Chapter 4: Interactice Notebook for Students

Ram Gopal, Dan Philps, and Tillman Weyde

Summer 2022

Table of Contents

# Functions

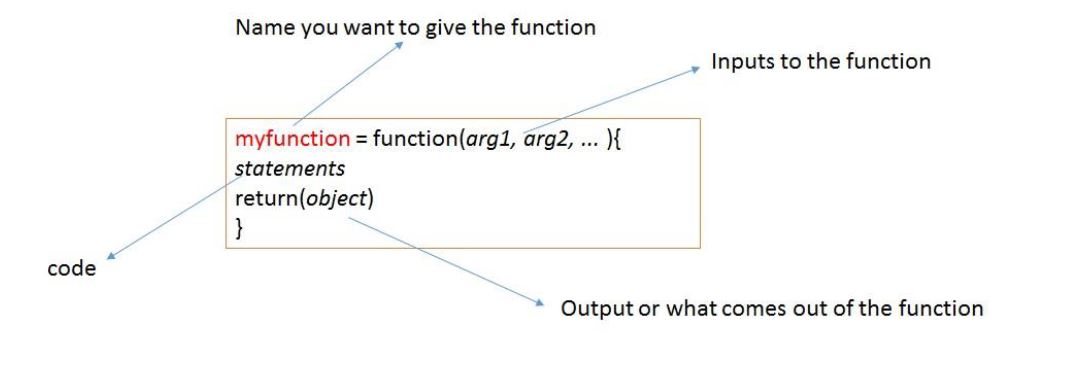
Thus far, we have used many functions available in base R as well as in packages that we have explored. Examples of these functions include mean(), std(), integrate, and others. You can create your own functions, as well. Let us do an example.

There is no function available in base R to compute geometric mean. Geometric mean of positive numbers ,,… is defined as:

Now we will attempt to create a function ourselves. Before we do this, let us write the code to compute the geometric mean of a set of numbers.

v = c(1,33,4,29,9,90)  
gmean = (prod(v))^(1/length(v))  
gmean

## [1] 12.08

The above code works fine to compute the geometric mean. What we would like to do is package this into a function so it can be used repeatedly in this script or others that we may write. This is the anatomy of a function:  


geomean = function(v){  
gmean = (prod(v))^(1/length(v))  
return(gmean)  
}

That is how you create a function! Once the function is created, you can send any input as a vector and our function will calculate the answer. Here is an example of how we call the function.

x = c(1,3,54,9,29,234,6,2,8,2456,2,8)  
geomean(x)

## [1] 13.52

As another example, let us compute the harmonic mean which is defined as:

$${n}\over{{1\over{a\_1}}+...+{1\over{a\_n}}}$$

The following code will give us the harmonic mean.

d = c(2,4,63,6,3)  
g = 1/d  
hmean = length(d)/(sum(g))  
hmean

## [1] 3.95

Now, let us create a function for harmonic mean.

harmean = function(v){  
 a = 1/v  
 hmean = length(v)/(sum(a))  
 return(hmean)  
}

Now we will test both of the functions.

v = c(2,23,3456,1,4,6)  
geomean(v)

## [1] 12.5

harmean(v)

## [1] 3.061

Let us try another example. Suppose we have a vector and we want to find how many elements of the vector are above a cutoff number, say 6. The following code can be used to find this.

v = c(4,6,23,6,375,9)  
cutoff = 6  
vsub = v[v>cutoff]  
vsub

## [1] 23 375 9

length(vsub)

## [1] 3

Next, let us package this into a function. You should notice that this function should have two inputs: the vector and the cutoff value.

vecsize = function(v,cutoff = 1){  
 vsub = v[v>cutoff]  
 return(length(vsub))  
}

Let us see how this function works.

v = c(.2,5,2,.6,3,57,34,5)  
vecsize(v,10)

## [1] 2

You will notice that in the function, we have set a default cutoff value of 1 by using cutoff = 1. If you do not specify a cutoff value when using the function, it defaults to 1. For example

vecsize(v)

## [1] 6

Let us do another example. Suppose we have a vector and we want to see all the elements between 5 and 100. The following code can be used to find this.

vec1 = c(2,2,6,3,546,2346,22,34,7,21,4)  
cutoff1 = 5  
cutoff2 = 100   
vsub = vec1[vec1>cutoff1 & vec1<cutoff2]  
vsub

## [1] 6 22 34 7 21

Now, let us turn this into a function.

