Chapter 6
Loops and Functions

When reading this book for the first time, you may skip this chapter, as building functions[1] and programming loops[2] are probably not among the first R procedures you want to learn, unless these subjects are your prime interests. In general, people perceive these techniques as difficult, hence the asterisk in the chapter title. Once mastered, however, these tools can save enormous amounts of time, especially when executing a large number of similar commands.

6.1 Introduction to Loops

One of R's more convenient features is the provision for easily making your own functions. Functions are useful in a variety of scenarios. For example, suppose you are working with a large number of multivariate datasets, and for each of them you want to calculate a diversity index. There are many diversity indices, and new ones appear regularly in the literature. If you are lucky, the formula for your chosen diversity index has already been programmed by someone else, and, if you are very lucky, it is available in one of the popular packages, the software code is well documented, fully tested, and bug free. But if you cannot find software code for the chosen diversity index, it is time to program it yourself!

If you are likely to use a set of calculations more than once, you would be well advised to present the code in such a way that it can be reused with minimal typing. Quite often, this brings you into the world of functions and loops (and conditional statements such as the `if` command).

The example presented below uses a dataset on owls to produce a large number of graphs. The method involved is repetitive and time consuming, and a procedure that will do the hard work will be invaluable.

---

[1] A function is a collection of codes that performs a specific task.

[2] A loop allows the program to repeatedly execute commands. It does this by iteration (iteration is synonymous with repetition).

A.F. Zuur et al., *A Beginner's Guide to R*, Use R,
DOI 10.1007/978-0-387-93837-0_6, © Springer Science+Business Media, LLC 2009

Developing this procedure requires programming and some logical thinking. You will need to work like an architect who draws up a detailed plan for building a house. You should definitely not begin entering code for a function or loop until you have an overall design.

You also must consider how foolproof your function needs to be. Do you intend to use it only once? Should it work next year on a similar dataset (when you have forgotten most settings and choices in your function)? Will you share it with colleagues?

Functions often go hand in hand with loops, as they both help to automate commands.

Suppose you have 1000 datasets, and for each dataset you need to make a graph and save it as a jpeg. It would take a great deal of time to do this manually, and a mechanism that can repeat the same (or similar) commands any number of times without human intervention would be invaluable. This is where a loop comes in. A plan for the 1000 datasets could be

```
For i is from 1 to 1000:
        Extract dataset i
        Choose appropriate labels for the graph for dataset i
        Make a graph for dataset i
        Save the graph for dataset i
End of loop
```

Note that this is not R code. It is merely a schematic overview, which is the reason that we put the text in a box and did not use the " > " symbol and the `Courier New` font that we have been using for R code. The sketch involves a loop, meaning that, once the code is syntax correct, R executes 1000 iterations, with the first iteration having $i = 1$, the second iteration $i = 2$, and in the final iteration $i = 1000$. In each iteration, the commands inside the loop are executed.

This plan has only four steps, but, if we want to do more with the data, it may make sense to group certain commands and put them in a function. Suppose we not only want a graph for each dataset, but also to calculate summary statistics and apply a multivariate analysis. We will very quickly end up with 10–15 commands inside the loop, and the code becomes difficult to manage. In such a scenario, using functions can keep the code simple:

```
For i is from 1 to 1000:
        Extract dataset i
        Execute a function to calculate summary statistics for dataset i.
        Execute a function to make and save a graph for dataset i.
        Execute a function that applies multivariate analysis on dataset i.
End of loop
```

Each function is a small collection of commands acting on individual datasets. Each function works independently, unaffected by what happens elsewhere, and does only what it has been told to do. There is a mechanism in place to allow only the dataset into the function and to return information for this dataset. Once programmed, the function should work for any dataset. Program it once, and, if all goes according to plan, you never have to think about it again.

Just as a house can be designed to be built in different ways, your plan can take more than one approach. In the sketch above, we created a loop for *i* from 1 to 1000, which, in each iteration, extracts data and passes the data to a function. You can also do it the other way around:

> Execute a function to calculate summary statistics for each dataset.
> Execute a function to make and save a graph for each dataset.
> Execute a function to apply multivariate analysis on each dataset.

Each function will contain a loop in which the data are extracted and subjected to a series of relevant commands. The building of the code depends entirely on personal programming style, length of the code, type of problem, computing time required, and so on.

Before addressing the creation of functions, we focus on loops.

6.2 Loops

If you are familiar with programming languages like FORTRAN, C, C++, or MATLAB,[3] you are likely to be familiar with loops. Although R has many tools for avoiding loops, there are situations where it is not possible. To illustrate a situation in which a loop saves considerable time, we use a dataset on begging behaviour of nestling barn owls. Roulin and Bersier (2007) looked at nestlings' response to the presence of the mother and the father. Using microphones inside, and a video camera outside, the nests, they sampled 27 nests, studying vocal begging behaviour when the parents bring prey. A full statistical analysis using mixed effects modelling is presented in Roulin and Bersier (2007) and also in Zuur et al. (2009).

For this example, we use "sibling negotiation," defined as the number of calls by the nestlings in the 30-second interval immediately prior to the arrival of a parent, divided by the number of nestlings. Data were collected between 21.30 hours and 05.30 hours on two consecutive nights. The variable `ArrivalTime` indicates the time at which a parent arrived at the perch with prey.

Suppose that you have been commissioned to write a report on these data and to produce a scatterplot of sibling negotiation versus arrival time for each nest, preferably in jpeg format. There are 27 nests, so you will need to produce,

[3] These are just different types of programming languages, similar to R.

and save, 27 graphs. This is not an uncommon type of task. We have been involved in similar undertakings (e.g., producing multiple contour plots for >75 bird species in the North Sea). Keep in mind that they may ask you to do it all again with a different plotting character or a different title! Note that R has tools to plot 27 scatterplots in a single graph (we show this in Chapter 8), but assume that the customer has explicitly asked for 27 separate jpeg files. This is not something you will not want to do manually.

6.2.1 Be the Architect of Your Code

Before writing the code, you will need to plan and produce an architectural design outlining the steps in your task:

1. Import the data and familiarise yourself with the variable names, using the `read.table`, `names`, and `str` commands.
2. Extract the data of one nest and make a scatterplot of begging negotiation versus arrival time for this subset.
3. Add a figure title and proper labels along the *x*- and *y*-axes. The name of the nest should be in the main header.
4. Extract data from a second nest, and determine what modifications to the original graph are needed.
5. Determine how to save the graph to a jpeg file.
6. Write a loop to extract data for nest *i*, plot the data from nest *i*, and save the graph to a jpeg file with an easily recognized name.

