ISOM 3390: Business Programming in R

Instructor: Bingjie Qian (PhD candidate)

Summer 2020, HKUST

Topic 2: Variables and Data Structures

2.1 Variables

The most basic and crucial element of R would be a **variable**, which can be assigned a number, a string, a vector, a matix, a data frame, and others.

Variables can be thought of as containers which refer to any type of objects.

We can use `<-` or `=` to assign values to variables.

```
variable_1 = 12
my.variable <- "ISOM 3390"
```

Naming Conventions of variables:

- Variable names consist of letters, digits, period (.) and underscore (_)
- Cannot start with digit or underscore
- Do not use other special characters or space

2.2 Basic Data Types

Some Basic Data Types

- **Logical**: a binary value (TRUE/FALSE, or T/F).

```
lg_1 = TRUE  # equivalent to lg_1 = T
class(lg_1)

## [1] "logical"
```

- **Numeric**: a numeric value. Two most common numeric types are Integer and Double.
  - **Integer**: a whole number (positive, negative or zero).
  - **Double**: a double precision floating point number.
  - In R, numeric values are stored as Double by default. To create an integer, you must specify explicity by placing an `L` directly after the number.
  - A Double type can store the special values **`Inf`**, **`-Inf`**, and **`NaN`**, which represent "positive infinity", "negative infinity", and "not a number".

```
num_1 = 7        # Double, numeric values are stored as Double by default
class(num_1)

## [1] "numeric"

typeof(num_1)

## [1] "double"

num_2  = 7L      # Integer
class(num_2)

## [1] "integer"

typeof(num_2)

## [1] "integer"

1/0

## [1] Inf

-1/0

## [1] -Inf

0/0

## [1] NaN
```

- **Character**: a sequence of characters, called "String" in other programming languages.

```
char_1 = "Hello, world"
class(char_1)

## [1] "character"
```

R represents *"Not Applicable"* or *"Missing"* values with **`NA`**.

```
ms = NA
ms

## [1] NA
```

Some Basic Built-In Operations

R has many built-in operators that can be applied to variables.

- **Arithmetic operators**

```
# use R as a calculator
5 + 2      # addition

## [1] 7

5 - 2      # subtraction
```

```
## [1] 3

5 * 2       # multiplication

## [1] 10

5 / 2       # division

## [1] 2.5

5 %/% 2     # integer division

## [1] 2

5 %% 2      # modulus, the remainder of 5 divided by 2

## [1] 1

5 ^ 2       # exponentiation

## [1] 25
```

- **Comparison operators**

```
5 > 2       # return a logical value (TRUE or FALSE)

## [1] TRUE

5 < 2

## [1] FALSE

5 >= 2

## [1] TRUE

5 <= 2

## [1] FALSE

5 == 2

## [1] FALSE

5 != 2

## [1] TRUE
```

- **Logical operators**

!: negation
&: and
|: or

```
! 5 < 2
```

```
## [1] TRUE

5 < 2 & 3 * 4 == 12

## [1] FALSE

5 < 2 | 3 * 4 == 12

## [1] TRUE
```

2.3 Data Structures

The data we deal with in practice is often related in some way. We can organize the data in some structure to mirror its semantics:

- Items in a shopping cart
- Districts of Hong Kong
- Users of an online community
- ...

In R, we use **data structures** to organize related data.

2.3.1 Vectors

A vector is an ordered collection of values, all of the same type. It is the most basic data structure in R.

Creating Vectors

We can create a vector with the function `c()` (short for combine):

```
v_char = c("hello", "world")   # create a vector of characters
v_lg = c(TRUE, T, F)           # create a vector of logical values
v1 = c(1, -2, 5.5)             # create a vector of numeric values
v2 = c(2, 4, 2e3)
v3 = c(v1, v2)                 # create a new vector by combining 2 vectors
```

Use `:` to create a vector with every integer from start to end (a sequence):

```
2:5

## [1] 2 3 4 5

-2:5

## [1] -2 -1  0  1  2  3  4  5

5:-2

## [1]  5  4  3  2  1  0 -1 -2
```

Unlike other programming languages, R has no scalar data types. Individual numbers or strings are vectors of length 1.

```
12        # a vector of length 1

## [1] 12

"hello"

## [1] "hello"
```

Check the type and length of a vector (which depends the type of its elements):

```
class(v_char)

## [1] "character"

typeof(v_char)

## [1] "character"

length(v_char)

## [1] 2
```

Test if a vector is a given type with `is.*()` function.

```
is.integer(v_char)

## [1] FALSE

is.character(v_char)

## [1] TRUE

is.logical(v_lg)

## [1] TRUE

is.numeric(v3)

## [1] TRUE
```

Get a brief description of the structure of an object with `str()` (short for structure):

```
str(v_char)

##  chr [1:2] "hello" "world"

str(v3)

##  num [1:6] 1 -2 5.5 2 4 2000
```

Operations on Vectors

Many operations in R are *vectorized*.

