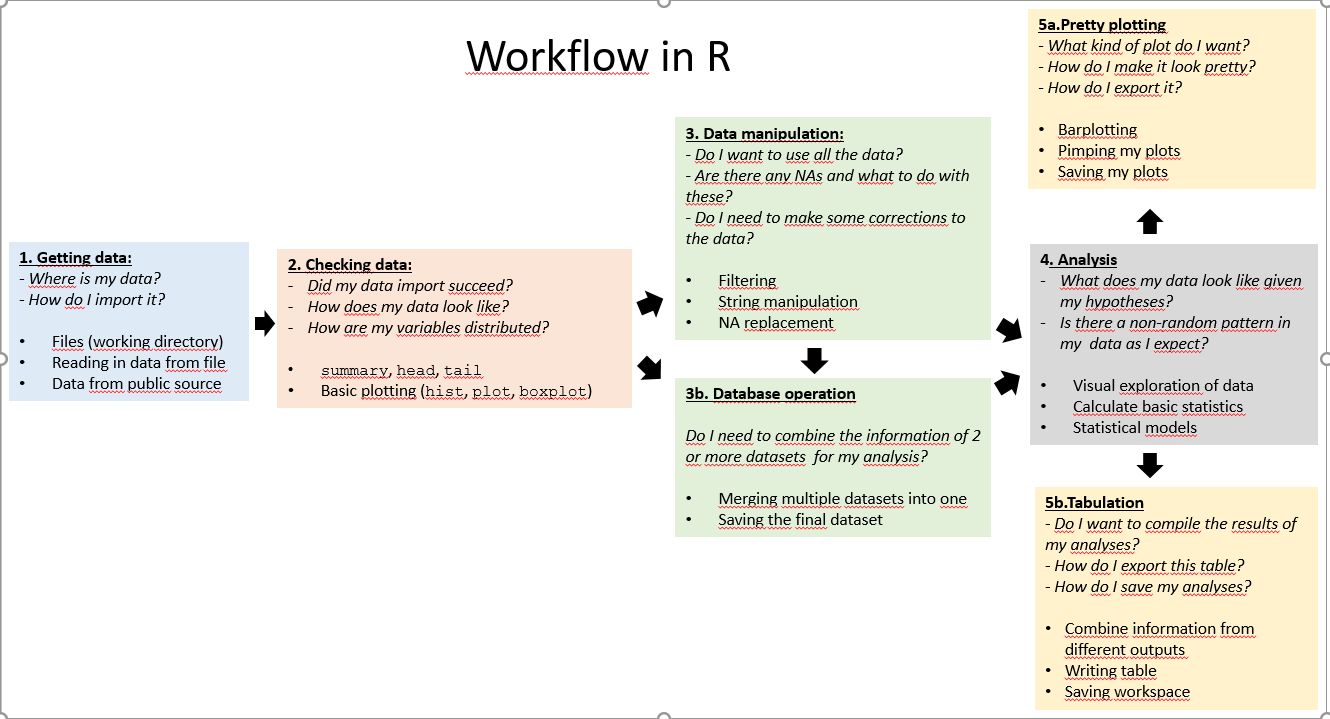
# Part II: R workflow: starting to do stuff with your data

In this second part, we start by explicitly introducing the R workflow (Figure 5.0). The workflow is perhaps the main advantage of working in R. The capacity to get your information (data) into R and then perform all subsequent steps in R, e.g. quick plotting, calculations, doing statistical analyses, making publication-quality graphs. In part I, we literally raced through an entire workflow, from getting the data (operation 1) to checking the data (operation 2) with some analyses (operation 4), albeit a very restricted workflow.



*Figure 5.0: Flow diagram of five distinct sets of operations (in different colors) which together define a workflow of many typical coding jobs. The diagram is not exhaustive, however, and both less and more operations may be required.*

Given data, the R script allows you to keep the source information in the original shape, read it into R’s “brain” for further processing (filtering, analyses) and prepare high-quality output. The original data and your script together provide a reproducible and transparent record of what was done to arrive at the final output. Setting up a workflow through coding may seem cumbersome, but has a number of advantages. Let’s consider, for example, data processing. Essentially all data science work requires what is called data manipulation (e.g. filtering), where you start with the original data and prepare it for the analysis you want to perform. Different analyses typically require slightly different data manipulation. A major advantage of coding is that you have a record of what steps were taken. If you prepare your data for analysis using a manual “clicking” approach (e.g. in a spreadsheet program), your actions leave no record and you may end up with many datafiles for different analyses that slightly differ from each other in ways that is difficult to remember. Given you have coded your workflow, you can (1) easily make adjustments if you realize something was not correct and re-run the workflow. (2) As long as the original data file is kept intact, you can run the script to produce exactly the same outcome at any point in time. (3) You can share your code with others who can then follow exactly what you did and who can also add to it; thus, your workflow is open and transparent. By getting to work in R you hence gain flexibility, reproducibility and transparency.

