R Basics

Last Updated 28 Sep 2022

Dr Wei Miao

2022-09-28

Table of contents

1 Hello R

1.1 Bilingual arrangements at MSc BA

- Primary language is Python
    - Programming (MSIN00143), Business Strategy (MSIN0093), Machine Learning electives
- Secondary language is R
    - Marketing Analytics (MSIN0094), Operations Analytics (MSIN0095), Statistical Foundations (MSIN0096)

1.2 A brief history of R

- R project was initiated by **R**obert Gentleman and **R**oss Ihaka (Univ of Auckland) in 1991; both are statisticians, who later made the language open-source.
- Since 1997, R has been developed by the R Core Team on CRAN.

- As of January 2022, it has 18,728 contributed packages. As of March 2022, R ranks 11th in the TIOBE index[1]; the language peaked in 8th place in August 2020.

1.3 Why learn R?

- Super powerful data analytics and visualizations, including[2]
  - Data wrangling (`dplyr`) and data visualization (`ggplot`)
  - Econometrics (numerous packages)
  - Predictive analytics (numerous packages)
- Write beautiful reports/dissertations/presentations using `quarto`
  - Write your MSc dissertation (highly recommended; super efficient)
  - Effortlessly build websites. I built and maintain my personal website and the marketing course website all in R.

1.4 One-One comparison with Python

Table 1: R versus Python

	R	Python
Language purpose	R is a **statistical language** specialized in the **data analytics and visualization**. Best for data science, may not be robust for production environment.	Python is a **general-purpose language** that is used for the deployment and development of various projects. Best for production environment.
Data analytics	R is better at **statistical models** and **econometrics**.	Python is better at **machine learning** due to support from PyTorch and TensorFlow.
IDEs	RStudio	Many options such as Jupyter Notebook, Spyder, Pycharm, etc.
Targeted users	Primary users of R include **researchers** in academia and **data scientists**, who heavily rely on data analyses and visualization.	Primary users of python include **developers and programmers.**

[1] A measure of programming language popularity

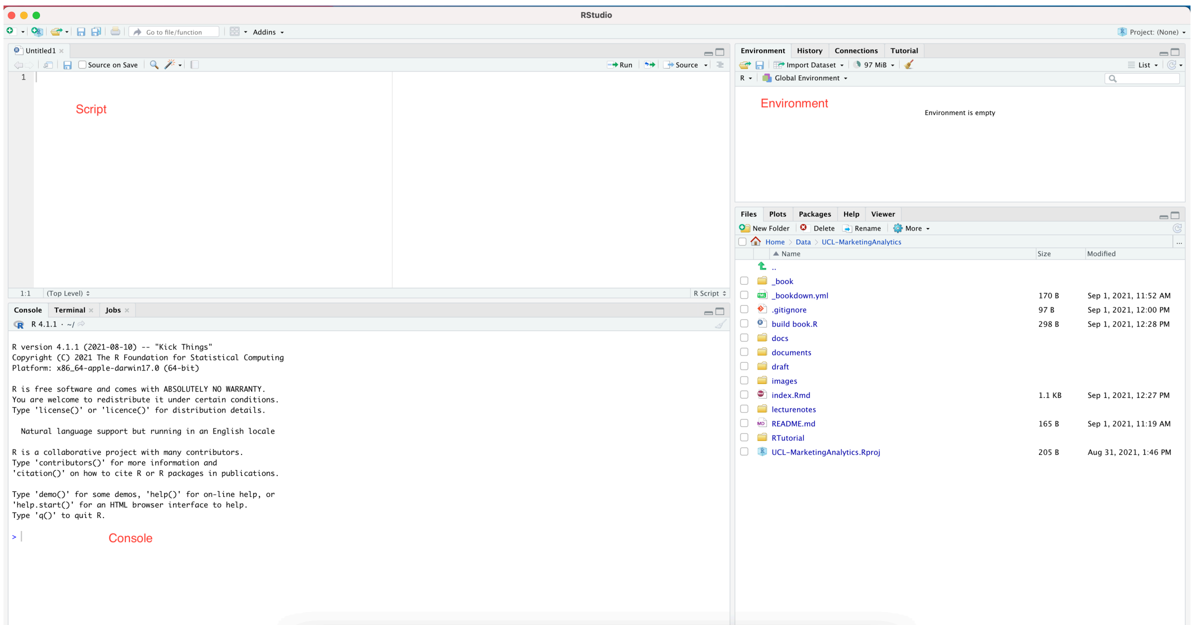
[2] There are many R-exclusive packages, such as the state-of-the-art causal machine learning library `grf` , which we will learn in the final week.