When we apply an operation to a vector, the operation is applied to each element in the vector. In other words, it works *element-wise*.

```
v1 = c(1, 3, 5, 7)
v1 + 2                # the operation (plus 2) is applied to all elements in
v1

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

When we apply an operation to two vectors, R will match the elements of the two vectors *pairwise*, and the operation is applied each pair of elements.

```
v2 = c(2, 4, 6, 8)
v1 * v2   # the operation (multiplication) is applied to corresponding
elements in v1 and v2

## [1]  2 12 30 56

v1 < v2

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

When the vectors do not have the same length, a *recycling rule* that repeats the shorter vector till it fills in the size of the longer one will apply.

```
v3 = c(0, 1)
v1 + v3

## [1] 1 4 5 8
```

[Task 1: Creating sequences]

(a) Create a sequence of numbers from 3 to 12 in descending order, save it as a variable `x`.

(b) Create a sequence of numbers from 3 to 30 in increments of 3, save it as a variable `y`.

Tips: Sequences created with the `:` operator are always in increments or decrements of 1. In addition to this, R provides a sequence function, `seq()` that allows us to generate more flexible sequences.

To learn the use of the `seq()` function, type `?seq` into the console in RStudio to pull up a Help file.

(c) Create a sequence of all even nunbers from 4 to 42 in descending order, save it as a variable `z`, then determine the type and length of `z`.

Tips: use `seq()`, `typeof()` and `length()`.

(d) Output the results of `x + z` and `y * z` and determine the type(s) of the product and the sum.

[End of Task 1]

Coercion rules

- **Implicit coercion:**

When elements of different types are combined into a single vector, they will be coerced to the most flexible type.

Types from the least to the most flexible are: logical, integer, double, and character.

```
str(c(12, TRUE))

##  num [1:2] 12 1

str(c("hello", 12, TRUE))

##  chr [1:3] "hello" "12" "TRUE"
```

- **Explicit coercion:**

Objects can be explicitly coerced from one type to another using `as.*()` function.

Nonsensical coercion results in `NA`.

```
x = 0:6
typeof(x)

## [1] "integer"

as.logical(x)

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

typeof(as.double(x))

## [1] "double"

as.character(x)

## [1] "0" "1" "2" "3" "4" "5" "6"

as.numeric(c("a", "b", "c"))      # nonsensical coercion

## Warning: NAs introduced by coercion

## [1] NA NA NA
```

[Task 2: Coercion rules]

Test your knowledge of the vector coercion rules and determine the type of the output of the following uses of `c()`:

`c(1, FALSE)` `c("R programming", 3390)` `c(TRUE, 1e3L)`

[End of Task 2]

Subsetting vectors:

Subsetting or indexing allows us to pull out the element(s) that we're interested in from a vector.

There are 3 ways to index a vector:

- Integer indices;
- Logical vectors;
- Names (later; we have to set the element names first).

```
x = c(2, 4, 3, 5, 0)
x[3]            # A positive integer retrieves the element at the specified
position

## [1] 3

x[-3]           # A negative integer omits the element at the specified
position

## [1] 2 4 5 0

x[c(3, 1, 3)]    # Retrieve multiple elements

## [1] 3 2 3

x[c(-3, -1)]     # Omit multiple elements

## [1] 4 5 0

x[c(T, T, F, F, T)]   # A logical vector selects elements where the
corresponding logical value is TRUE

## [1] 2 4 0
```

Set the element names of a vector.

```
v_char2 = c("R Programming", "Machine Learning")
v_char2

## [1] "R Programming"    "Machine Learning"

names(v_char2) = c("Course 1", "Course 2")   # this is to add a "names"
attribute to the vector
v_char2

##           Course 1           Course 2
##    "R Programming" "Machine Learning"

v_char2["Course 1"]             # retrieve element(s) with names
```

```
##        Course 1
## "R Programming"
```

```
v_char2[c("Course 2", "Course 1", "Course 3")]
```

```
##           Course 2           Course 1               <NA>
## "Machine Learning"    "R Programming"                 NA
```

