MiCM Workshop Series

R - Beyond the Basics

Efficient Coding

- Yi Lian

- March 7, 2020

Link to workshop material https://github.com/ly129/MiCM2020 (https://github.com/ly129/MiCM2020)

Outline

1. **An overview of efficiency**
    - General rules
    - R-specific rules
    - R object types (if necessary)
    - Record runtime of your code
2. **Efficient coding**
    - Powerful functions in R

    ```
    - aggregate(), by(), apply() family
    - ifelse(), cut() and split()
    ```

    - Write our own functions in R

    ```
    - function()
    ```

    - Examples
        - Categorization, conditional operations, etc..
3. **Exercises**

Important note! There are MANY advanced and powerful packages that do different things. There are too many and they are too diverse to be covered in this workshop.

Here is a list of some awesome packages. https://awesome-r.com/ (https://awesome-r.com/)

1.1 General rules

1.2 R-specific rules

1.3 R data types and structures

1.3.1 R data types

```
- numeric
    - integer
    - double precision (default)
- logical
- character
- factor
- ...
```

In [1]:

```
# double
class(5); is.double(5)
```

'numeric'

TRUE

In [2]:

```
# integer
class(5L); is.double(5L)
```

'integer'

FALSE

In [3]:

```
# How precise is double precision?
options(digits = 22) # show more digits in output
print(1/3)
options(digits = 7) # back to the default
```

```
[1] 0.3333333333333333148296
```

In [4]:

```
object.size(rep(5, 10))
object.size(rep(5L, 10))
```

```
176 bytes
```

```
96 bytes
```

In [5]:

```
# logical
class(TRUE); class(F)
```

'logical'

'logical'

In [6]:

```
# character
class("TRUE")
```

'character'

In [7]:

```
# Not important for this workshop
fac <- as.factor(c(1, 5, 11, 3))
fac
```

1 5 11 3

▶ **Levels**:

In [8]:
```
class(fac)
```

'factor'

In [9]:
```
# R has an algorithm to decide the order of the levels
fac.ch <- as.factor(c("B", "a", "1", "ab", "b", "A"))
fac.ch
```

B a 1 ab b A

▶ **Levels**:

1.3.2 R data structures

```
- Scalar *
- Vector
- Matrix
- Array
- List
- Data frame
- ...
```

In [10]:
```
# Scalar - a vector of length 1
myscalar <- 5
myscalar
```

5

In [11]:
```
class(myscalar)
```

'numeric'

In [12]:
```
# Vector
myvector <- c(1, 1, 2, 3, 5, 8)
myvector
```

1 1 2 3 5 8

In [13]:
```
class(myvector)
```

'numeric'

In [14]:
```
# Matrix - a 2d array
mymatrix <- matrix(c(1, 1, 2, 3, 5, 8), nrow = 2, byrow = FALSE)
mymatrix
```

A matrix:
2 × 3 of
type dbl

1	2	5
1	3	8

In [15]:
```
class(mymatrix)
```

'matrix'

In [16]:
```
str(mymatrix)
```

```
 num [1:2, 1:3] 1 1 2 3 5 8
```

In [17]:
```
# Array - not important for this workshop
myarray <- array(c(1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144), dim = c(2, 2, 3))
print(myarray) # print() is not needed if run in R or Rstudio.
```

```
, , 1

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

, , 2

     [,1] [,2]
[1,]    5   13
[2,]    8   21

, , 3

     [,1] [,2]
[1,]   34   89
[2,]   55  144
```

In [18]:
```
class(myarray)
```

'array'

In [19]:
```
# List - very important for the workshop
mylist <- list(Title = "R Beyond the Basics",
               Duration = c(2, 2),
               sections = as.factor(c(1, 2, 3, 4)),
               Date = as.Date("2020-03-06"),
               Lunch_provided = FALSE,
               Feedbacks = c("Amazing!", "Great workshop!", "Yi is the best!", "Wow!")
)
print(mylist) # No need for print if running in R or Rstudio
```

```
$Title
[1] "R Beyond the Basics"

$Duration
[1] 2 2

$sections
[1] 1 2 3 4
Levels: 1 2 3 4

$Date
[1] "2020-03-06"

$Lunch_provided
[1] FALSE

$Feedbacks
[1] "Amazing!"        "Great workshop!" "Yi is the best!" "Wow!"
```

In [20]:
```
class(mylist)
```

'list'

In [21]:
```
# Access data stored in lists
mylist$Title
```

'R Beyond the Basics'

In [22]:
```
# or
mylist[[6]]
```

'Amazing!' 'Great workshop!' 'Yi is the best!' 'Wow!'

