# Part V: Repeatedly carrying out the same set of instructions

In the previous parts, our focus has been on working through the workflow (Figure 5.0) linearly. That is, each operation is performed only once and then the program moves to the next operation. In this part, we start using R to repeatedly carry out one or more operations. Knowing some coding to handle this kind of automation is of course a clear advantage as you will otherwise find yourself repeating the same procedure many times, with clear risk for error.

## Chapter 12. Coding for loops

A for loop is a classic programming structure to repeatedly carry out the same set of instructions. In this chapter we take a look at this feature.

### 12.1 Initialization: data.frame

Before we start coding for loops, we cover how to construct an empty object and then fill it, one part at a time.

Suppose we wish to store information on a variable v1 in a data.frame called dat.

> dat$v1<-1

Error in dat$v1 <- 1 : object ‘dat’ not found

R complains: dat is not defined. You therefore first need to define dat. For example, as an empty data.frame.

> dat=data.frame()

> dat

data frame with 0 columns and 0 rows

> rbind(dat,data.frame(number=c(1,2,3),name=c(“Bob”,”Bob”,”Harriet”)))

number name

1 1 Bob

2 2 Bob

3 3 Harriet

The above tells R that dat is an empty data.frame. You can then use, as before, the functions rbind() or cbind() to add data to this empty data.frame. We can then continue expanding the data.frame dat. Remember that only a data.frame with the correct tags can be added to a data.frame (for more on data.frame see chapter 5.2.4).

> dat<-rbind(dat,data.frame(number=c(4,5,6),name=c("Jim","Alice","Bob")))

> dat

number name

1 1 Bob

2 2 Bob

3 3 Harriet

4 4 Jim

5 5 Alice

6 6 Bob

### 12.2 Initialization: list

We have encountered a list when using tapply() and in improving the axes of barplot() and plot() in chapter 8. A list is very flexible in storing information, even more flexible than a data.frame. It can contain various pieces of information. For example, a vector, matrix and data.frame

> myList<-list(

+ myVec=c(1,2,3)

+ , myMat=matrix(c(1,2,3,4),2,2)

+ , myDf=data.frame(students=c("Bill","Miranda","Laura"), ages=c(22,21,19)))

> myList

$myVec

[1] 1 2 3

$myMat

[,1] [,2]

[1,] 1 3

[2,] 2 4

$myDf

students ages

1 Bill 22

2 Miranda 21

3 Laura 19

> str(myList)

List of 3

$ myVec: num [1:3] 1 2 3

$ myMat: num [1:2, 1:2] 1 2 3 4

$ myDf :'data.frame': 3 obs. of 2 variables:

..$ students: chr [1:3] "Bill" "Miranda" "Laura"

..$ ages : num [1:3] 22 21 19

We can index a list by using $ and the tag name or by using double square brackets [[ ]] instead of single square brackets []), but otherwise following the same logic as for other classes

> myList$myMat

[,1] [,2]

[1,] 1 3

[2,] 2 4

> myList[[”myMat”]]

[,1] [,2]

[1,] 1 3

[2,] 2 4

> myList[[2]]

[,1] [,2]

[1,] 1 3

[2,] 2 4

To initialize, you may specify an empty list and fill this list with your information. Afterwards the list can be coerced to a data.frame. For example,

> #empty list

> dat2=list(number=NULL,name=NULL)

> dat2

$v1

NULL

$v2

NULL

> #and add

> dat2$number=c(1,2,3)

> dat2$name=c("Bob","Bob","Harriet")

> dat2

$number

[1] 1 2 3

$name

[1] "Bob" "Bob" "Harriet"

> #and expand

> dat2$number=c(dat2$number,c(4,5,6))

> dat2$name=c(dat2$name, c("Jim","Alice","Bob"))

> dat2

$number

[1] 1 2 3 4 5 6

$name

[1] "Bob" "Bob" "Harriet" "Jim" "Alice" "Bob"

The above uses NULL (chapter 1.5) to define dat2$number and dat2$name as elements in a list that are empty. After defining, information can be added. The list dat2 can be easily coerced into a data.frame

> #make into data.frame

> dat2<-data.frame(dat2)

> dat2

number name

1 1 Bob

2 2 Bob

3 3 Harriet

4 4 Jim

5 5 Alice

6 6 Bob

In chapters 12.1 and 12.2, two possible variants are demonstrated to initialize and add information sequentially, but there are other approaches. It depends on what you want to do which option to use.

### 12.3 Loop

Often you want to repeat the same operation a number of times. The classical programming structure for doing this is a so-called for loop. The coding of a for loop is here approached in small steps. In general, this is how you should approach coding such complicated procedures.