   If you can implement this algorithm, you are a good architect!

6.2.2 Step 1: Importing the Data

The following code imports the data and shows the variable names and their status. There is nothing new here in terms of R code; the `read.table`, `names`, and `str` commands were discussed in Chapters 2 and 3.

```
> setwd("C:/RBook/")
> Owls <- read.table(file = "Owls.txt", header = TRUE)
> names(Owls)
[1] "Nest"                "FoodTreatment"
[3] "SexParent"           "ArrivalTime"
[5] "SiblingNegotiation" "BroodSize"
[7] "NegPerChick"
> str(Owls)
'data.frame':   599 obs. of  7 variables:
 $ Nest              : Factor w/ 27 levels ...
```

```
 $ FoodTreatment     : Factor w/ 2 levels ...
 $ SexParent         : Factor w/ 2 levels ...
 $ ArrivalTime       : num 22.2 22.4 22.5 22.6 ...
 $ SiblingNegotiation: int 4 0 2 2 2 2 18 4 18 0 ...
 $ BroodSize         : int 5 5 5 5 5 5 5 5 5 5 ...
 $ NegPerChick       : num 0.8 0 0.4 0.4 0.4 0.4 ...
```

The variables `Nest`, `FoodTreatment`, and `SexParent` are defined using alphanumerical values in the ascii file, and therefore R considers them (correctly) as factors (see the output of the `str` command for these variables).

6.2.3 Steps 2 and 3: Making the Scatterplot and Adding Labels

To extract the data from one nest, you first need to know the names of the nests. This can be done with the `unique` command

```
> unique(Owls$Nest)
 [1] AutavauxTV        Bochet       Champmartin
 [4] ChEsard           Chevroux     CorcellesFavres
 [7] Etrabloz          Forel        Franex
[10] GDLV              Gletterens   Henniez
[13] Jeuss             LesPlanches  Lucens
[16] Lully             Marnand      Moutet
[19] Murist            Oleyes       Payerne
[22] Rueyes            Seiry        SEvaz
[25] StAubin           Trey         Yvonnand
27 Levels: AutavauxTV Bochet Champmartin ... Yvonnand
```

There are 27 nests, and their names are given above. Extracting the data of one nest follows the code presented in Chapter 3:

```
> Owls.ATV <- Owls[Owls$Nest=="AutavauxTV", ]
```

Note the comma after `Owls$Nest==" AutavauxTV"` to select rows of the data frame. We called the extracted data for this nest `Owls.ATV`, where ATV refers to the nest name. The procedure for making a scatterplot such as that needed to show arrival time versus negotiation behaviour for the data in `Owls.ATV` was discussed in Chapter 5. The code is as follows.

```
> Owls.ATV <- Owls[Owls$Nest == "AutavauxTV", ]
> plot(x = Owls.ATV$ArrivalTime,
       y = Owls.ATV$NegPerChick,
       xlab = "Arrival Time", main = "AutavauxTV"
       ylab = "Negotiation behaviour)
```

You will be plotting the variable `ArrivalTime` versus `NegPerChick` from the data frame `Owls.ATV`, hence the use of the $ sign. The resulting graph is presented in Fig. 6.1. So far, the procedure requires no new R code.

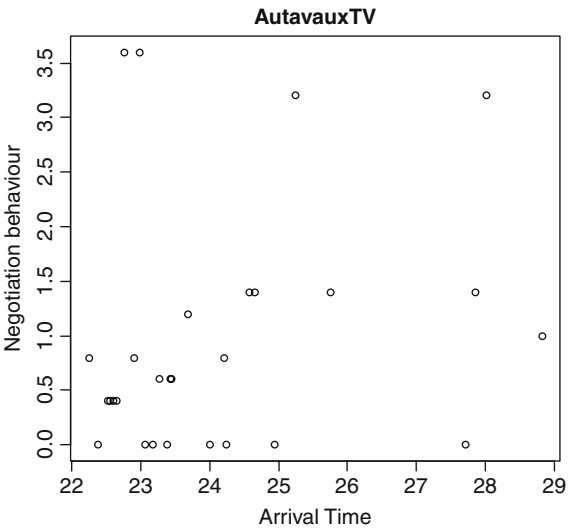

Fig. 6.1 Scatterplot of arrival time (*horizontal axis*) versus average negotiation behaviour per visit (*vertical axis*) for a single nest (AutavauxTV). Time is coded from 22 (22.00) to 29 (4.00). Measurements were conducted on two consecutive nights

6.2.4 Step 4: Designing General Code

To investigate the universality of the code, go through the same procedure for data from another nest. The code for the second nest requires only a small modification; where you entered `AutavauxTV`, you now need `Bochet`.

```
> Owls.Bot <- Owls[Owls$Nest == "Bochet", ]
> plot(x = Owls.Bot$ArrivalTime,
       y = Owls.Bot$NegPerChick,
       xlab = "Arrival Time",
       ylab = "Negotiation behaviour", main = "Bochet")
```

The graph is not shown here. Note that we stored the data from this particular nest in the data frame `Owls.Bot`, where "Bot" indicates "Bochet." If you were to make the same graph for another nest, you need only replace the main title and the name of the data frame and the actual data (the loop will do this for us).

The question is, in as much as you must do this another 25 times, how can you minimise the typing required? First, change the name of the data frame to something more abstract. Instead of `Owls.ATV` or `Owls.Bot`, we used `Owls.i`. The following construction does this.

```
> Owls.i <- Owls[Owls$Nest == "Bochet", ]
> plot(x = Owls.i$ArrivalTime,
       y = Owls.i$NegPerChick, xlab = "Arrival Time",
       ylab = "Negotiation behaviour", main = "Bochet")
```

Instead of a specific name for the extracted data, we used a name that can apply to any dataset and pass it on to the `plot` function. The resulting graph is not presented here. The name "Bochet" still appears at two places in the code, and they need to be changed each time you work with another dataset. To minimise typing effort (and the chance of mistakes), you can define a variable, `Nest.i`, containing the name of the nest, and use this for the selection of the data and the main header:

```
> Nest.i <- "Bochet"
> Owls.i <- Owls[Owls$Nest == Nest.i, ]
> plot(x = Owls.i$ArrivalTime, y = Owls.i$NegPerChick,
     xlab = "Arrival Time", main = Nest.i,
     ylab = "Negotiation behaviour")
```

In order to make a plot for another nest, you only need to change the nest name in the first line of code, and everything else will change accordingly.

