R Basics

Dr Wei Miao

September 27, 2024

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), 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 almost 20k contributed packages.
- As of 2024, R is ranked 18th in the TIOBE index[1].

[1] A measure of programming language popularity

1.3 Why learn R?

- Super powerful data analytics and visualizations, including[2]
  - Data wrangling (`dplyr`) and data visualization (`ggplot`)
  - Statistics and Econometrics (major advantage of R over Python)
  - Predictive analytics such as machine learning
- Write beautiful reports/dissertations/presentations using `Quarto`
  - Write your MSc dissertation
  - 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 (Intergrated Development Environment)	RStudio	Many options such as Jupyter Notebook, Spyder, Pycharm, etc.
Targeted users	Primary users of R include **data scientists** and **researchers** in academia, who heavily rely on data analyses and visualization.	Primary users of python include **developers and programmers.**

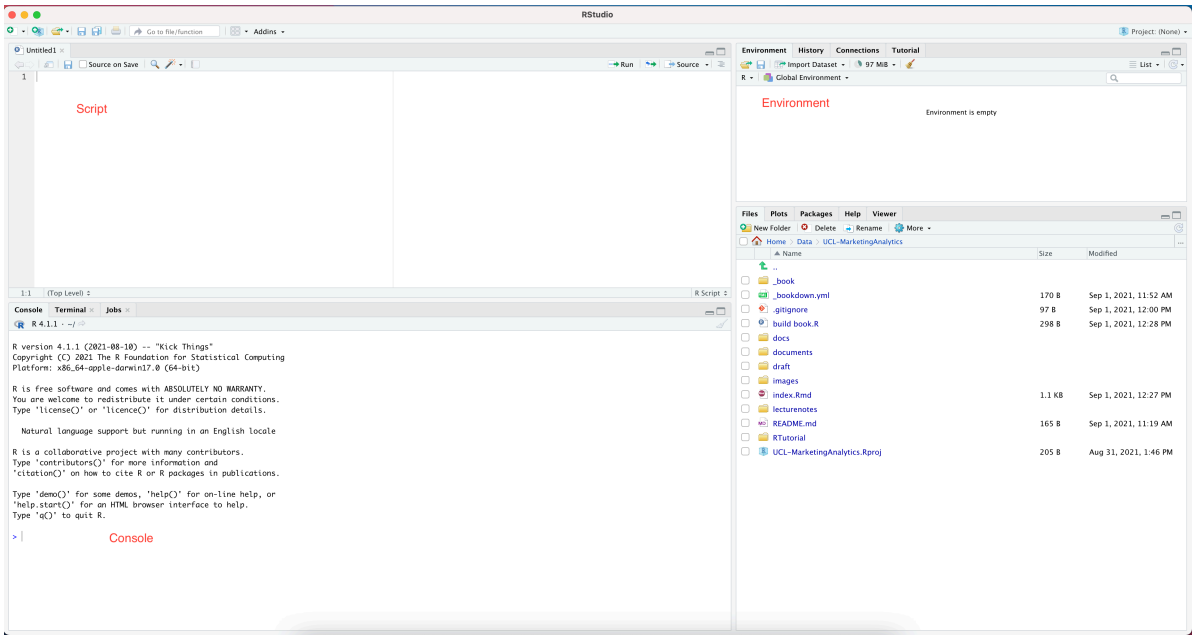
1.5 A first look at the RStudio Interface

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

RStudio is so far the best R IDE. And it's interface consists of the following major panels (clockwise from top left):

[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.

- ***script***: (top left) where you do the coding
- ***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
- ***console***: (bottom left) where you can run commands interactively with R and see code outputs