1.5 A first look at the RStudio Interface

R is the **programming language**, and we need a “place” to write codes. This place is called an **Integrated development environment (IDE)**.

RStudio is THE BEST R IDE to date. And it’s interface consists of the following:

- ***script***: (top left) where you do the coding
- ***console***: (bottom left) where you can run commands interactively with R and see code outputs
- ***environment***: (top right) a list of named objects that we have generated
- ***history***: (top right) the list of past commands that we have used
- ***help***: (bottom right) user manuals of functions available in R
- ***package***: (bottom right) a collection of ready-to-use packages written by others



1.6 Where to write R codes (I): Console

- You can write codes *interactively* in the R console. See an example: Type the following code into your console and see what happens.

  ```
  print('Hello World')
  ```

  ```
  [1] "Hello World"
  ```

- Often used for simple exploratory tasks, where you don’t need to keep a record of codes.
  - check summary statistics; inspect datasets; etc.

1.7 Where to write R codes (II): `.R` script

- `R script` is a text-readable file ending with `.R` suffix. See an example.
  - codes can be run line-by-line or *sourced* altogether

> **!** Important
>
> All texts in the script will be treated as R codes unless commented out.

- Often used for project development and deployment, where you don't need to communicate results to others

1.8 Where to write R codes (III): `.qmd` script

- `Quarto`[3] files have a `.qmd` suffix. You can think of Quarto as Microsoft Word that can run R codes.
- Quarto can create dynamic content with Python and R, conveniently combining data analytics work with beautiful reporting.
  - Quarto can be thought of as the R equivalent of Jupyter Notebook but is much more powerful.
  - We will be mainly using Quarto in the marketing analytics module. You can also use Quarto to do your assignments, write your dissertation, and build your own blogging websites.
- Let's create a new quarto file together!

2 Introduction to Quarto

2.1 YAML header

- You can think of YAML header as a MS Word template, which determines how your final report looks like (font, font size, color, margins, etc.).
- The YAML header is typically at the beginning of a document, separated from the main text by three dashes (`---`). YAML will not appear in the final report.
- To make life easier, I will set `YAML` headers for all `.qmd` files for you in Marketing Analytics module.

[3] Why the name Quarto? "We wanted to use a name that had meaning in the history of publishing and landed on Quarto, which is the format of a book or pamphlet produced from full sheets printed with eight pages of text, four to a side, then folded twice to produce four leaves. The earliest known European printed book is a Quarto, the Sibyllenbuch, believed to have been printed by Johannes Gutenberg in 1452–53."

2.2 Authoring with normal texts

RStudio provides two ways to edit a quarto file (1) **visual mode** and (2) **source mode**.

- RStudio's visual editor offers an WYSIWYM (Microsoft Word like) authoring experience for markdown
  - recommended and easier to learn; we will be using this mode in class
  - check the rich formatting tools we can use for authoring a report
- In the source mode, you can edit the file using markdown syntax
  - optional; for advanced users once you're familiar with the markdown syntax