6.2.5 Step 5: Saving the Graph

You now need to save the graph to a jpeg file (see also the help file of the `jpeg` function):

1. Choose a file name. This can be anything, for example, `''AnyName.jpg''`.
2. Open a jpeg file by typing `jpeg(file = ''AnyName.jpg'')`.
3. Use the `plot` command to make graphs. Because you typed the `jpeg` command, R will send all graphs to the jpeg file, and the graphic output will not appear on the screen.
4. Close the jpeg file by typing: `dev.off()`.

You can execute multiple graphing commands in Step 3 (e.g., `plot`, `lines`, `points`, `text`) and the results of each will go into the jpeg file, until R executes the `dev.off` (device off) command which closes the file. Any graphing command entered after the `dev.off` command will not go into the jpeg file, but to the screen again. This process is illustrated in Fig. 6.2.

At this point, you should consider where you want to save the file(s), as it is best to keep them separate from your R working directory. In Chapter 3 we discussed how to set the working directory with the `setwd` command. We set it to "C:/AllGraphs/" in this example, but you can easily modify this to your own choice.

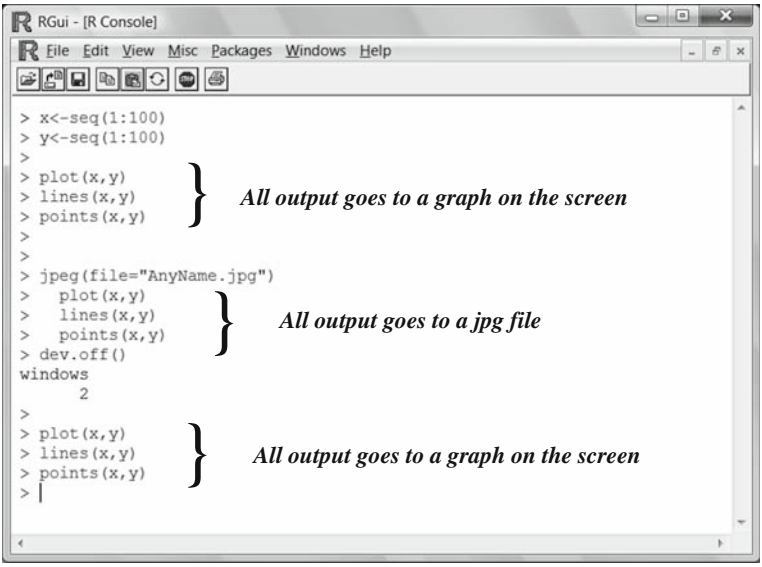

Fig. 6.2 Summary of the `jpeg` and `dev.off` commands. The results of all graphing commands between the `jpeg` and `dev.off` commands are sent to a jpg file. The *x*- and *y*-coordinates were arbitrarily chosen

The final challenge is to create a file name that automatically changes when we change the name of the nest (the variable `Nest.i`). You will need a file name that consists of the nest name (e.g., Bochet) and the file extension jpg. To connect "Bochet" and ".jpg" with no separation between these two strings (i.e., "Bochet.jpg") use the `paste` command:

```
> paste(Nest.i, ".jpg", sep = "")
[1] "Bochet.jpg"
```

The output of the `paste` command is a character string that can be used as the file name. You can store it in a variable and use it in the `jpeg` command. We called the variable `YourFileName` in the code below, and R sends all graphic output created between the `jpeg` and `dev.off` commands to this file.

```
> setwd("C:/AllGraphs/")
> Nest.i <- "Bochet"
> Owls.i <- Owls[Owls$Nest == Nest.i, ]
> YourFileName <- paste(Nest.i, ".jpg", sep="")
> jpeg(file = YourFileName)
> plot(x = Owls.i$ArrivalTime, y = Owls.i$NegPerChick,
     xlab = "Arrival Time", main = Nest.i,
     ylab = "Negotiation behaviour")
> dev.off()
```

Once this code has been executed, you can open the file *Bochet.jpg* in your working directory with any graphic or photo editing package. The help file for the `jpeg` function contains further information on increasing the size and quality of the jpeg file. Alternative file formats are obtained with the functions `bmp`, `png`, `tiff`, `postscript`, `pdf`, and `windows`. See their help files for details.

6.2.6 Step 6: Constructing the Loop

You still have to modify the variable `Nest.i` 27 times, and, each time, copy and paste the code into R. Here is where Step 6 comes in, the loop. The syntax of the `loop` command in R is as follows.

```
for (i in 1 : 27) {
     do something
     do something
     do something
   }
```

"Do something" is not valid R syntax, hence the use of a box. Note that the commands must be between the two curly brackets { and }. We used 27 because there are 27 nests. In each iteration of the loop, the index *i* will take one of the values from 1 to 27. The "do something" represent orders to execute a specific command using the current value of *i*. Thus, you will need to enter into the loop the code for opening the jpeg file, making the plot, and closing the jpeg file for a particular nest. It is only a small extension of the code from Step 5.

On the first line of the code below, we determined the unique names of the nests. On the first line in the loop, we set `Nest.i` equal to the name of the *i*th nest. So, if *i* is 1, `Nest.i` is equal to `''AutavauxTV''`; *i* = 2 means that `Nest.i = ''Bochet'';` and, if *i* is 27, `Nest.i` equals `''Yvonnand''` The rest of the code was discussed in earlier steps. If you run this code, your working directory will contain 27 jpeg files, exactly as planned.

```
> AllNests <- unique(Owls$Nest)
> for (i in 1:27){
 Nest.i <- AllNests[i]
 Owls.i <- Owls[Owls$Nest == Nest.i, ]
 YourFileName <- paste(Nest.i, ".jpg", sep = "")
 jpeg(file = YourFileName)
 plot(x = Owls.i$ArrivalTime, y = Owls.i$NegPerChick,
      xlab = "Arrival Time",
      ylab = "Negotiation behaviour", main = Nest.i)
 dev.off()
 }
```

Do Exercise 1 in Section 6.6. This is an exercise in creating loops, using a temperature dataset.

