STAT 2005 – Programming Languages for Statistics

Tutorial 5 Debugging, Outer and Objects

2020

LIU Ran

Department of Statistics, The Chinese University of Hong Kong

1 Debugging

Sometimes, we have errors in a defined function and want to check the values inside the function. However, if we call the function directly, we cannot see the values inside the function except the return value. We need other methods to achieve this goal.

Here are three ways to view the values of the variables inside the function.

1.1 `browser()`

Use **`browser()`** command at the beginning of your function or before the loop to run the code line by line. (Similar with debugging in C/C++)

```
my_function <- function(){

  # want to see the change of result during the loop
  browser()

  # record each time's mean value
  result = NULL
  for (i in 1:10) {
    x <- rnorm(10)
    result = c(result,mean(x))
  }

  return(result)
}

my_function()
```

You will see the result appending one by one. Specifically for loops, if you don't want to run the code line by line, and want each time the code will stop at the exact line, you can use the breakpoint. You can do this in RStudio by clicking to the left of the line number in the editor, or by pressing Shift+F9 with your cursor on the desired line.

```
 3 my_function <- function() {
 4
 5   # browser()
 6
 7   result = NULL
 8   for (i in 1:10) {
 9     x <- rnorm(10)
10     result = c(result,mean(x))
11   }
12   return(result)
13 }
14
15 my_function()
```

You can see each time the code will stop at the breakpoint line. But remember that unlike the **`browser()`**, you must use the **`source`** command to activate the breakpoints.

1.2 print or cat

You can print the variables inside the function:

```
my_function <- function(){

  # record each time's mean value
  result = NULL
  for (i in 1:10) {
    x <- rnorm(10)
    result = c(result,mean(x))
    print(result) # cat(result)
  }

  return(result)
}

my_function()
```

It is a direct way to check the values inside the function. But when there are many variables to be checked, it is quite messy.

1.3 message and sink

Because sometimes, we print some values just for debugging. After ensuring our code is good, we want to remove the print results which are just for debugging and only remain the output results we want to show. Here we can use **message** command to achieve this goal.

```

set.seed(2005)
my_function <- function(){

  # record each time's mean value
  result = NULL
  for (i in 1:10) {
    x <- rnorm(10)
    result = c(result,mean(x))
    print(i)
    message(result)
  }

  return(NULL)
}

con <- file("test.log", open = "wt")

# record the messages and outputs
# all the output and message will be recorded into the file test.log
# and there are no texts in the console
# append: whether or not the file will be overwritten
sink(con, append=F, type=c('message'))
sink(con, append=F, type=c('output'))

# just record the outputs
# sink(con, append=F, type=c('output'))

my_function()

# After recording, reset the output and message record setting,
# Following outputs will be displayed in the console again
sink()
sink(type="message")

print('All setting returns to the default')
```

Remark 1.1. **sink** command is very useful for a huge function's debugging which may have many messages to present, but it is also more complicated than the previous two methods. If your function is small, you can choose the much simpler one.

Remark 1.2. Besides **message**, you can also use **wanring** and **stop** to present different levels' information.

2 Outer

The outer product of the arrays X and Y is the array A with dimension c(dim(X), dim(Y)) where element A[c(arrayindex.x, arrayindex.y)] = FUN(X[arrayindex.x], Y[arrayindex.y], ...).

FUN must be a vectorized function expecting at least two arguments and returning a value with the same length as the first (and the second).

An example in class:

```
# get the elements on the right diagonal
slash <- function(X) {
    m<-ncol(X)
    n<-nrow(X)
    # remain the elements whose summation of indexes is equal to 5
    # discard other elements
    (outer(1:n,1:m,"+")==min(m,n)+1)*X
}
> matrix(1:20,ncol=4)
     [,1] [,2] [,3] [,4]
[1,]    1    6   11   16
[2,]    2    7   12   17
[3,]    3    8   13   18
[4,]    4    9   14   19
[5,]    5   10   15   20

>slash(matrix(1:20,ncol=4))
     [,1] [,2] [,3] [,4]
[1,]    0    0    0   16
[2,]    0    0   12    0
[3,]    0    8    0    0
[4,]    4    0    0    0
[5,]    0    0    0    0
```

Another example: if we want to create a matrix $a_{ij} = i + j$ and the dimension is `c(3,3)`:

```
> dimension = 3
> A = outer(1:dimension,1:dimension,'sum')
Error in dim(robj) <- c(dX, dY) :
  dims [product 9] do not match the length of object [1]

> dimension = 3
> A = outer(1:dimension,1:dimension,'+')
     [,1] [,2] [,3]
[1,]    2    3    4
[2,]    3    4    5
[3,]    4    5    6
```

This is because your function must return a value with the same length as the first (and the second) when it is a vector. In the **outer** command, we do not calculate the function value by elements. We will do it in a vectorized version:

```
> sum(c(1,2,3),1)
[1] 7

# a vector with the same length
> c(1,2,3) + 1
[1] 2 3 4
```

So, When you use the **outer** command, please first check your input function. Does it satisfy the requirements of **outer**?

3 Objects and Classes in R(optional)

Last week, we talked about the function can have different effects on objects with different classes. Let's have a brief review about it.

Next, due to the time limitation, I will give a very brief introduction of OOP in R.

