

Statistical Computing and Data Visualization in R

Lecture 3

Programming in R

Defining functions

- You can define your own functions in R

```
> logit = function(x){  
+   z = log(x/(1-x))  
+   return(z)}  
> logit(runif(3, 0, 1))  
[1] -4.3336057 -2.7145211 -0.6638038  
> norm = function(x) sqrt(x%*%x)  
> norm(1:4)  
[1] 5.477226
```

- There might be multiple formal arguments, and their value can be predefined

```
> f = function(x, a=1, b=0) a*x^2+b  
> f(2)           #Alternatively, f(x=2)  
[1] 4  
> f(2,2)         #This is equivalent to f(x=2, a=2)  
[1] 8  
> f(2,2,2)  
[1] 10
```

Defining functions

- Sometimes it is useful to return values *invisibly* ...

```
> f1 = function(x) x
> f2 = function(x) invisible(x)
> f1(1)      #Normal behavior
[1] 1
> f2(1)      #Return "invisible" value
> a = f2(2)  #The value is assignable!
> a
[1] 2
```

- To see internal calculations in a function you need to print it ...

```
> f1 = function(x){
+   paste("Attempt 1 to show value of x =", x)
+   print(paste("Attempt 2 to show value of x =", x))
+   return(invisible(0))}
> f1(3)
[1] "Attempt 2 to show value of x = 3"
```

Defining Functions: Argument Matching

- R functions arguments can be matched positionally or by name.

```
> f = function(x, y) {  
>   print(paste("x =", x, "y =", y))  
>   return(invisible(0)) }  
> f(2, 3)  
[1] "x = 2 y = 3"  
> f(x=2, y=3)  
[1] "x = 2 y = 3"  
> f(y=3, x=2)  
[1] "x = 2 y = 3"
```

- You can mix positional matching with matching by name.

```
> f(x=2, 3)  
[1] "x = 2 y = 3"  
> f(y=3, 2)  
[1] "x = 2 y = 3"
```

Defining Functions: Argument Matching

- R does not check for formal parameters in the definition if they are not used by the function (this is called lazy evaluation)

```
> f = function(a, b){  
>   return(a)}  
> f(3)          # No error reported  
[1] 3  
> f(3,2)  
[1] 3
```

- Functions can be passed as arguments to other functions!

```
> a = c(1,2,-1,4,0)  
> summarizex = function(x, ff){  
>   z = ff(x)  
>   return(z)}  
> summarizex(x, mean)  
[1] 1.2  
> summarizex(x, median)  
[1] 1
```

Defining Functions: Scoping

- R allows for free variables in functions (i.e., variables that are not formal arguments or defined inside the function).

```
> b = 4
> f = function(x, y) {
>   z = x + y/b          # Avoid free variables!
>   return(z) }
> f(3,2)
[1] 3.5
```

- R does lexical scoping (i.e., the values of free variables are searched for in the environment in which the function was defined). Important to remember (for example, when functions are defined inside other functions).
- Most often the function is called from the same environment in which it was defined, so lexical scoping and dynamic scoping are identical.

Defining Functions: Scoping

- Free variables might be handy for some purposes, but my recommendation is that you avoid them unless you know exactly what you are doing!
 - If the value of a symbol is not found in the environment in which a function was defined, then the search is continued in the parent environment.
 - After the top-level environment, the search continues down the search list until we hit the empty environment.
 - If a value for a given symbol cannot be found once the empty environment is arrived at, then an error is thrown.
- The `search()` function tells you what the search list is.

Defining Functions: Scoping

- An example of lexical scoping

```
> b = 4
> g = function(x, y){
>   b = 3
>   f = function(x, y){
>     z = x + y/b
>     return(z) }
>   w = f(x,y)
>   return(w) }
> g(3,2)
[1] 3.666667
```

- If R did static scoping the answer would have been 3.5 as before.
- Moving up in search space.

```
> b = 4
> g = function(x, y){
>   f = function(x, y){
>     z = x + y/b
>     return(z) }
>   w = g(x,y)
>   return(w) }
> g(3,2)
[1] 3.5
```


Defining Functions: Argument Matching

- The ... argument indicates a variable number of arguments that are usually passed on to other functions.
 - Example: `plot(x, y, ...)`
- Arguments that appear after ... on the argument list must be named explicitly and cannot be partially matched.

```
> args(paste)
function (... , sep = " ", collapse = NULL)
NULL
> paste("a", "b", " Hello ")
[1] "a b Hello "
> paste("a", "b", sep=" Hello ")
[1] "a Hello b"
```