In this second part, we look in more depth at operations 3 (data manipulation), and 4 (analysis) in the R workflow (Figure 5.0).

## Chapter 5. Re-cap on your script and the class of R objects

### 5.1 Structure your script

Some general observations on coding to start with. Your R script is a program coded in R language. R is a platform for programming. A program is a list of instructions to be carried out by the computer. It is like a recipe for when you are cooking.

1. Mix 6 dl flower with 2.5 dl milk, some salt and 3 eggs.

2. Let the batter stand for some minutes while you heat up a frying pan

3. Bake your pancakes in butter.

The programs in R are, of course, for solving numerical problems. A program has a logical sequence for carrying out the instructions, which you can often group in sections. The next section (next set of instructions) often depends on the previous section being completed. For example, you first must read in the data before you can plot it. Annotation (use of comments behind the hashtag #) is a way to clarify the different sections in your code. Actually, even before writing any R code, you can use annotations to plan the different steps your script should be doing. In general, it is always clever to annotate as if you would be writing the code for a person without a clue of what the program will be doing. This is because it is easy to forget what the lines were about even if you have spent a lot of time working out how to code it, you may not remember it after days/weeks/months. Annotation can be added prior or behind the line with code. You can visually structure your code by creating ‘blocks’ using various symbols and white-lines.

This is especially handy for larger programs that do multiple things in a row. For example

#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#\*\* This code reads in data and does something complicated,

#\*\* plots and returns a value

#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#----- SECTION 1 ----------

# Reading in the data #####

#data consists of the following variables

#1. Name

#2. Sex

#3. Score

… code …

#------ SECTION 2 ----------------------------------------

# Getting the first really difficult calculations done ###

# defining the variables needed

… code …

# calculating their fancy statistical thingy

… code …

#------ SECTION 3 ----------------------------------------

# Plotting to see if it all makes sense ###

.. code code code ..

Another aspect which helps to increase clarity is to choose variable names which have an intuitive meaning. Thus, avoid having a, b, c, d; instead these could be data, regressionModel, populationSize, outputAnalysis, etc. R is really specific in reading what you type in; it generally does not guess about your meaning and also takes into account upper and lower cases. This applies to functions, but also to the objects you make yourself. So, the object numberOfStudents is something else for R than numberofstudents. It is therefore handy to be consistent in your “style”. If you for example have a data file read into R as an object called myData, and the columns of your datafile have names, it is clever to either have them all start with a capital (or then not, as long as you are consistent). These columns can be accessed by the dollar sign $. So, you will have data$Sex and data$Name and not some kind of mix (e.g. data$Sex and data$name). This avoids having to guess and scrolling up and down to check how you defined variable names. Of course, if you use R Studio for coding in R, you will notice that R Studio will try to help you by giving suggestions, but consistency is still a clever habit.

Take home message: Think about *meaningful identifying names*, add *comments* and make a transparent *layout.* Importantly, a structured layout can help you plan the script before you start the “real” coding, as well as afterwards aid in understanding what was coded. A clearly structured script can be “read” by anyone with knowledge in R. Again, even when you are not coding the program for someone else but just for your own use, it is still very useful to have all this structure and explanation when you again look at the code some time later. It is surprising how quickly you forget what was done and why in your script.

### 5.2 More on R objects

R is an object oriented programming language. An object is anything saved in R’s brain. Easiest way to think of an object is that it is anything left of the assign operator “<-“. Objects have different classes. We have encountered some basic classes and in this section more detail is provided

### 5.2.1 Vector

A main trick in programming is learning to think in terms of *vectors* and *matrices*. A vector is a collection of numbers (R: class: “numeric”) or strings (R class “character”) or factorial levels (R class “factor”. It is written in text using square brackets. For example, the vector



is a row vector containing three numbers (1, 2 and 3). A vector can also be written as a column vector. For example,



In R, these are formed – for example – as c(1,2,3) for the vector *v*. R does not work with column vectors *per se*.

In R, we do not have single numbers, or scalars, or single pieces of text (strings), but all these are viewed as a vector. A matrix is, in a way, a collection of vectors stacked in rows or columns, and when you import data, you can interpret all your data for a given variable as a vector. In R *the ‘lowest level’ is always a vector*. The number (mathematicians would say, scalar) 1, for example, is in R simply a vector with only one number. It is the vector [1]. When programming, it is clever to start thinking in terms of vectors as much as possible. This is not always so easy. If we have data on individuals, and for each of these we have measured a lot of different variables, we tend to think in terms of these individuals (the rows of our datafile) rather than the data being a series of column vectors each containing the information of one particular aspect. Yet, this is in some ways how R sees it. This if why we have data$Sex and data$Name. The name of the individuals in the data is just one vector for R. Most of the basic operations which are of interest, such as filtering, plotting or calculating with your data, all involve operations on vectors. Thinking in vectors will therefore allow more efficient, but – more importantly – more transparent code.

