :1   Mean   : 76.25  
##                 3rd Qu.: 85.00  
##                 Max.   :100.00  
```

Defining your working directory

You can define a certain folder as your working directory
Note that you should use a foreword slash (/ or //) and not a backslash (\)

```r
setwd("D:/Users/shtien/Dropbox/R_workshop/alex")
getwd()
## [1] "D:/Users/shtien/Dropbox/R_workshop/alex"
dat <- read.csv("df_example.csv")
```

Getting the list of files in your working directory

```
list.files()
##  [1] "df_example.csv"                   "header.tex"
##  [3] "images"                           "Intro to R.pptx"
##  [5] "Intro_to_R.pdf"                   "Intro_to_R.R"
##  [7] "Intro_to_R.Rmd"                   "Intro_to_R.syncte
##  [9] "Intro_to_R.tex"                   "logical_operators
## [11] "math_operators.PNG"               "recycle_rool.PNG"
## [13] "RStudio-Logo-Blue-Gradient.png"   "Rstudio_component
## [15] "Special_values.PNG"               "tex2pdf.1036"
## [17] "tex2pdf.3276"                     "tex2pdf.3916"
## [19] "tex2pdf.5392"                     "tex2pdf.6384"
## [21] "tex2pdf.6840"                     "try.Rmd"
## [23] "try.tex"                          "view_example.PNG"
```

Lists

Lists are different from atomic vectors because their elements can be of any type, including lists. You construct lists by using list() instead of c():

```
x <- list(1:3, "a", c(TRUE, FALSE, TRUE), c(2.3, 5.9))
str(x)
## List of 4
##  $ : int [1:3] 1 2 3
##  $ : chr "a"
##  $ : logi [1:3] TRUE FALSE TRUE
##  $ : num [1:2] 2.3 5.9
```

Lists

The typeof() a list is list. You can test for a list with is.list() and coerce to a list with as.list(). You can turn a list into an atomic vector with unlist(). If the elements of a list have different types, unlist() uses the same coercion rules as c().

Lists are used to build up many of the more complicated data structures in R. For example, both data frames (described in data frames) and linear models objects (as produced by lm()) are lists.

List slicing

We retrieve a list slice with the single square bracket “[]” operator. The following is a slice containing the second member of x, which is a copy of s.

```r
# First we will create a list
n = c(2, 3, 5)
s = c("aa", "bb", "cc", "dd", "ee")
b = c(TRUE, FALSE, TRUE, FALSE, FALSE)
x = list(n, s, b, 3)   # x contains copies of n, s, b

x[2]
## [[1]]
## [1] "aa" "bb" "cc" "dd" "ee"
```

Lists

```
# With an index vector, we can retrieve a slice with multip

x[c(2, 4)]
## [[1]]
## [1] "aa" "bb" "cc" "dd" "ee"
##
## [[2]]
## [1] 3
```

Lists - Subsetting

In order to reference a list member directly, we have to use the double square bracket "[[]]" operator. The following object x[[2]] is the second member of x.

```
x[[2]]
## [1] "aa" "bb" "cc" "dd" "ee"

# We can modify its content directly.
x[[2]][1] = "ta"
x[[2]]
## [1] "ta" "bb" "cc" "dd" "ee"
```

Useful links and books

You can read and learn more from the following links:

- https://support.rstudio.com/hc/en-us/articles/201141096-Getting-Started-with-R
- http://tryr.codeschool.com/

Try to look for answers for your R question in Google.

Almost everything you asked yourself someone else asked before :)