Let’s assume we have data on 2 sites

> dat.sites=data.frame(site=c("A1","A1","A2","A2","A2"),x=c(2,3,3,4,3),y=c(1,12,3,8,9))

> dat.sites

site x y

1 A1 2 1

2 A1 3 12

3 A2 3 3

4 A2 4 8

5 A2 3 9

The objective is to produce a data.frame that holds the mean for each variable for each site using a for() loop. In general, a for loop will repeat the same instructions for as many times as is specified. To work ourselves towards the for loop, it is clever to start by coding one step at a time. The first step is to figure out how to perform the operation we want to code for the first site in a way that is as general as possible.

Let’s start by creating a vector that contains all the sites

> sites<-c(“A1”,”A2”)

[1] "A1" "A2"

Then, we work out how to handle the first site. To filter the data only for the first site, subset() can be used

> MySubset<-subset(dat.sites,site==sites[1]) #only date on the 1-th site

> MySubset

site x y

1 A1 2 1

2 A1 3 12

and the mean can be calculated for each variable

> mean(MySubset$x)

[1] 2.5

> mean(MySubset$y)

[1] 6.5

and all the above can of course be entered into a data.frame

> data.frame(site=sites[1]

+ ,mean.x=mean(MySubset$x)

+ , mean.y=mean(MySubset$y))

site mean.x mean.y

1 A1 2.5 6.5

The above is coded sufficiently general to repeat for the second site, only by changing one number. Below only the indexing of sites is changed from 1 to 2.

> #for first site

> MySubset<-subset(dat.sites,site==sites[1]) #only date for the 1st site

> data.frame(site=sites[1] #only date for the 1st site

+ ,mean.x=mean(MySubset$x)

+ , mean.y=mean(MySubset$y))

site mean.x mean.y

1 A1 2.5 6.5

> # we can repeat this procedure for the second site just by changing one number

> MySubset<-subset(dat.sites,site==sites[2]) #only date on the 2-nd site

> data.frame(site=sites[2] #only date on the 2-nd site

+ ,mean.x=mean(MySubset$x)

+ , mean.y=mean(MySubset$y))

site mean.x mean.y

1 A2 3.333333 6.666667

Now we are ready for the for loop! A for loop for the above coded case has the syntax

for (i in 1:2) {CODE}

The loop will re-run the code written between the curly brackets ({CODE}) for as many times as is specified in the for command. For the above, there are 2 sites, which are given as the vector 1:2 and the index, arbitrarily called i is said to be “in” this vector. As a consequence, in the first “round” of the for loop, i is 1, and in the second “round” i will be 2. After that, the for loop is ready and the script continues after the closing curly bracket }. The code developed above already does this. It can be coded explicitly using the index as

> # -- first round of "loop"

> i=1

> MySubset<-subset(dat.sites,site==sites[i]) #only date on the i-th site

> data.frame(site=sites[i]

+ ,mean.x=mean(MySubset$x)

+ , mean.y=mean(MySubset$y))

site mean.x mean.y

1 A1 2.5 6.5

> # -- second round of "loop"

> i=2

> MySubset<-subset(dat.sites,site==sites[i]) #only date on the 2-nd site

> data.frame(site=sites[i]

+ ,mean.x=mean(MySubset$x)

+ , mean.y=mean(MySubset$y))

site mean.x mean.y

1 A1 3.333333 6.666667

As we are repeating this calculation of the mean for each site, one by one, we need to store each value. One alternative to storing your data in a data.frame is to initialize sites.means as an empty data.frame, and then fill it with the information needed using rbind(). The need of first defining sites.means as an empty data.frame and then fill it was covered in chapter 14.1. The above can hence be modified to

> sites.means=data.frame()

> # -- and again our explicitly coded loop but now r-binding

> # -- first round of "loop"

> i=1

> MySubset<-subset(dat.sites,site==sites[i]) #only date on the i-st site

> sites.means<-rbind(sites.means,data.frame(site=sites[i]

+ ,mean.x=mean(MySubset$x)

+ , mean.y=mean(MySubset$y)))

> # -- second round of "loop"

> i=2

> MySubset<-subset(dat.sites,site==sites[i]) #only date on the i-th site

> sites.means<-rbind(sites.means,data.frame(site=sites[i]

+ ,mean.x=mean(MySubset$x)

+ , mean.y=mean(MySubset$y)))

> sites.means

site mean.x mean.y

1 A1 2.500000 6.500000

2 A2 3.333333 6.666667

The script above is an explicitly coded for loop. It has a colouring scheme to illustrate that the exact same lines of coding are repeated (the code in brown above), and the only aspect that has changed is the value of i. Thus, we can now place this as CODE within the for (i in 1:2) {CODE} loop which then repeats the CODE (brown coloured code) for the different values of i, analogous to what is coded explicitly above.