Many built-in functions are what is called vectorized: The operation is applied to the entire vector or matrix in one go. For example,

> x<-c(1,2,3,4)

> 2\*x

[1] 2 4 6 8

### 5.2.2. Factor

A factorial variable in statistics is when a variable is considered to have levels (e.g. control and treatment). These may be ordinal (there is an order: low, medium, high) or nominal (no order: black, red, blue). R will, however, interpret all text and alphanumeric data that you read into R’s “brain” as characters.

> myData<-read.csv2(”MyData.csv”)

> str(myData)

‘data.frame’: 10 obs. of 4 variables:

$ individual: chr “A” “B” “C” “D” ...

$ colour : chr “blue” “blue” “red” “blue” ...

$ mass : num 1 1.2 1.7 1.4 1.5 1.1 1.9 1.5 1.3 1.4

$ length : int 10 9 5 9 12 13 12 7 8 10

You may want to use a vector as a factorial vector. A character or numerical vector can be coerced into a factorial vector using the command as.factor().

> as.factor(c(“a”,”b”,”c”))

[1] a b c

Levels: a b c

> as.factor(c(1,2,3))

[1] 1 2 3

Levels: 1 2 3

Of course, another option is to read in the data specifying that you want all strings to be considered as factors. Consider again the myDataF object from Part I.

> myDataF<-read.csv2(”MyData.csv”, stringAsFactors=TRUE)

> str(myDataF)

‘data.frame’: 10 obs. of 4 variables:

$ individual: Factor w/ 10 levels “A”,”B”,”C”,”D”,..: 1 2 3 4 5 6 7 8 9 10

$ colour : Factor w/ 2 levels “blue”,”red”: 1 1 2 1 1 2 2 1 1 2

$ mass : num 1 1.2 1.7 1.4 1.5 1.1 1.9 1.5 1.3 1.4

$ length : int 10 9 5 9 12 13 12 7 8 10

R will code factors as 1,2,3,… in alphabetic order (which may not be the order in your data or the order you want: You can of course change the levels in R; see the help for levels()). A factorial vector can be coerced to be a character or numeric vector

> as.character(myDataF$colour)

[1] “blue” “blue” “red” “blue” “blue” “red” “red” “blue” “blue” “red”

> as.numeric(myDataF$colour)

[1] 1 1 2 1 1 2 2 1 1 2

Hence, when a factorial vector is changed to numeric, R interprets it in terms of the levels.

### 5.2.3. matrix

We have in Part I encountered this class. It is used for storage in two dimensions of information where all elements need to be either numeric or character. For example,

> matrix(c(1,2,3,4),2,2)

[,1] [,2]

[1,] 1 3

[2,] 2 4

stores the number 1 to 4 in a 2 x 2 matrix. It may seem redundant to specify the matrix has to be 2 x 2 given we provide a vector with 4 numbers. However, this specification is needed because R will recycle the information provided to fill the matrix to be the dimension specified. For example, storing the numbers 1 to 4 in a 2 x 3 matrix produces

matrix(c(1,2,3,4),2,3)

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

[1,] 1 3 1

[2,] 2 4 2

Warning message:

In matrix(c(1, 2, 3, 4), 2, 3) :

data length [4] is not a sub-multiple or multiple of the number of columns [3]

Note that R produces a warning, yet still performs the operation. Warnings like this are typical of R; its warns your request is odd although it will comply with it. Recycling is handy in some cases. For example, it allows easily creating a 2x2 matrix of ones

> matrix(1,2,2)

[,1] [,2]

[1,] 1 1

[2,] 1 1

A second aspect to note is that R will by default fill the matrix by columns, not by rows. Bringing up the help for matrix explains

matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE,

dimnames = NULL)

as.matrix(x, ...)

## S3 method for class 'data.frame'

as.matrix(x, rownames.force = NA, ...)

is.matrix(x)

**Arguments**

|  |  |
| --- | --- |
| data | an optional data vector (including a list or [expression](http://127.0.0.1:13623/library/base/help/expression) vector). Non-atomic classed **R** objects are coerced by [as.vector](http://127.0.0.1:13623/library/base/help/as.vector) and all attributes discarded. |
| nrow | the desired number of rows. |
| ncol | the desired number of columns. |
| byrow | logical. If FALSE (the default) the matrix is filled by columns, otherwise the matrix is filled by rows. |

Thus, we need to override the default byrow=FALSE and specify byrow=TRUE in case we wish to fill the matrix by rows.