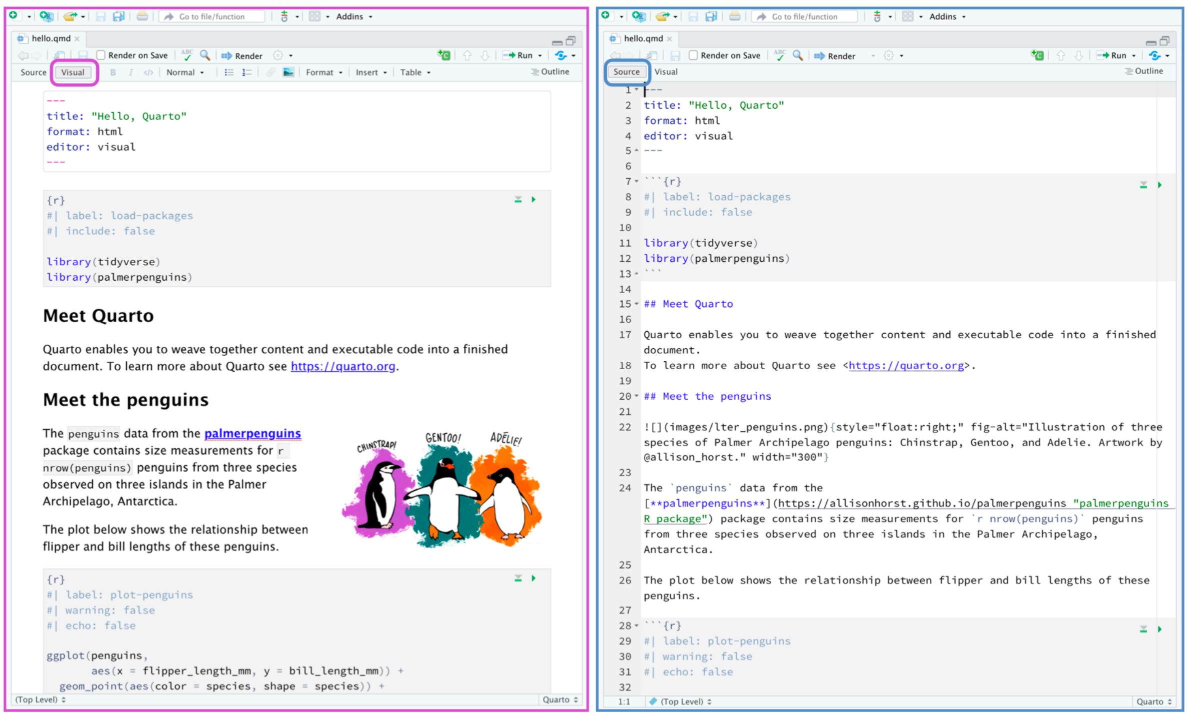


Figure 1: Visual Mode versus Source Mode

> **Exercise**
>
> Create a new quarto file from RStudio with the following level-1 and level-2 headers
>
> - Basics of R
> - Vectors
>   - Creating vectors

2.3 Coding with code blocks

- In qmd files, we write actual R codes in **code chunks** identified with `{r}`.
- You can run each code chunk interactively by clicking the render icon. RStudio executes the code and displays the results below the code chunks.
- To insert a code chunk, click `Insert` ->`Code Chunk` -> `R`.
- See an example and try on your computer!

```r
print('R is the Best Language! Better than Python! And dont tell David I said this!')
```

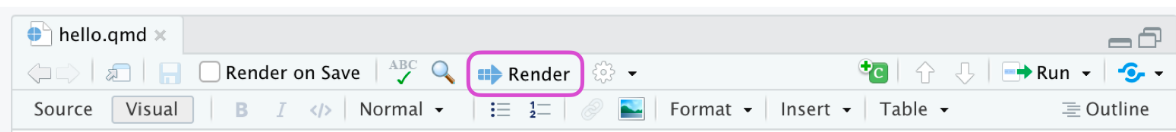
```
[1] "R is the Best Language! Better than Python! And dont tell David I said this!"
```

> **Exercise**
>
> Insert the above R code block in your quarto file under any section.

2.4 Rendering a report

At the end, when codes and main texts are ready, use the **`Render`** button in the RStudio IDE to render the file.

The rendered report will be in the same folder with your qmd file.



> **Exercise**
>
> Render your quarto file into a document and see how it looks like.

2.5 More learning resources for Quarto

- The available YAML fields vary based on document format
  - Here for YAML fields for PDF documents
  - Here for MS Word
  - Here for HTML documents
- Markdown syntax
  - Markdown basics

  - Markdown practice

- Quarto (recommended to be reviewed after-class)

  - Get started

3 Basics of R

3.1 Named objects

- R is an **object-oriented language**, so we will be working on named objects.
- We use the **left arrow `<-`** to create a named object, which assigns the **objects** on the RHS to the **name** on the LHS.[4]

  - The below code creates a new object called 'x' in the **environment**, which is a number 2.

```
x <- 3
x
```

```
[1] 3
```

- After an object is created, we can refer to the object by its name, and operates on it.

```
# Question: why Wei chooses these two numbers?
x^2
```

```
[1] 9
```

```
x^3
```

```
[1] 27
```

> **Exercise**
>
> Insert a code block in your quarto file, which does the following:
>
> - Create an object with name 'x' with value 2 + 2

[4] You can also use equal sign `=`, but it's recommended to stick with R's tradition.

3.2 Rules for object names

For a variable to be valid, it should follow these rules

- It should contain **letters**, **numbers**, and only **dot** or **underscore** characters.
- It *cannot* start with a number (eg: 2iota).

```
# 2iota <- 2
```

- It *cannot* start with a dot followed by a number (eg: .2iota).

```
# .iota <- 2
```

- It should not start with an underscore (eg: _iota).

```
# _iota <- 2
```

- It should not be a reserved keyword.

```
# mean <- 2
```

> **Tip**
>
> It's good practice to use memorable names to name an object
>
> - For instance, use prefix "df_" or "data_" to name datasets.