In [23]:
```
# Further
mylist$Duration[1]
mylist[[6]][2]
```

2

'Great workshop!'

In [24]:
```
# Elements in lists can have different data types
lapply(mylist, class) # We will talk about lapply() later
```

$Title
'character'
$Duration
'numeric'
$sections
'factor'
$Date
'Date'
$Lunch_provided
'logical'
$Feedbacks
'character'

In [25]:
```
# Elements in list can have different lengths
lapply(mylist, length)
```

$Title
1
$Duration
2
$sections
4
$Date
1
$Lunch_provided
1
$Feedbacks
4

In [26]:
```
# Data frames - most commonly used for analyses
head(mtcars)
```

A data.frame: 6 × 11

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225	105	2.76	3.460	20.22	1	0	3	1

In [27]:
```
# Access a column (variable) in data frames
mtcars$mpg
```

21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26 30.4 15.8 19.7 15 21.4

1.4 Time your program in R

Illustrations of R rules for efficiency.

```
- proc.time(), system.time()
- microbenchmark
```

1.4.1 Vectorized operation vs. loop

Example Calculate the square root of 1 to 1,000,000 using vectorized operation vs. using a for loop.

In [28]:
```
# Vectorized operation
# system.time(operation)  returns the time needed to run the 'operation'
t <- system.time( x1 <- sqrt(1:1000000) )
head(x1)
```

1 1.4142135623731 1.73205080756888 2 2.23606797749979 2.44948974278318

In [29]:
```
# For loop
x2 <- rep(NA, 1000000)
t0 <- proc.time()
for (i in 1:1000000) {
    x2[i] <- sqrt(i)
}
t1 <- proc.time()

identical(x1, x2) # Check whether results are the same
```

TRUE

In [30]:
```
# As we can see, R is not very fast with loops.
t; t1 - t0
# ?proc.time
```

```
   user  system elapsed 
  0.006   0.004   0.010 

   user  system elapsed 
  0.067   0.002   0.069 
```

Take-home message

- Use vectorized operations rather than loops for speed in R.
- Loops are more intuitive though.
- Balance between
  - speed
  - your need for speed
  - your level of comfortableness with linear algebra
  - your level of laziness
  - your typing speed
  - ...
- Based on what you are doing
  - dealing with big dataset and expensive calculations?
  - running the code only once or potentially many many times?

1.4.2 Use established functions

Example Calculate the square root using sqrt( ) vs. our own implementation.

In [31]:
```
# microbenchmark runs the code multiple times and take a summary
# Use well-developped R function
library(microbenchmark)
result <- microbenchmark(sqrt(500),
                         500^0.5,
                         unit = "ns", times = 1000
                        )
summary(result)
# Result in nanoseconds
```

A data.frame: 2 × 8

expr	min	lq	mean	median	uq	max	neval
<fct>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
sqrt(500)	81	90	97.807	94	101	777	1000
500^0.5	155	165	181.637	170	176	5660	1000

In summary, keep the rules in mind, know what you want to do, test your program, time your program.

2. Efficient coding

R has many powerful and useful functions that we can use to achieve efficient coding and computing.

2.1 Powerful functions in R

Let's play with some data.

In [32]:
```
data <- read.csv("https://raw.githubusercontent.com/ly129/MiCM2020/master/sample.csv", header =
TRUE)
head(data, 8)
```

A data.frame: 8 × 8

X	Sex	Wr.Hnd	NW.Hnd	Pulse	Smoke	Height	Age
<int>	<fct>	<dbl>	<dbl>	<int>	<fct>	<dbl>	<dbl>
1	Male	21.4	21.0	63	Never	180.00	19.000
2	Male	19.5	19.4	79	Never	165.00	18.083
3	Female	16.3	16.2	44	Regul	152.40	23.500
4	Female	15.9	16.5	99	Never	167.64	17.333
5	Male	19.3	19.4	55	Never	180.34	19.833
6	Male	18.5	18.5	48	Never	167.00	22.333
7	Female	17.5	17.0	85	Heavy	163.00	17.667
8	Male	19.8	20.0	NA	Never	180.00	17.417