> matrix(c(1,2,3,4),2,2, byrow=TRUE)

[,1] [,2]

[1,] 1 2

[2,] 3 4

### 5.2.4 data.frame

We have seen that data is read into R as an object of class data.frame. A data.frame object can store information in two dimensions, but can hold both numeric and characters. We can also make a data.frame using the function (surprise!) data.frame(). For example, using vectors

> students<-c(“Bill”, “Miranda”)

> ages<-c(22,21)

> df1<-data.frame(students,ages)

> df1

students ages

1 Bill 22

2 Miranda 21

> str(df1)

'data.frame': 2 obs. of 2 variables:

$ students: chr "Bill" "Miranda"

$ ages : num 22 21

The above first used c() to make two vector objects (students and ages) and then combines these using data.frame() in a data.frame. The data.frame df1 now consists of a character variable and of a numeric one. The names of these variables in the data.frame (R uses the term *tag* for these “variable names”) are taken to be the names of the two vectors (students and ages). Consulting the help for data.frame explains more details

**Usage**

data.frame(..., row.names = NULL, check.rows = FALSE,

check.names = TRUE, fix.empty.names = TRUE,

stringsAsFactors = default.stringsAsFactors())

**Arguments**

|  |  |
| --- | --- |
| ... | these arguments are of either the form value or tag = value. Component names are created based on the tag (if present) or the deparsed argument itself. |

---

|  |  |
| --- | --- |
| stringsAsFactors | logical: should character vectors be converted to factors? The ‘factory-fresh’ default is FALSE, but this can be changed by setting [options](http://127.0.0.1:13623/library/base/help/options)(stringsAsFactors = TRUE). |

From this help info, we can see that stringsAsFactors is also at play here. Similarly to when we are reading in data, the function data.frame() will assume that stringsAsFactors = FALSE. To see what happens if we override the default by specifying stringsAsFactors = TRUE.

> df2<-data.frame(students,ages,stringsAsFactors = TRUE)

> df2

students ages

1 Bill 22

2 Miranda 21

> str(df2)

'data.frame': 2 obs. of 2 variables:

$ students: Factor w/ 2 levels "Bill","Miranda": 1 2

$ ages : num 22 21

Now we have created a data.frame called df2 which converts the character vector students to a factorial vector with 2 levels; level 1 codes “Bill” and level 2 “Miranda”.

We can index a data.frame in multiple ways, similar to vectors and matrices

> df1[1]

students

1 Bill

2 Miranda

> df1[,1]

[1] "Bill" "Miranda"

> df1[,"students"]

[1] "Bill" "Miranda"

> df1$students

[1] "Bill" "Miranda"

From the above, note that the first option df1[1] returns the first column of df1 as a data.frame. The other options are three ways to return this column as a character vector. Of these three ways, using the $ sign is clearly the “safest” in the sense that using the name of the column (students) is clear whereas using the number of the column is more confusing coding. In R Studio, after you typed the dollar sign $, the program will also show you the different tags to choose from which reduces typos. On the other hand, in programming, especially if the script needs to be generic, indexing with numbers often is handy. Again, there are several ways to achieve the same outcome.

Once we have a data.frame, we can add information to it. The function rbind(), which stands for row bind, can be used to add rows. Consider for example the addition of data on an extra student to df1

> df1<-rbind(df1,data.frame(students="Laura",ages=19))

> df1

students ages

1 Bill 22

2 Miranda 21

3 Laura 19

> str(df1)

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

$ students: chr "Bill" "Miranda" "Laura"

$ ages : num 22 21 19

In the above, we have defined a new df1 to be the old df1 with an extra row. For adding an extra row to work properly, a data.frame with the same tags must be add ed to the existing data.frame. This is why data.frame(students=”Laura”,ages=19) is added to the existing data.frame df1. Not using the same tags (i.e. names for the variables), will produce an error.

> rbind(df1,data.frame(student=”Laura”,age=19))

Error in match.names(clabs, names(xi)) :

names do not match previous names

Suppose we want to add a new column based on the existing information, say, the difference in age of each student to age = 20. An easy way to add a new column to an existing data.frame is to directly define it using the $ sign. Thus,

> df1$ageDiff<-df1$ages-20

> df1

students ages ageDiff

1 Bill 22 2

2 Miranda 21 1

3 Laura 19 -1

## Chapter 6. Elementary filtering of data

After reading in the data, one often needs to change some aspects of it. In data science, this is called data manipulation. The term “data manipulation” sounds like a bad, unscientific idea. The term is used, however, to describe operations in which your dataset is altered as to become suited for analysis. Alteration include selecting, inserting, deleting and updating data. One example of data manipulation we performed in the previous sections was “extracting” part of the data using indexing. Data manipulation approaches are often needed before analyses are carried out. At the very least, some kind of filtering is often required.