3.3 Functions

- In R, a **function** takes object(s) as input, run specific actions on the object(s) defined by the function, and then return an outcome object.
  - The example below shows the function `mean`, which computes the average of several numbers.

```
a <- 1:3 # which generates a sequence 1,2,3
a
```

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

```
mean(a)
```

```
[1] 2
```

- We will heavily rely on functions to conduct data analyses. For how to use a new function, search the function in RStudio's `help` panel.

  - **Description**: what the function does in a nutshell
  - **Usage**: how to call the function
  - **Arguments**: how you would like to run the function
  - **Value**: what will be returned
  - **Examples**: examples of how to use the function

> **Exercise**
>
> 1. Search and learn the usage of function "sum"
> 2. Insert a code block in your quarto file to compute the sum of vector 1:3

3.4 Collection of functions: Packages

The base R already has many useful built-in functions to perform basic tasks, but as data scientists, we need more.

To perform certain tasks (such as a machine learning model), we can definitely write our own code from scratch, but it takes lots of (unnecessary) effort. Fortunately, many packages have been written by others for us to directly use.

- Install the package using the built-in function `install.packages()`. R will download the package.

```
install.packages('praise')
```

```
Error in contrib.url(repos, "source"): trying to use CRAN without setting a mirror
```

- Load the packages using `library()`. Every time you restart the RStudio, packages need to be reloaded.

```
library(praise)
```

- Now that the package is loaded, you can use the functions in it. `praise()` is a function in the `praise` package.

```
praise()
```

```
[1] "You are kickass!"
```

> **Tips**
>
> Installation of a package is only needed for the first time. After installation, just need to reload the packages using library() every time your restart RStudio.

3.5 Comment codes

You can put a # before any code, to indicate that any codes after the # on the same line are your comments, and will not be run by R.

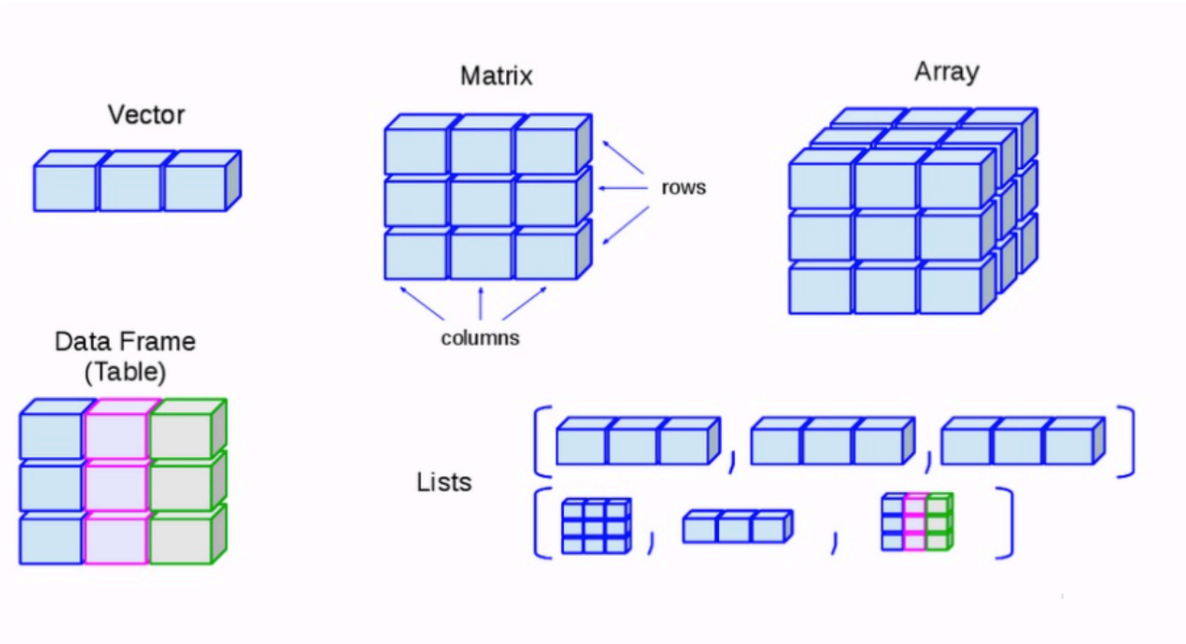
It's a good practice to often comment your codes, so that you can help the future you to remember what you were trying to achieve.

```
# print("Support Wei for an iPhone 14 Pro!")

# Below, x will be 1 rather than 1+1
x <- 1 # +1
```

3.6 Data structures

Below are the complete list of objects in R.