> # for loop filling empty data.frame

> # which are the sites. This needs to be defined beforehand

> sites<-unique(dat.sites$site)

> # define an empty data.frame to hold the data

> sites.means=data.frame()

> for (i in 1:length(sites)) {

+ MySubset<-subset(dat.sites,site==sites[i]) #only date on the i-th site

+ sites.means<-rbind(sites.means

+ , data.frame(site=sites[i]

+ , mean.x=mean(MySubset$x)

+ , mean.y=mean(MySubset$y)))

+ }

> sites.means

site mean.x mean.y

1 A1 2.500000 6.500000

2 A2 3.333333 6.666667

The above uses a loop for (i in 1:length(sites)) { CODE } to repeat what is the “CODE” for each value of i. The value of i is stated to be “in” the vector 1:length(sites) which is the vector 1:2 as there are 2 unique sites (i.e. A1, A2).

Let’s look once more at how the for loop works. In the first round, i = 1, each line of code between the curly brackets printed in brown colour above is processed until the closing curly bracket. The program then starts again from the beginning, but now in this second round, i = 2. After the 2nd time this for() loop stops since i was specified to run from 1 to 2; the program hence continues with the next line after the closing curly bracket. In general, of course, the for() loop can be repeated as many times as needed.

In the code above, the for() function is used in a “classic” fashion (for i in 1:length(sites))) in which an index number, arbitrarily called i , is allowed to vary from 1 to as many sites as there are. The identity of the site is then identified by using the index number, thus specifying sites[i], meaning that in the first round we are considering sites[1] and in the second round of the loop, we are considering sites[2], etc. Such a construction is not necessary in R. This is because the for function already implies to R that we have a vector and that we assign the consecutive values of this vector to i. In the above code, the vector is 1:length(sites). Alternatively, we can use the vector sites, and assign its consecutive values to i; in which case i is first assigned “A1”, then “A2”. Using this formulation, we would not need to identify the specific site as sites[i], but would use i instead. This second approach is provided below, but note that to avoid confusion with the example above the code below does not use i, but assign to s the consecutive entries in the vector sites. Script would look like

> #as above but using vector directly

> sites.means=data.frame()

> for (s in sites) {

+ MySubset<-subset(dat.sites,site==s)

+ sites.means<-rbind(sites.means

+ , data.frame(site=s

+ , mean.x=mean(MySubset$x)

+ , mean.y=mean(MySubset$y)))

+ } #for s

> sites.means

site mean.x mean.y

1 A1 2.500000 6.500000

2 A2 3.333333 6.666667

These two implementations work otherwise the same and it is a matter of taste how you do it.

Lastly, please note that this example is simply meant to illustrate a for() loop. It codes what of course can be easily done in a single line using e.g. aggregate().

> sites.means.aggregate<-aggregate(cbind(X,Y)~site, dat, mean)

### 12.4 Example of a for() loop in repeatedly plotting

A for() loop is used to repeat what is between the curly brackets. We can for example create a plot of y against x in dat.sites for each site

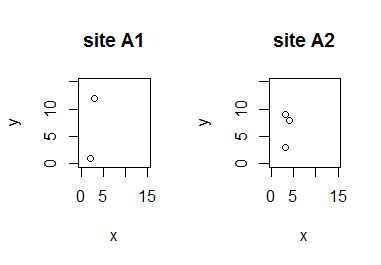
> layout(matrix(c(1,2),1,2))

> for (s in unique(dat.sites$site)) {

+ plot(y~x, data=subset(dat.sites,site==s),ylim=c(0,15),xlim=c(0,15),main=paste("site",s))

+ }

which produces a plot with two panels (Figure 14.1). In the above the two-panel plot is specified using the function layout(), and the function paste() is used to define the plot’s main title to be printed.



*Figure 14.1. Two plots in one panel of data in* dat.sites *per site made using a for loop*

### 12.5 Example of a for loop and logical operator

Let’s return to the data.frame dat

> dat

number name

1 1 Bob

2 2 Bob

3 3 Harriet

4 4 Jim

5 5 Alice

6 6 Bob

Suppose we want to use a for loop to extract the minimal value in dat$number for each unique entry in dat$name. As there is multiple times the same element “Bob” in dat$name, we first need to compile the unique entries.

> unique.names<-unique(dat$name)

> unique.names

[1] “Bob” “Harriet” “Jim” “Alice”

We then can find the elements in dat$number that belong to each unique element in dat$name. Say, we start with the first one

> # find the rows with the first unique value

> index=which(dat$name==unique.names[1])

> index

[1] 1 2 6

> # the minimum value in dat$number that belong to unique.names[1]

> min(dat$number[index])

[1] 1

We have now worked out how to obtain the minimal of the dat$number values belonging to the first unique element in dat$name. The above can be placed in a for loop to carry out the same procedure repeatedly.