The function `attributes()` gives all additional properties (attributes) for more complex R objects, e.g., named vectors, factors, matrices, etc.

```
attributes(v_char2)
```

```
## $names
## [1] "Course 1" "Course 2"
```

```
names(v_char2)
```

```
## [1] "Course 1" "Course 2"
```

Factors:

An important type of augmented vectors is **factors**, which are designed to represent **categorical** data and can be created with the `factor()` function.

```
gender_1 <- c("Male", "Female", "Male", "Female", "Female", "Female")
gender_1
```

```
## [1] "Male"   "Female" "Male"   "Female" "Female" "Female"
```

```
typeof(gender_1)
```

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

```
str(gender_1)              # a vector of characters
```

```
##  chr [1:6] "Male" "Female" "Male" "Female" "Female" "Female"
```

```
gender_2 <- factor(gender_1)
gender_2
```

```
## [1] Male   Female Male   Female Female Female
## Levels: Female Male
```

```
typeof(gender_2)
```

```
## [1] "integer"
```

```
str(gender_2)              # a vector of categorical data
```

```
##  Factor w/ 2 levels "Female","Male": 2 1 2 1 1 1
```

Factors can only contain pre-defined values. We can't use values that are not in levels.

```
gender_2[3] <- "Yes"
```

```
## Warning in `[<-.factor`(`*tmp*`, 3, value = "Yes"): invalid factor level,
NA
## generated

gender_2         # NA produced due to nonsensical operation

## [1] Male   Female <NA>   Female Female Female
## Levels: Female Male
```

Factors have two attributes, **levels** and **class**.

```
attributes(gender_2)

## $levels
## [1] "Female" "Male"
##
## $class
## [1] "factor"

levels(gender_2)    # levels defines the set of allowed values

## [1] "Female" "Male"

class(gender_2)

## [1] "factor"

class(gender_1)

## [1] "character"
```

The order of the levels can be set using the **levels** and **ordered** arguments of the `factor()` function. This can be important in modeling and visualization where the 1st level is treated as the baseline level.

```
gender_3 <- factor(gender_1, levels = c("Male", "Female"), ordered = TRUE)
gender_3

## [1] Male   Female Male   Female Female Female
## Levels: Male < Female
```

For factors, we can count the occurrences of possible values using `table()` function.

```
table(gender_3)

## gender_3
##   Male Female
##      2      4
```

The `table()` function can also be used to obtain cross-tabulation of several factors.

```
age <- factor(c("adult", "adult", "juvenile", "juvenile", "adult", "adult"))
table(gender_3, age)
```

```
##          age
## gender_3 adult juvenile
##   Male       1        1
##   Female     3        1
```

[Task 3: Factors]

(a) Create an ordered factor that will be printed by R as follows:

```
[1] b b a c b a c a
Levels: c < b < a
```

(b) Use the `table()` function to count the occurrences of `a`, `b`, and `c` in the factor created in (a).

[End of Task 3]

2.3.2 Matrices

Matrices are like **two-dimensional** vectors, organizing values into rows and columns.

Creating matrices

We can create a matrix with the `matrix()` function.

```
m1 <- matrix(1:6, nrow = 2, ncol = 3)   # by default, the matrix is filled by
columns
m1

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

m2 <- matrix(1:6, nrow = 2, ncol = 3, byrow = TRUE)  # set the argument
"byrow" to TRUE
m2

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

class(m2)

## [1] "matrix" "array"

typeof(m2)

## [1] "integer"

attributes(m2)
```

```
## $dim
## [1] 2 3

dim(m2)

## [1] 2 3
```

We can also create a matrix from a vector by using the assignment form of `dim()`, which adds a dim attribute.

```
m3 <- 1:6             # first create a vector
dim(m3) <- c(3, 2)    # then add the dim attribute
m3

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

Modify a matrix in place by setting `dim()`.

```
dim(m3) <- c(1, 6)
m3

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

Many of the functions for vectors have generalizations for matrices.

- `length()` for vectors generalizes to `nrow()` and `ncol()` for matrices.

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

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

length(m)

## [1] 6

nrow(m)

## [1] 3

ncol(m)

## [1] 2
```

- `names()` for vectors generalizes to `rownames()` and `colnames()` for matrices.

```
rownames(m) <- c("a", "b", "c")
colnames(m) <- c("A", "B")
m

##   A B
## a 1 4
## b 2 5
## c 3 6
```

- `c()` for vectors generalizes to `cbind()` and `rbind()` for matrices. `cbind()` and `rbind()` are concatenation functions that respect `dim` attributes.