6.3 Functions

The principle of a *function* may be new to many readers. If you are not familiar with it, envision a function as a box with multiple holes on one side (for the input) and a single hole on the other side (for the output). The multiple holes can be used to introduce information into the box; the box will act as directed upon the information and feed the results out the single hole. When a function is running properly, we are not really interested in knowing how it obtains the results. We have already used the `loess` function in Chapter 5. The input consisted of two variables and the output was a list that contained, among other things, the fitted values. Other examples of existing functions are the `mean`, `sd`, `sapply`, and `tapply`, among others.

The underlying concept of a function is sketched in Fig. 6.3. The input of the function is a set of variables, A, B, and C, which can be vectors, matrices, data frames, or lists. It then carries out the programmed calculations and passes the information to the user.

The best way to learn how to use a function is by seeing some examples.

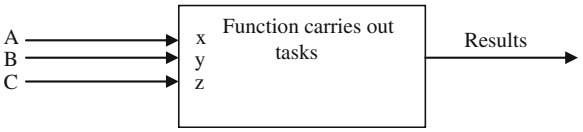

Fig. 6.3 Illustration of the principle of a function. A function allows for the input of multiple variables, carries out calculations, and passes the results to the user. According to the order in which the variables are entered, A, B, and C are called x, y, and z within the function. This is called positional matching

6.3.1 Zeros and NAs

Before executing a statistical analysis, it is important to locate and deal with any missing values, as they may present some difficulties. Certain techniques, such as linear regression, will remove any case (observation) containing a missing value. Variables with many zeros cause trouble as well, particularly in multivariate analysis. For example, do we say that dolphins and elephants are similar because they are both absent on the moon? For a discussion on double zeros in multivariate analysis, see Legendre and Legendre (1998). In univariate analysis, a response variable with many zeros can also be problematical (See the Zero Inflated Data chapter in Zuur et al., 2009).

We recommend creating a table that gives the number of missing values, and the number of zeros, per variable. A table showing the number of missing values (or zeros) per case is also advisable. The following demonstrates using R code to

create the tables, but before continuing, we suggest that you do Exercise 2 in Section 6.6, as it guides you through the R code in this section.

Our example uses the vegetation data from Chapter 4. We imported the data with the `read.table` command, and used the `names` command to see the list of variables:

```
> setwd("C:/RBook/")
> Veg <- read.table(file = "Vegetation2.txt",
                    header = TRUE)
> names(Veg)
 [1] "TransectName" "Samples"      "Transect"
 [4] "Time"         "R"            "ROCK"
 [7] "LITTER"       "ML"           "BARESOIL"
[10] "FallPrec"     "SprPrec"      "SumPrec"
[13] "WinPrec"      "FallTmax"     "SprTmax"
[16] "SumTmax"      "WinTmax"      "FallTmin"
[19] "SprTmin"      "SumTmin"      "WinTmin"
[22] "PCTSAND"      "PCTSILT"      "PCTOrgC"
```

The first four variables contain transect name, transect number, and time of survey. The column labelled R contains species richness (the number of species) per observation. The remaining variables are covariates.

Suppose you want a function that takes as input a data frame that contains the data, and calculates the number of missing values in each variable. The syntax of such a function is

```
NAPerVariable <- function(X1) {
  D1 <- is.na(X1)
  colSums(D1)
}
```

If you type this code into a text editor and paste it into R, you will see that nothing happens. The code defines a function with the name `NAPerVariable`, but it does not execute the function. This is done with the command

```
> NAPerVariable(Veg[,5:24])
      R     ROCK   LITTER       ML  BARESOIL  FallPrec
      0        0        0        0         0         0
SprPrec  SumPrec  WinPrec  FallTmax   SprTmax   SumTmax
      0        0        0        0         0         0
WinTmax FallTmin  SprTmin   SumTmin   WinTmin   PCTSAND
      0        0        0        0         0         0
PCTSILT  PCTOrgC
      0        0
```

We omitted the first four columns of the data frame Veg, as these contain the transect and time information. There appear to be no missing values in the listed variables. Take a closer look at what is going on inside the function. The first, and only, argument of the function is X1. We assume that the variables are in columns and the observations in rows. The command is.na(X1) creates a Boolean matrix of the same dimension as X1, with the value TRUE if the corresponding element of X1 is a missing value and FALSE if not. The colSums function is an existing R function that takes the sum of the elements in each column (variable). Normally, colSums is applied to a data matrix with numbers, but if it is applied to a Boolean matrix, it converts a TRUE to 1, and a FALSE to 0. As a result, the output of colSums(D1) is the number of missing values per variable.

If you replace the colSums command with the rowSums command, the function gives the number of missing values per observation.

6.3.2 Technical Information

There are a few aspects of the function that we need to address: first, the names of the variables used inside the function. Note that we used X1 and D1. You may wonder why the code inside the function runs at all, as X1 seems to come out of the blue. The application here is called *positional matching*. The first and, in this case, only, argument in NAPerVariable, is a subset of the data frame Veg. Inside the function, these data are allocated to X1, because X1 is the first variable in the argument of the function. Hence, X1 contains columns 5 – 24 of the data frame Veg.

The principle of positional matching was illustrated in Fig. 6.1. The external variables A, B, and C are called x, y, and z within the function. R knows that x is A, because both are the first argument in the call to the function. We have already seen this type of action with the arguments in the plot, lines, and loess functions. The reason for changing the variable designations is that you should not use names within a function that also exist outside the function. If you make a programming mistake, for example, if you use D1 <- is.na(X) instead of D1 <- is.na(X1), R will look first inside the function for the values of X. If it does not find this variable inside the function, it will look outside the function. If such a variable exists outside the function, R will happily use it without telling you. Instead of calculating the number of missing values in the variable Veg, it will show you the number of missing values in X, whatever X may be. The convention of using different, or new, names for the variables inside a function applies to all variables, matrices, and data frames used in the function.

A second important aspect of functions is the form in which the resulting information is returned to the user. FORTRAN and C++ users may assume that this is done via the arguments of the function, but this is not the case. It is the information coded for on the final line of the function that is returned. The

function NAPerVariable has colSums(D1) on the last line, so this is the information provided. If you use

```
> H <- NAPerVariable(Veg[ , 4 : 24])
```

H will contain the number of missing values in vector format. If the final line of the function is a list, then H will be a list as well. In an example presented later in this chapter, we see that this is useful for taking back multiple variables (see also Chapter 3).

As always, you should document your code well. Add comments (with the # symbol) to the function, saying that the data must be in an "observation by variable" format, and that it calculate the number of missing values per column.

You should also ensure that the function will run for every possible dataset that you may enter into it in the future. Our function, for example, will give an error message if the input is a vector (one variable) instead of a matrix; colSums only works if the data contain multiple columns (or at least are a matrix). You need to document this, provide an understandable error message, or extend the function so that it will run properly if the input consists of a vector.

6.3.3 A Second Example: Zeros and NAs

The red king crab *Paralithodes camstchaticus* was introduced to the Barents Sea in the 1960 s and 1970 s from its native area in the North Pacific. The leech *Johanssonia arctica* deposits its eggs into the carapace of this crab. The leech is a vector for a trypanosome blood parasite of marine fish, including cod. Hemmingsen et al. (2005) examined a large number of cod for trypanosome infections during annual cruises along the coast of Finnmark in North Norway. We use their data here. The data included the presence or absence of the parasite in fish as well as the number of parasites per fish. Information on the length, weight, age, stage, sex, and location of the host fish was recorded. The familiar read.table and names functions are used to import the data and show the variable names:

```
> setwd("c:/RBook/")
> Parasite <- read.table(file = "CodParasite.txt",
                         header = TRUE)
> names(Parasite)
[1] "Sample"     "Intensity"  "Prevalence" "Year"
[5] "Depth"      "Weight"     "Length"     "Sex"
[9] "Stage"      "Age"        "Area"
```

Because we already copied and pasted the function NAPerVariable into R in Section 6.3.1, there is no need to do this again. To obtain the number of missing values per variable, type

```
> NAPerVariable(Parasite)
Sample  Intensity Prevalence        Year       Depth
     0         57          0           0           0
Weight     Length         Sex      Stage         Age
     6          6           0           0           0
  Area
     0
```

There are 57 missing values in the variable `Intensity`, and 6 in each of the variables `Length` and `Weight`.

In a statistical analysis, we would model the number of parasites as a function of year and length or weight, sex, and location of host fish. This is typically done with generalised linear modelling for count data. Problems may occur if the response variable is zero inflated (too many zeros). Therefore, we need to determine how many zeros are in each variable, especially in `Intensity`. Our first attempt is the function

```
ZerosPerVariable <- function(X1) {
  D1 = (X1 == 0)
  colSums(D1)
}
```

It is similar to the earlier function `NAPerVariable`, except that `D1` is now a matrix with values TRUE if an element of `X1` equals 1, and FALSE otherwise. To execute the function, use

```
> ZerosPerVariable(Parasite)
Sample  Intensity Prevalence        Year       Depth
     0         NA        654           0           0
Weight     Length         Sex      Stage         Age
    NA         NA          82          82          84
  Area
     0
```

There are 654 fish with no parasites, and 82 observations with a value of 0 for Sex. The fact that Sex and Stage have a certain number of observations equal to 0 is a matter of coding; these are nominal variables. So it is not a problem. There are NAs for the variables `Intensity`, `Weight`, and `Length`. This is because the `colSums` function gives NA as output if there is an NA anywhere in the variable. The help file of `colSums` (obtained by typing `?colSums`) shows that the option `na.rm = TRUE` can be added. This leads to:

```
ZerosPerVariable <- function(X1) {
  D1 = (X1 == 0)
```

```
  colSums(D1, na.rm = TRUE)
}
```

Missing values are now ignored because of the `na.rm = TRUE` option. To execute the new function, we use

```
> ZerosPerVariable(Parasite)
Sample  Intensity Prevalence       Year      Depth
     0        654        654          0          0
Weight     Length        Sex      Stage        Age
     0          0         82         82         84
  Area
     0
```

The output now shows no observations with `weight` or `length` equal to 0, and this makes sense. The fact that both `Intensity` and `Prevalence` have 654 zeros also makes sense; absence is coded as 0 in the variable `Prevalence`.

6.3.4 A Function with Multiple Arguments

In the previous section, we created two functions, one to determine the number of missing values per variable and another to find the number of zeros per variable. In this section, we combine them and tell the function to calculate the sum of the number of observations equal to zero or the number of observations equal to NA. The code for the new function is given below.

```
VariableInfo <- function(X1, Choice1) {
   if (Choice1 == "Zeros"){ D1 = (X1 == 0) }
   if (Choice1 == "NAs")  { D1 <- is.na(X1)}
   colSums(D1, na.rm = TRUE)
}
```

The function has two arguments: `X1` and `Choice1`. As before, `X1` should contain the data frame, and `Choice1` is a variable that should contain either the value “Zeros” or “NAs.” To execute the function, use

```
> VariableInfo(Parasite, "Zeros")
Sample  Intensity Prevalence       Year      Depth
     0        654        654          0          0
Weight     Length        Sex      Stage        Age
     0          0         82         82         84
  Area
     0
```

For the missing values, we can use

```
> VariableInfo(Parasite, "NAs")
Sample  Intensity Prevalence        Year       Depth
     0         57          0           0           0
Weight     Length         Sex      Stage         Age
     6          6          0           0           0
  Area
     0
```

As you can see, the output is the same as in the previous section. So, the function performs as we intended. We can also allocate the output of the function to a variable in order to store it.

```
> Results <- VariableInfo(Parasite, "Zeros")
```

If you now type `Results` into the console, you will get the same numbers as above. Figure 6.4 gives a schematic overview of the function up to this point. The function takes as input the data frame `Parasite` and the character string `"Zeros"`, and internally calls them `X1` and `Choice1`, respectively. The function then performs its calculations and the final result is stored in `D1`. Outside the function, the results are available as `Results`. Once everything is perfectly coded and bug free, you can forget about `X1`, `Choice1`, and `D1`, and what is going on inside the function; all that matters is the input and the results.

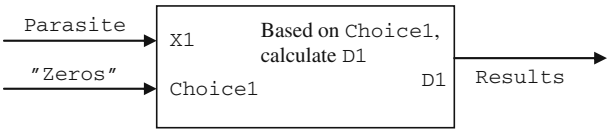

Fig. 6.4 Illustration of the function to calculate the number of zeros or the number of missing values of a dataset. Due to positional matching, the data frame `Parasite` and the argument "`Zeros`" are called `X1` and `Choice1` within the function

The only problem is that our current function is not robust against user error. Suppose you make a typing mistake, spelling "Zeros" as "zeroos":

```
> VariableInfo(Parasite, "zeroos")
Error in inherits(x, "data.frame"): object "D1" not
found
```

The variable `Choice1` is equal to the nonexistent "zeroos", and therefore none of the commands is executed. Hence, `D1` has no value, and an

error message is given on the last line. Another possible mistake is to forget to include a value for the second argument:

```
> VariableInfo(Parasite)
Error in VariableInfo(Parasite): argument "Choice1" is
missing, with no default
```

The variable `Choice1` has no value; the code crashes at the first line. The challenge in making a function is anticipating likely errors. Here, we have seen two silly (but common) mistakes, but the function can be written to provide a safety net for these types of errors.

6.3.5 Foolproof Functions

To make a foolproof function, you have to give it to hundreds of people and ask them all to try it and report any errors, or apply it on hundreds of datasets. Even then, you may be able to crash it. But there are a few common things you can do to make it as stable as possible.

6.3.5.1 Default Values for Variables in Function Arguments

The variable `Choice1` can be given a default value so that if you forget to enter a value for `Choice1`, the function will do the calculations for the default value. This is done with

```
VariableInfo <- function(X1, Choice1 = "Zeros") {
  if (Choice1 == "Zeros"){ D1 = (X1 == 0) }
  if (Choice1 == "NAs")  { D1 <- is.na(X1)}
  colSums(D1, na.rm = TRUE)
}
```

The default value is now "Zeros." Executing this function without specifying a value for `Choice1` produces valid output. To test it, type

```
> VariableInfo(Parasite)

Sample  Intensity Prevalence       Year      Depth
     0        654        654          0          0
Weight     Length        Sex      Stage        Age
     0          0         82         82         84
  Area
     0
```

To calculate the number of missing values, use as before:

```
> VariableInfo(Parasite, "NAs")
```

In this case, the second `if` command in the function is executed. The output of this command is not shown here. Don't forget to write a help file to document the default value!

6.3.5.2 Misspelling

We also want a function that executes the appropriate code, depending on the value of `Choice1`, and gives a warning message if `Choice1` is not equal to "Zeros" or "NAs". The following code does just that.

```
VariableInfo <- function(X1, Choice1 = "Zeros") {
   if (Choice1 == "Zeros"){ D1 = (X1 == 0)  }
   if (Choice1 == "NAs")   { D1 <- is.na(X1)}
   if (Choice1 != "Zeros" & Choice1 != "NAs") {
       print("You made a typo")} else {
             colSums(D1, na.rm = TRUE)}
}
```

The third `if` statement will print a message if `Choice1` is not equal to either "Zeros" or "NAs". If one of these conditions is TRUE, then the `colSums` command is executed. To see it in action, type:

```
> VariableInfo(Parasite, "abracadabra")

[1] "You made a typo"
```

Note that internally the function is doing the following steps.

> If A then blah blah
> If B then blah blah
> If C then blah blah, ELSE blah blah

A professional programmer will criticise this structure, as each `if` statement is inspected by R, even if the argument is "Zero" and only the first `if` statement is relevant. In this case, this does not matter, as there are only three `if` statements which won't take much time, but suppose there are 1000 `if` statements, only one of which needs to be executed. Inspecting the entire list is a waste of time. The help file for the `if` command, obtained by `?if`, provides some tools to address this situation. In the "See also" section, there is a link to the `ifelse` command. This can be used to replace the first two commands in the function:

```
> ifelse(Choice1 == "Zeros", D1 <- (X1 == 0),
                             D1 <- is.na(X1))
```

If the value of Choice1 is equal to "Zeros", then the D1 <-(X1 == 0) command is executed, and, in all other situations, it is D1 <- is.na(X1). Not exactly what we had in mind, but it illustrates the range of options available in R. In Section 6.4, we demonstrate the use of the if else construction to avoid inspecting a large number of if statements.

Do Exercise 2 in Section 6.6 on creating a new categorical variable with the ifelse command, using the owl data.

6.4 More on Functions and the if Statement

In the following we discuss passing multiple arguments out of a function and the ifelse command, with the help of a multivariate dataset. The Dutch government institute RIKZ carried out a marine benthic sampling program in the summer of 2002. Data on approximately 75 marine benthic species were collected at 45 sites on nine beaches along the Dutch coastline. Further information on these data and results of statistical analyses such as linear regression, generalised additive modelling, and linear mixed effects modelling, can be found in Zuur et al. (2007, 2009).

The data matrix consists of 45 rows (sites) and 88 columns (75 species and 13 explanatory variables). You could apply multivariate analysis techniques to see which species co-occur, which sites are similar in species composition, and which environmental variables are driving the species abundances. However, before doing any of this, you may want to start simply, by calculating a diversity index and relating this index to the explanatory variables.

A diversity index means that, for each site, you will characterise the 75 species with a single value. There are different ways of doing this, and Magurran (2004) describes various diversity indices. We do not want to engage in a discussion of which is better. You only need to develop an R function that takes as input an observation-by-species matrix, potentially with missing values, and a variable that tells the function which diversity index to calculate. To keep it simple, we limit the code to three indices. Interested readers can extend this R function and add their own favourite diversity indices. The three indices we use are:

1. Total abundance per site.
2. Species richness, defined as the number of different species per site.
3. The Shannon index. This takes into account both the presence/absence nature of the data and the actual abundance. It is defined by

$$H_i = -\sum_i^m p_{ij} \times \log_{10} p_{ij}$$

p_{ij} is calculated by

$$p_{ij} = \frac{Y_{ij}}{\sum_{j=1}^{n} Y_{ij}}$$

where p_{ij} is the proportion of a particular species j at site i, and m (in the first equation) is the total number of species. The total number of species is n.

6.4.1 Playing the Architect Again

Just as with the previous example presented in this chapter, begin by making a sketch of the tasks to be carried out.

1. Import the data and investigate what you have in terms of types of variables, variable names, dimension of the data, and so on.
2. Calculate total abundance for site 1. Repeat this for site 2. Automate this process, making the code as general as possible. Use elegant and efficient coding.
3. Calculate the different number of species for site 1. Repeat this process for site 2. Automate this process, and make the code as general as possible.
4. Do the same for the Shannon index.
5. Combine the code, and use an `if` statement to choose between the indices. Use elegant coding.
6. Put all the code in a function and allow the user to specify the data and the diversity index. The function should return the actual index and also indicate which diversity index was chosen (as a string).

In the following, we transform this sketch into fully working R code.

6.4.2 Step 1: Importing and Assessing the Data

Import the RIKZ data, separate the species data from the environmental data, and determine the size of the data with the following R code.

```
> Benthic <- read.table("C:/RBook/RIKZ.txt",
                        header = TRUE)
> Species <- Benthic[ , 2:76]
> n <- dim(Species)
> n
[1] 45 75
```

The first column in the data frame `Benthic` contains labels, columns 2–76 contain species data, and columns 77–86 are the explanatory variables. The

species data are extracted and stored in the data frame Species. Its dimension is 45 rows and 75 columns, and these values are obtained and stored in n using the dim command. To save space, results of the names and str command are not shown here; all variables are coded numerically.

6.4.3 Step 2: Total Abundance per Site

Calculate the sum of all species at site 1 by using

```
> sum(Species[1, ], na.rm = TRUE)

[1] 143
```

The total number of species at site 1 is 143. The same can be done for site 2:

```
> sum(Species[2, ], na.rm = TRUE)

[1] 52
```

To avoid typing this command 45 times, construct a loop that calculates the sum of all species per site. Obviously, we need to store these values. The following code does this.

```
> TA <- vector(length = n[1])
> for (i in 1:n[1]){
    TA[i] <- sum(Species[i, ], na.rm = TRUE)
  }
```

The vector TA is of length 45 and contains the sum of all species per site:

```
> TA

 [1] 143  52  70 199 67 944 241 192 211 48 35
[12]   1  47  38  10  1  47  73   8  48  6 42
[23]  29   0  43  33 34  67  46   5   7  1  1
[34] 102 352   6  99 27  85   0  19  34 23  0
[45]  11
```

Three sites have no species at all, whereas at one site the total abundance is 944. Note that you must define TA as a vector of length 45 before constructing the loop or TA[i] will give an error message (see the code above). You also need to ensure that the index i in the loop is indeed between 1 and 45; T[46] is not defined. Instead of using length = 45 in

the vector command, we used `length = n[1]`; remember that the task was to make the code as general as possible. The loop is what we call the brute force approach, as more elegant programming, producing identical results, is given by:

```
> TA <- rowSums(Species, na.rm = TRUE)
> TA

 [1] 143  52 70 199 67 944 241 192 211 48 35
[12]   1  47 38  10  1  47  73   8  48  6 42
[23]  29   0 43  33 34  67  46   5   7  1  1
[34] 102 352  6  99 27  85   0  19  34 23  0
[45]  11
```

The `rowSums` command takes the sum for each row. Note that this requires only one line of coding and also involves less computing time (albeit for such a small dataset the difference is very small), and is preferable to the loop.

6.4.4 Step 3: Richness per Site

The number of species at site 1 is given by

```
> sum(Species[1, ] > 0, na.rm = TRUE)

[1] 11
```

There are 11 different species at site 1. `Species[1, ] > 0` creates a Boolean vector of length 75 with elements `TRUE` and `FALSE`. The function `sum` converts the value `TRUE` to 1, and `FALSE` to 0, and adding these values does the rest.

For site 2, use

```
> sum(Species[2, ] > 0, na.rm = TRUE)

[1] 10
```

To calculate the richness at each site, create a loop as for total abundance. First define a vector `Richness` of length 45, then execute a loop from 1 to 45. For each site, richness is determined and stored.

```
> Richness <- vector(length = n[1])
> for (i in 1:n[1]){
```

```
   Richness[i] <- sum(Species[i, ] > 0, na.rm = TRUE)
  }
> Richness

 [1] 11 10 13 11 10 8  9 8 19 17 6  1 4 3 3
[16]  1  3  3  1  4 3 22 6  0  6 5  4 1 6 4
[31]  2  1  1  3  4 3  5 7  5  0 7 11 3 0 2
```

The elegant approach uses the `rowSums` command and gives the same result:

```
> Richness <- rowSums(Species > 0, na.rm = TRUE)
> Richness

 [1] 11 10 13 11 10 8  9 8 19 17 6  1 4 3 3
[16]  1  3  3  1  4 3 22 6  0  6 5  4 1 6 4
[31]  2  1  1  3  4 3  5 7  5  0 7 11 3 0 2
```

6.4.5 Step 4: Shannon Index per Site

To calculate the Shannon index, we need only three lines of elegant R code that include the equations of the index:

```
> RS <- rowSums(Species, na.rm = TRUE)
> prop <- Species / RS
> H <- -rowSums(prop * log10(prop), na.rm = TRUE)
> H

 [1]  0.76190639  0.72097224  0.84673524
 [4]  0.53083926  0.74413939  0.12513164
 [7]  0.40192006  0.29160667  1.01888185
[10]  0.99664096  0.59084434  0.00000000
              < Cut to reduce space>
```

We could have used code with a loop instead. The calculation can be done even faster with the function "`diversity`", which can be found in the `vegan` package in R. This package is not part of the base installation; to install it, see Chapter 1. Once installed, the following code can be used.

```
> library(vegan)
> H <- diversity(Species)
> H
```

```
        1         2         3         4         5
1.7543543 1.6600999 1.9496799 1.2223026 1.7134443
        6         7         8         9        10
0.2881262 0.9254551 0.6714492 2.3460622 2.2948506
       11        12        13        14        15
1.3604694 0.0000000 0.4511112 0.5939732 0.9433484
       16        17        18        19        20
0.0000000 0.7730166 0.1975696 0.0000000 0.8627246
                < Cut to reduce space>
```

Note that the values are different. The `diversity` help file shows that this function uses the natural logarithmic transformation, whereas we used the logarithm with base 10. The `diversity` help file gives instructions for changing this when appropriate.

A limitation of using the `vegan` package is that this package must be installed on the computer of the user of your code.

6.4.6 Step 5: Combining Code

Enter the code for all three indices and use an `if` statement to select a particular index.

```
> Choice <- "Richness"
> if (Choice == "Richness") {
     Index <- rowSums(Species >0, na.rm = TRUE)}
> if (Choice == "Total Abundance") {
     Index <- rowSums(Species, na.rm = TRUE) }
> if (Choice=="Shannon") {
     RS <- rowSums(Species, na.rm = TRUE)
     prop <- Species / RS
     Index <- -rowSums(prop*log10(prop), na.rm = TRUE)}
```

Just change the value of `Choice` to`'' Total Abundance''` or" `Shannon''` to calculate the other indices.

6.4.7 Step 6: Putting the Code into a Function

You can now combine all the code into one function and ensure that the appropriate index is calculated and returned to the user. The following code does this.

```
Index.function <- function(Spec, Choice1){
  if (Choice1 == "Richness") {
```

```
    Index <- rowSums(Spec > 0, na.rm = TRUE)}
  if (Choice1 == "Total Abundance") {
    Index <- rowSums(Spec, na.rm = TRUE) }
  if (Choice1 == "Shannon") {
    RS <- rowSums(Spec, na.rm = TRUE)
    prop <- Spec / RS
    Index <- -rowSums(prop * log10(prop),
                      na.rm = TRUE)}
  list(Index = Index, MyChoice = Choice1)
  }
```

The if statement ensures that only one index is calculated. For small datasets, you could calculate them all, but for larger datasets this is not good practice. Before executing the code, it may be wise to ensure that none of the variables within the function also exists outside the function. If they do, remove them with the rm command (see Chapter 1), or quit and restart R. We renamed all input variables so that no duplication of variable names is possible. In order to execute the function, copy the code for the function, paste it into the console, and type the command:

```
> Index.function(Species, "Shannon")

$Index

 [1] 0.76190639 0.72097224 0.84673524 0.53083926
 [5] 0.74413939 0.12513164 0.40192006 0.29160667
 [9] 1.01888185 0.99664096 0.59084434 0.00000000
[13] 0.19591509 0.25795928 0.40969100 0.00000000
[17] 0.33571686 0.08580337 0.00000000 0.37467654
[21] 0.37677792 1.23972435 0.62665477 0.00000000
[25] 0.35252466 0.39057516 0.38359186 0.00000000
[29] 0.58227815 0.57855801 0.17811125 0.00000000
[33] 0.00000000 0.12082909 0.08488495 0.43924729
[37] 0.56065567 0.73993117 0.20525195 0.00000000
[41] 0.65737571 0.75199627 0.45767851 0.00000000
[45] 0.25447599

$MyChoice

[1] "Shannon"
```

Note that the function returns information from of the final command, which in this case is a list command. Recall from Chapter 2 that a list allows us to combine data of different dimensions, in this case a variable with 45 values and also the selected index.

Is this function perfect? The answer is no, as can be verified by typing

```
> Index.function(Species, "total abundance")
```

The error message produced by R is

```
Error in Index.function(Species, "total abundance"):
   object "Index" not found
```

Note that we made a typing error in not capitalizing "total abundance". In the previous section, we discussed how to avoid such errors. We extend the function so that it inspects all `if` statements and, if none of them is executed, gives a warning message. We can use the `if else` command for this.

```
Index.function <- function(Spec,Choice1){
  if (Choice1 == "Richness") {
    Index <- rowSums(Spec > 0, na.rm = TRUE) } else
  if (Choice1 == "Total Abundance") {
    Index <- rowSums(Spec, na.rm = TRUE) } else
  if (Choice1 == "Shannon") {
    RS <- rowSums(Spec, na.rm = TRUE)
    prop <- Spec / RS
    Index <- -rowSums(prop*log(prop),na.rm=TRUE)} else {
       print("Check your choice")
       Index <- NA }
  list(Index = Index, MyChoice = Choice1)}
```

R will look at the first `if` command, and, if the argument is `FALSE`, it will go to the second `if` statement, and so on. If the variable `Choice1` is not equal to "Richness", "Total Abundance", or "Shannon", the function will execute the command,

```
print("Check your choice")
Index <- NA
```

You can replace the text inside the `print` command with anything appropriate. It is also possible to use the `stop` command to halt R. This is useful if the function is part of a larger calculation process, for example, a bootstrap procedure. See the help files on `stop`, `break`, `geterrmessage`, or `warning`. These will help you to create specific actions to deal with unexpected errors in your code.

Table 6.1 R functions introduced in this chapter

Function	Purpose	Example
`jpeg`	Opens a jpg file	`jpeg(file = "AnyName.jpg")`
`dev.off`	Closes the jpg file	`dev.off()`
`function`	Makes a function	`z <- function(x, y){ }`
`paste`	Concatenates variables as characters	`paste("a", "b", sep = " ")`
`if`	Conditional statement	`if (a) { x<-1 }`
`ifelse`	Conditional statement	`ifelse (a, x<-1, x<-2)`
`if elseif`	Conditional statement	`if (a) { x<-1 } elseif (b) { x<-2 }`

6.5 Which R Functions Did We Learn?

Table 6.1 shows the R functions that were introduced in this chapter.

6.6 Exercises

Exercise 1. Using a loop to plot temperature per location.

In Section 6.2, sibling negotiation behaviour was plotted versus arrival time for each nest in the owl data. A graph for each nest was created and saved as a jpg file. Do the same for the temperature data; see Exercise 4.1 for details. The file *temperature.xls* contains temperature observations made at 31 locations (denoted as stations in the spreadsheet) along the Dutch coastline. Plot the temperature data versus time for each station, and save the graph as a jpg file.

Exercise 2. Using the `ifelse` command for the owl data.

The owl data were sampled on two consecutive nights. If you select the data from one nest, the observations will cover both nights. The two nights differed as to the feeding regime (satiated or deprived). To see observations from a single night, select all observations from a particular nest and food treatment. Use the `ifelse` and `paste` functions to make a new categorical variable that defines the observations from a single night at a particular nest. Try rerunning the code from Exercise 1 to make a graph of sibling negotiation versus arrival time for observations of the same nest and night.

Exercise 3. Using the `function` and `if` commands with the benthic dataset.

In this exercise we provide the steps for the function that was presented in Section 6.4: the calculation of diversity indices. Read the introductory text in Section 6.4 on diversity indices. Import the benthic data and extract columns 2–76; these are the species.

Calculate total abundance at site 1. Calculate total abundance at site 2. Calculate total abundance at site 3. Calculate the total abundance at site 45. Find a function that can do this in one step (sum per row). Brute force may work as well (`loop`), but is less elegant.

Calculate the total number of *different* species in site 1 (species richness). Calculate species richness for site 2. Do the same for sites 3 and 45. Find a function that can do this in one step.

Create a function using the code for all the diversity indices. Make sure that the user can choose which index is calculated. Ensure that the code can deal with missing values.

If you are brave, add the Shannon index. Apply the same function to the vegetation data.