Defining functions

- If you need to define functions that you regularly use, it is useful to put them in an easily accessible file and “source” the whole file when needed:

```
> source(file="myfunctions.R")
```

- Make sure the file myfunctions.R in your working directory, or add the path for the location of the path in the call

Control structures: Loops

- Loops can be easily constructed in R:

```
> for(i in 1:3){  
+   print(2*i)}  
[1] 2  
[1] 4  
[1] 6
```

- The values the index can take can be a vector of any type:

```
> counties = ("CA", "NE", "OR")  
> for(i in counties){  
+   print(i)}  
[1] "CA"  
[1] "NE"  
[1] "OR"
```

Control structures: Loops

- R does not like zero-length loops!!!

```
> n = 1
> for(i in 1:n){
+   print(i) }
[1] 1
> n = 0
> for(i in 1:n){
+   print(i) }
[1] 1
[1] 0
```

- This is a very common source of problems with your code, and there is no simple solution (except an if statement).

Control structures: Loops

- An alternative to `for` loops is `while` loops. They are executed as long as the condition in the `while` continues to be satisfied.

```
> x = 1
> while(x<10){
+   x = x^2 + 1
+   print(x)}
[1] 2
[1] 5
[1] 26
```

- In a `while` loop, the condition is evaluated before each iteration, i.e., no instruction is ever executed if the condition is not satisfied before the first iteration. If you want the condition to be evaluated after, use a `repeat` loop instead!

Control structures: Loops

- The instructions `break` and `next` can be used to alter the natural flow of a loop.
 - `break` exists the loop immediately, e.g., before all indexes have been iterated on
 - `next` skips the remaining operations in a given iteration.
- Both `break` and `next` apply only to the innermost of nested loops.

Efficient loops

- There are multiple ways to fill a vector using a loop

```
> x = numeric(0)
> for(i in 1:10){
+   x[i] = i}      #Bad (slow)!
>
> x = numeric(0)
> for(i in 1:10){
+   x = c(x,i)}    #Even worse (slower)!
>
> x = numeric(10)
> for(i in 1:10){
+   x[i] = i}      #Better!
```

- Take-home: Memory allocation is very expensive!

Avoiding loops

- For loops in R are *very* slow:

```
> x = rexp(1000000, rate=1)
> logown = function(x) {
+   z = rep(0, length(x))
+   for(i in 1:length(x)) {
+     z[i] = log(x[i])
+   }
+   return(z) }
```

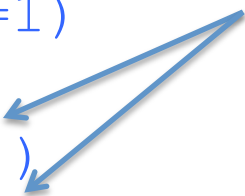
```
> system.time(log(x))
```

user	system	elapsed
0.011	0.001	0.011


```
> system.time(logown(x))
```

user	system	elapsed
1.024	0.008	1.014

Slightly better to store
length(x) in a variable
than to recalculate
multiple times. Try it!



Two orders of
magnitude worse!!!



Avoiding loops

- Loops make sense when each iteration is dependent on the results of previous iterations
- When iterations of the loop are independent, they can be often avoided by using the `apply` family of functions. However they are not necessarily faster than regular for loops.
- Another option is to vectorize your own functions.

Avoiding loops

- The apply function works along any subset of dimensions of an array:

```
> x = matrix(rnorm(6000), nrow=10000, ncol=6)
> apply(x, 2, mean) #Easy to read
[1] -0.004258705 -0.000315459  0.002038574
-0.004258705 -0.000315459  0.002038574
> z = numeric(6)      #"C style", more convoluted
> for(i in 1:6){
+   z[i] = mean(x[,i])
+ }
> z
[1] -0.004258705 -0.000315459  0.002038574
-0.004258705 -0.000315459  0.002038574
```

- A number of “relatives”: lapply, tapply, parapply, sapply, etc ...

Avoiding loops

- Apply can be used with functions that require multiple arguments

```
> x = matrix(rnorm(6000), ncol=3)
> apply(x, 2, quantile, c(0.025,0.975))
```

	[,1]	[,2]	[,3]
2.5%	-1.949164	-1.963706	-1.853970
97.5%	1.982152	1.913266	1.925459

- In these cases, the first argument of apply must correspond to the first argument of the function to be applied (FUN). Other arguments can be appended to apply, in the same order they appear in the definition of FUN.