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"
```

- Used **for simple exploratory, unstructured tasks**, where you don't need to keep a record of codes.
  - e.g., summary statistics; check variable values, etc.

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

- `Quarto`[3] files have a `.qmd` suffix. You can think of Quarto as **Microsoft Word that can run R codes**.
- If you have experience with Jupyter Notebook, Quarto is the R equivalent of Jupyter Notebook, but just much more powerful.
- Quarto can create dynamic contents with R, conveniently combining data analytics work with beautiful reporting.
- Now, let's create a new quarto file together! Name it "MyFavoriateShow.qmd" and save it to your download folder.

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 always at the beginning of a document, separated from the main text by three dashes (`---`). YAML does not appear in the final report.

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 a Microsoft-Word like experience for you to write R codes.
  - Check the rich formatting tools we can use for authoring a report
- If you are familiar with markdown syntax, you can use the **source mode** to write the report (optional; for advanced users only).

[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."

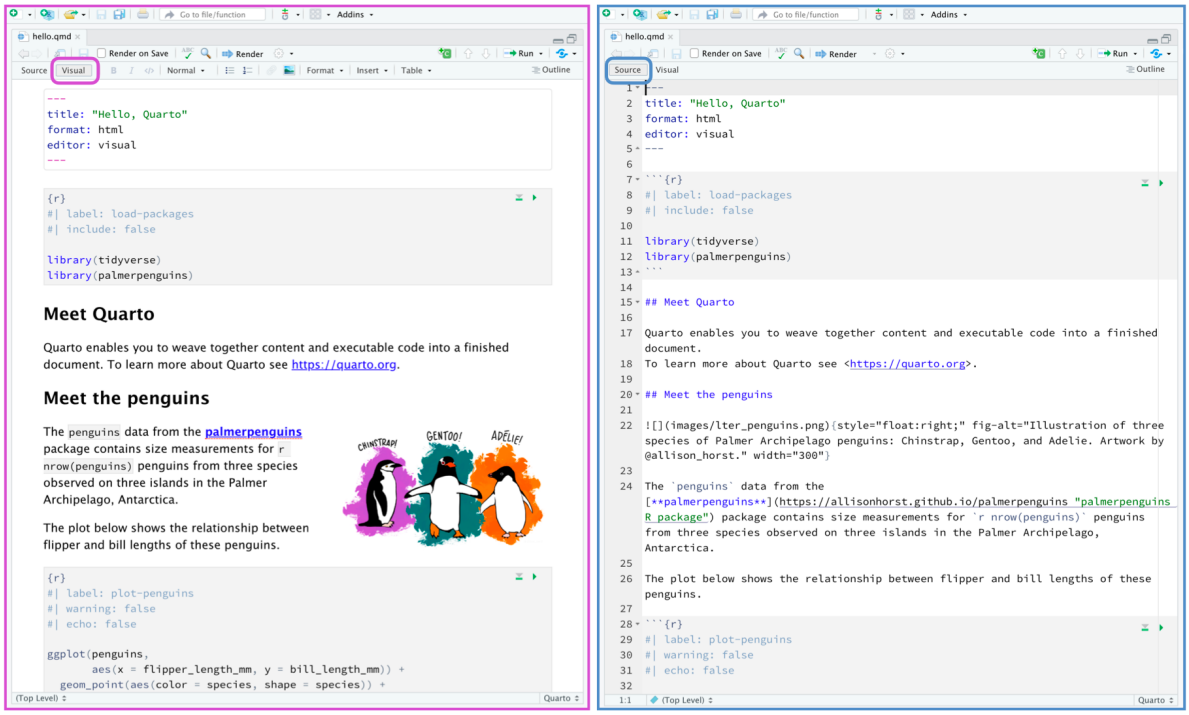


Figure 1: Visual Mode versus Source Mode

> **Exercise**
>
> Create a new quarto file from RStudio with the following level-1 and level-2 headers
> Level 1: Slowhorse Season 4
> Level 2: Episode 1: Identity Theft
> Body: A London bombing puts Taverner under pressure. When River grows concerned for his grandfather, Louisa encourages him to go for a visit.

2.3 Coding with code blocks

- In qmd files, we write R codes in so-called **code chunks** (sometimes referred to as code cells or code blocks) identified with `{r}`.
- To insert a code chunk, click `Insert` ->`Code Chunk` -> `R`. You can also use the shortcut `Ctrl + Alt + I` or `Cmd + Option + I`.

> **Caveat**
>
> Leave the first line as {r} only, do not write anything else on the first line!

- You can run each code chunk interactively by clicking the green solid triangle (run current code chunk). RStudio executes the codes in the code chunk and displays the results.

- See an example and try on your computer!