> #above in a for loop

> #define empty data.frame to hold the results

> minimum.values<-data.frame()

> for (n in unique.names) {

+ index=which(dat$name==n)

+ minimum.values<-rbind(minimum.values

+ , data.frame(u.name=n, min.val=min(dat$number[index])))

+ }

> minimum.values

u.name min.val

1 Bob 1

2 Harriet 3

3 Jim 4

4 Alice 5

Again, this example is to explain the logic of a for() loop. The same can be coded easily using aggregate().

> aggregate(number~name, dat,min)

name number

1 Bob 1

2 Harriet 3

3 Alice 5

4 Ji m 4

These examples illustrate the coding of a for() loop, but they also highlight that R has powerful functions to repeatedly carry out instructions that we have become familiar with in earlier parts. These examples also demonstrate a division in programming “style”. At this point, however, the main interest is in becoming more familiar with how for() loops work.

## Chapter 13. Array and tabulation

In the R Workflow of this course, tabulation is included in analyses (set of operations 4, Figure 5.0). Tabulation is required for many statistical analyses, but it is also a powerful way to describe and to some extent check your data. Array is a class of object that quickly becomes relevant with tabulation of data with more than 2 levels. In this chapter we start by looking at array and then continue with tabulation and cross-classification of data.

### 13.1 Array

A vector has one dimension, and a matrix has two dimensions. An array is a general extension of this same concept to *n*-dimensions. For example a 3-dimensional array can be thought of as matrices (rows and columns) stacked on top of each other. In >3 dimensions, my imagination does not accommodate visualization anymore, but R is happy to accept it. For example, an array consisting of NA values in 4 dimensions

> array(NA, dim=c(2,2,2,2))

, , 1, 1

[,1] [,2]

[1,] NA NA

[2,] NA NA

, , 2, 1

[,1] [,2]

[1,] NA NA

[2,] NA NA

, , 1, 2

[,1] [,2]

[1,] NA NA

[2,] NA NA

, , 2, 2

[,1] [,2]

[1,] NA NA

[2,] NA NA

In the above, the first two dimensions are row and column (2 rows and 2 columns). The third and fourth dimension of this array each also hold two “layers”. These are printed above as “,,1,1” for the rows and columns in the first layer of the third dimension and first layer of the second dimension. Similarly “,,2,1” prints the rows and columns of the second layer for third dimension and the first layer for the fourth dimension, and so on. Similar to vector and matrix, an array is either filled with numbers or with characters, but not a combination of the two. One way to fill the array is by indexing

> myArray<-array(NA, dim=c(2,3,2))

> myArray[,,1]<-matrix(c(1,2,3,4,5,6),2,3,byrow=TRUE)

> myArray[,,2]<-matrix(6+c(1,2,3,4,5,6),2,3,byrow=TRUE)

> myArray

, , 1

[,1] [,2] [,3]

[1,] 1 2 3

[2,] 4 5 6

, , 2

[,1] [,2] [,3]

[1,] 7 8 9

[2,] 10 11 12

The above thus first defines an empty array and then fills it with two matrices consisting of two rows and three columns “stacked” in the third dimension.

### 13.2 Function apply()

The function apply() follows a similar logic as tapply(). However, instead of specifying as the second argument for which subsets a function is to be applied, apply() asks for the dimension (1 = row, 2 = column, 3 = 3rd dimension, etc) over which a function is to be applied. This function works well for example when we have a matrix or an array. For example, we have an array filled with numbers

> myArray<-array(NA, dim=c(2,3,4))

> myArray[,,1]<-matrix(c(1,2,3,4,5,6),2,3,byrow=TRUE)

> myArray[,,2]<-matrix(2+c(1,2,3,4,5,6),2,3,byrow=TRUE)

> myArray[,,3]<-matrix(4+c(1,2,3,4,5,6),2,3,byrow=TRUE)

> myArray[,,4]<-matrix(6+c(1,2,3,4,5,6),2,3,byrow=TRUE)

> myArray

, , 1

[,1] [,2] [,3]

[1,] 1 2 3

[2,] 4 5 6

, , 2

[,1] [,2] [,3]

[1,] 3 4 5

[2,] 6 7 8

, , 3

[,1] [,2] [,3]

[1,] 5 6 7

[2,] 8 9 10

, , 4

[,1] [,2] [,3]

[1,] 7 8 9

[2,] 10 11 12

Now we want to calculate the number of values larger than 3 along different dimensions.