Avoiding loops

- Apply is cleaner but not necessarily faster than a for loop:

```
> x = matrix(rexp(6000000), ncol=6000)
> loopsum=function(x) {
+   n = dim(x)[2]
+   z=numeric(n)
+   for(i in 1:n){
+     z[i] = mean(x[,i])
+   }
+   return(z)
> system.time(loopsum(x))
   user  system elapsed 
0.108    0.008    0.104 
> system.time(apply(x, 2, sum)) #About the same!
   user  system elapsed 
0.104    0.015    0.108
```

- Also, it might not work for functions with multiple arguments!

Avoiding loops

- For a few row and/or column operations there is a fast alternative (loop built in low-level C)

```
> x = matrix(rexp(6000000), ncol=6000)
```

```
> system.time(apply(x, 2, sum))
```

user	system	elapsed
0.104	0.015	0.108

```
> system.time(colSums(x))
```

user	system	elapsed	
0.007	0.000	0.008	#Much better!

Avoiding loops

- Another common problem is standardization of matrices, i.e., applying the same operation to each column/row, but using a different argument for each (i.e., dividing each row by a different number).

```
> xo = x = matrix(rexp(6000000), ncol=6000)
> system.time(for(i in 1:6000){
>   x[,i] = x[,i]/sum(x[,i]))}
      user  system elapsed
0.231    0.031    0.249
```

- There is a built-in function that can help

```
> system.time(scale(x, center=FALSE,
scale=colSums(x)))
      user  system elapsed
0.157    0.006    0.152
```

Avoiding loops

- Recycling can be used to vectorize complex operations.
- For example we can easily divide each **row** by a different number

```
> S = matrix(1, 2, 2)
> x = c(2, 4)
> S/x
```

```
      [,1] [,2]
[1,] 0.50 0.50
[2,] 0.25 0.25
```

(by default, R fills matrices by column)

- Fast and useful, but I am not a fan because it makes your code very hard to read.

Avoiding loops

- Consider evaluating a function over a two-dimensional grid of arguments:

```
> x = seq(-2,2,length=50)
> y = seq(-2,2,length=50)
> f = function(x,y) x^2 + y^2
> f(x,y) = function(x,y) x^2 + y^2
[1] 8.0000000000 7.360266556 ... #50 values
```

- Standard vectorization evaluates the function only on the diagonal elements of the grid! To get what you want

```
> z = expand.grid(x,y)
> f(z[,1], z[,2])
[1] 8.0000000000 7.680133278 ... #2500 values
```


Avoiding loops

- The vectorized version is faster!

```
> system.time(f(z[,1], z[,2]))
  user  system elapsed 
    0      0      0  #So small, it is rounded down!

> fgrid = function(x,y){
>   n1 = length(x)
>   n2 = length(y)
>   M  = matrix(NA,n1,n2)
>   for(i in 1:n1){
>     for(j in 1:n2){
>       M[i,j] = f(x[i],y[j])}}
>   return(M) }
> system.time(fgrid(x,y))
  user  system elapsed 
0.028  0.001  0.023
```

- An alternative way to vectorize (returns results in matrix form).

```
> system.time(outer(x,y,f))
  user  system elapsed 
    0      0      0
```

Avoiding loops

- Another function to keep in mind is `Vectorize`.
- I personally have had to use `Vectorize` just a couple of times in my life (to vectorize functions that contain complex internal conditional statements, see next few slides), but it is useful to know that it exists.

Conditional statements

- Conditional statements allow you execute certain functions only if a given condition is satisfied.

```
> x = 7
> if(x <= 10){
>   print("Less or equal to 10!")}
> else{
>   print("Greater than 10!")}
[1] "Less or equal to 10!"
```

- The **and** operator in R is **&** and the **or** operator is **|** .

```
> x = 7
> if(x <= 10 & x >5){
>   print("Between 5 and 10!")}
[1] "Between 5 and 10!"
```

Conditional statements

- If/then/else statements are not vectorized!

```
> x = c(1,3,-1)
> if(x < 2){
>   print("Hello")}
> else{
>   print("Bye")}
[1] "Hello"
```

Warning message:

```
In if (x < 2) { :
  the condition has length > 1 and only the
  first element will be used
```

- The `ifelse` function provides a way to vectorize some conditionals

```
> ifelse(x < 2, "Hello", "Bye")
[1] "Hello" "Bye"   "Hello"
```

Conditional statements

- `ifelse` function can be nested!

```
> ifelse(x < 2, ifelse(x < 0, "Hello", ""),  
"Bye")  
[1] ""      "Bye"    "Hello"
```

- There are other forms of vectorizing conditional statements (e.g., by using logical vectors)

```
> z = c("Bye", "Hello")  
> z[as.numeric(x<2)+1]  
[1] "Hello" "Bye"    "Hello"
```

- Sometimes you will need to have regular if/then/else loops inside a for/while/repeat loop!