```
print('R is the Best Language! Way better than Python! The battle is on!')
```

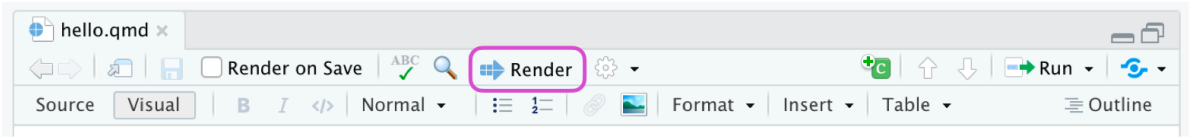
```
[1] "R is the Best Language! Way better than Python! The battle is on!"
```

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

2.4 Rendering a report

At the end, when the Quarto document (including 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 (Optional)

- 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, the keyboard shortcut for `<-` for windows users is `Alt` + `-`, or for mac users, `Option` + `-`.
- The `<-` is an assignment operator, which assigns the **R objects** on the RHS to the **name** on the LHS.[4]
- The below code creates a new object called 'x' in R; x is a numeric object; its value is 3.

```
# create an object x with value 3
x <- 3
```

- After an object is created, we can refer to the object by its name

```
# print out x
x
```

```
[1] 3
```

- We can also perform operations on the object

```
# Question: hmmm, why does 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 the formula of 2 + 2

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

3.2 Rules for naming object

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), or a dot, or an underscore.

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

- It should not be a reserved word in R (eg: mean, sum, etc.).

```
# mean <- 2
```

> Tips
>
> In the future, 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

- A **function** usually takes (one or several) objects as input, run specific operations on the object(s) defined by the function, and then return an output.
- For instance, an R's built-in function `sqrt()` takes a number as input, and returns the square root of the number. Let's use it on object `x`.

```
sqrt(x)
```

```
[1] 1.732051
```

- 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.

> Exercise
>
> 1. Search and learn the usage of function "log()".
> 2. Insert a code block in your quarto file to compute the logarithm of x.

3.4 Collection of functions: Packages

The base R already comes with 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.

- To download a package, hit `Tools -> Install Packages` in RStudio, and type the package name in the pop-up window. Now, download the package `praise`.
- To load the packages, we need to type `library()`.

```
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 riveting!"
```

> Tips
>
> - Packages need to be downloaded only once, but need to be loaded every time you restart the 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("Let's fund Wei for an iPhone 16 Pro Max as a birthday gift!")
```

```
# Is x 1 or 2 below?
x <- 1 # +1
```

4 Data structures

4.1 Data types

Numeric

- We can use R as a calculator for numeric objects

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

```
[1] 0.9162907
```

```
num2^2
```

```
[1] 6.25
```

```
exp(num2)
```

```
[1] 12.18249
```

Logical (TRUE, FALSE):

- Logical objects are used to store logical values, such as `TRUE` and `FALSE`.

```
num2 <- 2.5

# larger than 2?
num2 > 2
```

```
[1] TRUE
```

```
# smaller than 2?
num2 < 2
```

```
[1] FALSE
```

```
# equal to 2?
num2 == 2
```

```
[1] FALSE
```

```r
# not equal to 2?
num2 != 2
```

```
[1] TRUE
```

- Sometimes, we may need to operation on multiple relational operations. We can use **logical operators** to combine multiple relational operations.

```r
T & F # and
```

```
[1] FALSE
```

```r
T | F # or
```

```
[1] TRUE
```

```r
!T # not
```

```
[1] FALSE
```

- For instance, we may want to know if a number is between 3 and 8.

