AUA CS 108, Statistics, Fall 2019

R Lab Session 01

Michael Poghosyan
YSU, AUA
michael@ysu.am, mpoghosyan@aua.am

25 Aug 2019

Installing R

- **R** is freeware

Installing R

- **R** is freeware
- To install R, visit https://www.r-project.org/.

Installing R

- **R** is freeware
- To install R, visit https://www.r-project.org/.
- *Recommendation*: Install also **R Studio** (freeware): https://www.rstudio.com/

Installing R

- **R** is freeware
- To install R, visit https://www.r-project.org/.
- *Recommendation*: Install also **R Studio** (freeware): https://www.rstudio.com/
- I will use the following version: R version 3.6.1 (2019-07-05)

Installing R

- **R** is freeware
- To install R, visit https://www.r-project.org/.
- *Recommendation*: Install also **R Studio** (freeware): https://www.rstudio.com/
- I will use the following version: R version 3.6.1 (2019-07-05)
- To find some R Studio shortcuts, hit $Alt + Shift + K$

Installing R

- **R** is freeware
- To install R, visit https://www.r-project.org/.
- *Recommendation*: Install also **R Studio** (freeware): https://www.rstudio.com/
- I will use the following version: R version 3.6.1 (2019-07-05)
- To find some R Studio shortcuts, hit *Alt* + *Shift* + *K*

Here is a link to a reference book for **R**: An Introduction to R

Running a command in the Console

You can start working with **R** in a Console, say, in **R Studio** or in R Terminal.

Running a command in the Console

You can start working with **R** in a Console, say, in **R Studio** or in R Terminal. The Console shows the "$>$" sign, and you can write a command at that line and execute it by hitting *Enter*.

Running a command in the Console

You can start working with **R** in a Console, say, in **R Studio** or in R Terminal. The Console shows the "$>$" sign, and you can write a command at that line and execute it by hitting *Enter*.

The output will be dispayed on the next line, and start with a number in a parenthesis, say,

```
2+3
```

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

Running a command in the Console

Here '##' sign is for the slides, **R** will not show that sign. It is to emphasize the **R** output line. Also note that the **R** code in my slides will be written in a color (and will not start by the ">" sign) [1].

[1] I am using the **R Markdown**, which gives a possibility to integrate *R* code with **LaTeX**.

Running a command in the Console

Here '##' sign is for the slides, **R** will not show that sign. It is to emphasize the **R** output line. Also note that the **R** code in my slides will be written in a color (and will not start by the ">" sign) [1].

The symbol **[1]** after the **##** sign in the output shows the position in the output of the first element in the row. Let me explain by examples:

```r
x <- c(12,3,43,24); x
```

```
## [1] 12  3 43 24
```

[1] I am using the **R Markdown**, which gives a possibility to integrate *R* code with **LaTeX**.

Running a command in the Console

Here '##' sign is for the slides, **R** will not show that sign. It is to emphasize the **R** output line. Also note that the **R** code in my slides will be written in a color (and will not start by the ">" sign) [1].

The symbol **[1]** after the **##** sign in the output shows the position in the output of the first element in the row. Let me explain by examples:

```
x <- c(12,3,43,24); x
```

```
## [1] 12  3 43 24
```

Here the first element in the output row, 12, is the first element in the output, so this line starts by [1].

[1] I am using the **R Markdown**, which gives a possibility to integrate *R* code with **LaTeX**.

Running a command in the Console

Now, let us run the following:

```
seq(7, 120, 2)
```

```
##  [1]   7   9  11  13  15  17  19  21  23  25  27  29  31
## [20]  45  47  49  51  53  55  57  59  61  63  65  67  69
## [39]  83  85  87  89  91  93  95  97  99 101 103 105 107
```

Running a command in the Console

Now, let us run the following:

```r
seq(7, 120, 2)
```

```
##  [1]   7   9  11  13  15  17  19  21  23  25  27  29  31
## [20]  45  47  49  51  53  55  57  59  61  63  65  67  69
## [39]  83  85  87  89  91  93  95  97  99 101 103 105 107
```

Here, in the output, the command prints all odd numbers from 7 to 120.

[2]This is in my slides, which I am prepared using **RMarkdown**. If you will run the code in **R** console, you can have just 3 lines (btw, not all elements are shown in the slide, the output is wide for the slides)

Running a command in the Console

Now, let us run the following:

```r
seq(7, 120, 2)
```

```
##  [1]   7   9  11  13  15  17  19  21  23  25  27  29  3
## [20]  45  47  49  51  53  55  57  59  61  63  65  67  6
## [39]  83  85  87  89  91  93  95  97  99 101 103 105 10
```

Here, in the output, the command prints all odd numbers from 7 to 120. As you can see, the output is given in 4 lines[2]. [1] in the first line shows that the first element in that row, 7, is the no. 1 element in the output. [18] at the beginning of the second line means that the first element in that row, 41, is the 18-th element in the output, and hence, 43 is the 19-th one. So 109 is the 52-nd element in the output, 111 is the 53-rd one, and you can calculate that we have 57 elements in the output in total.

