STAT 2005 – Programming Languages for Statistics

Tutorial 4 Attributes

2020

LIU Ran

Department of Statistics, The Chinese University of Hong Kong

1 Define Functions

Aim: Write $f(x) = x^2$ and $g(x) = x^2 + 2x + 1$.

```
> f = function(x){
+     y = x^2
+     return(y)
+ }
> f(-3:3) # Only if the calculating process in your function allows the vector input.
[1] 9 4 1 0 1 4 9
> g = function(x){
+     f(x)+2*x+1 # x^2+2*x+1
+ }
> g(-1) # you can call the defined function in another function
[1] 0
```

Remark 1.1. Previous defined functions can be used in new functions' definitions. Therefore, it is better to separate one huge function into several small functions. It will be helpful for debugging and reading.

You can also use the function itself(recursion):$f(n) = n! = n(n-1)(n-2)... * 2 * 1$, so we can get the recurrence formula $f(n) = n * f(n-1) = n * (n-1) * f(n-2) = ...$

```
# Calculate the factorial
Factorial<-function(n){
  if(n == 1){
    return(1)
  }
  else{
    # each time the argument will minus one,
    # and the result will be multiplied by the current value
    return(n*Factorial(n-1))
  }
}
# end at the point one when you call Factorial(1)
```

Please note the difference between the case that the argument is a vector and the case that there are several arguments. It is quite misleading.

```
> sum_of_three<-function(a,b,c){
+     return(a+2*b+3*c)
+ }
> sum_of_three(1:3) # input a vector
Error in sum_of_three(1:3) : argument "b" is missing, with no default
# Use the comma to separate the arguments
> sum_of_three(1,2,3) # input three arguments by order, the first is a, the second is b ...
[1] 14
> sum_of_three(a=1,c=3,b=2) # you can also use the keyword to input arguments
[1] 14
```

If there are the default values, it is better to use the keyword to input values.

```
# add the defualt value in the middle
> sum_of_three<-function(a,b=2,c){
+     return(a+2*b+3*c)
+ }
> sum_of_three(1,3)
Error in sum_of_three(1, 3) : argument "c" is missing, with no default
> sum_of_three(a=1,c=3)
```

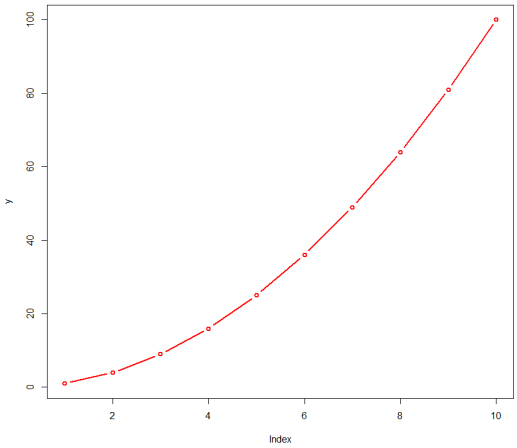
```
[1] 14
# The function will receive the keyword first,
# and the remaining arguments will receive the input by order.
> sum_of_three(1,a=3,3) # a=3,b=1,c=3; keyword is 'a', remain 'b' and 'c'
[1] 14
> sum_of_three(1,b=5,3) # a=1,b=5,c=3; keyword is 'b', remain 'a' and 'c'
[1] 20
# not recommend the mixture input(order and keyword)
```

The three dots ellipsis arguments `...`: it means that the function is designed to take any number of named or unnamed arguments.

```
> s<-function(multiplier, ...){
+   print(..1) # print the first argument in the ellipsis arguments
    # instead of using print(...) directly
+   print(list(...)) # list(a=1,b=2,c=3), transform the ellipsis arguments into a list
+   return(multiplier*sum(...))
+ }
> s(2,a=1,b=2,c=3) # 2*sum(a=1,b=2,c=3) = 12
[1] 1
$a
[1] 1

$b
[1] 2

$c
[1] 3

[1] 12

# Also we can use the keyword, the remaining arguments will be included in the ellipsis
> s(a=1,multiplier = 3,2) # list(a=1,2)
[1] 1
$a
[1] 1

[[2]]
[1] 2

[1] 9
```

The ellipsis arguments are often used in the function for drawing. Because there are many parameter settings in the plot command. It is convenient to use three dots arguments to receive the settings.

```
# f(x) = x^2
> draw_square<-function(data,...){
+   y = data^2 # get the square of data
+   plot(y,...) # transfer the parameters to the function 'plot'
+ }
> draw_square(seq(1,10),col='red',type='b',lwd=2)
# no need for defining arguments 'col','type','lwd'
```



Remark 1.2. Use  to insert indents in functions/loops, which is helpful for understanding and debugging.