```r
num2 >= 3 & num2 <= 8
```

```
[1] FALSE
```

Character:

- Characters are enclosed within a pair of quotation marks.
- Single or double quotation marks can both work.
- If even a character may contain numbers, it will be treated as a character, and R will not perform any mathematical operations on it.

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

4.2 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.

4.3 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
```

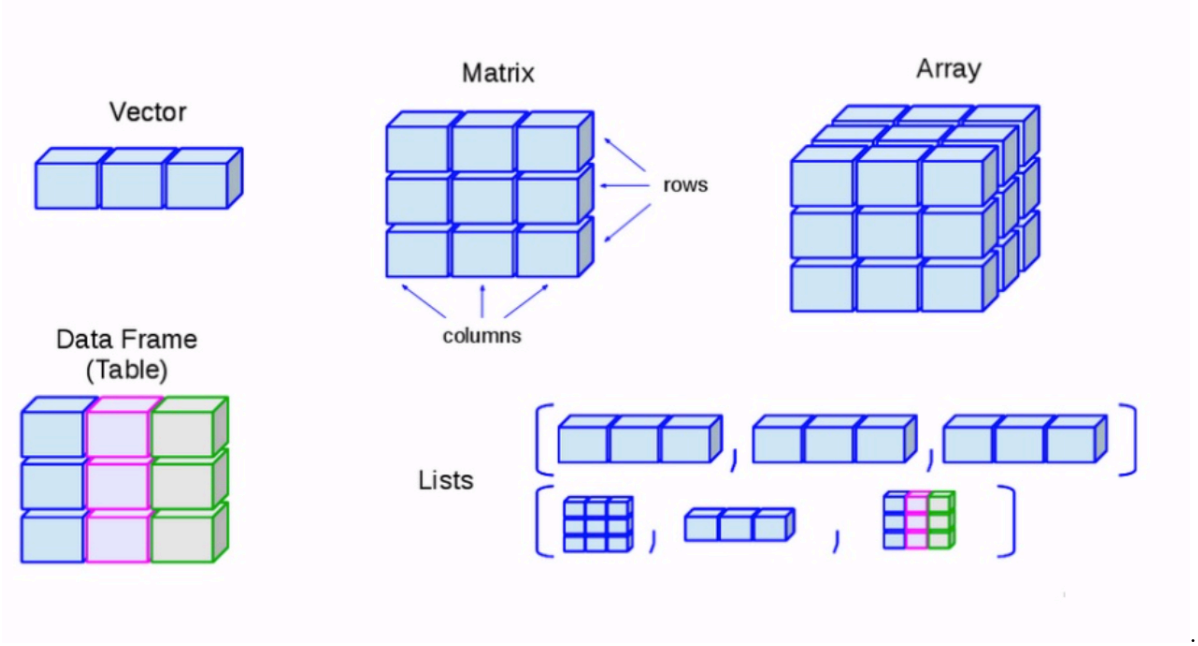
- We can convert `a` to a numeric value. To convert from character to numeric, we use `as.numeric()`

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

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

5 Vectors

Below are the complete list of objects in R.



We will use vector and matrix most frequently in this course.

5.1 Creating vectors

Creating vectors: c()

- In R, a **vector** is a collection of elements of the same data type, which is often used to store a variable of a dataset. For instance, a vector can store the income of a group of people, the final grades of students, etc.
- Vector can be created using the function `c()` by listing all the values in the parenthesis, separated by comma ',.'
- c() stands for "combine".

```
Income <- c(1, 3, 5, 10)
Income
```

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

- Vectors must contain elements **of the same data type**. If not, it will automatically convert elements into the same type (usually character type).

```
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()`

```
x <- c('R',' is', ' the', ' best', ' language')
length(x)
```

```
[1] 5
```

Creating numeric sequences: seq()

It is also possible to easily create sequences with patterns

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

```
# 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 the step is 1, there's a convenient way using `:`

```
1:5
```

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

Combine multiple vectors into one: c()

- Sometimes, we may want to combine multiple vectors into one. For instance, we may have collected income data from two different sources, and we want to combine them into one vector.
- We can use `c()` to combine different vectors; this is very commonly used to concatenate vectors.

```
Income1 <- 1:3
Income2 <- c(10, 15)
```

```
c(Income1,Income2)
```

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

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

5.2 Indexing and subsetting

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

```
# create a vector of income for 4 lecturers at UCL

income <- c(5000, 5500, 6000, 9000)
```

- Extract a single element: use the index of the element

```
# what is the income of the 3rd lecturer?
income[3]
```

```
[1] 6000
```

- Extract multiple elements: use a vector of indices

```
# what are the incomes of the 1st, 3rd, and 4th lecturers?
income[c(1,3,4)]
```

```
[1] 5000 6000 9000
```

5.3 Element-wise arithmetic operations

R is a **vectorized** language, which broadcasts operations to all elements in a vector. This behavior is also called element-wise operations, or broadcasting.