3.1 S3 and S4 classes

There are two main systems to define a class, S3 and S4. S3 is the older system which is informal. For example, an object can have several classes. To be more formal and rigorous, people propose a new system S4 to replace the S3.

Nowadays, there are still some methods and classes in R with S3 classes. It is because of the compatibility for old versions. But we are encouraged to use the S4 system. So, here I will focus on the concepts about S4 system.

3.2 Some descriptions

1. A class is a description(type) of a thing. A new class can be defined using `setClass()`. For example, you can state that **person** and **student** are two classes. (?setClass)
2. An object is an instance of a class. Objects can be created using `new()`. For example, **John** and **David** could be two objects with the class **student**, i.e., John and David are two students.(?new)
3. A generic function is an R function which dispatches methods. (e.g. print, plot) Actually, the generic functions do not do any computation. The only thing they do is to take the data and figure out which class the data comes from. Finally, find the specific method for this class and call this method.(e.g. print.htest) (?setGeneric)
4. A method is the implementation of a generic function for an object of a particular class. (e.g. print.htest) (?setMethod)

3.3 Reason for a new class

The reason why we want to create a new class is to represent new types of data(e.g. gene expression, space-time, hierarchical, sparse matrices). They may have very important properties to show and there may be more efficient way to deal with these kinds of data. Therefore, we need to create a new class and new methods for these kinds of specific data.

3.4 An example

Define two classes 'person' and 'student':

```
# define a class 'person' which has properties: name, age and sex
# slots are properties of a class(store data)
person <- setClass(Class = 'person', slots = c(name = 'character',
                                    age = 'numeric',
                                    sex = 'character'))

# create an object with the class 'person'
# p1 = new('person', name = 'John', age = 23, sex = 'Male')
p1 = person(name = 'John', age = 23, sex = 'Male')

# get the name of this object p1
p1@name

# define a class 'student' which have the same slots with the 'person' class and
# one more slot 'gpa'(inherit)
```

```
# son class will inherit the slots of the father class
# use contains to achieve this goal
student <- setClass(Class = 'student', slots = c(gpa = 'numeric'),
                    contains = 'person')

# create an object with the class 'student'
p2 = new('student', name = 'David',gpa = 3.9, age = 22, sex = 'Male')
p2 = student(name = 'David',gpa = 3.9, age = 22, sex = 'Male')
p2@gpa
```

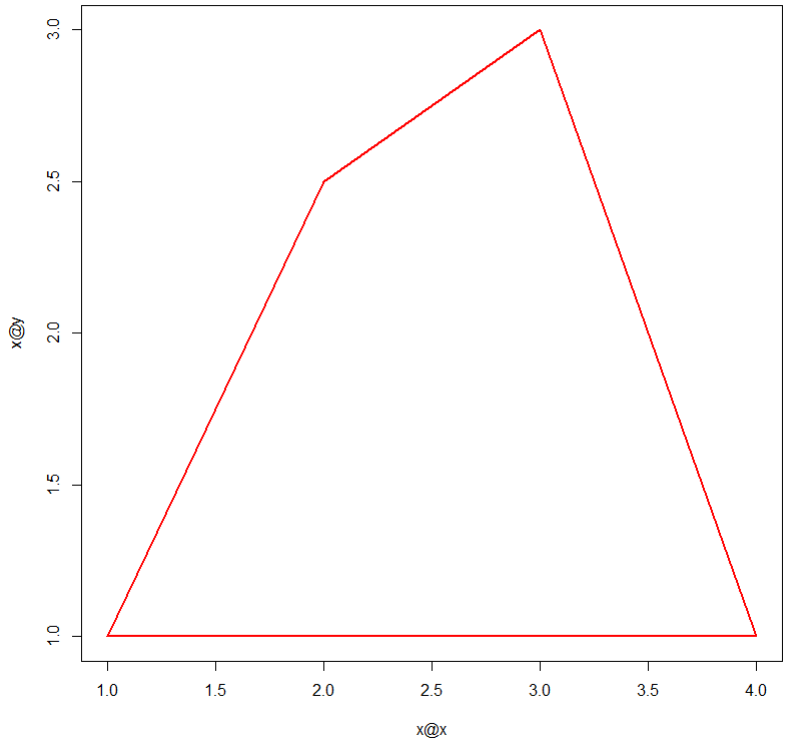
3.5 Another example

We define a polygon class and extend the generic function `plot` to have a method specialized for the class polygon.

```
# slots: x coordinate, y coordinate
setClass('polygon',slots = c(x = 'numeric', y = 'numeric'))

# extend the already defined generic function plot
# to define the method for the new class 'polygon'
setMethod('plot','polygon',
          # here the arguments for plot function should not change
          # actually, we don't use y here
          function(x,y,...){
            plot(x@x,x@y,type = 'n')
            # want to get a closed fgure
            # so we add the line between the first point and the final point
            xp <- c(x@x, x@x[1])
            yp <- c(x@y, x@y[1])
            lines(xp,yp,...)
          })

p<-new('polygon', x = c(1,2,3,4), y = c(1,2.5,3,1))
plot(p,col = 'red',lwd = 2)
```



Remark 3.1. More materials:

1. More examples: Bioconductor (a free, open source and open development software project for the analysis and comprehension of genomic data)
2. Some videos: a lecture by Prof. Peng