[2]This is in my slides, which I am prepared using **RMarkdown**. If you will run the code in **R** console, you can have just 3 lines (btw, not all elements are shown in the slide, the output is wide for the slides)

Running a command in the Console

Working in the Console is not too hard. Say, *pi* is the π, and “^” means “to the power”:

```
pi^2
```

```
## [1] 9.869604
```

Hitting the up/down arrows in Console will run over the Commands History.

Running a command in the Console

Working in the Console is not too hard. Say, *pi* is the π, and "^" means "to the power":

```
pi^2
```

```
## [1] 9.869604
```

Hitting the up/down arrows in Console will run over the Commands History. But it is not too convenient to work in the Console.

Running a command in the Console

Working in the Console is not too hard. Say, *pi* is the π, and "^" means "to the power":

```
pi^2
```

```
## [1] 9.869604
```

Hitting the up/down arrows in Console will run over the Commands History. But it is not too convenient to work in the Console.

This is because, for instanse, you can write at most one command in the Console (unless you put ";" sign between 2 commands), and also the code written in the Console will not be saved (well, will be saved in the History, in fact).

Running a command in the Console

Working in the Console is not too hard. Say, *pi* is the π, and "^" means "to the power":

```
pi^2
```

```
## [1] 9.869604
```

Hitting the up/down arrows in Console will run over the Commands History. But it is not too convenient to work in the Console.

This is because, for instanse, you can write at most one command in the Console (unless you put ";" sign between 2 commands), and also the code written in the Console will not be saved (well, will be saved in the History, in fact).

So we will write our code in **.r** files, so we can run, update, change our code when we wish to.

And please send your **R** homework as **.r** files $\ddot\smile$

Creating/Opening an R Script file

To create a New **R** Script:

- use *File* $\rightarrow$ *New File* $\rightarrow$ *R Script*

Creating/Opening an R Script file

To create a New **R** Script:

- use *File* $\rightarrow$ *New File* $\rightarrow$ *R Script*
- or hit *Ctrl* + *Shift* + *N*

Creating/Opening an R Script file

To create a New **R** Script:

- use *File* $\rightarrow$ *New File* $\rightarrow$ *R Script*
- or hit *Ctrl* + *Shift* + *N*
- or just create a file *SomeName.R* by using your favourite text editor (and open it in, say, R Studio)

Creating/Opening an R Script file

To create a New **R** Script:

- use *File* $\rightarrow$ *New File* $\rightarrow$ *R Script*
- or hit *Ctrl* + *Shift* + *N*
- or just create a file *SomeName.R* by using your favourite text editor (and open it in, say, R Studio)

To load your saved script from R:

- use *File* $\rightarrow$ *Open File...*

Running a line/part of a code

To run the command on some line of your R Script

- just put the cursor at that line and press *Ctrl* + *Enter*

Running a line/part of a code

To run the command on some line of your R Script

- just put the cursor at that line and press *Ctrl* + *Enter*
- or hit the Run button when the cursor is on that line

Running a line/part of a code

To run the command on some line of your R Script

- just put the cursor at that line and press *Ctrl* + *Enter*
- or hit the Run button when the cursor is on that line

To run a block of commands of your R Script

- select that block and press *Ctrl* + *Enter*

Running a line/part of a code

To run the command on some line of your R Script

- just put the cursor at that line and press *Ctrl* + *Enter*
- or hit the Run button when the cursor is on that line

To run a block of commands of your R Script

- select that block and press *Ctrl* + *Enter*
- Say, *Ctrl* + *A*, *Ctrl* + *Enter* will execute the whole script. The same can be done by *Ctrl* + *Alt* + *R*

Commenting and Clearing the Console

- Use the symbol # to write a comment

  Example:

```
# 1+2+3
4+5+6
```

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

Commenting and Clearing the Console

- Use the symbol # to write a comment

  Example:

```r
# 1+2+3
4+5+6
```

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

- when you are in the console, use *Ctrl* + *L* to clear the console (without deleting the variables and data you have)

R Basic Operations and Commands

Here are some simple calculation examples. Try to run the following lines in the **R** Console:

```r
pi*sqrt(10)+exp(4)
```

```
## [1] 64.53274
```

```r
2^10
```

```
## [1] 1024
```

```r
sin(2*pi)
```

```
## [1] -2.449213e-16
```

Defining Variables

Usually, we are giving some names to quantities we calculate, to use them later. In **R** this can be done in two ways:

```r
x = 10+20
```

or

```r
x <- 10+20
```

[3]You can read more about the difference between these assignment operators here or here