```
x<-1:4
y<-11:14
z<-91:94
cbind(x, y, z)

##      x  y  z
## [1,] 1 11 91
## [2,] 2 12 92
## [3,] 3 13 93
## [4,] 4 14 94

rbind(x, y, z)

##   [,1] [,2] [,3] [,4]
## x    1    2    3    4
## y   11   12   13   14
## z   91   92   93   94
```

Subsetting matrices

The elements of a matrix can be selected through a similar indexing scheme as in vectors, but this time with both row and column indices.

```
m

##   A B
## a 1 4
## b 2 5
## c 3 6

m[1, 2]                    # return a vector

## [1] 4

m["b", 2]

## [1] 5

m["c", c(2, 1)]            # return a (named) vector

## B A
## 6 3
```

```
m[c(1, 3), c("B", "A")]    # return a (named) matrix

##   B A
## a 4 1
## c 6 3
```

By default, when a single element or a single column/row of a matrix is retrieved, it is returned as a vector with the column/row names removed. This behavior can be turned off by setting `drop = FALSE`.

```
r1 = m[c(1, 3), 2]                  # return a vector
r2 = m[c(1, 3), 2, drop = FALSE]    # return a matrix
class(r1)

## [1] "integer"

class(r2)

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

Omitting any dimension returns full columns or rows.

```
m[1, ]       # equivalent to m[1, c(1, 2)]

## A B
## 1 4

m[, 2]       # equivalent to m[c(1, 2, 3), 2]

## a b c
## 4 5 6
```

[Task 4: Matrices]

(a) Define a variable `x.vec` to contain the integers 1 through 100. Add up the numbers in `x.vec`, by calling a built-in R function `sum()` (`?sum`)

(b) Convert `x.vec` into a matrix with 20 rows and 5 columns, and store this as `x.mat`. Here `x.mat` should be filled out in the default order (columnwise).

Check the dimensions of `x.mat`, and the data type as well.

Compute the sums of each of the 5 columns of `x.mat`, by calling a built-in R function `colSums()` (`?colSums`).

(c) Extract and display rows 1, 5, and 17 of `x.mat`, with a single line of code. Answer the following questions, each with a single line of code:

1. How many elements in row 2 of `x.mat` are larger than 40?
2. How many elements in column 3 are in between 45 and 50?
3. How many elements in column 5 are odd?

Tips: take advantage of the `sum()` function applied to Boolean vectors.

[End of Task 4]

2.3.3 Arrays

Arrays are extensions of matrices to more than 2 dimensions; used to maintain **multi-dimensional** data.

Creating arrays

We can create an array with the `array()` function.

```
a1 <- array(1:24, dim = c(3, 4, 2))
a1
```

```
## , , 1
## 
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
## 
## , , 2
## 
##      [,1] [,2] [,3] [,4]
## [1,]   13   16   19   22
## [2,]   14   17   20   23
## [3,]   15   18   21   24
```

We can also create an array from a vector by using the assignment form of `dim()`.

```
a2 <- 1:12
dim(a2) <- c(3, 2, 2)
a2
```

```
## , , 1
## 
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
## 
## , , 2
## 
##      [,1] [,2]
## [1,]    7   10
## [2,]    8   11
## [3,]    9   12
```

Name the different dimensions of a matrix with `dimnames()` by specifing a list of character vectors.

```
dimnames(a2) <- list(c("a", "b", "c"), c("one", "two"), c("A", "B"))
a2

## , , A
##
##   one two
## a   1   4
## b   2   5
## c   3   6
##
## , , B
##
##   one two
## a   7  10
## b   8  11
## c   9  12
```

The `c()` function clears `dim` and `dimnames` attributes and **flattens** multidimensional arrays (and matrices as well), returning vectors.

```
c(a2)    # equivalent to as.vector(a2)

##  [1]  1  2  3  4  5  6  7  8  9 10 11 12

c(m)

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

2.3.4 Lists

While matrices are extremely useful for processing and storing large datasets, they have several limitations. For example, only one data type is allowed in a matrix. If we would like to put together columns of numeric values and of characters in the same matrix, numeric values will be forced into characters.

Both **lists** and **data frames** are more flexible data structures which allow different data types to be stored in a single data structure.

Creating Lists

We can create a list with the function `list()`.

The elements of a list are **ordered** and may also have a name attached to them.