### 6.1. ifelse() and relational operators

Conditional statement can be used to manipulate data. These are instructions like “*If* you have fresh tomatoes, boil these shortly, peel off their skin, slice them and fry them, *else* open a can of tomatoes and fry its content.” There clearly can also be more options. R uses the Booleans TRUE and FALSE to distinguish statements. For example, the above text can be translated to R using the function ifelse(), which consists of (1) a logical statement (“If you have tomatoes…”), which (2) if TRUE, performs a certain instruction (boil, peel, slice, fry), but (3) if FALSE carries out another (open a can and fry its content). For example

> ifelse(myData$colour==”blue”,”blue”,”not blue”)

[1] “blue” “blue” “not blue” “blue” “blue” “not blue” “not blue” “blue”

[9] “blue” “not blue”

returns a vector of “blue” and “not blue” for blue and non-blue individuals respectively. Note the double ‘equal to’ mark == which is used by R to distinguish ‘=’ (which you can use to define an argument or an object) from ‘is equal to?’. Ask help for == to get an overview of relational operators

*Box 6.1. Help page for == (shortened)*

|  |  |
| --- | --- |
| Comparison {base} | R Documentation |

**Relational Operators**

**Description**

Binary operators which allow the comparison of values in atomic vectors.

**Usage**

x < y

x > y

x <= y

x >= y

x == y

x != y

**Arguments**

|  |  |
| --- | --- |
| x, y | atomic vectors, symbols, calls, or other objects for which methods have been written. |

Thus, in the call to ifelse(), the first argument specifies a logical vector

> myData$colour=="blue"

[1] TRUE TRUE FALSE TRUE TRUE FALSE FALSE TRUE TRUE FALSE

The second argument to ifelse()then specifies for each element in this vector what the output should be if TRUE; the second argument specifies the output if FALSE. That is, ”blue” and ”not blue” respectively.

The above displayed help (Box 6.1) specifies, among others, the use of the exclamation mark ! to indicate negation (i.e. the “reverse” or NOT != means “not equal to”).

> 3==3

[1] TRUE

> 3!=3

[1] FALSE

The use of ! as a reversal works for all logical vectors.

> c(T,T,F,F)

[1] TRUE TRUE FALSE FALSE

> !c(T,T,F,F)

[1] FALSE FALSE TRUE TRUE

The ifelse() operation can be used to recode data. For example to add an extra column where the colour information is recoded

> myData$colour.recoded<-ifelse(myData$colour==”blue”,”blue”,”not blue”)

> myData$colour.recoded

[1] “blue” “blue” “not blue” “blue” “blue” “not blue” “not blue” “blue”

[9] “blue” “not blue”

Here, the vector myData$colour was recoded into a vector of strings myData$colour.recoded.

The conditional statement allows us to recode columns in the data.frame in one line. For example, all individuals with mass exceeding 1.3 are to be left untouched, or if below or equal to 1.3 are coded as NA

> ifelse(myData$mass>1.3,myData$mass,NA)

[1] NA NA 1.7 1.4 1.5 NA 1.9 1.5 NA 1.4

Also a factorial vector can be recoded using the above formulation. Consider, for example, the data.frame myDataF which resulted from reading in the datafile with the argument stringsAsFactor=TRUE. We can use ifelse()to recode the factorial vector

> myDataF$colour

[1] blue blue red blue blue red red blue blue red

Levels: blue red

> ifelse(myDataF$colour=="blue","blue","not blue")

[1] "blue" "blue" "not blue" "blue" "blue" "not blue"

[7] "not blue" "blue" "blue" "not blue"

Note, however, that this operation produces a character vector (and not a factorial vector)

> class(ifelse(myDataF$colour=="blue","blue","not blue"))

[1] "character"

### 6.2 subset() and relational operators

We can use indexing in combination with a relational operator to extract data on the blue individuals from the object myData

> myData[myData$colour==”blue”,]

individual colour mass length colour.recoded

1 A blue 1.0 10 blue

2 B blue 1.2 9 blue

4 D blue 1.4 9 blue

5 E blue 1.5 12 blue

8 H blue 1.5 7 blue

9 I blue 1.3 8 blue

The convenience function in R for this kind of operations is subset(): The help for this function is shown in Box 6.2.1.

The above subsetting of myData can hence be alternatively performed by using the function subset() and introducing the conditional as the second argument (subset)

Box 6.2.1. Help page for subset (shortened)

|  |  |
| --- | --- |
| subset {base} | R Documentation |

**Subsetting Vectors, Matrices and Data Frames**

**Description**

Return subsets of vectors, matrices or data frames which meet conditions.

**Usage**

subset(x, ...)

## Default S3 method:

subset(x, subset, ...)

## S3 method for class 'matrix'

subset(x, subset, select, drop = FALSE, ...)

## S3 method for class 'data.frame'

subset(x, subset, select, drop = FALSE, ...)

