Class2_Cont

Dr. Pacifique

2025-10-05

Control structures

Grouping

Control structures in R allow you to control the flow of execution of a series of R expressions. Basically, control structures allow you to put some "logic" into your R code, rather than just always executing the same R code every time. Control structures allow you to respond to inputs or to features of the data and execute different R expressions accordingly. Commonly used control structures are • if and else: testing a condition and acting on it • for: execute a loop a fixed number of times • while: execute a loop while a condition is true • repeat: execute an infinite loop (must break out of it to stop) • break: break the execution of a loop • next: skip an interaction of a loop

if (condition){ ## do something } ## continue with the rest of the code.

if(condition){ do something

} else { do something else

}

You can also create a series of test by following the initial if with a number of else ifs

if(condition){ do something

}else if (condition 2){ Do something different }else{ do something different }

Function on R

```
f<-function(x) x^2
formals(f)

## $x

body(f)

## x^2

environment(f)

## <environment: R_GlobalEnv>
```

ifelse(test,yes,No)

```
pvalues<-c(.867,0.0054,0.0018,0.1572,0.0183,0.5386)
results<-ifelse(pvalues<0.05,"Significant"," Not significant")
results
```

```
## [1] " Not significant" "Significant"      "Significant"      " Not s
ignificant"
## [5] "Significant"      " Not significant"
```

```
x<-runif(1,0,10)
if(x>3){
  y<-10
}else {
  y<-0
}
```

The value of y is a set depending on whether x>3 or not. This can also be achieved by

```
y<-if (x>3){
  10
}else {
  0
}
```

####For

```
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
```

```
x<-c("a","b","c","d")

x[3]
```

```
## [1] "c"
```

```
for(i in 1:5){
  print(x[i])
}
```

```
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
## [1] NA
```

```
for(i in 1:5)print(1:i)
```

```
## [1] 1
## [1] 1 2
## [1] 1 2 3
## [1] 1 2 3 4
## [1] 1 2 3 4 5
```

```
for(i in 5:1)print(1:i)
```

```
## [1] 1 2 3 4 5
## [1] 1 2 3 4
## [1] 1 2 3
## [1] 1 2
## [1] 1
```

while Loops

It begins by testing a condition, if it is true, then they execute the loop body. once the loop body is executed, the condition is tested again, until the condition is false. after which the loop exits.

```
count<-1
while (count<10){
  print(count)
  count<-count+1
}
```

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

```
z<-5
set.seed(1)
while(z>= 3 && z<=10){
  coin<-rbinom(1,1,0.5)
  if (coin==1){
    z=z+1
  } else {
```

```
    z<-z-1
  }
}
print(z)

## [1] 2
```

next, break

This is used to skip an iteration of a loop

```
for (i in 1:100){
  print(1:i)
  if (i>20){
    break
}
}

## [1] 1
## [1] 1 2
## [1] 1 2 3
## [1] 1 2 3 4
## [1] 1 2 3 4 5
## [1] 1 2 3 4 5 6
## [1] 1 2 3 4 5 6 7
## [1] 1 2 3 4 5 6 7 8
## [1] 1 2 3 4 5 6 7 8 9
##  [1]  1  2  3  4  5  6  7  8  9 10
##  [1]  1  2  3  4  5  6  7  8  9 10 11
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21
```

Function

functionname<-function(parameters){ statements return(value) }

f1<-function(x){ result<-x^2+2 return(result) }

f2<-function(x,y){ result<-x^{2+y}2-4 return(result) }

```
f<-function(x,y){
  result<-x+(2*y)+3
  return(result)
}
```

```
f(2,3)
```

```
## [1] 11
```

```
f(2,3)
```

```
## [1] 11
```

You can use args() function to view the parameter names and default values

Exercises

Make functions that calculate summary statistics

Make a function to calculate two sample t test

Applying functions to matrices and data frame.

```
a<-4
sqrt(4)
```

```
## [1] 2
```

```
b<- c(1,243,5.754,2.987)
round(b)
```

```
## [1]   1 243   6   3
```

```
c<-matrix(runif(12),nrow=3)
c
```

```
##           [,1]      [,2]      [,3]      [,4]
## [1,] 0.5995658 0.8273733 0.1079436 0.8209463
## [2,] 0.4935413 0.6684667 0.7237109 0.6470602
## [3,] 0.1862176 0.7942399 0.4112744 0.7829328
```

```
log(c)
```

```
##            [,1]       [,2]       [,3]       [,4]
## [1,] -0.5115495 -0.1894993 -2.2261462 -0.1972976
## [2,] -0.7061487 -0.4027686 -0.3233632 -0.4353160
## [3,] -1.6808394 -0.2303698 -0.8884946 -0.2447085
```

```
mean(c)
```

```
## [1] 0.5886061
```

Notice that the mean of matrix c results in a scalar (0.444). the mean() take the average of all 12 elements in the matrix.But what if you want the three row means or the four column means?

R provides a function, apply() that allows to apply an arbitrary function to any dimension of a matrix, array or data frame. The format for the apply() function is -

apply(x, MARGIN, FUN,...) where x stands for the data object, Margin can be 1(rows) and 2(columns) ## Col/Row Sums and Means

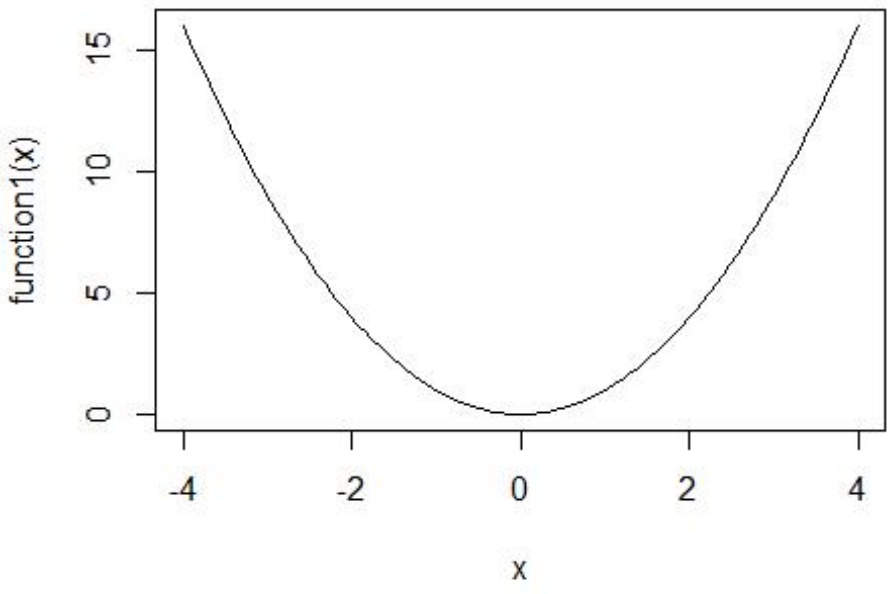
• rowSums = apply(x, 1, sum) • rowMeans = apply(x, 1, mean) • colSums = apply(x, 2, sum) • colMeans = apply(x, 2, mean)

lappy function

lapply() takes three inputs:,x,a list,a function, and..., It applies to each element of the list and returns a new list. lapply(x,f,...). It is called function because it takes function as an argument. Assume we have a data frame df. instead of assigning the result pf lappliy() to df, we will assign them to df[] to ensure we get a data frame.

```
fix_missing<-function(x){
  x[x==-99]<-NA
  x
}
function1<-function(x){
  x^2
}

curve(function1,-4,4)
```




```
#df[]<-lappy(df,fix_missing)
```

It works for any number of columns.There is no way to accidentally miss a column

There is no way to accidentally treat one column differently than another

It easy to generalize this technique to a subset of columns

```
#df[1:5]<- lapply(df[1:5],fix_missing )
```

sapply

sapply() and vapply(), variants of lapply() that produces vectors,matrices, and arrays as Output, instead of lists map(), and mapply which iterate over multiple input data structures in parallel

