R Library: Introduction to functions

The R program (as a text file) for the code on this page.
In order to see more than just the results from the computations of the functions (i.e. if you want to see the functions echoed back in console as they are processed) use the **echo**=T option in the **source** function when running the program.

```
source("c:/stat/intro_function.txt", echo=T)
```

One of the aspects of R which makes these packages different from other statistical packages is that they are based on the computer language S. In other words, we have an entire computer language at our disposal when we program in R which allows us to easily and elegantly write virtually any function that we want to implement. This page is designed to help the novice R user get a general idea of how to write basic functions.

Basic set up for functions:

```
function.name <- function(arguments) {
purpose of function
i.e. computations involving the arguments
}
```

Creating a function, called **f1**, which adds a pair of numbers.

```
Example 1:

f1 <- function(x, y) {
  x+y
}

f1( 3, 4)
[1] 7
```

If we have a function which performs multiple tasks and therefore has multiple results to report then we have to include a **return** statement (with **c()**) inside the function is order to see all the results. In the following example the function **f.bad** does not have a **return** statement and thus only reports the last of the computations whereas the function **f.good** has a **list** statement and thus reports all the results.
BEWARE: The return statement exits the function. Thus, it is important to include the return statement at the end of the function!

```
Example 2:

f.bad <- function(x, y) {
 z1 <- 2*x + y
 z2 <- x + 2*y
 z3 <- 2*x + 2*y
 z4 <- x/y
}

f.bad(1, 2)
[1] 0.5

f.good <- function(x, y) {
 z1 <- 2*x + y
 z2 <- x + 2*y
 z3 <- 2*x + 2*y
 z4 <- x/y
 return(c(z1, z2, z3, z4))
}

f.good(1, 2)
$z1:
[1] 4

$z2:
[1] 5
```

```
$z3:
[1] 6

$z4:
[1] 0.5
```

Furthermore, when we have a function which performs multiple tasks (i.e. computes multiple computations) then it is often useful to save the results in a list. Now we can access each result separately by using the list indices (double square brackets).
Note: The variables **z1** and **z2** exist only inside the function **f2** and you can not refer to them outside the function. Thus, we can not make a call to **f3(2, 5)$z1** as is demonstrated at the end of the example.

```
Example 3:

f2 <- function(x, y) {
  z1 <- x + y
  z2 <- x + 2*y
  list(z1, z2)
}

f2(2, 5)
[[1]]:
[1] 7

[[2]]:
[1] 12


f2(2, 5)[[1]]
[1] 7

f2(2, 5)[[2]]
[2] 12

f2(2, 5)$z1
NULL
```

We are using the same function as before but now we name the elements in the list of results. We then have a choice of accessing the results using either the list indices or the names of the elements in the list.

```
Example 4:

f3 <- function(x, y) {
  z1 <- x + y
  z2 <- x + 2*y
  list(result1=z1, result2=z2)
}

f3(2, 5)
$result1:
[1] 7

$result2:
[1] 12

f3(2, 5)$result1
[1] 7

f3(2, 5)$result2
[1] 12
```

It is often convenient to store the result of function in an object. Let's store the results of the function **f3** applied to the pair (1, 4) in an object called **y** which in this case will be a list. If we need to see the names for the objects in the list **y** then we apply the **names** function to **y**. We can access the results stored in the list **y** either by the name of the elements or by the list indices.

```
Example 5:

y <- f3(1, 4)
names(y)
[1] "result1" "result2"

y$result2
```

```
[1] 9

y[[2]]
[1] 9
```