**Arguments**

|  |  |
| --- | --- |
| x | object to be subsetted. |
| subset | logical expression indicating elements or rows to keep: missing values are taken as false. |
| select | expression, indicating columns to select from a data frame. |
| drop | passed on to [ indexing operator. |
| ... | further arguments to be passed to or from other methods. |

> subset(myData,colour==”blue”)

individual colour mass length colour.recoded

1 A blue 1.0 10 blue

2 B blue 1.2 9 blue

4 D blue 1.4 9 blue

5 E blue 1.5 12 blue

8 H blue 1.5 7 blue

9 I blue 1.3 8 blue

We may also wish to use the third argument (select) to restrict the columns. Perhaps to only include individual, mass and length

> subset(myData,colour==”blue”, c(individual,mass,length))

individual mass length

1 A 1.0 10

2 B 1.2 9

4 D 1.4 9

5 E 1.5 12

8 H 1.5 7

9 I 1.3 8

The last argument (drop) is specified to by default be FALSE. This means that the class of the object returned by subset will be the same as the object it is given. Thus,

> subset(myData,colour==”blue”,individual)

individual

1 A

2 B

4 D

5 E

8 H

9 I

returns a data.frame, whereas

> myData[myData$colour=="blue","individual"]

[1] "A" "B" "D" "E" "H" "I"

returns a vector.If we would want the call to subset() to return a vector, the specification drop = TRUE should be specified, thereby allowing a “drop” in class from data.frame to (in this case) a vector.

> subset(myData,colour==”blue”,individual, drop=TRUE)

[1] A B D E H I

Levels: A B C D E F G H I J

The drop argument can be used in a number of functions. It can also be used when indexing

> myData[myData$colour=="blue","individual",drop=F]

individual

1 A

2 B

4 D

5 E

8 H

9 I

Hence, the drop argument allows control over the class of the output produced.

Conditional statements can be combined using the function & (stands for “and”). Statements then need to be placed inside brackets.

> subset(myData,(colour==”blue”)&(mass>1.3),c(individual,mass))

individual mass

4 D 1.4

5 E 1.5

8 H 1.5

or the function | (typically termed “bar”, which stands for “or”)

> subset(myData,(colour==”blue”)|(mass>1.3),c(individual,colour,mass))

individual colour mass

1 A blue 1.0

2 B blue 1.2

3 C red 1.7

4 D blue 1.4

5 E blue 1.5

7 G red 1.9

8 H blue 1.5

9 I blue 1.3

10 J red 1.4

Consult the help for e.g. & (or any other logical operator) to obtain a list of logical operators

|  |  |
| --- | --- |
| Box 6.2.2 Help for & (shortened)  Logic {base} | R Documentation |

**Logical Operators**

**Description**

These operators act on raw, logical and number-like vectors.

**Usage**

! x

x & y

x && y

x | y

x || y

xor(x, y)

---- shortened

Clearly, there are many logical operators (Box 6.2.2) and these operators cover various possible logical combinations. They return whether these combinations are TRUE or FALSE. Again, the and (&) operator returns TRUE when both values are TRUE (note that below T and F are shorthand for TRUE and FALSE)

> T&T

[1] TRUE

> T&F

[1] FALSE

If used with vectors with several elements as in the subset() examples above, the & operator will compare each element in the first vector to its corresponding element in the other

> c(T,T)&c(T,F)

[1] TRUE FALSE

By doubling the & operator, only the first element in the vectors are compared

> c(T,T)&&c(T,F)

[1] TRUE

In contrast, the operator or (|) will be TRUE if one of multiple conditions are TRUE

> T|T

[1] TRUE

> T|F

[1] TRUE

> F|T|F

[1] TRUE

Lastly, xor(x,y) is TRUE if either x or y is TRUE

> xor(T,T)

[1] FALSE

> xor(F,F)

[1] FALSE

> xor(T,F)

[1] TRUE

## Chapter 7. Elementary analysis: repeated calculations for subsets

In this chapter we make a start with analysis by introducing some basic R functions for calculating descriptive statistics.

### 7.1 aggregate()

Suppose we want to calculate mean of mass of blue and for red individuals separately. We could use subsetting (or indexing) to extract information on these two groups and calculate the means

> mean(subset(myData,colour==”blue”,mass)[,”mass”])

[1] 1.316667

> mean(subset(myData,colour==”red”,mass)[,”mass”])

[1] 1.525

In the above lines, we first subset the data to be only the mass of blue individuals (which creates a data.frame) and then also index this data.frame for column [,”mass”] to provide the function mean with the data as a vector. The alternative, as discussed in section 6.2, is to use the select argument for subset() to only return information on mass and specify drop=TRUE

> mean(subset(myData,colour==”blue”,mass,drop=TRUE))

[1] 1.316667

> mean(subset(myData,colour==”red”,mass,drop=TRUE))

[1] 1.525

Although the above formulations clearly work, it is handy to use one of R’s functions for repeating analysis for different subsets. For example, the function aggregate()(Box 7.1)