```
int <- 12345L                    # integer
char <- "Charles"                # character
vec <- c(90, 80, 85)             # vector of numeric values
student1 <- list(int, char, marks = vec)    # a list of 3 elements, the last
element is named. Alternatively, student1 <- list(12345L, "Charles", marks =
c(90, 80, 85))

student1

## [[1]]
## [1] 12345
##
## [[2]]
## [1] "Charles"
##
## $marks
## [1] 90 80 85

class(student1)

## [1] "list"

typeof(student1)

## [1] "list"

str(student1)

## List of 3
##  $      : int 12345
##  $      : chr "Charles"
##  $ marks: num [1:3] 90 80 85

names(student1)

## [1] ""      ""      "marks"

names(student1)[1:2] <- c("id", "name")    # set the names of the first two
elements
str(student1)

## List of 3
##  $ id   : int 12345
##  $ name : chr "Charles"
##  $ marks: num [1:3] 90 80 85
```

Lists can be concatenated using the `c()` function.

```
student1 <- c(student1, list(gender = "male", age = 19))
str(student1)

## List of 5
##  $ id    : int 12345
##  $ name  : chr "Charles"
##  $ marks : num [1:3] 90 80 85
##  $ gender: chr "male"
##  $ age   : num 19
```

When combining a vector and a list, `c()` will coerce the vector to a list before combining them.

```
student1 <- c(student1, c(parent1 = "Ana", parent2 = "Mike"))
str(student1)
```

```
## List of 7
##  $ id     : int 12345
##  $ name   : chr "Charles"
##  $ marks  : num [1:3] 90 80 85
##  $ gender : chr "male"
##  $ age    : num 19
##  $ parent1: chr "Ana"
##  $ parent2: chr "Mike"
```

Lists are *recursive*: a list can contain other lists.

```
students <- list(student1, list(id = 12344, name = "Karen"))  # the list
contains two lists
students
```

```
## [[1]]
## [[1]]$id
## [1] 12345
##
## [[1]]$name
## [1] "Charles"
##
## [[1]]$marks
## [1] 90 80 85
##
## [[1]]$gender
## [1] "male"
##
## [[1]]$age
## [1] 19
##
## [[1]]$parent1
## [1] "Ana"
##
## [[1]]$parent2
## [1] "Mike"
##
##
## [[2]]
## [[2]]$id
## [1] 12344
##
## [[2]]$name
## [1] "Karen"
```

```
str(students)

## List of 2
##  $ :List of 7
##   ..$ id     : int 12345
##   ..$ name   : chr "Charles"
##   ..$ marks  : num [1:3] 90 80 85
##   ..$ gender : chr "male"
##   ..$ age    : num 19
##   ..$ parent1: chr "Ana"
##   ..$ parent2: chr "Mike"
##  $ :List of 2
##   ..$ id  : num 12344
##   ..$ name: chr "Karen"
```

Subsetting lists

We can extract elements from a list with `[ ]`, which returns a list.

```
student1[1]              # return a list containing the 1st element

## $id
## [1] 12345

student1[c(1, 3)]        # return a list containing the 1st and 3rd elements

## $id
## [1] 12345
##
## $marks
## [1] 90 80 85

str(student1[c(1, 3)])

## List of 2
##  $ id   : int 12345
##  $ marks: num [1:3] 90 80 85
```

The `[[]]` and `$` (by name only) operators allow us to extract a list element. This type of indexing can only access one element at a time.

```
student1[[1]]          # return the 1st element, which is an integer

## [1] 12345

# the following three are equivalent
student1[[3]]          # return the 3rd element, which is a vector

## [1] 90 80 85

student1[["marks"]]

## [1] 90 80 85
```

```
student1$marks

## [1] 90 80 85
```

List subsetting can be *recursive*: subset the subset

```
# the following three are equivalent
students[[1]][[3]]

## [1] 90 80 85

students[[1]]$marks

## [1] 90 80 85

students[[c(1, 3)]]

## [1] 90 80 85
```

We can add new elements to a list using the `[[]]` or `$` notation.

```
student1$award <- TRUE   # alternatively, student1[["award"]] <- TRUE
str(student1)

## List of 8
##  $ id     : int 12345
##  $ name   : chr "Charles"
##  $ marks  : num [1:3] 90 80 85
##  $ gender : chr "male"
##  $ age    : num 19
##  $ parent1: chr "Ana"
##  $ parent2: chr "Mike"
##  $ award  : logi TRUE
```

A list can be flattened into a vector using the `unlist()` function.

- By default this will coerce different types to a common type.
- Each element of this vector will have a name generated from the name of the list element.