In [33]:
```
summary(data)
```

```
       X              Sex         Wr.Hnd          NW.Hnd          Pulse       
 Min.   :  1.00   Female:47   Min.   :13.00   Min.   :12.50   Min.   : 40.00  
 1st Qu.: 25.75   Male  :53   1st Qu.:17.50   1st Qu.:17.45   1st Qu.: 50.25  
 Median : 50.50               Median :18.50   Median :18.50   Median : 71.50  
 Mean   : 50.50               Mean   :18.43   Mean   :18.39   Mean   : 69.90  
 3rd Qu.: 75.25               3rd Qu.:19.50   3rd Qu.:19.52   3rd Qu.: 84.75  
 Max.   :100.00               Max.   :23.20   Max.   :23.30   Max.   :104.00  
                                                              NA's   :6       
   Smoke        Height           Age       
 Heavy: 6   Min.   :152.0   Min.   :16.92  
 Never:79   1st Qu.:166.4   1st Qu.:17.58  
 Occas: 5   Median :170.2   Median :18.46  
 Regul:10   Mean   :171.8   Mean   :20.97  
            3rd Qu.:179.1   3rd Qu.:20.21  
            Max.   :200.0   Max.   :73.00  
            NA's   :13                     
```

a1. Calculate the mean writing hand span of all individuals

```
mean(x, trim = 0, na.rm = FALSE, ...)
```

In [ ]:
```
```

a2. Calculate the mean height of all individuals, exclude the missing values

In [ ]:

In [ ]:

a3. Calculate the mean of all continuous variables

```
apply(X, MARGIN, FUN, ...)
```

In [34]:
```
# Choose the continuous variables
```

In [35]:
```
# Calculate the mean
```

b1. Calculate the count/proportion of females and males

```
table(...,
  exclude = if (useNA == "no") c(NA, NaN),
  useNA = c("no", "ifany", "always"),
  dnn = list.names(...), deparse.level = 1)

prop.table()
```

In [ ]:

b2. Calculate the count in each Smoke group

In [ ]:

b3. Calculate the count of males and females in each Smoke group

In [ ]:

In [ ]:

c1. Calculate the standard deviation of writing hand span of females

```
aggregate()
tapply()
by()
```

In [36]:
```
# aggregate() syntax 1
```

In [37]:
```
# aggregate() syntax 2
```

In [38]:
```
# by()
```

In [39]:
```
# tapply()
```

In [40]:
```
# Return a list using tapply()
```

aggregate( ), by( ) and tapply( ) are all connected. They give different types of output.

c2. Calculate the standard deviation of writing hand span of all different Sex-Smoke groups

In [ ]:

In [ ]:

c3. Calculate the standard deviation of writing hand and non-writing hand span of all Sex-Smoke groups

In [ ]:

In [ ]:

Let's try to figure out what aggregate( ) is doing

```
print()
```

In [ ]:

Exercise.

- Repeat b1-b3 using aggregate( )

In [ ]:

- Make histograms of writing hand span for all eight Sex-Smoke groups using aggregate( )

```
hist()
```

In [ ]:

d1. Categorize 'Age' - make a new binary variable 'Adult'

```
ifelse(test, yes, no)
```

In [41]:

```
vec <- 1:5
vec

ifelse(vec>3, yes = "big", no = "small")
```

1 2 3 4 5

'small' 'small' 'small' 'big' 'big'

In [ ]:

R has if (test) {opt1} else {opt2}, what is the advantage of ifelse( )?

In [42]:
```
if (data$Age >= 18) {
    data$Adult2 = "Yes"
} else {
    data$Adult2 = "No"
}
head(data)
```

```
Warning message in if (data$Age >= 18) {:
“the condition has length > 1 and only the first element will be used”
```

A data.frame: 6 × 9

X	Sex	Wr.Hnd	NW.Hnd	Pulse	Smoke	Height	Age	Adult2
<int>	<fct>	<dbl>	<dbl>	<int>	<fct>	<dbl>	<dbl>	<chr>
1	Male	21.4	21.0	63	Never	180.00	19.000	Yes
2	Male	19.5	19.4	79	Never	165.00	18.083	Yes
3	Female	16.3	16.2	44	Regul	152.40	23.500	Yes
4	Female	15.9	16.5	99	Never	167.64	17.333	Yes
5	Male	19.3	19.4	55	Never	180.34	19.833	Yes
6	Male	18.5	18.5	48	Never	167.00	22.333	Yes

In [43]:
```
# Delete Adult2
data <- subset(data, select=-c(Adult2))
```

ifelse( ) is vectorized!!!

d2. Categorize 'Wr.Hnd' into 5 groups - make a new categorical variable with 5 levels

```
1. =< 16: TP/XS
2. 16~18: P/S
3. 18~20: M/M
4. 20~22: G/L
5. >  22: TG/XL
```

Can we still use ifelse( )?

```
cut(x, breaks, labels = NULL, right = TRUE, ...)
```