> aggregate(mass~colour, data=myData, FUN=mean)

colour mass

1 blue 1.316667

2 red 1.525000

applies a function (specified by argument FUN) to each subset of data. In our case, the function mean() is used for mass for each subset of the data specified by colour. In the above, we use the so-called “formula” implementation (the tilde ~; y~x indicates y as a function of x) as well as specifying in detail the second and third argument. The above can be shortened to

> aggregate(mass~colour, myData,mean)

colour mass

1 blue 1.316667

2 red 1.525000

The above work because, as shown in the help for aggregate() (Box 7.1), under the formula method, aggregate() expects the first argument to be the formula (mass~colour), the second argument to be the data.frame (myData) and the third argument to be the function (mean). Given that we provide the correct information in the correct order, it is not needed not specify the argument’s name. The help page (Box 7.1) also shows that aggregate() can take arguments in different ways. You can take some time to try these various approaches; they differ in subtle ways in the output they produce. For simplicity, we here stick to the formula way of using this function.

Box 7.1. Help for aggregate (shortened)

|  |  |
| --- | --- |
| aggregate {stats} | R Documentation |

**Compute Summary Statistics of Data Subsets**

**Description**

Splits the data into subsets, computes summary statistics for each, and returns the result in a convenient form.

**Usage**

aggregate(x, ...)

## Default S3 method:

aggregate(x, ...)

## S3 method for class 'data.frame'

aggregate(x, by, FUN, ..., simplify = TRUE, drop = TRUE)

## S3 method for class 'formula'

aggregate(formula, data, FUN, ...,

subset, na.action = na.omit)

## S3 method for class 'ts'

aggregate(x, nfrequency = 1, FUN = sum, ndeltat = 1,

ts.eps = getOption("ts.eps"), ...)

**Arguments**

|  |  |
| --- | --- |
| x | an R object. |
| by | a list of grouping elements, each as long as the variables in the data frame x. The elements are coerced to factors before use. |
| FUN | a function to compute the summary statistics which can be applied to all data subsets. |
| simplify | a logical indicating whether results should be simplified to a vector or matrix if possible. |
| drop | a logical indicating whether to drop unused combinations of grouping values. The non-default case drop=FALSE has been amended for **R** 3.5.0 to drop unused combinations. |
| formula | a [formula](http://127.0.0.1:40914/help/library/stats/help/formula), such as y ~ x or cbind(y1, y2) ~ x1 + x2, where the y variables are numeric data to be split into groups according to the grouping x variables (usually factors). |
| data | a data frame (or list) from which the variables in formula should be taken. |
| subset | an optional vector specifying a subset of observations to be used. |

Suppose we have an additional categorical variable, habitat, which we add

> myData$habitat<-c(rep(”sunny”,5),rep(”shady”,5))

> myData

individual colour mass length colour.recoded habitat

1 A blue 1.0 10 blue sunny

2 B blue 1.2 9 blue sunny

3 C red 1.7 5 not blue sunny

4 D blue 1.4 9 blue sunny

5 E blue 1.5 12 blue sunny

6 F red 1.1 13 not blue shady

7 G red 1.9 12 not blue shady

8 H blue 1.5 7 blue shady

9 I blue 1.3 8 blue shady

10 J red 1.4 10 not blue shady

We then can calculate means in mass for all combinations of both these categories

> aggregate(mass~colour+habitat, myData,mean)

colour habitat mass

1 blue shady 1.400000

2 red shady 1.466667

3 blue sunny 1.275000

4 red sunny 1.700000

where + is used to add the second category in the formula (Box 7.1).

### 7.2 tapply()

Another useful function to perform similar operations as aggregate() is tapply(), pronounced as t-apply; it is one function in the so-called apply() family to which also aggregate() belongs. Investigation of its help (Box 7.2) shows that we cannot use the formula syntax for tapply() but have to pass it vectors

> tapply(myData$mass,myData$colour,mean)

blue red

1.316667 1.525000

> tapply(myData[,”mass”],myData$colour,mean)

blue red

1.316667 1.525000

The help page for tapply() is rather technical (Box 7.2). It includes the term “ragged array”. We will later get more familiar objects of class array. For the present, it is sufficient to realize that tapply() is a fairly general function which will produce an object of class array

> means.per.colour<-tapply(myData[,”mass”],myData$colour,mean)

> class(means.per.colour)

[1] “array”

To specify that the function is to be applied along multiple categories, the help (Box 7.2) clarifies that the second argument INDEX is to be provided as a list. A list is another basic class in R. Later on in the compendium we will take a closer look at objects of this class. For now, however, it is sufficient to realize that the function list() will bundle multiple vectors in a list. Giving such a list as the second argument then produces the descriptive statistics.