Defining Variables

Usually, we are giving some names to quantities we calculate, to use them later. In **R** this can be done in two ways:

```r
x = 10+20
```

or

```r
x <- 10+20
```

R community is usually using the last way to assign a value to a variable, and we will follow the community $\ddot\smile$ [3]

[3] You can read more about the difference between these assignment operators here or here

Defining Variables

Usually, we are giving some names to quantities we calculate, to use them later. In **R** this can be done in two ways:

```
x = 10+20
```

or

```
x <- 10+20
```

R community is usually using the last way to assign a value to a variable, and we will follow the community $\ddot\smile$ [3]

You can also do in this way:

```
10 -> y
z <- x + y
```

[3] You can read more about the difference between these assignment operators here or here

Defining Variables

You can see that in both cases, x is assigned the value (say, you can see this in the *Environment* tab of **R Studio**), but the result is not printed on the screen.

Defining Variables

You can see that in both cases, x is assigned the value (say, you can see this in the *Environment* tab of **R Studio**), but the result is not printed on the screen.

If you want to see the result of the assignment, just type X after the assignment:

```
x
```

```
## [1] 30
```

or use

```
print(x)
```

```
## [1] 30
```

Defining Variables

You can see that in both cases, x is assigned the value (say, you can see this in the *Environment* tab of **R Studio**), but the result is not printed on the screen.

If you want to see the result of the assignment, just type X after the assignment:

```r
x
```

```
## [1] 30
```

or use

```r
print(x)
```

```
## [1] 30
```

The other way is to run

```r
(x <- 10 + 20)
```

```
## [1] 30
```

Example

Here is a piece of code, just some calculations:

```r
x <- 20
y <- 2*(x-pi)
z <- sin(y)
z
```

```
## [1] 0.7451132
```

Example

Here is a piece of code, just some calculations:

```r
x <- 20
y <- 2*(x-pi)
z <- sin(y)
z
```

```
## [1] 0.7451132
```

You can write several commands in a line, by separating them by ";":

```r
x <- 10; y <- exp(2); z <- sin(1); print(x+y+z)
```

```
## [1] 18.23053
```

Example

Here is a piece of code, just some calculations:

```r
x <- 20
y <- 2*(x-pi)
z <- sin(y)
z
```

```
## [1] 0.7451132
```

You can write several commands in a line, by separating them by ";":

```r
x <- 10; y <- exp(2); z <- sin(1); print(x+y+z)
```

```
## [1] 18.23053
```

To remove the variable x from the memory (to "forget" the variable x), just run

```r
rm(x)
```

Working with Vectors

The following assignment

```
x <- c(1,3,5, -3, 2.3)
```

assigns x to be the vector $x = (1, 3, 5, -3, 2.3)$.

Working with Vectors

The following assignment

```
x <- c(1,3,5, -3, 2.3)
```

assigns x to be the vector $x = (1, 3, 5, -3, 2.3)$. And you can use then

```
x
```

```
## [1]  1.0  3.0  5.0 -3.0  2.3
```

Working with Vectors

The following assignment

```
x <- c(1,3,5, -3, 2.3)
```

assigns x to be the vector $x = (1, 3, 5, -3, 2.3)$. And you can use then

```
x
```

```
## [1]  1.0  3.0  5.0 -3.0  2.3
```

```
x[2]
```

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

Working with Vectors

The following assignment

```
x <- c(1,3,5, -3, 2.3)
```

assigns x to be the vector $x = (1, 3, 5, -3, 2.3)$. And you can use then

```
x
```

```
## [1]  1.0  3.0  5.0 -3.0  2.3
```

```
x[2]
```

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

```
x[-1]
```

```
## [1]  3.0  5.0 -3.0  2.3
```

Working with Vectors

The following will make an empty vector:

```r
x <- c()
```

Working with Vectors

The following will make an empty vector:

```
x <- c()
```

c in the vecor definition comes from **concatinate**.

Working with Vectors

The following will make an empty vector:

```
x <- c()
```

c in the vecor definition comes from **concatinate**.

Using c, you can add an element to a vector:

```
x <- c(x, 2)
x
```

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

x was an empty vector, not we have added an element, and now x=(2)=2.

Working with Vectors

The following will make an empty vector:

```r
x <- c()
```

c in the vecor definition comes from **concatinate**.

Using c, you can add an element to a vector:

```r
x <- c(x, 2)
x
```

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

x was an empty vector, not we have added an element, and now x=(2)=2.