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

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

```
x + 2
```

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

```
x * 2
```

```
[1]  2  6 16 14
```

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

5.4 Elementwise relational operations

- Besides arithmetic operations, we can also perform relational operations on vectors.

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

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

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

```
incomeUCL <-  c(6000, 4600, 7000, 9100,10000)
incomeImperial <- c(5000, 4500, 6000, 9000,10000)
incomeUCL > incomeImperial
```

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

5.5 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
```

```
2 %in% x
```

```
[1] FALSE
```

5.6 After-class exercise

1. Create a vector of 10 numbers from 1 to 10, and extract the 2nd, 4th, and 6th elements.
2. Create a vector of 5 numbers from 1 to 5, and check if 3 is in the vector.
3. Now the interest rate is 0.1, and you have 1000 pounds in your bank account. Calculate the amount in your bank account after 1 year, 2 years, and 3 years, respectively.

6 Matrices

6.1 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

```
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 column and discard the remaining extra elements once fully filled, with a warning message

```
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
```

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: combine matrices `cbind()` and `rbind()`

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

- `cbind()` does the column binding

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

a
```

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

```
cbind(a, a) # column bind
```

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

- `rbind()` does the row binding

```
rbind(a, a) # row bind
```

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

6.2 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.

```
x <- matrix(1:9, nrow = 3, ncol = 3)
x
```

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

- Extract the element in the 2nd row, 3rd column.
  - use **square bracket** with a coma inside `[ , ]` to indicate subsetting; the argument before coma is the row index, and the argument after the coma is the column index.

        * 2 is specified for row index, so we will extract elements from the first row
        * 3 is specified for column index, so we will extract elements from the the second column
        * Altogether, we extract a single element in row 2, column 3.

```
x[2,3] # the element in the 2nd row, 3rd column
```

```
[1] 8
```

- If we leave blank for a dimension, we extract all elements along that dimension.
    - if we want to take out the entire first row
        * 1 is specified for the row index
        * column index is blank

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

```
[1] 1 4 7
```

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

6.3 Matrices: operations

Apply a math function to a matrix

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

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

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

```
y
```

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

- Functions will be vectorized over all elements in a matrix

```r
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 element-wise addition and element-wise multiplication

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

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

```r
x * x
```

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

- If we want to perform the matrix multiplication as in linear algebra, we need to use **%*%**
  - x and y must have conforming dimensions

```r
x
```

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

```r
y
```

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

```r
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

```r
x
```

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

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

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

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

```r
x
```

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

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

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

7 Data Frames

7.1 Data Frames: creating data.frame

Data Frames: create dataframe using `data.frame()`

- You can think of `data.frame` as a spreadsheet in excel.

```
df <-  data.frame(id = 1:4,
  name = c("David", "Karima", "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 102128.59 TRUE
2  2 Karima  99207.27 TRUE
3  3   Anil  99291.95 TRUE
4  4    Wei  98835.83 TRUE
```

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

7.2 Data Frames: Basics

- Each row stands for an `observation`; each column stands for a `variable`.
- Each variable 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.

7.3 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()`

```
str(df)
```

```
'data.frame':   4 obs. of  4 variables:
 $ id  : int  1 2 3 4
 $ name: chr  "David" "Karima" "Anil" "Wei"
 $ wage: num  102129 99207 99292 98836
 $ male: logi  TRUE TRUE TRUE TRUE
```

- Get the variables names of the data frame

```
names(df)
```

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

8 Other data structures (Optional)

8.1 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
```

8.2 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)
```

8.3 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
```

9 Programming Basics: Flow Control

9.1 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:

```
a <- 15

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

larger_than_10
```

```
[1] TRUE
```

Example 2:

```
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"
```

9.2 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.

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

9.3 Nested loops

We can also nest loops into other loops.

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

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

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

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

```
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
```

9.4 User-Defined 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:

```
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:

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

magic(1,3)
```

```
[1] 4
```

9.5 A comprehensive example

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

```
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