Remark 1.3. `return` is only useful inside functions. It can be omitted if the last line in the function is the desired output.

Remark 1.4. The `plot` function's arguments also have the three dots argument. Getting familiar with the three dots will deeper the understanding of some complex functions.

2 `if`

Aim: Write an indicator function $\mathbb{1}(x, b) = \begin{cases} 1 & \text{, if } x \geq b \\ 0 & \text{, if } x < b \end{cases}$.

```
> I = function(x, b){
+     if (x>=b){
+         out = 1
+     } else {
+         out = 0
+     }
+     out
+ }
> I(1, 3)
[1] 0
# input a vector
> I(1:5, 2)
[1] 0
Warning message:
In if (x >= b) { :
  the condition has length > 1 and only the first element will be used
> I = Vectorize(I)
> I(1:5, 2)
[1] 0 1 1 1 1
> I(c(-1, 4, 9), 4:6)
[1] 0 0 1
```

For the vectorized version of if-else, you can also use `ifelse`:

```
> I = function(x, b){
+   ifelse(x>=b,1,0)
+ }
> I(1:5, 2)
[1] 0 1 1 1 1
> I(c(-1, 4, 9), 4:6)
[1] 0 0 1
```

Remark 2.1. The 'and' operator `&` is vectorized but `&&` is not.

Remark 2.2. `Vectorize` can enable a function to support vector inputs. Though it is quite convenient, it can cause serious mistakes sometimes. A better way is to define a function that can support vector inputs at the very beginning.

3 Switch case

`switch(expression, case1, case2, case3....)`

Here, the expression is evaluated and based on this value, the corresponding item in the list is returned. If the value evaluated from the expression matches with more than one item of the list, switch() function returns the first matched item.

```
> switch(2,"red","green","blue")
[1] "green"
> switch('a',a="red",b="green",c="blue")
[1] "red"
```

It is useful for the case which has several complex pipelines(functions):

```
# simple pipelines example
FUNC<-function(data, pipeline_sel){
  switch(pipeline_sel,
         '1' = mean(data), # Function1
         '2' = sd(data),   # Function2
         '3' = sum(data))  # Function3
}
FUNC(c(1,2,3),'3') # return 6
```

4 Loops

4.1 `for` Loop

Aim: Compute $S = \sum_{i=1}^{100} i$.

```
> S = 0
> for(i in 1:100){
+     # already set the beginning, the ending and each step
+     S = S+i
+ }
> S
[1] 5050
```

4.2 `while` Loop

Aim: Compute $S = \sum_{i=1}^{100} i$.

```
> S = 0
> i = 1
> while (i<=100){
+     S = S+i
+     i = i+1 # should update the i
+ }
> S
[1] 5050
```

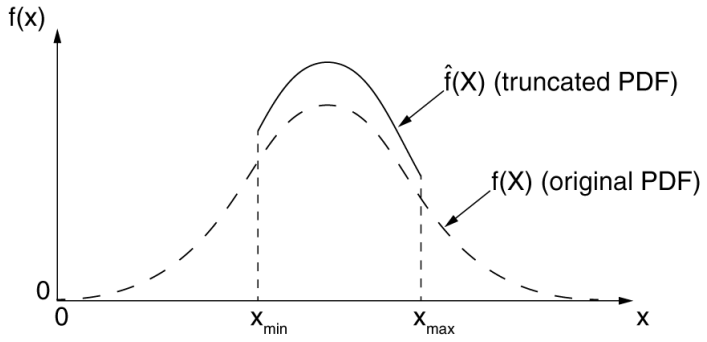
Remark 4.1. Please remember to set the stopping criterion every time when using the loops. Unstoppable loops may cause very serious problems.

Remark 4.2. If you only know the stopping criterion instead of the ending times, use `repeat` or `while` instead of `for`.

Remark 4.3. Although using loop can simplify the problem a lot, it is much slower than vectorized operations in general. Therefore, avoid using loops inside loop unless there is no other choice.

5 An example

We want to get samples from a truncated normal distribution: the truncated normal distribution is the probability distribution derived from that of a normally distributed random variable by bounding the random variable from either below or above (or both).




```r
# get samples from a truncated normal distribution
rnorm_trunc<-function(lower, upper, count, mean, sd){
  sample <- rnorm(count, mean = mean, sd = sd)
  looped <- 0

  # exclude samples outside the interval
  # re-sample the same size of bad samples from the original distribution

  # any() will return False as long as one element is False
  while (any(sample < lower | sample > upper)) {

    # get the number of bad samples
    bad <- (sample < lower | sample > upper) # logical indexs
    bad_values <- sample[bad]
    count_bad <- length(bad_values)

    # re-sample from the original distribution
    new_vals <- rnorm(count_bad, mean = mean, sd = sd)

    # fix the samples
    sample[bad] <- new_vals

    # add a stopping criterion to avoid some extreme cases
    # if after 10000*count times, we still cannot find a proper sample vector.
    if(looped > 10000*count){
      return(NA)
    }
    looped <- looped + 1
  }
  return(sample)
}

> rnorm_trunc(lower = -1,upper = 0,count = 5,mean = 0,sd = 1)
[1] -0.2781313 -0.8784863 -0.8221632 -0.4314080 -0.4344664
```