Another important function when dealing with big data is split() -tapply

Titanic data

```
str(Titanic)
```

```
##  'table' num [1:4, 1:2, 1:2, 1:2] 0 0 35 0 0 0 17 0 118 154 ...
##  - attr(*, "dimnames")=List of 4
##   ..$ Class   : chr [1:4] "1st" "2nd" "3rd" "Crew"
##   ..$ Sex     : chr [1:2] "Male" "Female"
##   ..$ Age     : chr [1:2] "Child" "Adult"
##   ..$ Survived: chr [1:2] "No" "Yes"
```

```
View(Titanic)
apply(Titanic,c(1,2),sum )
```

```
##       Sex
## Class  Male Female
##   1st   180    145
##   2nd   179    106
##   3rd   510    196
##   Crew  862     23
```

```
options(digits = 2)
apply(Titanic,c(1,2),sum)[3:4,]
```

```
##       Sex
## Class  Male Female
##   3rd   510    196
##   Crew  862     23
```

```
apply(Titanic,c(1,4),sum)[3:4,]
```

```
##       Survived
## Class   No Yes
##   3rd  528 178
##   Crew 673 212
```

```
apply(Titanic,c(1,2,4),sum)
```

```
## , , Survived = No
##
```

```
##       Sex
## Class  Male Female
##   1st   118      4
##   2nd   154     13
##   3rd   422    106
##   Crew  670      3
##
## , , Survived = Yes
##
##       Sex
## Class  Male Female
##   1st    62    141
##   2nd    25     93
##   3rd    88     90
##   Crew  192     20
```

```
#apply(Titanic,c(1,2,4),sum)[3:4,]
apply(Titanic,c(1,2,4),sum)[3:4,,]
```

```
## , , Survived = No
##
##       Sex
## Class  Male Female
##   3rd   422    106
##   Crew  670      3
##
## , , Survived = Yes
##
##       Sex
## Class  Male Female
##   3rd    88     90
##   Crew  192     20
```

```
ftable(apply(Titanic,c(1,4,2),sum)[3:4,,])
```

```
##                Sex Male Female
## Class Survived
## 3rd   No            422    106
##       Yes            88     90
## Crew  No            670      3
##       Yes           192     20
```

```
ftable(apply(Titanic,c(1,4,2),sum)[3:4,,])[1:2,]
```

```
##      [,1] [,2]
## [1,]  422  106
## [2,]   88   90
```

```
ftable(apply(Titanic,c(1,4,2),sum)[3:4,,])[3:4,]
```

```
##      [,1] [,2]
## [1,]  670    3
## [2,]  192   20
```

```
digit=2
prop.table(ftable(apply(Titanic,c(1,4,2),sum)[3:4,,])[1:2,], margin = 2)
```

```
##      [,1] [,2]
## [1,] 0.83 0.54
## [2,] 0.17 0.46
```

```
prop.table(ftable(apply(Titanic,c(1,4,2),sum)[3:4,,])[3:4,], margin = 2)
```

```
##      [,1] [,2]
## [1,] 0.78 0.13
## [2,] 0.22 0.87
```

```
matrix(prop.table(ftable(apply(Titanic,c(1,4,2),sum)[3:4,,])[1:2,], mar
gin = 2),nrow = 2, dimnames = list(dimnames(Titanic)$Survived,dimnames
(Titanic)$Sex))
```

```
##     Male Female
## No  0.83   0.54
## Yes 0.17   0.46
```

```
matrix(prop.table(ftable(apply(Titanic,c(1,4,2),sum)[3:4,,])[3:4,], mar
gin = 2),nrow = 2, dimnames = list(dimnames(Titanic)$Survived,dimnames
(Titanic)$Sex))
```

```
##     Male Female
## No  0.78   0.13
## Yes 0.22   0.87
```

```
### Create dataset
Student<-c("John ncuti", "Angela bakame", "Bruce wizeye","Alexis aganze
", "claude Rukundo", "Joel Kagabo", "Mary ineza")
Math<-c(600,412,358,495,512,410,522)
Science<-c(95,99,80,82,75,89,77)
English<-c(25,22,18,20,29,30,27)
roster<-data.frame(Student,Math,Science,English,stringsAsFactors = FALS
E)

### standardize variables and obtains the performance scores because th
ey are reported on different scale( With widely differing means and sta
ndard deviations, we need to make them comparable before we combine the
m.)
z<-scale(roster[,2:4])

### performance of each students using  rowmeans and adding them to ros
ter using cbind()
score<-apply(z,1,mean)
roster<-cbind(roster,score)
```

```
### Grades  the students: quantile function gives the percentile rank o
f each student's performance score check the cutoff of A
y<-quantile(score,c(.8,.6,.4,.2))
## create a grade variable us
roster$grade[score>=y[1]]<-"A"
roster$grade[score<y[1]&score>=y[2]]<-"B"
roster$grade[score<y[2]& score>=y[3]]<-"C"
roster$grade[score<y[3]& score>=y[4]]<-"D"
roster$grade[score<y[4]]<-"F"
#### Dealing with names
name<-strsplit((roster$Student),"")
lastname<-sapply (name,"[",2)
firstname<-sapply(name,"[",1)
roster<-cbind(firstname, lastname,roster[,-1])
roster<-roster[order(lastname,firstname),]
roster
```

```
##   firstname lastname Math Science English  score grade
## 7         M        a  522      77      27  0.085     C
## 4         A        l  495      82      20 -0.352     F
## 5         c        l  512      75      29  0.118     B
## 2         A        n  412      99      22  0.076     D
## 1         J        o  600      95      25  0.904     A
## 6         J        o  410      89      30  0.289     A
## 3         B        r  358      80      18 -1.119     F
```

Aggregation and reshaping

transpose

```
cars<-mtcars[1:5, 1:4]
cars
```

```
##                   mpg cyl disp  hp
## Mazda RX4          21   6  160 110
## Mazda RX4 Wag      21   6  160 110
## Datsun 710         23   4  108  93
## Hornet 4 Drive     21   6  258 110
## Hornet Sportabout  19   8  360 175
```

```
t(cars)
```

```
##      Mazda RX4 Mazda RX4 Wag Datsun 710 Hornet 4 Drive Hornet Sporta
bout
## mpg         21            21         23             21
  19
## cyl          6             6          4              6
   8
## disp       160           160        108            258
 360
## hp         110           110         93            110
 175
```

aggregate data

aggregate() collapse data in R using one or more by variables and a defined function

```
options(digits=3)
attach(mtcars)
aggdata<-aggregate(mtcars,by=list(cyl,gear),FUN=mean,na.rm=TRUE)
```

Reshape

Step 1: install the package reshape2 step 2: melt data step 3: Cast the melted data into any shape you desire

During the cast, you can aggregate the data with any function you wish.

```
id<-c(1,1,2,2)
time<-c(1,2,1,2)
x1<-c(5,3,6,2)
x2<-c(6,5,1,4)
mydata<-data.frame(id,time,x1,x2)
```