> myArray>3

, , 1

[,1] [,2] [,3]

[1,] FALSE FALSE FALSE

[2,] TRUE TRUE TRUE

, , 2

[,1] [,2] [,3]

[1,] FALSE TRUE TRUE

[2,] TRUE TRUE TRUE

, , 3

[,1] [,2] [,3]

[1,] TRUE TRUE TRUE

[2,] TRUE TRUE TRUE

, , 4

[,1] [,2] [,3]

[1,] TRUE TRUE TRUE

[2,] TRUE TRUE TRUE

> apply(myArray>3,1,sum)

[1] 8 12

> apply(myArray>3,2,sum)

[1] 6 7 7

> apply(myArray>3,3,sum)

[1] 3 5 6 6

Thus, using apply() to sum the number of values larger than 3 across the first dimension – apply(myArray>3,1,sum) - computes the total for each of the two rows across all other dimensions. Using apply() to sum the number of values larger than 3 across the second dimension – apply(myArray>3,2,sum) - computes the total for each of the three columns across all the dimensions in the array, and apply(myArray>3,3,sum) sums up values for each of the four layers in the 3rd dimension across all other dimensions. You can check this by counting the number of values that are TRUE for each of these dimensions above.

Remember: Working with a member in the apply() family requires functions. You can use one of R’s own functions, you can combine e.g. two functions in an anonymous function or you can write out your own function (topics covered in chapter 9. As we have seen before, all functions in the apply() family allow one to include additional arguments that are passed to the function called within apply.

### 13.3 Factorial data: Tabulation using table()

An array is often produced when tabulating data with many factors. Data often is factorial, meaning there are categories, which may be ordinal (i.e. there is an order e.g. small to large) or nominal (i.e. no order e.g. “red”, “brown”). We have already looked at tapply() and aggregate() as examples of functions used to analyze this type of data when summarizing information per level of a factor. In statistical applications, tabulation also is a basic tool for processing data prior to contingency testing (chi-square and higher dimensional contingency tables). Tabulation of multiple factorial variables can also be used for investigating sample sizes for each cross-categorical level in your data. Data exploration is an important aspect of all analyses. Essentially your analyses should start with you providing a description of your data which serves as a check for yourself but is also something that should be reported. The functions table(), xtabs() and split() are useful to this end.

The function table() will tabulate the numbers of observation for factorial levels in the data. For example,

> df3<-data.frame(individual=c(“ID1”,”ID1”,”ID2”,”ID2”,”ID3”), col=c(“blue”,”blue”,”blue”,”green”,”green”),habitat=c(“bush”,”field”,”field”,”bush”,”field”))

> df3

individual col habitat

1 ID1 blue bush

2 ID1 blue field

3 ID2 blue field

4 ID2 green bush

5 ID3 green field

> df3.tab<-table(df3)

> df3.tab

, , habitat = bush

col

individual blue green

ID1 1 0

ID2 0 1

ID3 0 0

, , habitat = field

col

individual blue green

ID1 1 0

ID2 1 0

ID3 0 1

> class(df3.tab)

[1] “table”

Thus, table() is used to tabulate (i.e. return the summed count of the number of cases) in a data.frame in the order of the variables in the data.frame provided. In the above example, it therefore returns what is in effect an array with in the order they appear: (1) individual in 1st dimension (row), (2) colour in 2nd dimension (column) and (3) habitat in the third dimension. We can use subset() or indexing to alter what we give to table() and hence what we get. For example, total number of cases per habitat (in rows) and colour (in columns):

> table(subset(df3,select=c(habitat,col)))

col

habitat blue green

bush 1 1

field 2 1

Alternatively, we can further process the df3.tab object produced above by the call to table(). This object resembles an array –like object (technically it is of class table). For example, to get the sample sizes break-down for the cross-classified categories

> colSums(df3.tab)

habitat

col bush field

blue 1 2

green 1 1

### 13.4 Tabulation using xtabs()

The function xtabs() is a flexible function for cross tabulation that uses the formula formulation, as we have used for aggregate(). Cross tabulation counts the number of cases for combinations of factorial levels in your data. The formula specifies which factors are to be cross tabulated. A formula object always includes the tilde ~ and one then needs to specify for which data the required formula is to be used. Many functions in R accept arguments as a formula object.

When using xtabs(), it is possible to specify the formula both in the format y ~ x + z to sum up variable y for each level of the factors x and z, as well as to specify ~x + z to tabulate (i.e. break down the sample sizes) for combinations of factor x and z. For example, the above detailed break down of cases per habitat and colour

> xtabs(~habitat+col,df3)

col

habitat blue green

bush 1 1

field 2 1

or to compute per individual what habitat they were recorded in

> xtabs(~individual+habitat,data=df3)

habitat

individual bush field

ID1 1 1

ID2 1 1

ID3 0 1

Which variable is placed in which dimension in the output of xtabs() follows the order in which the variables are specified (1st variable in 1st dimension, 2nd in 2nd, etc.).