Types of arguments

In all the functions created so far we have not put any restrictions on the types of arguments that we can use. This means that we can either use single numbers for each arguments as we have been doing in the examples, or x and y can be vectors or matrices. The only precaution is that when using vectors or matrices for both x and y then they must have the same dimension or else the computations will not be performed.

```
Example 6:

#Using vectors
v1 <- seq(1:5)
v1
[1] 1 2 3 4 5 6

v2 <- seq(2, 12, 2)
v2
[1]  2  4  6  8 10 12

f3(v1, v2)
$result1:
[1]  3  6  9 12 15 18

$result2:
[1]  5 10 15 20 25 30

#Using matrices
mat1 <- matrix( c(1, 2, 3, 4, 5, 6), ncol=2)
mat1
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

mat2 <- matrix(c(2, 4, 6, 8, 10, 12), ncol = 2)
mat2
     [,1] [,2]
[1,]    2    8
[2,]    4   10
[3,]    6   12

f3(mat1, mat2)
$result1:
     [,1] [,2]
[1,]    3   12
[2,]    6   15
[3,]    9   18

$result2:
     [,1] [,2]
[1,]    5   20
[2,]   10   25
[3,]   15   30
```

Default arguments

It is very easy and often very useful to specify default arguments in a function. In the following example the function **f4** is the same as **f3** except that the default arguments in the function are x=3 and y=2. By leaving the arguments blank in the call to **f4** we use the default arguments. If we call **f4** and list a pair of numbers as the arguments then the function will use the first number as x and the second as y. If we wish to change this ordering we can do this by using the x=value and y=value for the arguments and then the function will know how to match the numbers to the arguments.

```
Example 7:

f4 <- function(x=3, y=2) {
  z1 <- x + y
  z2 <- x + 2*y
  list(result1=z1, result2=z2)
}

#using the defaults values for the x and y arguments
f4()
$result1:
[1] 5

$result2:
[1] 7

#using the default value for the y argument
f4(1,  )$result1
[1] 3

f4(x=1)$result1
[1] 3

#using the default value for the x argument
f4(, 1)$result1
[1] 4

f4(y=1)$result1
[1] 4

#switching the order of the arguments
f4(y = 1, x = 2)$result2
[1] 4
```

Using for loops

The **for** loop is used when iterating through a list.

```
The basic structure of the for loop:

for(index in list){ commands }

Example 8:

for(i in 2:4) {
    print(i)
}
[1] 2
[1] 3
[1] 4
```

Unlike the general function the **for** loop does not have a **return** statement. When we just save the computation in an object then we will only be able to access the current value which will be the value after the loop has finished. If we wish to see the value at each iteration then we must use the **print** statement inside the loop. The list does not have to contain only numbers, any objects will work.

```
Example 9:

for(i in c(1, 3, 6, 9)) {
    z <- i + 1
}
z
[1] 10

#using the print statement to see result at each iteration
for(i in 3:5) {
    z <- i + 1
    print(z)
}
[1] 4
[1] 5
[1] 6

#The list does not have to contain numbers
cars <- c("Toyota", "Ford", "Chevy")
```

```
for(j in cars) {
    print(j)
}
[1] "Toyota"
[1] "Ford"
[1] "Chevy"
```

Including a **for** loop in a function. The following function will add the pairs of numbers. We use the **print** statement to view the results from each iteration inside the **for** loop and a **return** statement to see the results of the final y once we have exited the **for** loop. Note that we included the **return** statement at the end of the function because the **return** statement will exit the function.

```
Example 10:

f5 <- function(x) {
  for(i in 1:x) {
      y <- i*2
      print(y)
  }
  return(y*2)
}

f5(3)
[1] 2
[1] 4
[1] 6
[1] 12
```

Sometimes it is useful to have a **break** statement in the loop. This is often combined with an **if** function such that a **break** from the loop will occur if the condition specified in the **if** function is satisfied.

```
Example 11:

names1 <- c("Dave", "John", "Ann", "Roger", "Bill", "Kathy")
f.names <- function(x) {
        for(name in x){
          if(name=="Roger")
                break
          print(name)
        }
}

f.names(names1)
[1] "Dave"
[1] "John"
[1] "Ann"
```