In [44]:
```
cut.points <- c(0, 16, 18, 20, 22, Inf)

head(data)
# labels as default
```

A data.frame: 6 × 8

X	Sex	Wr.Hnd	NW.Hnd	Pulse	Smoke	Height	Age
<int>	<fct>	<dbl>	<dbl>	<int>	<fct>	<dbl>	<dbl>
1	Male	21.4	21.0	63	Never	180.00	19.000
2	Male	19.5	19.4	79	Never	165.00	18.083
3	Female	16.3	16.2	44	Regul	152.40	23.500
4	Female	15.9	16.5	99	Never	167.64	17.333
5	Male	19.3	19.4	55	Never	180.34	19.833
6	Male	18.5	18.5	48	Never	167.00	22.333

In [45]:
```
# Set labels to false
```

In [46]:
```
# Customized labels
label <- c("TP/XS", "P/S", "M/M", "G/L", "TG/XL")
```

e1. Calculate the mean Wr.Hnd span of each Hnd.group

In [ ]:
```
```

e2. Calcuate the mean Wr.Hnd span of each Hnd.group without using aggregate, by, tapply

```
split(x, f, ...)
lapply(X, FUN, ...)
sapply(X, FUN, ..., simplify = TRUE)
```

In [47]:
```
# cut.points <- c(0, 16, 18, 20, 22, Inf)
```

In [48]:
```
# lapply
```

In [49]:
```
# sapply
```

In [ ]:
```
```

In [50]:
```
# vapply *
# Safer than sapply(), and a little bit faster
# because FUN.VALUE has to be specified that length and type should match

# va <- vapply(Wr.Hnd.Grp, summary, FUN.VALUE = c("Min." = numeric(1),
#                                                  "1st Qu." = numeric(1),
#                                                  "Median" = numeric(1),
#                                                  "Mean" = numeric(1),
#                                                  "3rd Qu." = numeric(1),
#                                                  "Max." = numeric(1)))
# va
```

f. Calculate the 95% sample confidence intervals of Wr.Hnd in each Smoke group.

One variable for lower bound and one variable for upper bound.

$$CI = \bar{x} \pm t_{n-1,0.025} \times \sqrt{\frac{s^2}{n}}$$

where $\bar{x}$ is the sample mean and s^2 is the sample variance.

In [51]:
```
# aggregate(Wr.Hnd~Smoke, data = data, FUN = ...)
# tapply(X = data$Wr.Hnd, INDEX = list(data$Smoke), FUN = ...)
```

Unfortunately, I do not know any function in R that does this calculation.

But we know how to do it step by step.

In [52]:
```
# How many times did we aggregate according to the group? Can on aggregate only once?
```

Or, we can make our own function and integrate it into aggregate( ), by( ), or tapply( ) !!!

2.2 Write our own functions in R

A function takes in some arguments and gives some outputs

Arguments include

```
- inputs
- options
```

In [53]:
```
# The structure
func_name <- function(argument){
    statement
}
```

Example 1. Make a function for $f(x) = 2x$

In [54]:
```
# Build the function
times2 <- function(x) {
    fx = 2 * x
    return(fx)
}
# Use the function
times2(x = 5)
# or
times2(3)
```

10

6

Example 2. Make a function to calculate the integer division of a by b, return the integer part and the modulus.

In [55]:
```
# R has operators that do this
9 %/% 2
9 %% 2
```

4

1

```
floor( ) takes a single numeric argument x and returns a numeric vector containing the largest int
egers not greater than the corresponding elements of x.
```

In [56]:
```
int.div <- function(a, b){
    int <- floor(a/b)
    mod <- a - int*b
    return(list(integer = int, modulus = mod))
}
```

In [57]:
```
# class(result)
# Recall: how do we access the modulus?
result <- int.div(21, 4)
result$integer
```

5

In [58]:
```
int.div <- function(a, b){
    int <- a%/%b
    mod <- a%%b
    return(cat(a, "%%", b, ": \n integer =", int,"\n ------------------", " \n modulus =", mod,
"\n"))
}
int.div(21,4)
```

```
21 %% 4 :
 integer = 5
 ------------------
 modulus = 1
```

In [59]:
```
int.div <- function(a, b){
    int <- a%/%b
    mod <- a%%b
    return(c(a, b))
}
int.div(21, 4)
```

21 4

Example 3. Make the simplest canadian AI chatbot

A function can return something other than an R object, say some voice.

