MATH2411 T1B Tutorial 2

Functions in R

HOU Zhen and LI Yixin

2024-02-07

Reference: https://www.datacamp.com/tutorial/functions-in-r-a-tutorial

- Definition: In programming, functions are instructions organized together to carry out a specific task. The rationale behind functions is to create self-contained programs that can be called only when needed.
- Use functions whenever you expect to run a particular set of instructions **more than twice** in your code.

Built-in functions

```
print("Hello world!")
```

```
## [1] "Hello world!"
```

Use the example given in the first class

Create a vector x

```
x <- c( 970,  612, 1201, 1003,  666, 1088, 744,  898,  964, 1135,
        983, 1016, 1029, 1058, 1085, 1122, 1022, 623, 1197,  883)
```

We can calculte its mean, variance, minimum, maximum, median by using the built-in functions

```
# Print the mean and variance
print(c(mean(x),var(x)))
```

```
## [1]   964.95 31790.26
```

```
# Print the quantiles
print(fivenum(x))
```

```
## [1]  612.0  890.5 1009.5 1086.5 1201.0
```

Functions available in R packages

Will be introduced when used.

Important: You need to read the corresponding documents to check the notations

In application, it is **recommended** to use the built-in functions or functions available in R packages! However, one should be aware of the **conditions** under which the functions can be used.

Reason: Faster (with advance algorithm); Less error

User-Defined functions (UDF)

R functions normally adopt the following syntax:

```
function_name <- function(argument_1, argument_2) {
  function body
  return (output)
}
```

We can distinguish the four main elements:

- Function name.
- Arguments. Key for the function to know what data to take as input.
- Function body. Within curly brackets comes the body of the function, that is, the instructions to solve a specific task based on the information provided by the arguments.
- Return statement. The return statement is required if you want the function to save as variables the result or results following the operations in the function body.

It works just like the functions in math, for example $y = 2x$

```
# We can set a value as default
y <- function(x=0)
{
  return(2*x)
}
print(c(y(),y(100)))
```

```
## [1]   0 200
```

We can also create variables in functions to perform more complex tasks. (See the examples)

Global and Local variables

```
# y defined outside the function is a global variable
y <- 100

# A function without input
glo_loc <- function()
{
  # z defined inside the function is a local varialbe
  z <- 200

  # This y is local. It is different with the global y, though sharing the same name
  y <- 300
  return(c(y,z))
}

print(glo_loc())
```

```
## [1] 300 200
```

The variables (local y, z) defined in the functions are not available outside the function

```
# This y is the global y and its value is not changed by the function glo_loc
print(y)
```

```
## [1] 100
```

```
# There will be an error
# print(z)
```

We can change the global y inside a function by using «- (**NOT suggested**: Can make things complex: Codes difficult to read and errors difficult to find and solve)

```
change_y <- function()
{
  y <<- 400
}
print(y)
```

```
## [1] 100
```

```
change_y()
print(y)
```

```
## [1] 400
```

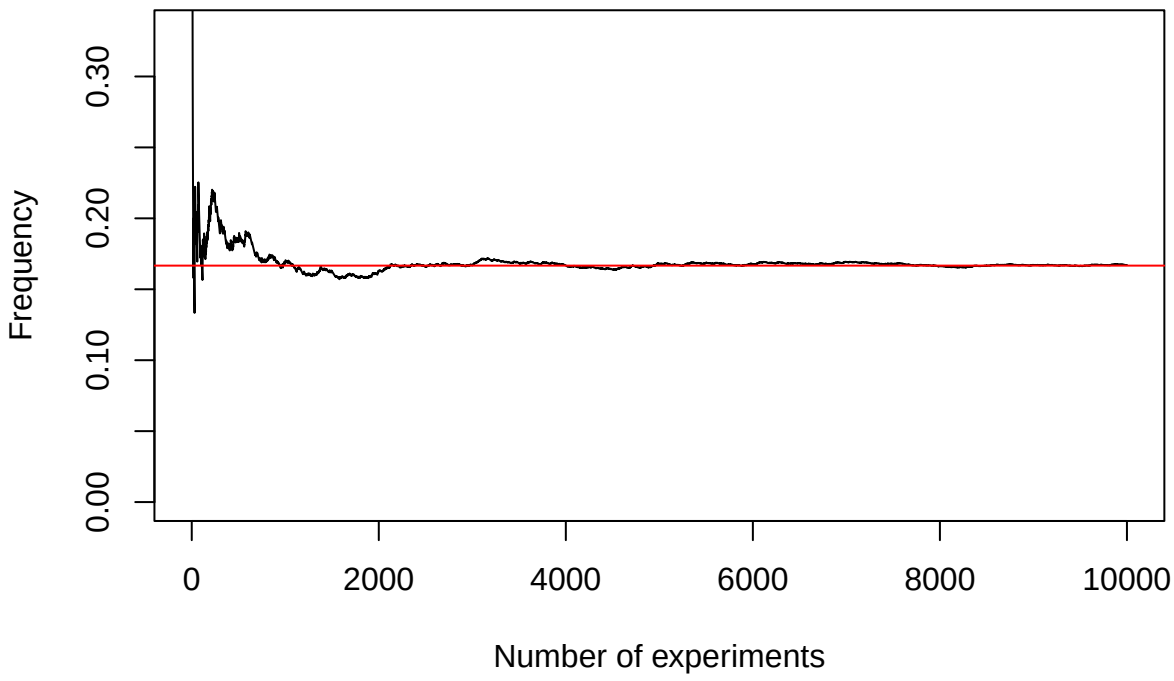
Real examples

Simulation: Toss a pair of dice

Based on P24 of the Slides in the Lecture

```
# To make the experiment repeatable
set.seed(2024)
toss <- function(n)
{
  # Input:
  # n  : The number of experienments
  # Output:
  # fre: The frequence of sum=7

  # Step 1: Generate the random numbers
  sam <- matrix(sample(1:6, 2*n, replace=TRUE), ncol=2)

  # Step 2: Calculate the sums
  sums <- rowSums(sam)

  # Step 3: Check the sums are equal to 7
  sum_equ_7 <- (sums==7)

  # Step 4: Calculate the frequency
  fre <- sum(sum_equ_7)/n

  # Optional: Draw the frequency plot
  freqs <- cumsum(sum_equ_7)/c(1:n)
  plot(freqs, ylim=c(0,1/3), type='l',
       xlab='Number of experiments',
       ylab='Frequency')
  abline(h=1/6, col='red')

  return(fre)
}
```

```
toss(10000)
```




```
## [1] 0.1671
```

Compare the result with

```
1/6
```

```
## [1] 0.1666667
```

Why are they seem close? How close is close? To answer these questions, you need to go further in probability and statistics.

Birthday Problem

Example 5i of Ross, S. M. (2010). A first course in probability (8th ed.). Pearson Prentice Hall.

If n people are present in a room ($n \leqslant 365$), what is the probability that no two of them celebrate their birthday on the same day of the year? (Suppose no one was born on 29/02)

The answer is

$$\frac{(365)(364)\cdots(365-n+1)}{365^n}$$

How to calculate it in a computer?

```
birthday <- function(n)
{
  return(prod(c(365:(365-n+1)))/365**n)
}
```

```r
birthday(20)
```

```
## [1] 0.5885616
```

A better way is to use the logrithm since it is a product of n terms, where n can be very large (which is very common in statistics)

```r
birthday_log <- function(n)
{
  return(exp(sum(log(c(365:(365-n+1))))-n*log(365)))
}

birthday_log(20)
```

```
## [1] 0.5885616
```

Compare the result when $n \geqslant 200$.