Intermediate = function(vec1, cutoff1, cutoff2){  
 vsub = vec1[vec1>cutoff1 & vec1<cutoff2]  
 return(vsub)  
}

Let us see how this works.

vec1 = c(2,2,6,3,546,2346,22,34,7,21,4)  
Intermediate(vec1,5,100)

## [1] 6 22 34 7 21

# Conditional Statements

Conditional statements are very useful and are an important element to any programming language. There are three varieties of syntax for conditional statements in R:

1. if (condition) {trueExpressions}
2. ifelse (condition,trueExpressions,falseExpressions)
3. if (condition){trueExpressions} else {falseExpressions}

Suppose we have a vector of numbers and we want to create another vector containing 0s and 1s. If the element in the first vector is 5 or above, we want to put a 1 for the corresponding element in the second vector. Otherwise, we want to put a 0. The appropriate code in R is

v1 = c(2,36,83,2,5,7,2,9,3,12)  
v2 = ifelse(v1>=5,1,0)  
print(v2)

## [1] 0 1 1 0 1 1 0 1 0 1

Let us do another example. In the MASS package, there is a data frame called whiteside. Let us take a look at it.

library(MASS)  
head(whiteside)

## Insul Temp Gas  
## 1 Before -0.8 7.2  
## 2 Before -0.7 6.9  
## 3 Before 0.4 6.4  
## 4 Before 2.5 6.0  
## 5 Before 2.9 5.8  
## 6 Before 3.2 5.8

Suppose we want to create a column in which temperatures below 5 are labeled cold and temperatures above 5 are labeled hot.

whiteside$hotcold = ifelse(whiteside$Temp<5,"cold","hot")  
head(whiteside)

## Insul Temp Gas hotcold  
## 1 Before -0.8 7.2 cold  
## 2 Before -0.7 6.9 cold  
## 3 Before 0.4 6.4 cold  
## 4 Before 2.5 6.0 cold  
## 5 Before 2.9 5.8 cold  
## 6 Before 3.2 5.8 cold

Next, let us create a summary table of average gas consumption when it is hot and cold, and with and without insulation. As before, we use the aggregate() function to summarize.

x = aggregate(whiteside$Gas,list(whiteside$Insul, whiteside$hotcold),mean)  
x

## Group.1 Group.2 x  
## 1 Before cold 5.940  
## 2 After cold 3.885  
## 3 Before hot 4.006  
## 4 After hot 2.680

It works well! As expected, gas consumption is high when it is cold and when there is no insulation. Notice that the column names are not very informative. The column names of a data frame can be changed using the colnames() function.

colnames(x)

## [1] "Group.1" "Group.2" "x"

colnames(x) = c("Insulation","Weather", "Average\_Gas\_Consumption")  
x

## Insulation Weather Average\_Gas\_Consumption  
## 1 Before cold 5.940  
## 2 After cold 3.885  
## 3 Before hot 4.006  
## 4 After hot 2.680

# Looping

## For Loops

Looping or iterations is another key element of programming. The syntax is as follows:

for (var in seq) {statements}

It is common to use i as the variable in the sequence. The command print allows you to write things out from inside the loop.