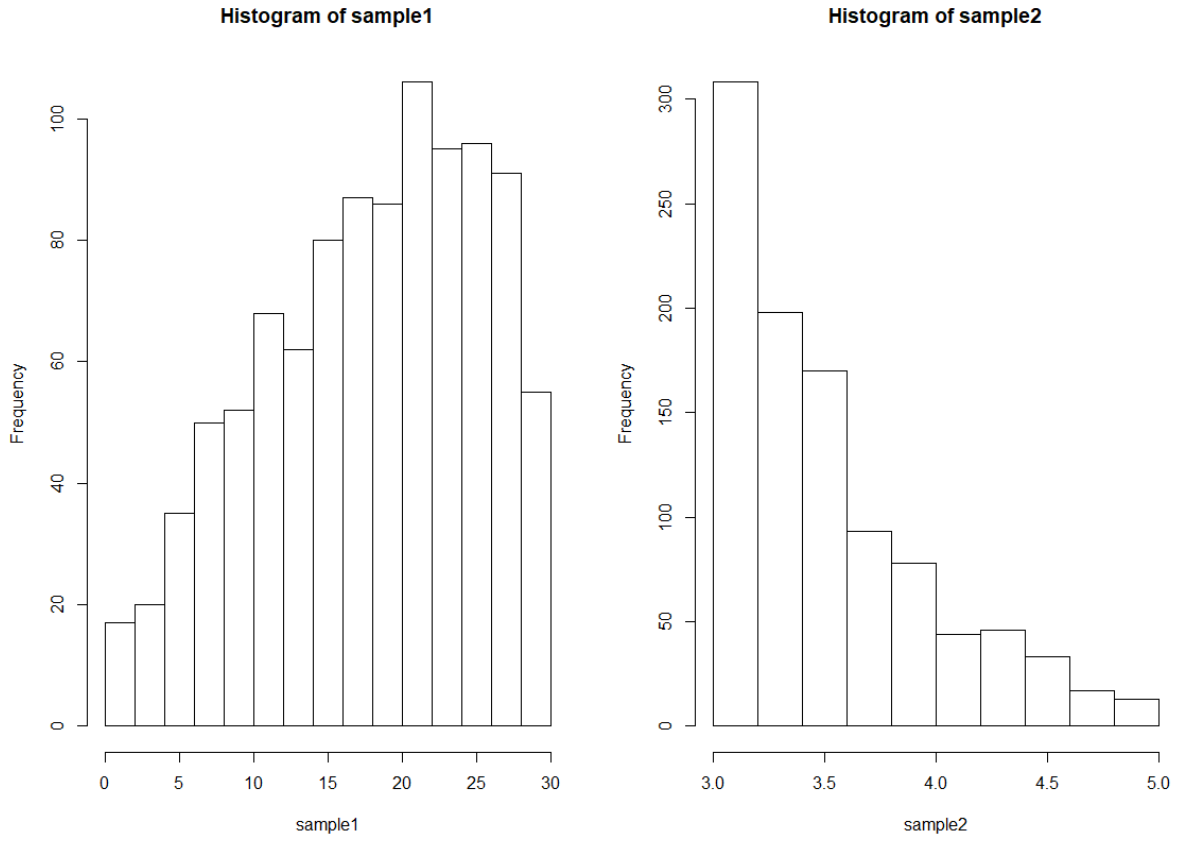
Similarly, if we want to get the truncated samples from any distribution:

Here, we'll use the ellipsis arguments `...` to collect an arbitrary set of parameters and pass them on to internal function calls. When defining a function to take ..., it is usually specified at last. So, we'll write a function called **`sample_trunc()`** that takes five parameters:

1. The lower limit, lower.

2. The upper limit, upper.
3. The sample size to generate, count.
4. The function to call to generate samples, **`sample_func()`**.
5. Additional parameters to pass on to **`sample_func()`**

```r
sample_trunc<-function(lower, upper, count, sample_func, ...){
  sample <- sample_func(count, ...)
  looped <- 0

  while (any(sample < lower | sample > upper)) {
    # fix the sample
    bad <- (sample < lower | sample > upper) # logical
    bad_values <- sample[bad]
    count_bad <- length(bad_values)

    new_vals <- sample_func(count_bad, ...)
    sample[bad] <- new_vals

    if(looped > 10000*count){
      return(NA)
    }
    looped <- looped + 1
  }
  return(sample)
}


sample1 <- sample_trunc(0, 30, 1000, rnorm, mean = 20, sd = 10)
sample2 <- sample_trunc(3, 5, 1000, rexp, rate = 1.5)


par(mfrow=c(1,2))
hist(sample1)
hist(sample2)
```



6 Attributes(optional)

This is the prior knowledge of the object-oriented programming(oop), which is optional for you in this course, but I think it is very important for you to learn how to code in an oop way.