In [60]:
```
# No need to worry about the details here.
# Just want to show that functions do not always have to return() something.
AIcanadian <- function(who, reply_to) {
    system(paste("say -v", who, "Sorry!"))
}
# AIcanadian("Alex", "Sorry I stepped on your foot.")
```

In [61]:
```
# Train my chatbot - AlphaGo style.
# I'll let Alex and Victoria talk to each other.
# MacOS has their voices recorded.
# chat_log <- rep(NA, 8)
# for (i in 1:8) {
#     if (i == 1) {
#         chat_log[1] <- "Sorry I stepped on your foot."
#         system("say -v Victoria Sorry, I stepped on your foot.")
#     } else {
#         if (i %% 2 == 0)
#             chat_log[i] <- AIcanadian("Alex", chat_log[i - 1])
#         else
#             chat_log[i] <- AIcanadian("Victoria", chat_log[i - 1])
#     }
# }
# chat_log
```

Example 4. Check one summary statistic by Smoke group of our 'data' data.

Function arguments can be basically anything, say another function.

In [62]:
```
data_summary <- function(func) {
    data <- read.csv("https://raw.githubusercontent.com/ly129/MiCM2020/master/sample.csv", head
er = TRUE)
    by(data = data$Wr.Hnd, INDICES = list(data$Smoke), FUN = func)
}
data_summary(mean)
```

```
: Heavy
[1] 18.43333
------------------------------------------------------------
: Never
[1] 18.31899
------------------------------------------------------------
: Occas
[1] 18.44
------------------------------------------------------------
: Regul
[1] 19.3
```

Example 5. Default argument value & stop execution

In [63]:
```
a_times_2_unless_you_want.something.else.but.I.refuse.3 <- function(a, b=2){
    if (b == 3) {
        stop("I refuse 3!")
    }

    if (b == 4) {
        warning("4 sucks too.")
    }

    a*b
}
```

In [64]:
```
a_times_2_unless_you_want.something.else.but.I.refuse.3(a = 5)
```

10

In [65]:
```
a_times_2_unless_you_want.something.else.but.I.refuse.3(a = 5, b = 4)
```

```
Warning message in a_times_2_unless_you_want.something.else.but.I.refuse.3(a = 5, :
“4 sucks too.”
```

20

In [66]:
```
# a_times_2_unless_you_want.something.else.but.I.refuse.3(a = 5, b = 3)
```

Exercise:

- Make a function to calculate sample confidence intervals (2.1 f)

In [ ]:

- Use the function in 1 with aggregate( ), by( ) or apply( ) to calculate the sample confidence intervals (2.1 f)

In [ ]:

3. Exercises

A fake dataset is generated. Results should make no biological sense.

In [67]:
```
set.seed(20200306)
N <- 200
height <- round(rnorm(n = N, mean = 180, sd = 10)) # in centimeter
weight <- round(rnorm(n = N, mean = 80, sd = 10)) # in kilograms
age <- round(rnorm(n = N, mean = 50, sd = 10))
treatment <- sample(c(TRUE, FALSE), size = N, replace = T, prob = c(0.3,0.7))
HF <- sample(c(TRUE, FALSE), size = N, replace = T, prob = c(0.1,0.9))

fake <- data.frame(height, weight, age, treatment, HF)
head(fake)
```

A data.frame: 6 × 5

height	weight	age	treatment	HF
<dbl>	<dbl>	<dbl>	<lgl>	<lgl>
186	92	60	FALSE	FALSE
155	74	58	FALSE	TRUE
182	79	62	FALSE	FALSE
178	101	54	FALSE	FALSE
182	72	54	FALSE	FALSE
159	66	41	FALSE	TRUE

1. (Vectorized operation) Calculate BMI for every individual

$$\text{BMI} = \text{weight}(kg)/\text{height}(m)^2$$

In [ ]:

2. (Categorization) BMI Categories:

```
- Underweight = <18.5
- Normal weight = 18.5–24.9
- Overweight = 25–29.9
- Obesity = BMI of 30 or greater
```

3. (*apply) Mean BMI of each BMI group

In [ ]:

```

```

4. (Aggregation) Proportion with heart failure in each BMI-treatment group

In [68]:

```
# Trick:
FALSE+TRUE+TRUE
```

2

In [ ]:

```

```

5. Write a function that allow user to specify

```
- a dataset
- the (binary) treatment variable
- the (binary) outcome variable
```

and return a cross-tabulation (a 2x2 table).

In [ ]:

```

```

5 Pro. The function should be able to check whether the treatment/outcome variables are binary or not. Continuous variables will be dichotomized based on a user-defined threshold.

In [ ]:

```

```

6. Specific task in your own research

In [ ]:

```

```