We can concatenate also several vectors:

```r
x <- c(1,2,3); y <- c(4,5,6); z <- c(x,y)
z
```

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

Basic Methods to make Vectors

Besides the above method, by specifying all elements of the vector, we can make some standard vectors using appropriate commands:

```r
(x <- 1:10)
```

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

```r
a <- seq(from=1, to=10, by=2); a
```

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

```r
b <- seq(from=1, to=10, length.out = 7); b
```

```
## [1]  1.0  2.5  4.0  5.5  7.0  8.5 10.0
```

Basic Methods to make Vectors

```r
y <- rep(1, 5); y
```

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

```r
z <- rep(1:3, 4); z
```

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

```r
t <- rep(1:3, each = 4); print(t)
```

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

```r
w <- rep(c(1,3), 2); w
```

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

Basic Methods to make Vectors

Another useful method to generate vectors is to choose random samples:

```
# Uniform Sample of size 10 from [0,3]
x <- runif(10, min = 0, max = 3)
x
```

```
##  [1] 0.4924354 1.1149195 2.6196978 2.4271227 2.8708527 2
##  [8] 0.8951109 0.2501558 2.2553890
```

```
# Normal Sample of size 15 with Mean 0 and Standard Deviat
x <- rnorm(15, mean = 0, sd = 2)
x
```

```
##  [1] -1.29047522  1.00992019  1.18918428 -1.24092485 -1.
##  [7]  0.69802600  1.63067232 -3.39887823 -0.13533879  2.
## [13]  0.08266733  5.27828355 -2.59519514
```

Basic Methods to make Vectors

Another useful method to generate vectors is to choose random samples:

```r
# Uniform Sample of size 10 from [0,3]
x <- runif(10, min = 0, max = 3)
x
```

```
##  [1] 0.4924354 1.1149195 2.6196978 2.4271227 2.8708527 2
##  [8] 0.8951109 0.2501558 2.2553890
```

```r
# Normal Sample of size 15 with Mean 0 and Standard Deviat
x <- rnorm(15, mean = 0, sd = 2)
x
```

```
##  [1] -1.29047522  1.00992019  1.18918428 -1.24092485 -1.
##  [7]  0.69802600  1.63067232 -3.39887823 -0.13533879  2.
## [13]  0.08266733  5.27828355 -2.59519514
```

Later we will talk about distributions in **R** and random samples.

Some operations on Vectors

For a vector

```r
x <- c(-1, 2.3, 10, 5)
```

we can calculate

```r
length(x) #The Lenght of a vector
```

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

```r
sum(x) # The sum of elements
```

```
## [1] 16.3
```

```r
cumsum(x) # The cumulative sum, x_1, x_1+x_2,...
```

```
## [1] -1.0  1.3 11.3 16.3
```

```r
prod(x)
```

```
## [1] -115
```

Some operations on Vectors

```r
cumprod(x)
```

```
## [1]   -1.0   -2.3  -23.0 -115.0
```

```r
mean(x) # The mean of the elements
```

```
## [1] 4.075
```

```r
max(x); min(x)
```

```
## [1] 10
```

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

```r
sin(x)
```

```
## [1] -0.8414710  0.7457052 -0.5440211 -0.9589243
```

```r
x^2
```

```
## [1]   1.00   5.29 100.00  25.00
```

Some operations on Vectors

Sorting a vector is easy:

```r
sort(x)
```

```
## [1] -1.0  2.3  5.0 10.0
```

Some operations on Vectors

Sorting a vector is easy:

```
sort(x)
```

```
## [1] -1.0  2.3  5.0 10.0
```

Sorting in the decreasing order is easy too:

```
sort(x, decreasing = TRUE)
```

```
## [1] 10.0  5.0  2.3 -1.0
```

Choosing a subvector

Let

```r
x <- c(3, -1, 4, 3, 2, 5)
```

Choosing a subvector

Let

```r
x <- c(3, -1, 4, 3, 2, 5)
```

Then we can use

```r
x[2] #The second element
```

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

```r
x[2:4] #x[2], x[3] and x[4]
```

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

```r
x[c(2,4)] #x[2] and x[4]
```

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

```r
x[-3] #everything without x[3]
```

```
## [1]  3 -1  3  2  5
```

Choosing a subvector

A little bit complicated example is

```
x[x>0] #all positive elements
```

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

Choosing a subvector

A little bit complicated example is

```
x[x>0] #all positive elements
```

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

You can use `x>0` to see what this command is doing:

```
x>0
```

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

Choosing a subvector

A little bit complicated example is

```
x[x>0] #all positive elements
```

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

You can use `x>0` to see what this command is doing:

```
x>0
```

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

Another example:

```
x[x%%2 == 0] #only the even elements of x
```

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

Making a DataFrame

DataFrame is one of the important objects in Data Analysis. It is a rectangular data set, similar to the MS Excel Table/Spreadsheet, if you know what it is $\ddot\smile$

Let us make a DataFrame in **R**:

```r
x <- c(22,43,16,38)
y <- c(76, 81, 55, 66)
df <- data.frame(age=x, weight=y)
```

And here is our DataFrame:

```r
df
```

```
##   age weight
## 1  22     76
## 2  43     81
## 3  16     55
## 4  38     66
```