Let us try an example. Say that you want to print the numbers 1-10 using a loop.

for (i in 1:10)  
{  
 print(i)  
}

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

Next, let us say you want to print the product of a term and the two terms before it. The code would be as follows:

for (i in 1:10)  
{  
 print(i\*(i-1)\*(i-2))  
}

## [1] 0  
## [1] 0  
## [1] 6  
## [1] 24  
## [1] 60  
## [1] 120  
## [1] 210  
## [1] 336  
## [1] 504  
## [1] 720

Now, let us say you want a vector that gives you the running sums the elements in vector . The code can be written as follows:

v1 = c(1,4,5,2,4,6,7,8,9,12,4,1,5,2)  
v2 = v1  
for (i in 2:length(v1))  
{  
 v2[i]=v2[i-1]+v1[i]  
}  
v2

## [1] 1 5 10 12 16 22 29 37 46 58 62 63 68 70

For the next example, we want to create a vector that gives you all the prime numbers between two integers of your choice. To do this, you will need to install the package schoolmath.

library(schoolmath)  
ans = c()  
a = 20  
b = 30  
for (i in a:b)  
{  
 if(is.prim(i)==TRUE){ans = c(ans,i)}  
}  
ans

## [1] 23 29

## Avoiding Loops

Even though we wrote the above code using loops, we won’t necessarily have to use loops in R. This is because R does vectorized calculations. This makes it easy to code without having to write loops many times. For example, when we did the sum of a vector, to add up all the numbers, we used sum(). In other programming languages which do not support vectorized calculations, you will have to write a loop to calculate the sum of the numbers in a vector. In general, use vectorized calculations and try to avoid using loops.

Now, we will rewrite the above code above without using a loop.

x = seq(20:30)  
y = is.prim(x)  
y

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

x[y]

## [1] 1 2 3 5 7 11

If you compare the two chunks of code, you will see that writing the code without using loops is much simpler and more elegant.

Now, suppose you have a vector and you want to calculate the following:

total = 0  
v = c(1,4,2,5,73,4,5)  
for (i in 1:(length(v)-1))  
{  
 total = total + (v[i]\*v[i+1])   
}  
total

## [1] 699

Again, there is a clever way to calculate this total without using loops by creating vectors and . The code is:

v1 = v[1:(length(v)-1)]   
v2 = v[2:length(v)]  
total = sum(v1\*v2)   
total

## [1] 699

In some cases, you will have no option but to create a loop. Here is an example.

Let us say you have a vector and you want a loop that flips the order of the elements in the vector.

v1 = c(72,3,57,2,8,24,7)  
v2 = c()  
n = length(v1)  
for (i in 1:n)  
{  
 v2[i] = v1[n +1 - i]  
}  
v2

## [1] 7 24 8 2 57 3 72

## While Loops

Another type of iteration is the while loop. The syntax in R is  
while(cond) expr

The while loop is used when you want to repeat a set of commands while a condition remains true.

Suppose you have a starting integer value, let us say 19. Now suppose you want to find ten prime numbers that follow 19. For this, the for loop is not useful. The logic to create this is as follows.

We will create one index called i that starts at 19 and will increment by 1 at each iteration. We will create another index primi which starts at 0 and will increment by 1 if a prime number is found. The while loop will continue until primi reaches 10.

library(schoolmath)  
startnumber = 300  
numprimes = 3  
i = startnumber + 1  
primi = 0  
primevec= c()  
while (primi<numprimes)  
{  
 if(is.prim(i)==TRUE){  
 primi = primi + 1  
 primevec = c(primevec,i)  
 }  
 i = i +1  
}  
print(primevec)

## [1] 307 311 313

Let us package this into a function.

nextprimes = function(startnumber, numprimes){  
 primi = 0  
 i = startnumber + 1  
 primevec = c()  
 while (primi<numprimes)  
 {  
 if(is.prim(i)==TRUE){  
 primi = primi + 1  
 primevec = c(primevec,i)  
 }  
 i = i +1  
 }  
 return(print(primevec))  
}

Now, let us test the function.

nextprimes(3,3)

## [1] 5 7 11

nextprimes(19,10)

## [1] 23 29 31 37 41 43 47 53 59 61

nextprimes(9,3)

## [1] 11 13 17