3.7 Data types

To make the best of the R language, you'll need a strong understanding of the basic data types and data structures and how to operate on them. Data structures are very important to understand because these are the objects you will manipulate on a day-to-day basis in R.

- Numeric (e.g.,2.5)
    - We can use R as a calculator for numeric objects

```r
# Numeric Vector
num2 <- 2.5
log(num2)
```

```
[1] 0.9162907
```

```r
num2^2
```

```
[1] 6.25
```

```r
exp(num2)
```

```
[1] 12.18249
```

- Logical (TRUE, FALSE)
  - `TRUE` is equivalent to 1 in R; `FALSE` is equivalent to 0.

```r
log1 <- TRUE
log2 <- FALSE
```

- Character (e.g. "Wei", "UCL", "1 + 1 = 3", "TRUE", etc.)
  - within a pair of quotation marks; single or double quotation marks can both work.

```r
str1 <- "1 + 1 = 2"
```

- Factor ("male", "female", etc.)
  - this is an important class for describing categories. We will discuss in more detail later in class when we learn linear regression.

```r
country <- c('UK','Spain','Italy','Multiverse')
factor(country)
```

```
[1] UK         Spain      Italy      Multiverse
Levels: Italy Multiverse Spain UK
```

3.8 Check data types using class()

We can use `class()` to check the type of an object in R.

```r
a <- '1+1'
class(a)
```

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

```r
b <- 1+1
class(b)
```

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

This is very useful when we first load data from external databases, we need to make sure variables are of the correct data types.

3.9 Data type: conversion

Sometimes, data types of variables from raw data may not be what we want; we need to change the data type of a variable to the appropriate one.

See the following example:

- `a` is a string, and we cannot use mathematical operations on it, or R will report errors.

```r
a <- '1'
class(a)
```

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

```r
a + 1
```

```
Error in a + 1: non-numeric argument to binary operator
```

- We can convert `a` to a numeric value. To convert from character to numeric, we use `as.numeric()`

```r
b <- as.numeric(a)
class(b)
```

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

4 Vectors

4.1 Creating vectors

Creating vectors: c()

Vector can be created using the function `c()` by listing all the values in the parenthesis, separated by comma ','.

```r
x <- c(1, 3, 5, 10)
x
```

```
[1]  1  3  5 10
```

```r
class(x)
```

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

Vectors must contain elements of the same data type. Otherwise, it will implicitly convert elements into the same type.

```r
x <- c(1, "intro", TRUE)

class(x)
```

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

Checking the number of elements in a vector: length()

You can measure the length of a **vector** using the command `length()`

```r
x <- c('R',' is', ' fun')
length(x)
```

```
[1] 3
```

```r
y <- c()
length(y)
```

```
[1] 0
```

Creating numeric sequences: seq() and rep()

It is also possible to easily create sequences with patterns

- use `seq()` to create sequence with fixed steps

```r
# use seq()
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 step is 1, there's a simpler way using :

```r
1:5
```

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

- use `rep()` to create repeated sequences.

```r
# replication using rep()
rep(c("A","B"), times = 5)
```

```
 [1] "A" "B" "A" "B" "A" "B" "A" "B" "A" "B"
```

Combine vectors

You can use `c()` to combine different vectors; this is very commonly used to concatenate vectors.

```r
x <- 1:3 # from 1 to 3
y <- c(10, 15) # 10 and 15
z <- c(x,y) # x first and then y
z
```

```
[1]  1  2  3 10 15
```

> **Exercise**
>
> Create a sequence of {1,1,2,2,3,3,3} using different methods.

4.2 Indexing and subsetting

We put the **index** of elements we would like to extract in a **square bracket `[ ]`**.[5]

- Which element is in the second position?

```r
x <- c(1,3,8,7)
x[2]
```

```
[1] 3
```

- What are the first 2 elements?

```r
x[1:2]
```

```
[1] 1 3
```

- What are the 1st, 3rd and 4th elements?

[5] Note that Python uses different ways to index and subset vectors and matrices.