Viewing/Editing DataFrames

Well, besides just typing the name of the DataFrame, or calling `print(df)`, one can use the following command to view the DataFrame:

```
View(df)
```

or to edit the DataFreame:

```
edit(df)
```

Accessing the columns/rows of a DataFrame

You can access the column “weight” by

```
df[,2]
```

```
## [1] 76 81 55 66
```

or by

```
df[,'weight']  # also df[,"weight"]
```

```
## [1] 76 81 55 66
```

or by

```
df$weight
```

```
## [1] 76 81 55 66
```

Some Examples

```r
mean(df$age)  # mean age
```

```
## [1] 29.75
```

```r
sum(df$weight) # sum of weights, total weight
```

```
## [1] 278
```

```r
length(df$weight) # the number of elements in df$weights
```

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

```r
sum(df$weight)/length(df$weight) # same as the mean(df$wei
```

```
## [1] 69.5
```

```r
sort(df$weight) # weights in the increasing order
```

```
## [1] 55 66 76 81
```

```r
sort(df$age, decreasing = T) # ages in the decreasing orde
```

R Built-In Datasets

The Basic **R** includes many Datasets to analyse. You can see the datasets supplied by the `datasets` package by running

```r
data()
```

Or, if you will run

```r
data(package="MASS")
```

it will show the avalable datasets in the **R** package `MASS`.

R Built-In Datasets

The Basic **R** includes many Datasets to analyse. You can see the datasets supplied by the `datasets` package by running

```
data()
```

Or, if you will run

```
data(package="MASS")
```

it will show the avalable datasets in the **R** package `MASS`.

For example, `cars` is one of the standard datasets in **R**. To see the content, just run

```
cars
```

```
##    speed dist
## 1      4    2
## 2      4   10
## 3      7    4
## 4      7   22
```

R Built-In Datasets

In our previous example, the dataset was too large to fit into the slide (try to run it in **R** console by yourself). To see the first/last few rows of the dataset, use

```r
head(cars) #first six rows
```

```
##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
## 4     7   22
## 5     8   16
## 6     9   10
```

or

```r
tail(cars, 1) #last 1 row
```

```
##    speed dist
## 50    25   85
```

Installing a Package

There are a lot of supplementary packages designed for extra tasks, giving us different functions and datasets. To install a package, run `install.packages("name_of_the_package")`. For example, to install the `ggplot2` package (for nice graphics, **gg=grammar of graphics**, not a taxi service), you need to run

```r
install.packages("ggplot2")
```

Using a function from a Package

If you want to use a function or a dataset from a package, you need to load that package first, then use the function/dataset. Say, we want to use the `rmvnorm` function from the package `mvtnorm` to generate a random sample of size 100 from the multivariate Normal Distribution. First, install `mvtnorm` by using

```r
install.packages("mvtnorm")
```

Using a function from a Package

If you want to use a function or a dataset from a package, you need to load that package first, then use the function/dataset. Say, we want to use the `rmvnorm` function from the package `mvtnorm` to generate a random sample of size 100 from the multivariate Normal Distribution. First, install `mvtnorm` by using

```
install.packages("mvtnorm")
```

Then run:

```
library(mvtnorm)
mu <- c(1,2) #The Mean of our Bivariate Normal Distributio
sigma <- matrix(c(4,2,2,3), ncol = 2) # The Covariance Mat
x <- rmvnorm(n = 100, mean = mu, sigma = sigma)
```

Here the command `library(mvtnorm)` is similar to `include` of `C++` or `import` of `Python`, and the above code will give an error without this line.

Using a function from a Package

Another way, without importing the package by the `library` command, is to use

```
x <- mvtnorm::rmvnorm(n = 100, mean = mu, sigma = sigma)
```

Using a function from a Package

Another way, without importing the package by the `library` command, is to use

```
x <- mvtnorm::rmvnorm(n = 100, mean = mu, sigma = sigma)
```

Anothe example:

```
MASS::SP500
```

will print the SP500 dataset of the `MASS` package (returns of the S&P500 index in 1990's).

Using a function from a Package

Another way, without importing the package by the `library` command, is to use

```
x <- mvtnorm::rmvnorm(n = 100, mean = mu, sigma = sigma)
```

Anothe example:

```
MASS::SP500
```

will print the SP500 dataset of the `MASS` package (returns of the S&P500 index in 1990's).

To see the list of loaded base/attached packages, you can use

```
sessionInfo()
```

Using Help

To open the **R**'s help page for some command, say, for the `sum` command, you can use on of the following options:

```
?sum
```

or

```
help(sum)
```

or, in **R Studio**, just put the cursor at some place on the word `sum` and hit *F1*. Anothe method is just to use the Google $\ddot\smile$

Making Plots

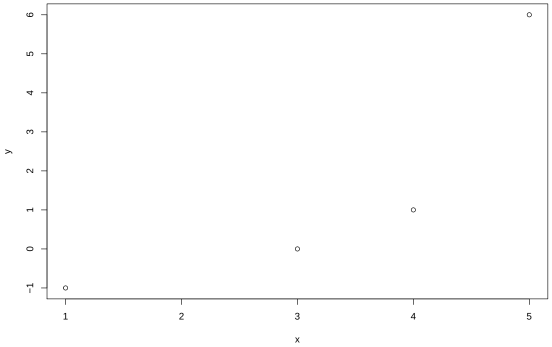
Here we will use the default package for plots[4].

[4]More you can get using the `ggplot2` package

Making Plots

Here we will use the default package for plots[4]. First, let us plot some points (x_i, y_i), $i = 1, ..., n$:

```r
x <- c(1,3,5,4); y <- c(-1, 0 ,6, 1)
plot(x,y)
```

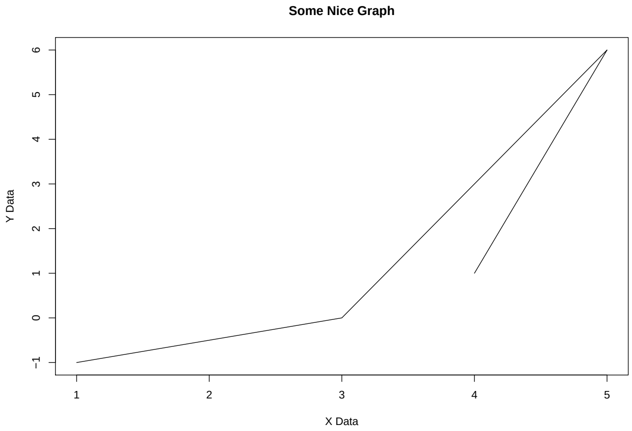


[4]More you can get using the ggplot2 package

Making Plots

Now, we join the points with line segments, add a title and axis labels to our graph:

```
plot(x,y, type = "l", main = "Some Nice Graph",
     xlab = "X Data", ylab = "Y Data")
```



Making Plots

Comments:

- `type` is the type of the plot. `type="l"` means that the type is set to lines. Try without the type parameter, or by `type = "h"`, `type = "s"`
- `main` is the title of the graph
- `xlab` and `ylab` are the axis labels (names)

Making Plots

In **R**, we have low-level and high-level graphical commands. The `plot` command is a low-level plot. If you will use 2 `plot` commands one after another, the last command will draw on a new canvas. So if you want to draw two graphs one over anoher, you cannot just use two `plot` commands consecutively.

[5]In fact, in **R Studio**, under the **Plots** tab, you can navigate through the plots

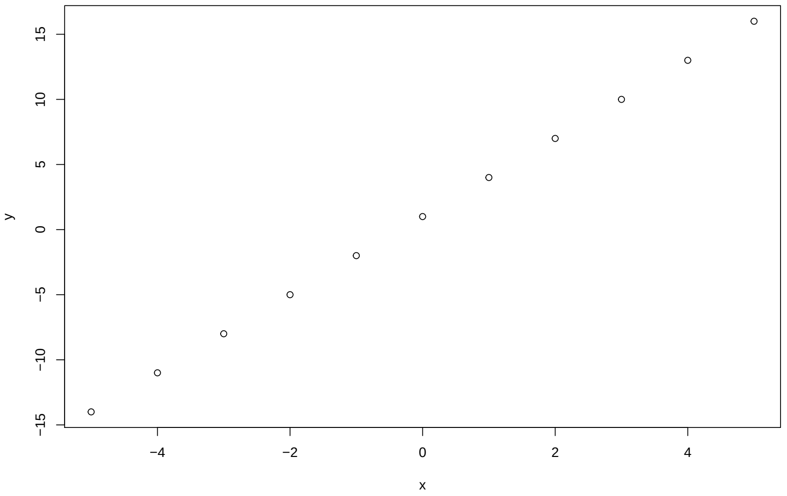
Making Plots

In **R**, we have low-level and high-level graphical commands. The `plot` command is a low-level plot. If you will use 2 `plot` commands one after another, the last command will draw on a new canvas. So if you want to draw two graphs one over anoher, you cannot just use two `plot` commands consecutively. In general, when using 2 low-level graphical commands, the last one will "overwrite" the previous plot [5]. But, when using high-level plotting commands after the low-level one, you will have the high-level plot added to the low-level one. High-level plot examples are `points` and `lines`.