All of the above tabulations are of course equivalent to using the functions tapply() or aggregate() to compute sample sizes as introduced earlier

> tapply(df3$individual,list(df3$col,df3$habitat),length)

bush field

blue 1 2

green 1 1

> aggregate(individual~habitat+col,df3,length)

habitat col individual

1 bush blue 1

2 field blue 2

3 bush green 1

4 field green 1

which produce an object of class array and class data.frame respectively.

For another example we can suppose we have data on the number of parasites of the individuals (i.e. not a factor) which is added to our data. Now, xtabs() can be used to compute the total number of parasites recorded for different combinations of the factorial variables.

> df3$parasites<-c(1,2,5,0,6)

> xtabs(parasites~col+habitat,data=df3)

habitat

col bush field

blue 1 7

green 0 6

Cross tabulation, because of its formula, presents an intuitive approach to quickly sum cases belonging to specific categories, which is typically handy to investigate how e.g. sample sizes or other variables that can be summed break down across different categories. Again, also tapply() and aggregate() or other functions can be used to sum. The main difference is that table(), xtabs() and also tapply() produce an “array-like” object (potentially multidimensional), whereas aggregate() produces a data.frame. Within the functions producing an “array-like” object, tapply() will state if specific cross-classified statistics are not available (NA), whereas table() and xtabs() will not do so. In some cases, this seemingly subtle distinction can be important. For example,

> tapply(df3$parasites, list(df3$individual,df3$col,df3$habitat),sum)

, , bush

blue green

ID1 1 NA

ID2 NA 0

ID3 NA NA

, , field

blue green

ID1 2 NA

ID2 5 NA

ID3 NA 6

From the above, it is clear that tapply() produces NA instead of 0 for combinations that do not exist in the data. That is, there are zero parasites for individual ID2 which has green colour and is in the habitat bush. In contrast, xtabs() will produces 0 to both indicate that 0 parasites were observed (for cross-classification ID2, green, bush) as well as when the cross-classification contains no data (NA above in the call to tapply(), but these are also zeros below in the call to xtabs()).

> xtabs(parasites~individual+col+habitat,df3,addNA = T)

, , habitat = bush

col

individual blue green

ID1 1 0

ID2 0 0

ID3 0 0

, , habitat = field

col

individual blue green

ID1 2 0

ID2 5 0

ID3 0 6

The function aggregate() will produce a 2-dimensional data.frame object, as illustrated by specifying the above breakdown using aggregate()

> aggregate(parasites~individual+col+habitat,data=df3,sum)

individual col habitat parasites

1 ID1 blue bush 1

2 ID2 green bush 0

3 ID1 blue field 2

4 ID2 blue field 5

5 ID3 green field 6

where also aggregate() produces only zeros for cross-classified cases for which there is data and thus really are zero. As a further consequence, aggregate() will not necessarily report all possible cross classifications, as in the second example above.

Care is also required when there are missing values (NA) on some levels of the cross classification you are interested in. For example, consider a data.frame with count of ticks made on 3 different individuals in 2 different patches of forest.

> #xtabs with characters

> df.ticks.char<-data.frame(ind=c(rep("ind1",2),rep("ind2",2),rep("ind3",2)),patch=rep(c(1,2),3),n.ticks=c(1,1,2,NA,NA,NA),stringsAsFactors = F)

> df.ticks.char

ind patch n.ticks

1 ind1 1 1

2 ind1 2 1

3 ind2 1 2

4 ind2 2 NA

5 ind3 1 NA

6 ind3 2 NA

> xtabs(n.ticks~ind+patch,df.ticks.char) #individual 3 is dropped

patch

ind 1 2

ind1 1 1

ind2 2 0

In the above, the individidual ID (ind) is a character vector. As we saw above, xtabs() here reports there were no ticks on individual ind2 in patch 2 although in fact the value was missing (NA). Furthermore, using xtabs() to sum the number of ticks for each individual in each patch will ignore individual ind3 as the number of ticks on this individual always was a missing value. In cases where it is important to include all the levels of a factorial variable in the cross classification, the variable has to be specified to be a factor. This is because for a factor, xtabs() will report the cross classification for each level of the factor. Hence,

> #xtabs with factors

> df.ticks.fac<-data.frame(ind=c(rep("ind1",2),rep("ind2",2),rep("ind3",2)),patch=rep(c(1,2),3),n.ticks=c(1,1,2,NA,NA,NA),stringsAsFactors = T)

> df.ticks.fac

ind patch n.ticks

1 ind1 1 1

2 ind1 2 1

3 ind2 1 2

4 ind2 2 NA

5 ind3 1 NA

6 ind3 2 NA

> xtabs(n.ticks~ind+patch,df.ticks.fac) #individual 3 is not dropped

patch

ind 1 2

ind1 1 1

ind2 2 0

ind3 0 0

Here, again, xtabs() will consider NA to be 0. However, when ind is defined as a factor, also ind3 which has only NAs (missing values) as the number of ticks will be included in the cross tabulation (instead of being excluded when ind is not a factor). Computing the total of ticks while allowing NA changes the output in such a way that all levels are specified and missing values are left blank.