> tapply(myData$mass, list(myData$colour,myData$habitat), mean)

shady sunny

blue 1.400000 1.275

red 1.466667 1.700

Box 7.2. Help page for tapply (shortened)

|  |  |
| --- | --- |
| tapply {base} | R Documentation |

**Apply a Function Over a Ragged Array**

**Description**

Apply a function to each cell of a ragged array, that is to each (non-empty) group of values given by a unique combination of the levels of certain factors.

**Usage**

tapply(X, INDEX, FUN = NULL, ..., default = NA, simplify = TRUE)

**Arguments**

|  |  |
| --- | --- |
| X | an **R** object for which a [split](http://127.0.0.1:40914/help/library/base/help/split) method exists. Typically vector-like, allowing subsetting with [[](http://127.0.0.1:40914/help/library/base/help/%5b). |
| INDEX | a [list](http://127.0.0.1:40914/help/library/base/help/list) of one or more [factor](http://127.0.0.1:40914/help/library/base/help/factor)s, each of same length as X. The elements are coerced to factors by [as.factor](http://127.0.0.1:40914/help/library/base/help/as.factor). |
| FUN | the function to be applied, or NULL. In the case of functions like +, %\*%, etc., the function name must be backquoted or quoted. If FUN is NULL, tapply returns a vector which can be used to subscript the multi-way array tapply normally produces. |

### 7.3 Different statistics per subset

In the above examples we only considered one descriptive statistic, the average as provided by the function mean(). Various other functions can of course be used to obtain other descriptive statistics. The statistic you want computed can be specified using the function argument to aggregate() or tapply(). For example, the standard deviation SD,

> sd.per.colour<-tapply(myData[,”mass”],myData$colour,sd)

> sd.per.colour

blue red

0.194079 0.35000

Again, internally tapply() and aggregate() construct subset specific vectors of the variable you pass to these functions. By using length() as the function, you can count the number of elements in each of these vectors which is your sample size (Note this does not necessarily work if there are missing values, see 9.3).

> n.per.colour<-tapply(myData[,”mass”],myData$colour,length)

> n.per.colour

blue red

1. 4

### 7.3 Exercises part II

**Exercise II.1**

a) Calculate the mean, SD and sample size of length for each colour in myData

b) Use aggregate() to calculate the means of both mass and length in myData for each colour in one command line

c) Code the construction of the following data.frame

Trial Experiment Measure

1 1 control 1

2 2 control 3

3 3 control 4

4 4 control 3

5 5 control 3

6 6 treatment 4

7 7 treatment 5

8 8 treatment 8

9 9 treatment 8

10 10 treatment 9

and use tapply() to calculate the mean, SD and sample size for each experimental group.

**Exercise II.2**

Write a script which

1. imports the datafile “ExperimentalData.txt” which you can find in Moodle
2. Performs some checks the data import was successful
3. Recodes the 4 groups to:

group Feeding.treatment Parasite.treatment

1 not fed removed

2 not fed retained

3 fed removed

4 fed retained

(d) Extracts all data where the parasite was removed

(e) Extracts data on all individuals which were “not fed” and parasite “retained” as well as “fed” and “removed” (one data.frame)

(f) Calculates the mean for “fed” and “not fed” treatment groups

(g) Calculates the variance for all each feeding and parasite treatment using aggregate()

**Exercise II.3**

Use the data in “Fruit\_tasting\_data\_2018.txt” from Moodle

Create four additional variables (i.e. “columns” in the data.frame holding the data) : score1.apples and score2.apples which hold score1 and score2 respectively only for apples, and score1.strawberies and score2.strawberries which contain score1 and score2 respectively but only for strawberries.

1. Calculate the mean for each of these four variables
2. Use subset() to create two datasets. One for apples and one for strawberries, but without any NA values
3. Make an object of class numeric containing score1 values for apples
4. Make an additional variable containing the name of each fruit tasted in the dataset, but as factors

**Exercise II.4**

Write a script which

1. defines an object that is a matrix with 10 columns containing the numbers 1 to 100
2. defines another object that is a matrix with 10 columns containing the numbers 1 to 100 such that the numbers increment by 1 for each row across the column (i.e. row 1 is 1, 2, 3, etc)
3. names the rows and columns of these two objects
4. Extracts the third row of each object
5. Calculates the mean of each column in each object

**Exercise II.5**

1. Create the following data.frame

Plot Species SoilType CationValue

P1 Acer Wet 10

P2.2 Acer Wet 12

P2.3 Quercus Dry 4

P3 Quercus Dry 3

P10 Acer Dry 1

P11 Quercus Wet 7

P12 Quercus Unknown 2

Use subset() to

1. Make an object containing only information for Quercus
2. Make an object containing only information for Quercus in dry soil types
3. Make an object containing only information for Quercus in dry and wet soil types
4. Make an object containing only information for Quercus in dry and Acer in dry and wet soil types
5. Make an object containing only the soil type and cation values for Acer
6. Make a vector of cation values for dry soil type