```r
x[c(1,3,4)]
```

```
[1] 1 8 7
```

4.3 Element-wise operations

R is a **vectorized** language, meaning by default it will do vector operation internally.

- If you operate on a vector with a single number, the operation will be applied to all elements in the vector

```r
x <- c(1,3,8,7)
x+2
```

```
[1]  3  5 10  9
```

```r
x^2
```

```
[1]  1  9 64 49
```

> **Caveats**
>
> When the length of vectors do not match, R will still do it for you without reporting error but a warning message. As you can see, even if the length of vectors does not match, R can still return an output but throws a warning message. It's important to check the warning messages when there is any!

```r
x <- c(1,3,8,7)

y <- c(1,3,4) # careful!!! does not report error
x + y
```

```
Warning in x + y: longer object length is not a multiple of shorter object
length
```

```
[1]  2  6 12  8
```

> **Exercise**
>
> Create a geometric sequence {2,4,8,16,32} using seq().

4.4 Relational operations

- We can compare a vector with a vector **of the same length**, which will do element-wise (element-by-element) comparison

```r
x <- c(1,3,8,7)
y <- c(2,3,7,8)
x > y
```

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

```r
x == y
```

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

- We can also compare a vector with a scalar, because R is vectorized

```r
x <- c(1,3,8,7)
x < 6 # is each element lower than 6?
```

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

```r
x == 10 # is the element equal to 10?
```

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

- Return the positions of elements that satisfy certain conditions: `which()`

```r
which(x == 8) # which element equals 8
```

```
[1] 3
```

```r
which.max(x) # which is the max element
```

```
[1] 3
```

```r
which.min(x)
```

```
[1] 1
```

Exercise

Find the minimum value of vector `x` using `which()`

- Sometimes, we may need to operation on multiple relational operations using `and` `or` `no`

  ```
  T & F # and
  ```

  ```
  [1] FALSE
  ```

  ```
  T | F # or
  ```

  ```
  [1] TRUE
  ```

  ```
  !T # not
  ```

  ```
  [1] FALSE
  ```

  - For instance, we may want to find out elements that are smaller than 8 **and** larger than 3.

```
which(x < 8 & x > 3 )
```

```
[1] 4
```

4.5 Special relational operation: `%in%`

- A special relational operation is `%in%` in R, which tests whether an element exists in the object.

```
x <- c(1,3,8,7)

3 %in% x
```

```
[1] TRUE
```

```
4 %in% x
```

```
[1] FALSE
```

4.6 After-class exercise

- Datacamp Introduction to R, finish the following:
  - Intro to basics
  - Vectors

5 Matrices

5.1 Matrices: creating matrices

Creating matrices: matrix()

- A matrix can be created using the command `matrix()`
  - the first argument is the vector to be converted into matrix
  - the second argument is the number of rows
  - the last argument is the number of cols (optional)

```r
matrix(1:9, nrow = 3, ncol = 3)
```

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

> **!** Important
>
> R by default inserts elements **vertically** by **columns**

- R will fill in the matrix by order and discard the remaining elements once fully filled

```r
matrix(1:9, nrow = 3, ncol = 2)
```

```
Warning in matrix(1:9, nrow = 3, ncol = 2): data length [9] is not a
sub-multiple or multiple of the number of columns [2]
```

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

- R will fill in the matrix by order and recycle to fill in the remaining elements

```r
matrix(1:9, nrow = 3, ncol = 4)
```

```
Warning in matrix(1:9, nrow = 3, ncol = 4): data length [9] is not a
sub-multiple or multiple of the number of columns [4]
```

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

Creating matrices: inserting by row

However, we can ask R to insert by rows by setting the `byrow` argument.

```
matrix(1:9, nrow = 3, ncol = 3, byrow = TRUE)
```

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

Creating matrices: concatenation of matrices cbind() and rbind()

We can use `cbind()` and `rbind()` to concatenate vectors and matrices into new matrices.

- `cbind()` does the column binding

```
x <- cbind(1:3, 4:6) # column bind
x
```

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

- `cbind()` can also operate on matrices.

```
cbind(x,x)
```

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

- `rbind()` does the row binding

```
rbind(7:9, 10:12) # row bind
```

```
     [,1] [,2] [,3]
[1,]    7    8    9
[2,]   10   11   12
```

5.2 Matrices: indexing and subsetting

Matrices have two dimensions: rows and columns. Therefore, to extract elements from a matrix, we just need to specify which row(s) and which column(s) we want.

```r
x
```

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

- Extract an element
    - 1 is specified for row index, so we will extract elements from the first row
    - 1 is specified for column index, so we will extract elements from the the second column
    - Altogether, we extract the single element in row 1, column 2.

```r
x[1,2] # the element in the 1st row, 2nd column
```

```
[1] 4
```

- If we leave blank for a dimension, we extract all elements of that dimension.
    - 1 is specified for row index, so we will extract elements from the first row
    - Nothing is specified for column index, so we will extract all elements from all columns
    - Altogether, we extract all elements in the first row

```r
x[1,] # all elements in the first row
```

```
[1] 1 4
```

> **Exercise**
>
> 1. Extract all elements in the second column
>
> 2. Extract all elements in the first and third rows

5.3 Matrices: operations

Let's use 3 matrices `x`, `y`, and `z`:

```r
x <- matrix(1:6, nrow = 3)
y <- matrix(1:6, byrow = T, nrow = 2)
```

- Functions will be vectorized over all elements in a matrix

```r
x
```

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

```
z<- x^2
z
```

```
     [,1] [,2]
[1,]    1   16
[2,]    4   25
[3,]    9   36
```

Matrices' operations: matrix addition and multiplication

- If the two matrices are of the same dimensions, they can do element-wise operations, including the *

```
x + z   # elementwise addition
```

```
     [,1] [,2]
[1,]    2   20
[2,]    6   30
[3,]   12   42
```

```
x * z
```

```
     [,1] [,2]
[1,]    1   64
[2,]    8  125
[3,]   27  216
```

- We can also use `%*%` to indicate matrix multiplication

```
x%*%y # matrix multiplication
```

```
     [,1] [,2] [,3]
[1,]   17   22   27
[2,]   22   29   36
[3,]   27   36   45
```

Matrices' operations: inverse and transpose

- We use `t()` to do matrix transpose

```
t(x) # transpose
```

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

- We use `solve()` to get the inverse of an matrix

```
solve(t(x)%*%x) # inverse; must be on a square matrix
```

```
           [,1]       [,2]
[1,]  1.4259259 -0.5925926
[2,] -0.5925926  0.2592593
```

6 Data Frames

6.1 Data Frames: creating dataframe

Data Frames: create dataframe using data.frame()

- Data Frame is the R object that we will deal with most of the time in the MSc program. You can think of `data.frame` as a spreadsheet in excel.

```
df <-  data.frame(id = 1:4,
  name = c("David", "Yongdong", "Anil", "Wei"),
  wage = rnorm(n=4, mean = 10^5, sd = 10^3),
  male = c(T, T, T, T)
  )
df
```

```
  id     name      wage male
1  1    David  99104.43 TRUE
2  2 Yongdong  99305.23 TRUE
3  3     Anil 101147.37 TRUE
4  4      Wei  99366.52 TRUE
```

- Data frames can also be created from external sources, e.g., from a csv file or database.

6.2 Data Frames: Basics

- Each row stands for an `observation`; each column stands for a `variable`.
- Each column should have a **unique** name.
- Each column must contain the same data type, but the different columns can store different data types.
  - compare with matrix?
- Each column must be of same length, because rows have the same length across variables.

6.3 Data Frames: check dimensions and variable types

- You can verify the size of the `data.frame` using the command `dim()`; or `nrow()` and `ncol()`

```
dim(df)
```

```
[1] 4 4
```

```
nrow(df)
```

```
[1] 4
```

```
ncol(df)
```

```
[1] 4
```

- You can get the data type info using the command `str()`

```
class(df)
```

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

```
str(df)
```

```
'data.frame':	4 obs. of  4 variables:
 $ id  : int  1 2 3 4
 $ name: chr  "David" "Yongdong" "Anil" "Wei"
 $ wage: num  99104 99305 101147 99367
 $ male: logi  TRUE TRUE TRUE TRUE
```

- Get the variables names

```r
names(df)
```

```
[1] "id"   "name" "wage" "male"
```

6.4 Data Frames: summary

- Summarize the data frame

```r
summary(df)
```

```
       id           name                wage             male        
 Min.   :1.00   Length:4           Min.   : 99104   Mode:logical  
 1st Qu.:1.75   Class :character   1st Qu.: 99255   TRUE:4        
 Median :2.50   Mode  :character   Median : 99336                 
 Mean   :2.50                      Mean   : 99731                 
 3rd Qu.:3.25                      3rd Qu.: 99812                 
 Max.   :4.00                      Max.   :101147                 
```

6.5 Data Frames: subsetting

Since a dataframe is essentially a matrix, all the subsetting syntax with matrices can be applied here.