> xtabs(n.ticks~ind+patch,df.ticks.char,na.action=na.pass) #individual 3 is dropped

patch

ind 1 2

ind1 1 1

ind2 2

ind3

This is a fairly detailed and exhaustive overview of tabulation. As is typical for R, it is clear that there are multiple solutions possible, allowing you as user to choose the format most useful to your need. The important aspect to keep in mind is of course that these different options may or may not produce different answers. Care is therefore needed to check that the answer you obtain makes sense given the data. For large datasets this is very difficult and it is good to check your code using a subset of the data.

### 13.4 Exercises part V

**Exercise V.1**

The file “dynamics.txt” contains results of 1000 simulations (rows) of population size over 100 years (in columns)

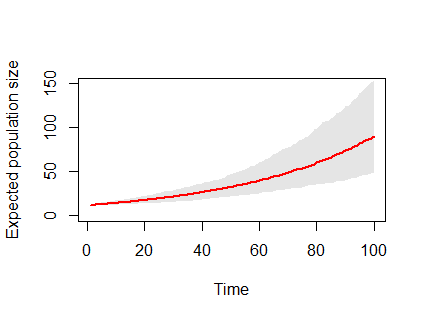
(a) Read in this information into R

(b) calculate the expected population size (i.e. the mean) for each of the 100 time steps

(c) plot the trajectory of the mean population size over time as good as you can

(d) Use the function quantile() to calculate the lower and upper quantiles that contain 95% of values over all 1000 simulations for each of the 100 time steps

(e) use the function polygon() to plot both the “envelope” in which 95% of simulated values lie as well as the mean, as in the following plot



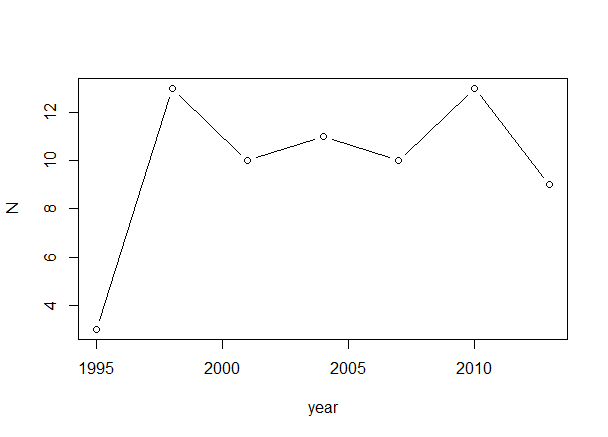
TIP: A number of functions draw or add things to your plot (like arrows, polygon, lines, text, segments, axis, …), but they all require that the plotting space is defined first using a plot command.

TIP2: The function polygon() will draw a line following the series of x and y coordinates in the order that you provide these coordinates to the function polygon(). In other words, it is conceptually like you are drawing it with a pencil in one go without taking the pencil from the paper.

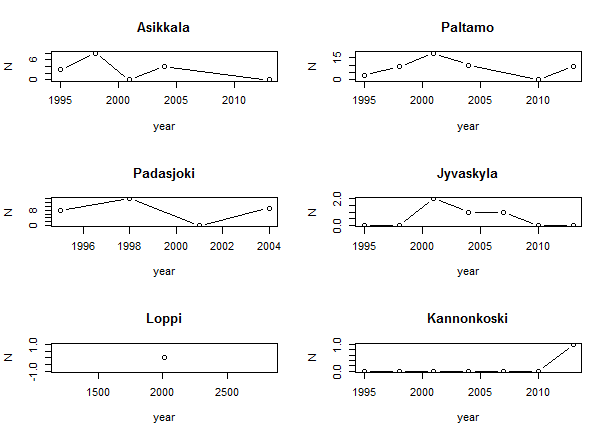
**Exercise V.2**

Read in the datafile “PopData” which contains counts of winter nests of beavers in Finland per municipality. The counts are made every third year, but are not always performed

1. How many counts are there per municipality? make a histogram of the number of counts
2. Plot the dynamics as a line with dots for the municipality Virrat. TIP: the function sort() or order() can be useful in making sure the data is in chronological order



1. Code a for loop to plot the dynamics of any six municipalities in a single plot with 6 panels with the name of the municipalities chosen printed above the plot. For the municipalities c(“Asikkala”, “Paltamo”, ”Padasjoki”, “Jyvaskyla”, ”Loppi”, ”Kannonkoski”) the following output should be produced



**Exercise V.3**

The zipped folder “outputData” contains files. Each file is named by the site and contains measures of that site.