Fun with Functions: Integration

- R can compute integrals of functions of one variable using adaptive quadrature:

```
> x = integrate(sin,0,pi) #Integral of sin(x)
in [0,pi]
> x
2 with absolute error < 2.2e-14
> names(x)
[1] "value"          "abs.error"
"subdivisions" "message"      "call"
> x$value
[1] 2
```

- You can use it on your own function:

```
> f = function(x) x^2 + 1
> integrate(f,-2,2)
9.333333 with absolute error < 1e-13
```

Fun with Functions: Integration

- The function might depend on multiple arguments, but you integrate only over the first and need to pass values for the other to integrate!

```
> f = function(x,b) x^2 + b  
> integrate(f,-2,2,1) ← This is the value of b  
9.333333 with absolute error < 1e-13
```

- The function to be integrated must accept a vector of inputs and produce a vector of function evaluations at those points.

```
> f = function(x) 2.0  
> integrate(f,0,1)  
Error in integrate(f, 0, 1) :  
  evaluation of function gave a result of wrong  
  length  
> integrate(Vectorize(f),0,1)  
2 with absolute error < 2.2e-14
```

Fun with Functions: Integration

- It works for a finite or infinite interval.

```
> g = function(x) {1/((x+1)*sqrt(x))}  
> integrate(g, 0, Inf)  
3.141593 with absolute error < 2.7e-05
```

- If the interval is infinite, state so explicitly!

```
> integrate(g, lower = 0, upper = 1000000, stop.on.error =  
FALSE)  
failed with message 'the integral is probably divergent'  
> integrate(dnorm, 0, 20)  
0.5 with absolute error < 3.7e-05  
> integrate(dnorm, 0, 200)  
0.5 with absolute error < 1.6e-07  
> integrate(dnorm, 0, 2000)  
0.5 with absolute error < 4.4e-06  
> integrate(dnorm, 0, 20000)  
0 with absolute error < 0  
> integrate(dnorm, 0, Inf)  
0.5 with absolute error < 4.7e-05
```


Fun with Functions: Optimization

- For univariate functions, the function optimize is highly efficient

```
> f = function(x,a) (x-a)^2
> optimize(f, c(0,1), 1/3)
$minimum
[1] 0.3333333
$objective
[1] 0
```

- You cannot explicitly use infinity here!

```
> optimize(f, c(-Inf, Inf), 1/3)
Error in optimize(f, c(-Inf, Inf), 1/3) : invalid
'xmin' value
```

- The default is minimization, but you can also maximize. Solutions close to the boundaries can be somewhat inaccurate

```
> optimize(f, c(0,1), 1/3, maximum=TRUE)
$maximum
[1] 0.9999339      #True value is 1.0
$objective
[1] 0.4443563
```

Fun with Functions: Optimization

- For multivariate functions, you can use `optim`

```
> f = function(x,a) (x[1]-a[1])^2 + (x[2]-a[2])^2 #Not
vectorized!!
> x0 = c(0,0)
> a = c(1,1)
> optim (x0, f, gr = NULL, a)
$par
[1] 0.9999874 1.0000754

$value
[1] 5.840565e-09

$counts
function gradient
      67      NA

$convergence
[1] 0

$message
NULL
```

Fun with Functions: Optimization

- By default, `optim` minimizes
- `optim` implements a number of different methods:
 - The default is an implementation of Nelder and Mead (1965), that uses only function values and is robust but relatively slow.
 - For continuous functions you should use conjugate gradient (“CG”) or a quasi-Newton method (“BFGS”). CG is good for large problems, but slightly more fragile. If no gradient is provided, it is numerically approximated.
 - Other options, including simulated annealing.