```
unlist(student1)    # return a (named) vector, all elements are coerced into
characters

##        id      name    marks1    marks2    marks3    gender       age
parent1
##   "12345" "Charles"      "90"      "80"      "85"    "male"      "19"
"Ana"
##   parent2     award
##    "Mike"    "TRUE"
```

[Task 5: Lists]

(a) Construct a nested list that will be printed as follows by `str()`. Name the resulting list `x.list`.

```
List of 3
 $ :List of 3
  ..$ : num [1:2] 2.3 5.9
  ..$ : Factor w/ 2 levels "a","b": 1 2 2 1 1
  ..$ : logi [1:3] TRUE FALSE TRUE
 $ :List of 1
  ..$ :List of 1
  .. ..$ : int [1:3] 1 2 3
 $ : int [1:2, 1:3] 20 17 19 16 18 15
```

(b) Complete the following tasks, each with a single line of code (Note: pay close attention to what is asked and use either single brackets `[ ]` or double brackets `[[ ]]` as appropriate):

1. Extract all but the 2nd first-level list element of `x.list` - seeking here a list as the final answer.

2. Use a **single pair** of double brackets `[[ ]]` to extract the only factor and output the following

```
[1] a b b a a
Levels: a b
```

3. Extract the last 2 columns of the matrix and output the following:

```
     [,1] [,2]
[1,]   19   18
[2,]   16   15
```

[End of Task 5]

2.3.5 Data frames

A data frame is like a spreadsheet or a database table, and is the most common way of storing data in R.

Usually, each row is a record/observation/instance, and each column is a field/variable/attibute.

Creating data frames

We can create a data frame with `data.frame()`, which takes *named vectors* as input.

```
default.data <- data.frame(income = c(1.5, 2.2, 2, 3), gender = c("M", "F",
"M", "F"), default = c(T, T, F, F))
default.data
```

```
##   income gender default
## 1    1.5      M    TRUE
## 2    2.2      F    TRUE
## 3    2.0      M   FALSE
## 4    3.0      F   FALSE
```

```
class(default.data)
```

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

```
typeof(default.data)
```

```
## [1] "list"
```

```
str(default.data)
```

```
## 'data.frame':    4 obs. of  3 variables:
##  $ income : num  1.5 2.2 2 3
##  $ gender : chr  "M" "F" "M" "F"
##  $ default: logi  TRUE TRUE FALSE FALSE
```

```
attributes(default.data)
```

```
## $names
## [1] "income"  "gender"  "default"
##
## $class
## [1] "data.frame"
##
## $row.names
## [1] 1 2 3 4
```

A data frame is *a list of equal-length vectors* and has a 2-dimensional structure. So it shares properties of both the *list* and the *matrix*.

```
nrow(default.data)
```

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

```
ncol(default.data)
```

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

```
rownames(default.data)
```

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

```
colnames(default.data)
```

```
## [1] "income"  "gender"  "default"
```

```
names(default.data)       # names() is equivalent to colnames()
```

```
## [1] "income"  "gender"  "default"
```

We can combine data frames using cbind() and rbind().

- When data frames are combined *column-wise*, the number of rows must match.
- When data frames are combined *row-wise*, both the number and names of columns must match.

```
cbind(default.data, data.frame(balance = c(200, 150, 94, 300), student = c(T, F, T, T)))
```

```
##   income gender default balance student
## 1    1.5      M    TRUE     200    TRUE
## 2    2.2      F    TRUE     150   FALSE
## 3    2.0      M   FALSE      94    TRUE
## 4    3.0      F   FALSE     300    TRUE
```

```
rbind(default.data, data.frame(income = c(1.3, 2.4), gender = c("M", "F"), default = c(F, T)))
```

```
##   income gender default
## 1    1.5      M    TRUE
## 2    2.2      F    TRUE
## 3    2.0      M   FALSE
## 4    3.0      F   FALSE
## 5    1.3      M   FALSE
## 6    2.4      F    TRUE
```

We can add new columns to a data frame. The new columns must have the same number of rows as the existing data frame.

