

R lesson 2 Basics of R: Programming

Conditioning
We have already seen some basic some form of conditioning in lesson 1 when indexing vectors \((x[x > 3])\). Comparison operators are:

- equal: `==`
- not equal: `!=`
- (strictly) greater: `>`, `>=`
- (strictly) less: `<`, `<=`

Logical operators are:

- and: `&` or `&&`
- or: `|` or `||`

These statements return a TRUE/FALSE which can be used for indexing

```r
> x=1:8
> x>3
[1] FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE
> x[x>3]
[1] 4 5 6 7 8
> x[(x>3) & (x<7)]
[1] 4 5 6
> x[(x>3) | (x<7)]
[1] 1 2 3 4 5 6 7 8
```

and for conditioning if/else statements. There are two ways to implement an if/else statement.

```r
> y=2
> ifelse(y==2,c("do this"),c("do that"))
[1] "do this"
> if(y==2){
+ c("do this")
+ }else{
+ c("do that")
+ }
[1] "do this"
```

Obviously there are occasions where an else statement is not required.

```r
> y=5
> if(y==2) y=c("now y is a string")
> y
[1] 5
```
Loops

The most commonly used loop structures in R are `for`, `while` and `apply` loops. Less common are `repeat` loops. The `break` function is used to break out of a loop and control transferred to the first statement outside that (inner-most) loop, and `next` halts the processing of the current iteration and advances the looping index.

Remark: `break` can be interchanged with `stop` that halts the current expression and returns an error message when there are no nested loops.

for

A (redundant) trivial example is:

```R
> for(i in 1:4) {
+   x[i]=0
+ }
> x
[1] 0 0 0 0 5 6 7 8
```

which obviously could more efficiently be done with `x[1:4]=1`. In this example we indexed `i = 1, 2, 3, 4`. However you can use any index:

```R
> x=1:8
> for(i in seq(3,7,2)) {
+   x[i]=0
+ }
> x
[1] 1 2 0 4 0 6 0 8
```

Next we illustrate the `break/stop` and `next` functions

```R
> z=NULL
> for(i in 1:10) {
+   if(i<5)z=i
+   if(i==5)stop("It is so fluffy")
+   if(i>5)z=i
+ }
Error: It is so fluffy
> z
[1] 4
```

```R
> z=rep(0,10)
> for(i in 1:10) {
+   if(i<5)z[i]=i
+   if(i==5)next
+   if(i>5)z[i]=i
+ }
> z
[1] 1 2 3 4 0 6 7 8 9 10
```
while

The while loop is used to execute expressions while a condition is true. It is commonly used in optimization algorithms until a certain tolerance level is obtained. Start with a value of 4 and add increments of 0.005 until a tolerance of less than 0.01 is reached.

```r
> x=4
> target=5
> k=1 # track number or iterations
> while((target-x)>0.01){
+     x=x+0.005
+     k=k+1
+ }
> x
[1] 4.995
> k
[1] 200
```

apply and its variations

The function `apply` is used to apply a function on certain dimensions of a matrix, array, data frame and any other structure of that form. We will focus on `apply` but other variations exist which we will briefly introduce.

As with most functions that take in as an argument other functions, it allows us to parse arguments for the input function.

```r
> M=array(rnorm(24),c(3,4,2),
+     dimnames=list(c("R1","R2","R3"),c("C1","C2","C3","C4"),c("D1","D2")))
> M
, , D1
   C1      C2      C3      C4
R1 0.5279391 0.6239913 -0.3019933 -0.6909477
R2 -0.5276262 1.5810012  0.6785096  0.7963933
R3  1.8838153 -0.7832629 -0.5741983 -0.5427939
, , D2
   C1      C2      C3      C4
R1 1.1449504 0.6239913 -0.3019933 -0.6909477
R2 -0.2516103 1.5810012  0.6785096  0.7963933
R3  0.2270848  0.6239913 -0.3019933 -0.6909477
> apply(M,2,mean,trim=0.05)
   C1     C2     C3     C4
0.5007588 -0.7408722  0.7937023  0.2078604
> apply(M,c(2,3),mean)
   C1     C2     C3     C4
0.5007588 -0.7408722  0.7937023  0.2078604
```

3
## Writing functions

Nearly everything you need to know can be explained with a couple of simple examples. First, begin with a function that takes a vector, performs a linear transformation $ax + b$ and calculates the mean. Its arguments are:

- **n** sample size that must be specified
- **a=1, b=0** mean and st.dev. that have default values if unspecified
- **...** (optional) additional arguments for any interior functions (e.g. for **mean**)

and it always returns the last object called.

```r
> set.seed(7363); x=rnorm(20)
> lt.mean = function(vec,a=1,b=0,...){
+   m = mean((a*vec+b),...)
+   m
+ }
> lt.mean(x)
[1] -0.201840597
> lt.mean(x,2,3)
[1] 2.596318806
> lt.mean(x,a=2,b=3)
[1] 2.674232821
```

The second time we run the function we specified $a = 2, b = 3$. If you are unsure of the order of each argument you may just as well run

```r
> lt.mean(vec=x,a=2,b=3)
[1] 2.596318806
> lt.mean(b=3,vec=x,a=2)
[1] 2.596318806
```

Finally we if we wish to calculate the 95% trimmed mean we simply need to parse the trim argument to the **mean** function (done via ...).

```r
> lt.mean(x,a=2,b=3,trim=0.05)
[1] 2.674232821
```
Remark: It is a good idea to add some statements that check if the inputted arguments are of the desired type/form.

What if we wish to output more than just one object where those objects are of different type? For example, let’s take the same vector and create a 4 × 5 matrix of the transformed values in addition to calculating the mean. The usual way is outputting a list.

```r
> lt.mm = function(vec,a=1,b=0,...){
+   y=a*vec+b
+   my=mean(y,...)
+   mat=matrix(my,4,5,byrow=TRUE)
+   list(mean=my,matrix=mat)
+ }
> out=lt.mm(x);out

$mean
 [1] -0.201840597

$matrix
[1,] -0.201840597 -0.201840597 -0.201840597 -0.201840597 -0.201840597
[2,] -0.201840597 -0.201840597 -0.201840597 -0.201840597 -0.201840597
[3,] -0.201840597 -0.201840597 -0.201840597 -0.201840597 -0.201840597
[4,] -0.201840597 -0.201840597 -0.201840597 -0.201840597 -0.201840597

> is.list(out)
[1] TRUE

> out[[1]]
[1] -0.201840597

> out$mean
[1] -0.201840597
```