Fun with Functions: Optimization

- `optim` is useful to compute maximum likelihood estimators:

```
> negloglik = function(theta, y){  
+   z = -sum(dweibull(y, shape=theta[1], scale=theta[2],  
log=TRUE))  
+   return(z)}  
> y = rweibull(100, shape=2, scale=1)  
> x0 = c(1.5,1.5)  
> optim(x0, negloglik, gr = NULL, y, method="BFGS")  
$par  
[1] 1.909326 1.014343  
  
$value  
[1] 63.43306  
  
$counts  
function gradient  
      28      9  
  
$convergence  
[1] 0  
  
$message  
NULL
```

Fun with Functions: Roots

- Finding the roots of *univariate* functions is relatively simple

```
> f = function(x, a) sin(a*x)-1/2  
> uniroot(f, interval=c(0,1), a=2)
```

```
$root
```

```
[1] 0.2618112
```

```
$f.root
```

```
[1] 2.049336e-05
```

```
$iter
```

```
[1] 5
```

```
$init.it
```

```
[1] NA
```

```
$estim.prec
```

```
[1] 6.103516e-05
```

Fun with Functions: Roots

- Only one root is returned when multiple are present

```
> f = function(x, a) sin(a*x)-1/2  
> uniroot(f, interval=c(0,4), a=2)
```

```
$root
```

```
[1] 3.403386
```

```
$f.root
```

```
[1] -1.10201e-05
```

```
$iter
```

```
[1] 6
```

```
$init.it
```

```
[1] NA
```

```
$estim.prec
```

```
[1] 6.103516e-05
```

- No easy way to control where the algorithm starts, so no control over which root is reported!

Fun with Functions: Roots

- When the function of interest is a polynomial, there is a specialized function that can report *all* roots

```
> z = c(-1,1,2)      #Coefficients
                        of  $2x^2 + x - 1$ 

> polyroot(z)
[1] 0.5-0i -1.0+0i

> z = c(1,-2,-1,2)    #Coefficients
                        of  $2x^3 - x^2 - 2x + 1$ 

> polyroot(z)
[1] 0.5+0i -1.0-0i 1.0-0i
```

- Note that polyroot always reports complex numbers
- No built-in function for solving non-linear systems of equations.

Fun with Functions: checking formal arguments

- Function execution might lead to different conditions:
 - Message: does not stop execution.
 - Warning: does not stop execution.
 - Error: **stops execution.**
- An example of a warning:

```
> log(-1)
```

```
[1] NaN
```

```
Warning message:
```

```
In log(-1) : NaNs produced
```


Fun with Functions: checking formal arguments

- An example of an error:

```
> printmessage = function(x){  
>   if(x > 1)  
>     print("x greater than 1")  
>   else  
>     print("x less or equal to 1")  
>   return(invisible(x))}  
> printmessage(2)  
[1] "x greater than 1"  
> printmessage(NA)  
Error in if (x > 1) print("x greater than 1")  
else print("x less or equal to 1") :  
  missing value where TRUE/FALSE needed
```

- Can you figure out the reason for the error?

Fun with Functions: checking formal arguments

- Dealing with the error:

```
> printmessage2 = function(x){  
>   if(is.na(x))  
>     warning("x is a missing value")  
>   else if(x > 1)  
>     print("x greater than 1")  
>   else  
>     print("x less or equal to 1")  
>   return(invisible(x))}  
> printmessage(2)  
[1] "x greater than 1"  
> printmessage(NA)  
Warning message:  
In printmessage(NA) : x is a missing value
```

Fun with Functions: checking formal arguments

- A different warning:

```
> printmessage3 = function(x) {  
>   if(is.na(x))  
>     warning("x is a missing value")  
>   else if(x > 1)  
>     print("x greater than 1")  
>   else  
>     print("x less or equal to 1")  
>   return(invisible(x)) }  
> x = c(NaN, 2)  
> printmessage(x)  
[1] "x is a missing value"
```

Warning message:

In if (is.na(x)) print("x is a missing value") else if
(x > 1) print("x greater than 1") else print("x less or
equal to 1") :

the condition has length > 1 and only the first
element will be used

Fun with Functions: checking formal arguments

- Dealing with the warning (not vectorized!):

```
> printmessage3 = function(x) {  
>   if(length(x) > 1)  
>     stop("x has length greater than 1")  
>   if(is.na(x))  
>     warning("x is a missing value")  
>   else if(x > 1)  
>     print("x greater than 1")  
>   else  
>     print("x less or equal to 1")  
>   return(invisible(x)) }  
> x = c(NaN, 2)  
> printmessage(x)  
Error in printmessage(x) : x has length greater  
than 1
```