```
default.data$education <- c("h", "l", "m", "h")
default.data
```

```
##   income gender default education
## 1    1.5      M    TRUE         h
## 2    2.2      F    TRUE         l
## 3    2.0      M   FALSE         m
## 4    3.0      F   FALSE         h
```

By default, `data.frame()` turns *strings* into *factors*. We can set `stringsAsFactors = FALSE` to suppress this behavior.

```
default.data <- data.frame(income = c(1.5, 2.2, 2, 3), gender = c("M", "F", "M", "F"), default = c(T, T, F, F), stringsAsFactors = FALSE)
str(default.data)
```

```
## 'data.frame':    4 obs. of  3 variables:
##  $ income : num  1.5 2.2 2 3
##  $ gender : chr  "M" "F" "M" "F"
##  $ default: logi  TRUE TRUE FALSE FALSE
```

Subsetting data frames

We can extract a column from a data frame, which returns a *vector*.

```
# the following expressions are equivalent
default.data$income

## [1] 1.5 2.2 2.0 3.0

default.data[["income"]]

## [1] 1.5 2.2 2.0 3.0

default.data[[1]]

## [1] 1.5 2.2 2.0 3.0

default.data[ , "income"]     # similar to subsetting matrices, omitting the
row dimension returns full rows.

## [1] 1.5 2.2 2.0 3.0
```

We can also extract multiple columns, which returns a *data frame*.

```
default.data[ , c("income", "default")]

##   income default
## 1    1.5    TRUE
## 2    2.2    TRUE
## 3    2.0   FALSE
## 4    3.0   FALSE

default.data[ , c(1, 3)]

##   income default
## 1    1.5    TRUE
## 2    2.2    TRUE
## 3    2.0   FALSE
## 4    3.0   FALSE
```

We can extract particular rows from the data frame by setting certain criteria.

```
default.data[default.data$income > 2, ]

##   income gender default
## 2    2.2      F    TRUE
## 4    3.0      F   FALSE
```

In the above example, the criteria `default.data$income > 2` actually returns a logical vector. Remember we can use logical vectors for indexing (refer to Chapter 2.3.1).

```
default.data$income > 2     # return a logical vector

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

Set more criteria in selecting rows.

```
default.data[default.data$default == T & default.data$income >= 2, ]

##   income gender default
## 2    2.2      F    TRUE
```

Combine the selection of rows and columns.

```
default.data[default.data$default == T & default.data$income >= 2, c(1, 3)]

##   income default
## 2    2.2    TRUE
```

We can simplify the typing of these criteria by using the function `attach()`; disable this facility using `detach()` once not needed.

```
attach(default.data)
default.data[default == T & income >= 2, c(1, 3)]

##   income default
## 2    2.2    TRUE
```

****Sorting data frames***

Data is sometimes easier to read when sorted. We can use the `order()` function to sort a data frame.

```
default.data

##   income gender default
## 1    1.5      M    TRUE
## 2    2.2      F    TRUE
## 3    2.0      M   FALSE
## 4    3.0      F   FALSE

default.data[order(default.data$income), ]      # sort in Increasing order by
default

##   income gender default
## 1    1.5      M    TRUE
## 3    2.0      M   FALSE
## 2    2.2      F    TRUE
## 4    3.0      F   FALSE

default.data[order(default.data$income, decreasing = T), ]    # set to
Decreasing order

##   income gender default
## 4    3.0      F   FALSE
## 2    2.2      F    TRUE
## 3    2.0      M   FALSE
## 1    1.5      M    TRUE
```

[Task 6: Data frames - cars data]

We'll use a very simple dataset to see how we can perform simple data processing and visualization. We're going to use the `cars` data that comes with the default installation of R and is maintained as a data frame object.

Let's first obtain an intuitive idea of what the data looks like by showing the first few records. Sometimes the dataset may be too large to show all the records in the console. The `head()` function returns the first `n` records, and the `tail()` function returns the last `n` records. `?head` if you want to know more.

```
head(cars)          # by default, subset the first 6 rows

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

head(cars, n=10)

##    speed dist
## 1      4    2
## 2      4   10
## 3      7    4
## 4      7   22
## 5      8   16
## 6      9   10
## 7     10   18
## 8     10   26
## 9     10   34
## 10    11   17
```

As you can see, this data frame contains 2 columns, containing values for two car features `speed` and `dist` (abbreviated for stopping distance), respectively.