```r
df$name # subset a column
```

```
[1] "David"    "Yongdong" "Anil"     "Wei"
```

```r
df[,c(2,3)] # can also subset like a matrix
```

```
      name      wage
1    David  99104.43
2 Yongdong  99305.23
3     Anil 101147.37
4      Wei  99366.52
```

We are interesting in the cylinders and the weights of inefficient cars (lower than 15 miles per gallon).

```r
poll_cars <- mtcars[mtcars$mpg<15, c("cyl", "wt")] # remember to assign the generated dataframe to
poll_cars
```

```
                    cyl    wt
Duster 360            8 3.570
Cadillac Fleetwood    8 5.250
Lincoln Continental   8 5.424
Chrysler Imperial     8 5.345
Camaro Z28            8 3.840
```

7 Other data structures (Optional)

7.1 Arrays

- We can use `array()` to generate a high-dimensional array
- Just like vectors and matrices, arrays can include only data types of the same kind.
- A 3D array is basically a combination of matrices each laid on top of other

```
x <- 1:4
x <- array(data = x, dim = c(2,3,2))
x
```

```
, , 1

     [,1] [,2] [,3]
[1,]    1    3    1
[2,]    2    4    2

, , 2

     [,1] [,2] [,3]
[1,]    3    1    3
[2,]    4    2    4
```

7.2 Lists

A list is an R object that can contain anything. List is pretty useful when you need to store objects for latter use.

```
x <- 1:2
y <- c("a", "b")
L <- list( numbers = x, letters = y)
```

7.3 Lists: indexing and subsetting

There are many ways to extract a certain element from a list.

- by index
- by the name of the element
- by dollar sign $

```
L[[1]] # extract the first element
```

```
[1] 1 2
```

```
L[['numbers']] # based on element name
```

```
[1] 1 2
```

```
L$numbers # extract the element called numbers
```

```
[1] 1 2
```

After extracting the element, we can work on the element further:

```
L$numbers[1:3] > 2
```

```
[1] FALSE FALSE    NA
```

8 Programming Basics

8.1 if/else

Sometimes, you want to run your code based on different conditions. For instance, if the observation is a missing value, then use the population average to impute the missing value. This is where `if/else` kicks in.

```
if (condition == TRUE) {
  action 1
} else if (condition == TRUE ){
  action 2
} else {
  action 3
}
```

Example 1:

```r
a <- 15

if (a > 10) {
larger_than_10 <- TRUE
} else {
  larger_than_10 <- FALSE
}

larger_than_10
```

```
[1] TRUE
```

Example 2:

```r
x <- -5
if(x > 0){
  print("x is a non-negative number")
} else {
  print("x is a negative number")
}
```

```
[1] "x is a negative number"
```

8.2 Loops

As the name suggests, in a loop the program repeats a set of instructions many times, until the stopping criteria is met.

Loop is very useful for repetitive jobs.

```r
for (i in 1:10){ # i is the iterator
  # loop body: gets executed each time
  # the value of i changes with each iteration
}
```

8.3 Nested loops

We can also nest loops into other loops.

```r
x <- cbind(1:3, 4:6) # column bind
x
```

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

```r
y <- cbind(7:9, 10:12) # row bind
y
```

```
     [,1] [,2]
[1,]    7   10
[2,]    8   11
[3,]    9   12
```

```r
z <- x

for (i in 1:nrow(x)) {
  for (j in 1:ncol(x)){
    z[i,j] <- x[i,j] + y[i,j]
  }
}

z
```

```
     [,1] [,2]
[1,]    8   14
[2,]   10   16
[3,]   12   18
```

8.4 Functions

A function takes the argument as input, run some specified actions, and then return the result to us.

Functions are very useful. When we would like to test different ideas, we can combine functions with loops: We can write a function which takes different parameters as input, and we can use a loop to go through all the possible combinations of parameters.

User-defined function syntax

Here is how to define a function in general:

```r
function_name <- function(arg1 ,arg2 = default_value){
  # write the actions to be done with arg1 and arg2
  # you can have any number of arguments, with or without defaults
  return() # the last line is to return some value
}
```

Example:

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

magic(1,3)
```

```
[1] 4
```

8.5 A comprehensive example

Task: write a function, which takes a vector as input, and returns the max value of the vector

```r
get_max <- function(input){
  max_value <- input[1]
  for (i in 2:length(input) ) {
    if (input[i] > max_value) {
      max <- input[i]
    }
  }

  return(max)
}

get_max(c(-1,3,2))
```

```
[1] 2
```

Exercise

Write your own version of `which.max()` function