[5]In fact, in **R Studio**, under the **Plots** tab, you can navigate through the plots

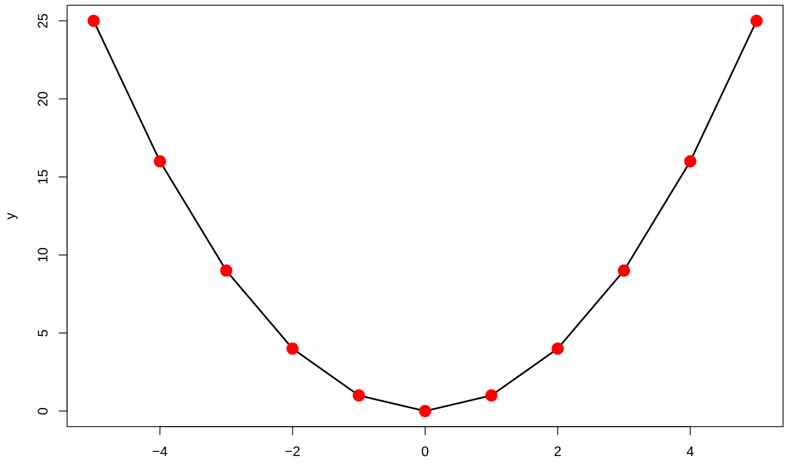
Making Plots, Example

```r
x <- -5:5; y <- 3*x+1
plot(x,y)
```



Making Plots

```r
x <- -5:5; y <- x^2
plot(x,y, type = "l", lwd = 2)
points(x,y, pch = 16, col = "red", cex = 2)
```



Making Plots

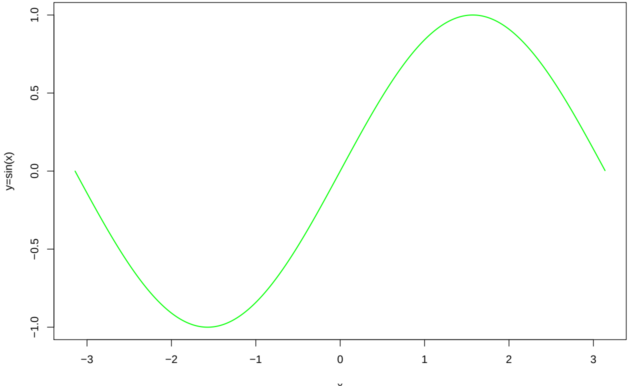
Comments:

- `lwd` is the *line width*
- `pch` is the *point character*, try to change 16 to other integers
- `col` is the *color*
- `cex` is the *character size*, *point size*

Making Plots

Let us plot the graph of $y = \sin(x)$, $x \in [-\pi, \pi]$:

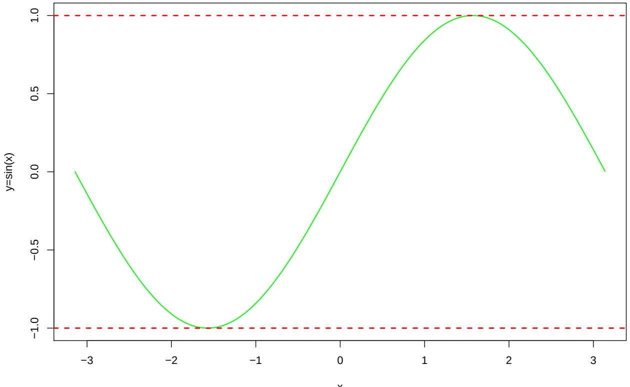
```r
x <- seq(from = -pi, to = pi, by = 0.01)
y <- sin(x)
plot(x,y, type = "l", lwd =1.5, col = 'green',
     xlab = "x", ylab = "y=sin(x)")
```



Making Plots

Now, let us add horizontal lines $y = \pm 1$ to previous plot:

```r
plot(x,y, type = "l", lwd =1.5, col = 'green',
     xlab = "x", ylab = "y=sin(x)")
abline(h = 1, lty = 2, lwd = 2, col = "red")
abline(h = -1, lty = 2, lwd = 2, col = "red")
```



Making Plots

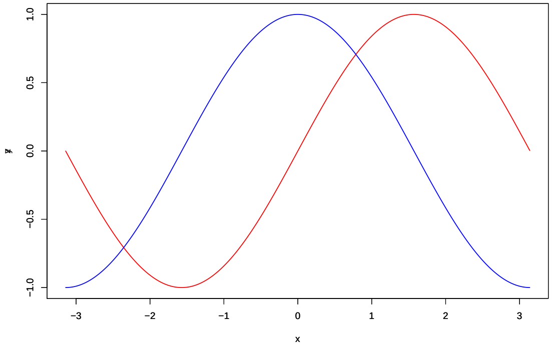
Comments:

- `abline` is to draw a line $y = a + b \cdot x$, you can give parameters $h = h_0$ for a horizontal line $y = h_0$ or $v = v_0$ for a vertical line $x = v_0$. `abline` is a high-level graphical command
- `lty` is the *line type*, try changing the values to see the difference (here we have a *dotted line*)

Making Plots

Now, let us draw two plots one over another:

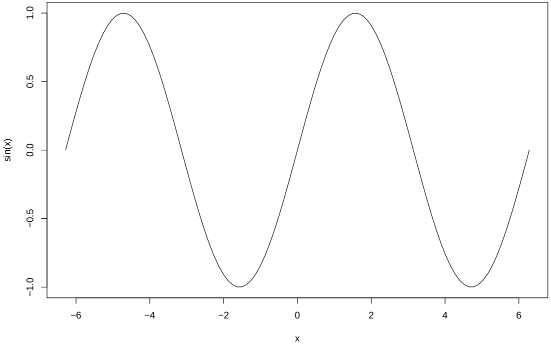
```
x <- seq(from = -pi, to = pi, by = 0.01)
y <- sin(x); z <- cos(x)
plot(x,y, type = "l", lwd =1.5, col = 'red')
par(new = TRUE) #setting a parameter to keep the previous p
plot(x,z, type = "l", lwd =1.5, col = 'blue')
```



Making Plots

Another method to draw graphs is to use the `curve` command:

```r
curve(sin, from = -2*pi, to = 2*pi)
```



Try also[6]

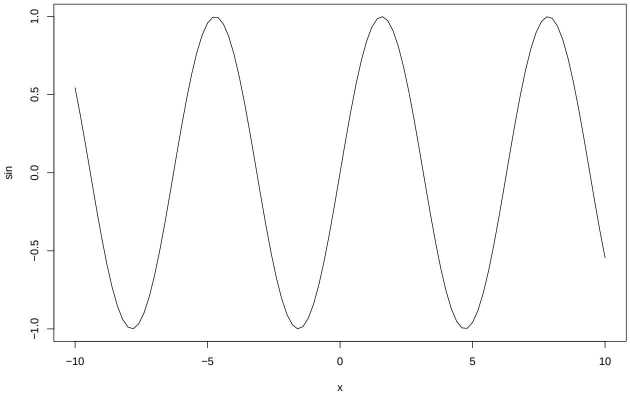
```r
curve(sin(x), from = -2*pi, to = 2*pi)
```

[6] If the variable x is not defined!

Making Plots

Yet another method to draw graphs is to use the `plot` command with function name:

```
plot(sin, -10, 10)
```



Defining Functions

It is easy to define a function in **R**. You just need to use the following template:

For example, let us define the function $z = x^2 + 2 * \arctan \frac{x}{x+\sin(x)}$:

```r
my.nicefunction <- function (x){
  z = x^2 +2* atan(x/(x+sin(x)))
  return(z)
}
```

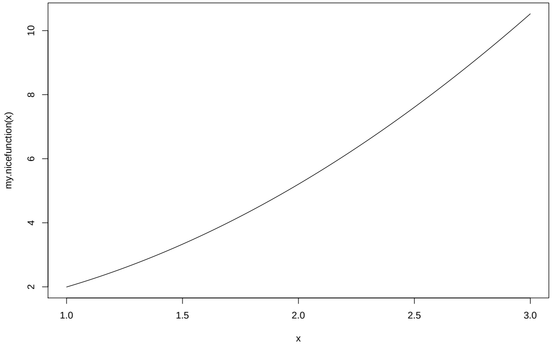
Defining Functions

Now, we can calculate our function's value at, say, the point 3, and plot its graph:

```
my.nicefunction(3)
```

```
## [1] 10.52485
```

```
curve(my.nicefunction, 1, 3) #or plot(my.nicefunction, 1, 3
```



Defining Functions

Now, let us define a function of 2 variables:

```
my_fun1 <- function (x, y){
  return(x+y^2)
}
```

In this case, to calculate the value at $(1, 2)$, you just write

```
my_fun1(1,2)
```

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

Defining Functions

You can define also functions with default values:

```r
my_fun2 <- function (x, y = 0){
  return(x+y^2)
}
```

Now, `my_fun(2)` will give `my_fun(2,0)`, and `my_fun(2, 4)` will calculate `my_fun(2,4)`:

```r
my_fun2(2)
```

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

```r
my_fun2(2,4)
```

```
## [1] 18
```

Defining Functions

By the way, if you will give default values to each variable, like here:

```r
my_fun3 <- function (x = 0, y = 0){
  return(x+y^2)
}
```

Then you can use the followings to calculate `my_fun3(2,1)`:

```r
my_fun3(2,1)
```

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

```r
my_fun3(x = 2, y = 1)
```

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

```r
my_fun3(y = 1, x = 2)
```

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

Defining Functions

Actually, most of the built-in **R** functions have named variables with default values. Say, use

You will see the help page, containing the usage of the function `rnorm`, in the form:

```
rnorm(n, mean = 0, sd = 1)
```

This means, that if you will use `rnorm(10)`, it will assume `mean = 0` and `sd = 1`. And you can mix the order of the *named* variables, say, run `rnorm(10, sd = 3, mean = -1)`. But please note that you cannot skip the first variable, the value of `n`, since it has no default value !