We can access the values of each column in the data frame using a `$` notation:

```
cars$speed    # return a vector that contains the values in the speed column

##  [1]  4  4  7  7  8  9 10 10 10 11 11 12 12 12 12 13 13 13 13 14 14 14 14
15 15
## [26] 15 16 16 17 17 17 18 18 18 18 19 19 19 20 20 20 20 20 22 23 24 24 24
24 25

cars$dist     # return a vector that contains the values in the dist column

##  [1]   2  10   4  22  16  10  18  26  34  17  28  14  20  24  28  26  34
34  46
## [20]  26  36  60  80  20  26  54  32  40  32  40  50  42  56  76  84  36
```

```
46  68
## [39]  32  48  52  56  64  66  54  70  92  93 120  85
```

You can save these two vectors as two variables before proceeding to answering the subsequent questions.

(a) Calculate the average (`mean()`) and standard deviation (`sd()`) of speed and distance.

(b) Normalize the `dist` data by subtracing individual values by mean and dividing the results by standard deviation.

(c) Select all even values greater than 50 in the `dist` column using Boolean indexing.

The expected output is

```
[1]  60  80  54  56  76  84  68  52  56  64  66  54  70  92 120
```

Tips:

- Recall the modulo operator we saw in class: `%%`. `x %% y` gives the remainder of `x` divided by `y`.
- Use comparion and logical operations to create a Boolean vector used for indexing.

[End of Task 6]

[Task 7: Data frames - survey data]

In this task, we're going to import an external dataset and explore it. The dataset `survey.csv` (csv is short for comma-separated values) can be downloaded from Canvas.

Some background knowledge:

Working directory

- Working directory is the default location on your computer that R is pointing at.
- If you want to read or save a file, you need to know where the current working directory is.
- Use `setwd()` to set up working directory. Use `getwd()` to check your current working directory.

Absolute path

- An absolute path is a path that describes the location of a file or folder regardless of the current working directory; in fact, it is relative to the root directory.

Relative path

- A relative path is a path that describes the location of a file or folder in relative to the current working directory.

In this task, I put both the R script (`2_Exercise.R`) and the dataset (`survey.csv`) in the folder "/Users/qbj/Desktop/Summer_2020/2_Data Structure" on my computer. You should replace it with your own folder path in the following codes.

Now, read the data from the file `survey.csv` with the function `read.table()`, and save it as a variable.

```
setwd("/Users/qbj/Desktop/Summer_2020/02_Data Structure")
survey = read.table("survey.csv", header = TRUE, sep = ",")   # return a data frame
```

The first argument of `read.table()` is the pathname to the `survey.csv` file. Here we use a relative path. You can replace it with the whole pathname (the absolute path: /Users/qbj/Desktop/Summer_2020/2_Data Structure/survey.csv).

The `header` argument is used to specify whether the file contains the names of the variables as its first line. If the data has names for each of the columns in the first row, set `header = TRUE`.

The `sep` argument is used to specify the field separator character. The default is white space. The funtion will automatically convert columns to logical, integer, numeric, complex vectors or factors as appropriate.

`?read.table` if you want to know more about the function.

(a) Each observation in the data frame corresponds to a survey respondent. Use `head()` and `tail()` to pick the first and last 5 respondents. Combine them together and display the result.

(b) Apply `typeof()`, `class()`, `str()`, `length()`, and `dim()` to `survey` to learn the structure of the data.

(c) Write code to determine how many survey respondents are from MARK or FINA.

(d) Write code to determine what percentage of survey respondents are from IS.

(e) Use `$` and `[]` notation to pull the `OperatingSystem` column from the survey data.

(f) Change the column name `Program` to `Major`. *Tips*: the assignment form of `names()`

(g) Use `table()` to determine the distributions of levels of `Rexperience` among respondents from different `Major`s.

The expected output is as follows:

```
        Basic competence Experienced Installed on machine Never used
   FINA                6           1                    6          4
```

```
  IS                    17          1                    12          12
  MARK                   4          0                     0           4
```

The `table()` function can take a data frame directly and builds contingency tables (i.e., count tables) showing counts at each combination of factor levels

Try the following code in console by yourself to see the output.

```
table(survey)
```

(h) Write code to get the `Major` and `Rexperience` of every student who watched at least 15 hours of TV (`TVhours`).

(i) Write code to sort survey respondents by `TVhours` in descending order.

[End of Task 7]

2.3.6 Summary of data structures

The data structures we have introduced in this chapter can be classified by dimensionality (one, two, or more) and whether they allow different data types (homogeneous or heterogeneous).

.	**Homogeneous**	**Heterogenenous**
1d	Vector	List
2d	Matrix	Data frame
nd	Array	