Using while loops

The **while** loop is used when you want to keep iterating as long as a specific condition is satisfied.

```
The basic structure of the while loop:

while(condition){ commands }

Example 12:

i <- 2
while(i <= 4) {
      i <- i+1
      print(i)
      }
[1] 2
[1] 3
[1] 4
```

Just as in the **for** loop we do not have a **return** statement inside the **while** loop. If we wish to see the results of each iteration then we have to use a **print** statement. Once we have exited the **while** loop then we can use the **return** statement. Since the **return** statement exits the function we will include the **return** statement at the end of the function.

```
Example 13:

f6 <- function(x) {
        i <- 0
  while(i < x) {
   i <- i+1
   y <- i*2
   print(y)
  }
  return(y*2)
}

f6(3)
[1] 2
[1] 4
[1] 6
[1] 12
```

It is rare to combine a **while** loop with a **break** statement since the function will only iterate as long as a specified condition is true. As an example we will rewrite the **f.names** function using a **while** loop rendering the **break** statement unnecessary.

Example 14:

```
names1 <- c("Dave", "John", "Ann", "Roger", "Bill", "Kathy")
f.names.while <- function(x) {
        i <- 1
        while( x[i] != "Roger"){
          print(x[i])
          i <- i+1
        }
}

f.names.while(names1)
[1] "Dave"
[1] "John"
[1] "Ann"
```

Using repeat loops

The **repeat** loop is an infinite loop and it is very often used in conjunction with a **break** statement.

The basic structure of the repeat loop:

```
repeat {
   commands
   if(condition)
        break
}
```

Example 15:

```
i <- 2
repeat {
        print(i)
        i <- i+1
        if(i > 4)
           break
}
[1] 2
[1] 3
[1] 4
```

We can re-write the **f.names** function using a **repeat** loop instead of a **for** loop.

Example 16:

```
names1 <- c("Dave", "John", "Ann", "Roger", "Bill", "Kathy")

f.names.repeat <- function(x)  {
        i <- 1
        repeat {
           print(x[i])
           i <- i+1
```

```
          if(x[i] == "Roger")
               break
        }
}

f.names.repeat(names1)
[1] "Dave"
[1] "John"
[1] "Ann"
```

The names function example above was included to parallel the other names function examples. A more realistic example for the repeat loop is where we are not at all concerned about the number of iteration, instead we would like to keep iteration until we have satisfied a specific criterion. In the following example we have a function which repeatedly draws samples with n=100 from a standard normal distribution. We would like to keep sampling until we have a sample with a mean which is within epsilon of zero. The function allows the user to specify epsilon.

Example 17:

```
random.sample1 <- function(epsilon) {
        i <- 0
        repeat {
          i = i+1
       mean.test <- abs( mean( rnorm(100) ) )
       if (mean.test < epsilon )
            break
               }
        list(mean=mean.test, number.iterations=i)
        }

random.sample1(0.0001)
$mean:
[1] 0.00001373388

$number.iterations:
[1] 6033
```

Ifelse function

The **ifelse** function is very handy because it allows the user to specify the action taken for the test condition being true or false. Like the **if** statement the **ifelse** function can be included in any function or loop.

The basic structure of the ifelse function

```
ifelse(test, action.if.true, action.if.false)
```

Example 18:

```
x <- seq(1:5)
ifelse(x < 3, "T", "F")
[1] "T" "T" "F" "F" "F"
```

Another example where we take the log of a sample with n=10 drawn from a standard normal distribution. Since we cannot take the log of negative numbers we censor all the values less than zero to be zero.
Note that we get an error warning of NA's since the all three components of the **ifelse** function is evaluated for each number in **norm2** even if we do not have any NA's in the list **log.normal**.

Example 19:

```
norm2 <- rnorm(10, mean = 2)
norm2
 [1]  1.5897000  2.1921638  1.1637615  1.3986778  1.4696180 -0.1433373
 [7]  1.6732941  0.3503897  2.2399934  2.2215101

log.normal <- ifelse(norm2 < 0, 0, log(norm2))
log.normal
 [1]  0.4635453  0.7848891  0.1516574  0.3355274  0.3850025  0.0000000
 [7]  0.5147942 -1.0487092  0.8064729  0.7981872
Warning messages:
  NAs generated in: log(x)
```

Passing an unspecified number of parameters to a function

We can pass an unspecified number of parameters to a function by using the ... notation in the argument list. However, the programmer should be careful about the order of the arguments when using the ... notation. Consider the functions **f1** and **f2** in the following example.

```
Example 20:

f1 <- function(x, ...) {
        y <- x+2
        return(y)
        #other commands
}

f2 <- function( ... , x) {
        y <- x+2
        return(y)
        #other commands
}
```

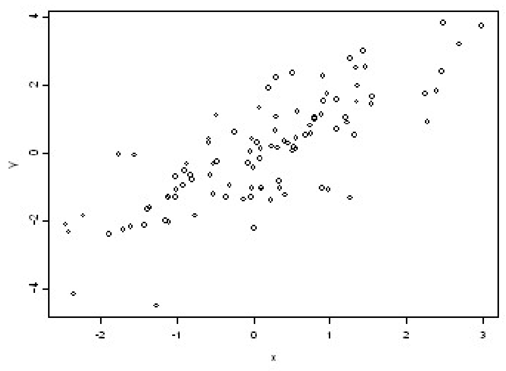
In **f1** we can pass a value for x either by specifying f1(3) or f1(x=3) and we will get the same results. But in **f2** we cannot pass the value for x by specifying f2(3) since f2 will now evaluate 3 as being part of the unspecified parameters. The only way to pass **f2** a value for x is by using the notation f2(x=3).

```
f1(3)
[1] 5

f1(x=3)
[1] 5

f2(3)
Problem in f2(3): argument "x" is missing with no default
Use traceback() to see the call stack

f2(x=3)
[1] 5
```

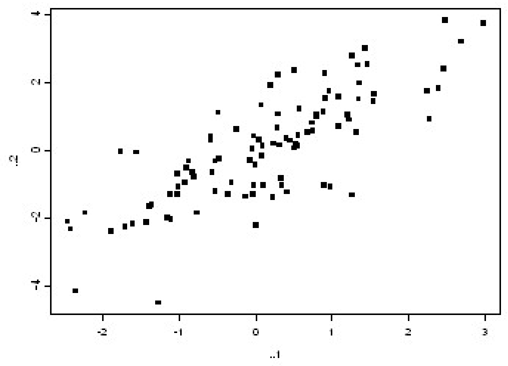
Modifying an already existing function

One of the most common tasks is to modify an existing function by changing only one or a few of the parameters in the existing function. In the following example we change the default symbol for a scatter plot from a diamond to a solid square in a function called **my.plot**

```
Example 21:

x <- rnorm(100)
y <- x + rnorm(100)

plot(x, y)
```



```
my.plot <- function(..., pch.new=15) {
        plot(..., pch=pch.new)
}

my.plot(x, y)
```



How to cite this page

Report an error on this page or leave a comment

IDRE RESEARCH TECHNOLOGY GROUP

High Performance Computing

Statistical Computing

GIS and Visualization

High Performance Computing
Hoffman2 Cluster
Hoffman2 Account Application
Hoffman2 Usage Statistics
UC Grid Portal
UCLA Grid Portal
Shared Cluster & Storage
About IDRE

GIS
Mapshare
Visualization
3D Modeling
Technology Sandbox
Tech Sandbox Access
Data Centers

Statistical Computing
Classes
Conferences
Reading Materials
IDRE Listserv
IDRE Resources
Social Sciences Data Archive

ABOUT CONTACT NEWS EVENTS OUR EXPERTS