Most packages are programmed in this way, if you are familiar with this oop, you will have a deeper understanding of them. And then, you may write your own package.

Both the names and the dimensions of matrices and arrays are stored in R as attributes of the object. These attributes can be seen as labeled values you can attach to any object.

You can define and extract an attribute by the function **`attr()`**.

```
> set.seed(2005)
> samples <- rnorm(10, mean = 0, sd = 10)

# define an attribute and assign a value to it
> attr(samples, 'disttype') <- 'norm'
> print(samples)
 [1]   9.640380  12.689196   4.731471   5.144074  -4.108097   0.694464  -2.861926   2.512387
 [9]   9.600874 -14.079819
attr(,"disttype")
[1] "norm"

# one way to nicely print the structure of a list (or other data object) is 'str'
# emphasize on the structure
> str(samples)
 num [1:10] 9.64 12.69 4.73 5.14 -4.11 ...
 - attr(*, "disttype")= chr "norm"

# extract the attribute
> attr(samples, "disttype"))
[1] "norm"

# another way, attributes() will return a list which contains all the attributes
> attributes(samples)$disttype
[1] "norm"
```

Many R functions return sorts of complex attribute-laden lists. Consider the **`t.test()`** function, which compares the means of two vectors for statistical equality(Hypothesis test):

```
set.seed(2005)
samp1 <- rnorm(100,mean = 10,sd = 5)
samp2 <- rnorm(100,mean = 8,sd = 5)
tresult <- t.test(samp1, samp2)

# it belongs to 'htest' class
> class(tresult)
[1] "htest"

# When printed, the result is a nicely formatted, human-readable result.
> print(tresult)

    Welch Two Sample t-test

data:  samp1 and samp2
t = 1.0739, df = 197.79, p-value = 0.2842
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.5789316  1.9634773
sample estimates:
mean of x mean of y
 9.319114  8.626841
```

If we run **`str(tresult)`**, however, we will find the true nature of tresult: it's a list.

```
> str(tresult)
List of 10
 $ statistic  : Named num 1.07
  ..- attr(*, "names")= chr "t"
 $ parameter  : Named num 198
```

```
  ..- attr(*, "names")= chr "df"
 $ p.value     : num 0.284
 $ conf.int    : num [1:2] -0.579 1.963
  ..- attr(*, "conf.level")= num 0.95
 $ estimate    : Named num [1:2] 9.32 8.63
  ..- attr(*, "names")= chr [1:2] "mean of x" "mean of y"
 $ null.value : Named num 0
  ..- attr(*, "names")= chr "difference in means"
 $ stderr      : num 0.645
 $ alternative: chr "two.sided"
 $ method      : chr "Welch Two Sample t-test"
 $ data.name   : chr "samp1 and samp2"
 - attr(*, "class")= chr "htest"
```

Given knowledge of this structure, we can easily extract specific elements, such as the confidence level:

```
> attr(tresult$conf.int,'conf.level')
[1] 0.95
```

Actually, the nice formatted result for **print** is because the class of tresult is **'htest'**.

```
> class(tresult) # use the 'class' function
[1] "htest"
> attr(tresult,'class') # get the attribute's value
[1] "htest"

> attr(tresult,'class')<-NULL # delete this attribute
> class(tresult) # back to the original data object
[1] "list"

> print(tresult) # print as a list instead of a nice formatted result
$statistic
       t
1.073929

$parameter
      df
197.7907

$p.value
[1] 0.284163

$conf.int
[1] -0.5789316  1.9634773
attr(,"conf.level")
[1] 0.95

$estimate
mean of x mean of y
 9.319114  8.626841

$null.value
difference in means
                  0

$stderr
[1] 0.6446171

$alternative
[1] "two.sided"

$method
[1] "Welch Two Sample t-test"

$data.name
[1] "samp1 and samp2"
```

The interpreter notices that the "class" attribute is set to "htest", and searches for another function with a different name to actually run: print.htest(). You can check it in **methods(print)**.

I will give more detail in the next tutorial.