Unzip all these files in a separate directory.

Write a script that reads in all files into R and calculates for each site the mean value as well as the number of measures taken. The function dir() will produce a vector with all file names in your working directory. Your code should produce a data.frame site.data that looks like

> head(site.data)

site meanValue n

1 Site1 38.888553 288

2 Site10 17.239733 254

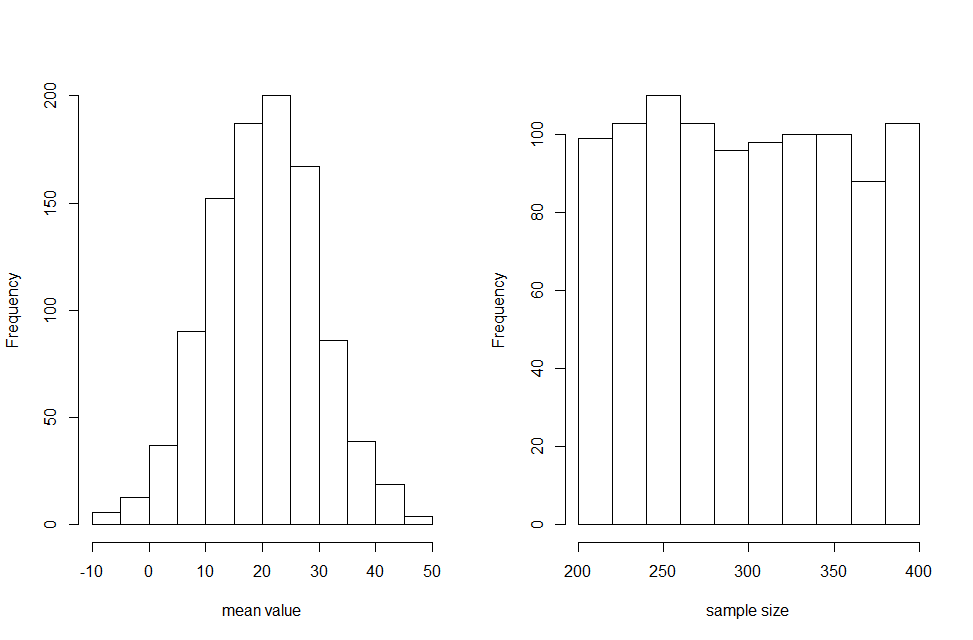
3 Site100 12.466849 226

4 Site1000 7.410668 269

5 Site101 11.737528 312

6 Site102 24.510496 326

Code the plotting of a histogram of all means and sample sizes that looks like this:



As always. Approach the solution in steps. dir() will produce a vector with all file names in your working directory. Start by coding reading in one file from that vector with filenames, and calculating the desired parameters. Then figure out how to add one more to this, and finally place in a for() loop.

**Exercise VI.4**

The file"DataBlueTits.csv" contains the variables: 1. Ring (ID code of each individual; the ID consists of a unique combination of numbers and letters), 2. Year (year the individual was caught), 3. Tarsus (a measure obtained on that individual when it was caught, although sometimes this measure was forgotten = NA).

Use this data to:

a) Perform a basic check of the data import

b) Tabulate for each individual (Ring) whether it was caught (1) or not (0) in each year

c) Use the tabulation you created in (b) to compute the number of years each individual was caught at least once and make a histogram of this information as well as a data.frame that should look like:

n.years frequency

1 1103

2 323

3 153

4 60

5 23

6 5

7 1

Thus, 1103 individuals were caught only in one year, 323 were caught in two years, etc. Note that individuals may have been caught several times in one year, but in the above table this is counted still only as one catch for that year. Observe that when you make a histogram, you also obtain the above information.

**Exercise VI.5**

The file"DataBlueTits.csv" contains the variables: 1. Ring (ID code of each individual; the ID consists of a unique combination of numbers and letters), 2. Year (year the individual was caught), 3. Tarsus (a measure obtained on that individual when it was caught, although sometimes this measure was forgotten = NA).

Use this data to:

a) Calculate in how many years the tarsus of each individual was measured at least once using either tapply(), table(), or xtabs() (you can choose which function), c:

times frequency

1 0 40

2 1 1070

3 2 322

4 3 150

5 4 59

6 5 22

7 6 4

8 7 1

where 40 individuals never had their tarsus measured (i.e. they were caught but not measured), 1070 individuals had their tarsus measured in one year at least once, 322 individuals had their tarsus measured in two years at least once per year, etc. TIP: This is easiest done by adjusting your code for step b in the previous exercise.

b) Combine the histograms of the data.frame you made in the previous exercise with the one made in this exercise in the same plot with the same scales, and labelled panels a and b as:

