LU 1: Getting started

Aafke Schipper

A.Schipper@science.ru.nl

Background

In this first learning unit, you will be introduced to the goals and set-up of this course. The unit will also introduce you to the interface of R and RStudio and some basic concepts, operators and functions. You will learn how to send commands to R and how you can send multiple commands via a script. After completing this unit, you have created and stored your first small program in R!

Learning goals

After this first learning unit you are:
. familiar with the goals, set-up and teaching methods of the course
. able to perform basic arithmetic operations
. able to assign objects
. able to perform basic vector and matrix operations
. able to store a sequence of operations in a small script
. familiar with the why and how of R styling

Starting R and RStudio

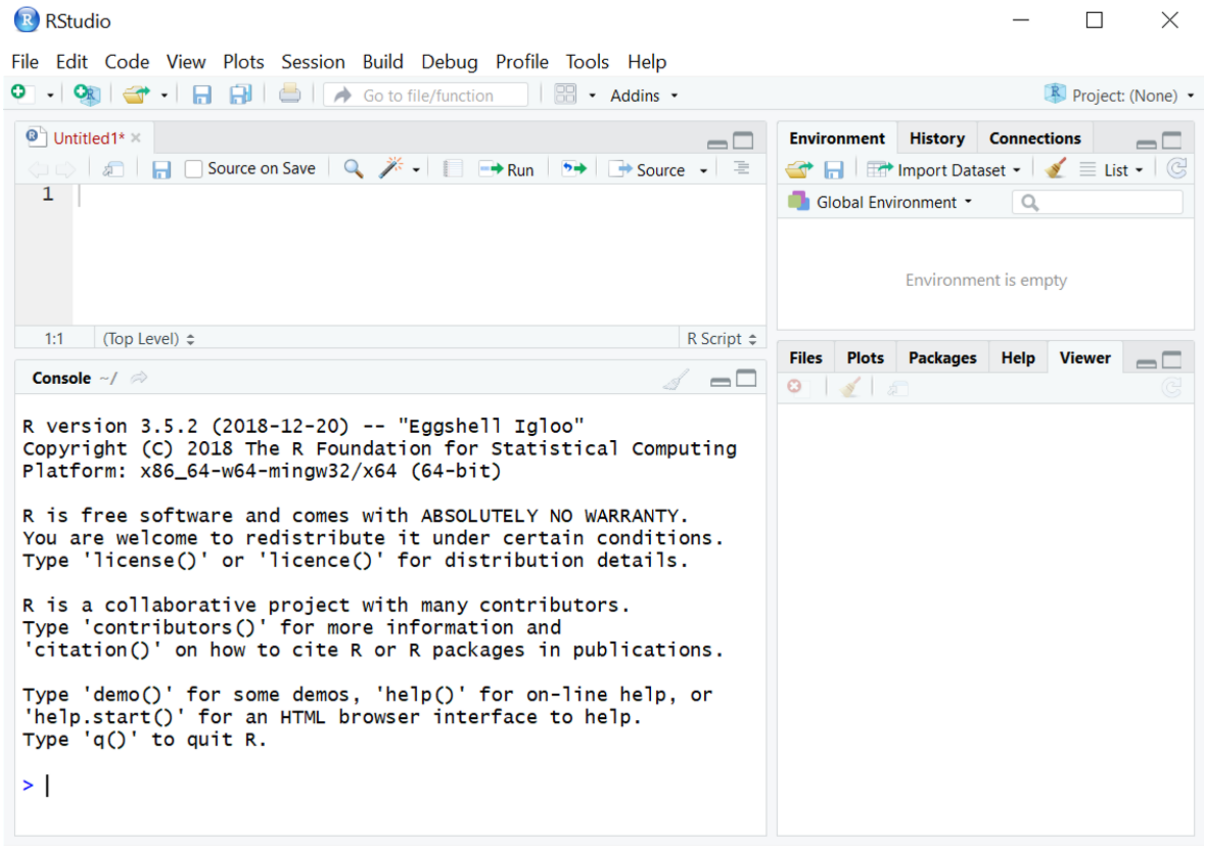
The very first thing to arrange, before anything else, is to ensure you have R and RStudio installed on your computer. RStudio is an application specifically designed to support R: it provides a user-friendly, platform-independent interface that integrates R with tools for plotting and data management and an editor to compile scripts.

In the computer rooms in the faculty, R is installed by default. If you wish to install R and RStudio on your own computer, you can download it here:

For R: http://www.r-project.org
For RStudio: https://www.rstudio.com/products/RStudio/

These sites also contain lots of useful information on R and RStudio, including documentation, manuals, news and info on for example conferences.

Once installed, we can open RStudio, and we will see an interface with four windows:



```
R version 3.5.2 (2018-12-20) -- "Eggshell Igloo"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> |
```

. The upper left window is the so-called Source window, where we can create, edit and store scripts.
. The lower left window (Console) shows the progress and outputs of running scripts and can also be used to send commands directly.
. The upper right window (Environment/History) gives an overview of all objects in the workspace (internal memory) and the history of the current R session.
. The lower right window (Files/Plots/Packages/Help/Viewer) provides the interface for managing files, plots and packages and access to documentation (Help).

The console window and some basic operations

We will now start providing some first very simple commands to R using the console window. You will see that R directly provides the output within the same window. Try for yourself a few basic arithmetic operations, like in the example below, and see what happens. You can simply use the common symbols adopted for these operations: + for summation, - for subtraction, * for multiplication and / for division.

```
4
```

```
## [1] 4
```

```
1 + 2
```

```
## [1] 3
```

```
1 + 2 + 3 + 4
```

```
## [1] 10
```

```
8 - 7
```

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

```
3 * 4
```

```
## [1] 12
```

```
1 / 0
```

```
## [1] Inf
```

```
0 / 0
```

```
## [1] NaN
```

So, this goes all well: R provides you with the output of the operations and tells you that dividing a number by zero gives infinity (Inf) whereas nothing divided by nothing is not a number (NaN). Nice! But using R only as a calculator is perhaps not the most useful… What more can we do?

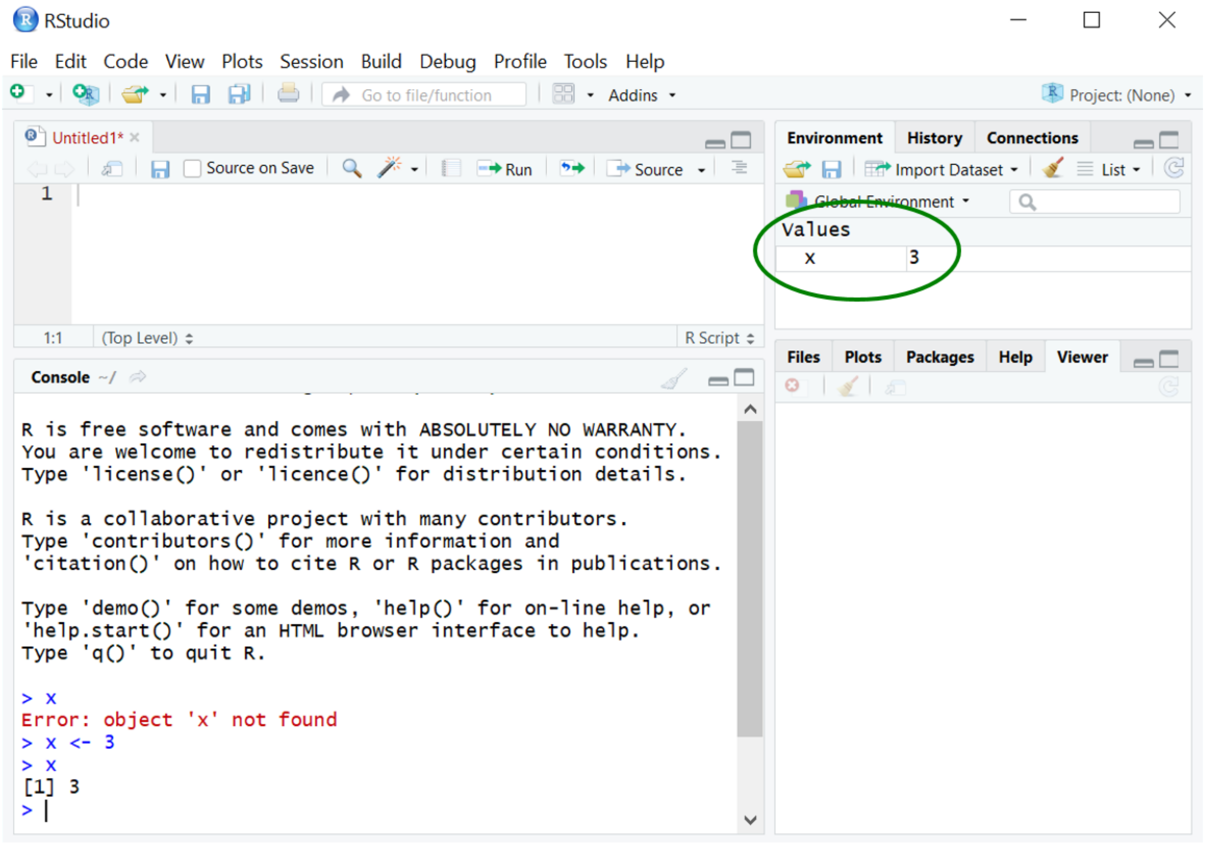
Assigning objects

One of the more useful things you can do with R is storing values as objects and then doing sequences of calculations on these objects. The convention in R is to use the notation <- to assign values to objects. For example, we can define an object x with value 3, as follows:

```
x <- 3
x
```

```
## [1] 3
```

So, object x is now defined and we see that it is stored in the internal memory (Environment).



```
Values
  x        3
```

```
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> x
Error: object 'x' not found
> x <- 3
> x
[1] 3
>
```

And then, after having defined objects, we can perform all kind of calculations (the outcomes of which we can also store as objects):

```
x <- 3
y <- 4
x + y
```

```
## [1] 7
```

```
z <- x + y
z
```

```
## [1] 7
```

Note that there are some naming conventions in R:
. R is case-sensitive, so object `a` is not the same as object `A`.
. Object names cannot start with a number
. Object names cannot contain special symbols

Creating more complex objects: vectors

Objects are not necessarily single values: they can also represent series of values. In R, a vector is a basic data structure that contains a sequence of data elements (called components or members). Vectors are typically created with the function `c()` (combine or concatenate), for example:

```
c(1, 2, 3, 4, 5)
```

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

You can also use a colon to define a regular sequence, as follows:

```
1:5
```

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

In order to explore the properties of an object, R contains various useful functions:
`str()` tells you the structure of an object
`length()` tells you the length of an object
`head()` gives you the first 6 components of an object
`tail()` gives you the last 6 components of an object

For example,

```
my.first.vector <- 1:50
length(my.first.vector)
```

```
## [1] 50
```

```
str(my.first.vector)
```

```
##  int [1:50] 1 2 3 4 5 6 7 8 9 10 ...
```

```
head(my.first.vector)
```

```
## [1] 1 2 3 4 5 6
```

```
tail(my.first.vector)
```

```
## [1] 45 46 47 48 49 50
```

It is also possible to retrieve a particular element from a vector based on its position:

```
my.next.vector <- c(2, 4, 6, 8, 10)
my.next.vector[3] # retrieve the third element
```

```
## [1] 6
```

And you can perform arithmetic operations with vectors, just like you can with scalars (single values), for example:

```
x <- 1:10
y <- 3
x + y
```

```
##  [1]  4  5  6  7  8  9 10 11 12 13
```

Exercise 1

1. Create a vector A of 10 consecutive even numbers starting from 2.
2. Create a second vector B of two values: 2 and 4.
3. Sum the two vectors. What happens?

Creating more complex objects: matrices

Most of the datasets you will work with in practice are more than single sequences of values (vectors), but typically consists of multiple sequences of values. A matrix is a collection of elements arranged into a two-dimensional structure with a fixed number of rows and a fixed number of columns. Suppose we want to create a matrix of two rows and three columns filled by odd numbers starting from 1, you can do the following:

```
my.first.matrix <- matrix(c(1, 3, 5, 7, 9, 11), nrow = 2, ncol = 3, byrow = TRUE)
my.first.matrix
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    7    9   11
```

As you see, the rows and columns each have an index, which you can use to select elements. For example, to retrieve the element in row 2 and column 3, you can do the following:

```r
my.first.matrix[2, 3] # get the element in row 2, column 3
```

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

It is also possible to retrieve an entire row or column, for example:

```r
my.first.matrix[1, ] # get the elements in row 1
```

```
## [1] 1 3 5
```

```r
my.first.matrix[ ,1] # get the elements in column 1
```

```
## [1] 1 7
```

Exercise 2

1. Create a 3x3 matrix named A filled row wise by the numbers 1 to 9.
2. Inspect the structure of the matrix.
3. Create a matrix B by dividing matrix A by 2.
4. Retrieve from B the elements in the second row and second and third column. Store these as vector C.
5. Sum the two selected elements. What is the answer?

Functions

Up to now, you have already encountered various examples of functions: instructions to carry out specific tasks. Further on in the course you will learn how to build functions yourself. Here, we will have a closer look at some built-in functions in R:

`c()` combines different elements into a vector
`sum()` calculates the sum of a series of numbers
`mean()` calculates the arithmetic mean of a series of numbers
`sd()` calculates the standard deviation of a series of numbers
`objects()` tells you which objects there are in the environment
`ls()` same: tells you which objects you have defined in the environment
`rm()` removes an object

Functions typically need input, which you define between the parentheses. For example,

```
x <- 1:5
y <- sum(x)
z <- mean(x)
```

Some functions go without input, so you can leave empty the space within the parentheses:

```
objects()
```

```
## [1] "my.first.matrix" "my.first.vector" "my.next.vector"  "x"
## [5] "y"               "z"
```

But how do you know what input you should and can feed into a function?

Finding help on functions and specifying arguments

Fortunately R has a built-in help system to consult. To get help on a particular function, you can use `help` or `?`. If you try, for example, `help(sum)`, you will get to the documentation of the function in the bottom right window, which explains the information (so-called arguments) you have to pass on to the function. In case of the sum function, for example, the documentation tells you that you need to pass one or more vectors of values (denoted by `...`) as input and that there is a second argument `na.rm`. This second arguments specifies how the function treats missing values (NA) or values that are not a number (NaN). If you do not specify this argument explicitly, R will use a default. The default arguments for each function are also described in the help documentation. In the case of the sum function, the default is that these NA and NaN values are not removed (which means that in case of NA or NaN values in your series of values, the resulting sum will be NA). For example,

```
y <- c(1, 2, 3, NA, 5)
sum(y)
```

```
## [1] NA
```

```
sum(x = y, na.rm = TRUE)
```

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

Exercise 3

1. Create a 2x2 matrix B filled column wise with the following contents: 2, 4, NA, 6.
2. Sum across the matrix.
3. Check the objects you have in the environment.
4. Remove all these objects using the `rm()` function. Use the `help` function if needed.

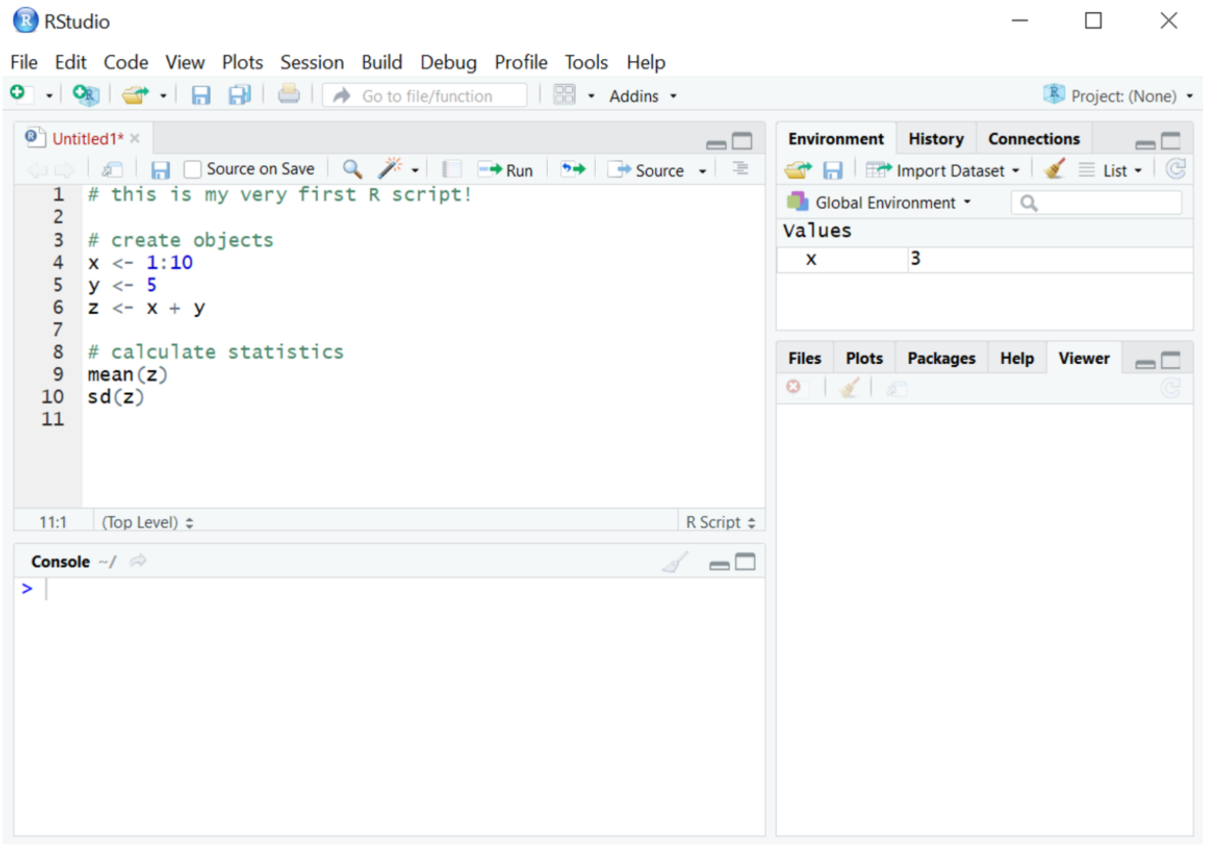
Exercise 4

1. Use the `help` function to find out how you can calculate weighted means in R using the function `weighted.mean`.
2. Apply the function to calculate the mean of 2, 4 and 6 weighted by, respectively, 0.25, 0.25 and 0.50.

The source window

So far we've been working only within the console, which is very interactive: we provided a command, R executed this, we did another command, R executed this again, and so on. However, this is not the most efficient and reproducible way of working: suppose the R environment is cleaned, all you did is lost and you have to reconstruct your work. So, as soon as data processing gets a bit more involved, it is recommended to create scripts to organise your work. It speeds up things and enhances reproducibility - one of the key requirement of scientific research.

Scripts are being created in the Source window (in the upper left). A script is basically a set of commands, commonly intertwined with annotation to provide a bit of explanation of what's happening. Comments are preceded by a hashtag, which tells R that the text that follows can be ignored in the execution. So, let's go to the Source window, and create our very first script by entering the code as shown in the picture below:

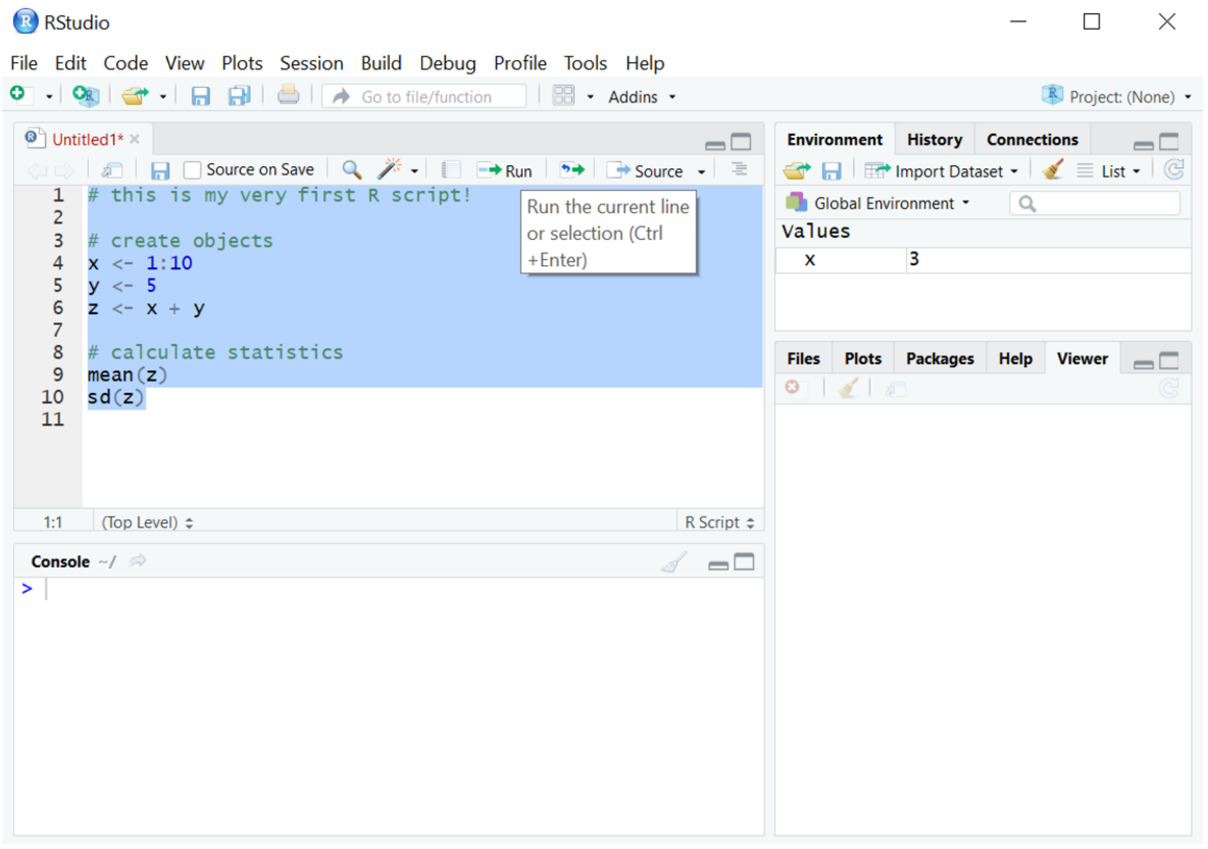


```
# this is my very first R script!

# create objects
x <- 1:10
y <- 5
z <- x + y

# calculate statistics
mean(z)
sd(z)
```

```
Values
  x          3
```

```
>
```

You will see that nothing is happening when you type the script (contrary to working in the console, when all commands are executed directly). In order to execute your script, you need to send it to the console. You can do this by selecting the script and then either press 'Run' or do Cntrl + Enter.

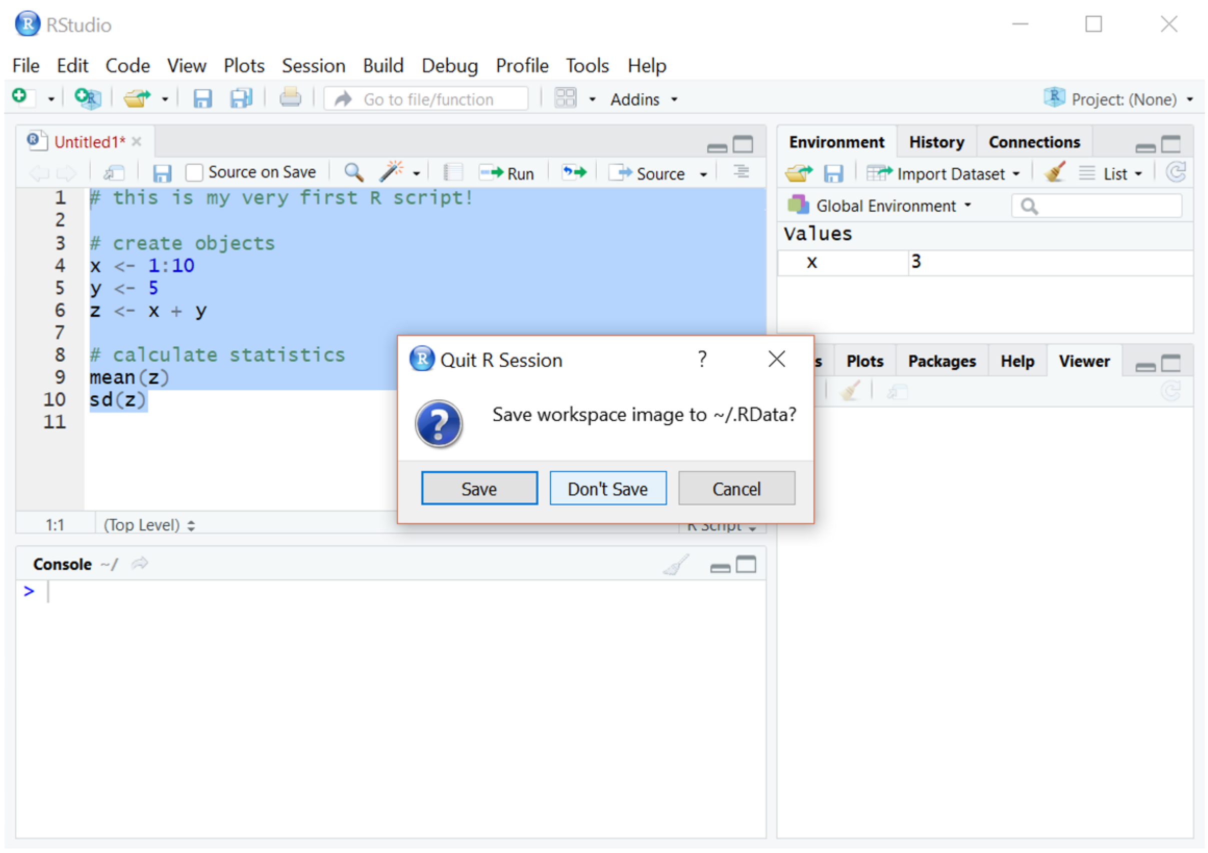

RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Project: (None)
Untitled1*
Source on Save
Run
Source
Run the current line or selection (Ctrl +Enter)
1:1
(Top Level)
R Script
Console ~/
Environment History Connections
Import Dataset
List
Global Environment
Files Plots Packages Help Viewer

```
# this is my very first R script!

# create objects
x <- 1:10
y <- 5
z <- x + y

# calculate statistics
mean(z)
sd(z)
```

```
Values
x     3
```

Saving scripts and quitting R

To save a script, you could use File - Save as and store your script in the directory of choice. If you use Save rather than Save As, R will save your script in the default working directory. To find out which one this is, you can use the function `getwd()` - without anything in the brackets as this function does not need any arguments. When saving, R scripts always get the extension .R. You can quit the R session via `q()`, Cntrl + Q, File - Quit session or just by clicking the X in the top right corner. When you quit, R will ask you whether you want to store the workspace. In most cases this is not really needed, as in a new session you could simply open the script you saved and re-generate the data in the environment.



```
# this is my very first R script!

# create objects
x <- 1:10
y <- 5
z <- x + y

# calculate statistics
mean(z)
sd(z)
```

Values

x	3

Save workspace image to ~/.RData?

Exercise 5

In this exercise we will create a small script that calculates intrinsic population growth rates of warm-blooded animals from their body mass, according to the following equation (Hendriks et al. 2007 Ecological Modelling 205:196-208; Hilbers et al. 2016 Ecology 97:615-626):

rm = ln(R0) . qt . gamma . m^-k

where rm is the intrinsic population growth rate (n/d); R0 is the life-time fecundity (number of individuals); qt is a body temperature correction factor relative to the standard temperature of 20°C (dimensionless), gamma is the average production coefficient (kg^k/d), m is the species' bodymass (kg) and k is the scaling exponent.

1. Create a new script that does the following:

- Create a vector with body mass values from 1 to 100 kg.
- Assign parameter values for the equation as follows (Hilbers et al. 2016 Ecology 97:615-626): R0 = 4.5; qt = 4.9; gamma = 0.27; k = 0.25
- Calculate intrinsic growth rates over the vector with body masses.
- Calculate a mean intrinsic growth rate over the body mass range.
- Compare this with the intrinsic growth rate of a species with a mean body mass.

2. Save your script.

Writing readable code

Finally, some words about good coding style. The primary reason to care about style is readability: adopting a consistent style and annotating your code will make it much easier to read and understand. This in turn not only helps to share your code with others, it also makes your own life much easier. Suppose you compile a script today and want to re-run it about half a year or so - this is a lot easier if you have clearly structured and annotated your code and used a consistent style! Styling is partly a matter of taste; often multiple options work. So, in principle it does not really matter what you pick, as long as your style is consistent. However, some conventions are quite broadly adopted, and it may ease the communication with others if you adopt those as well. For example, to assign objects, it is better to use `<-` than `=`, so `growth.rate <- 2` is better than `growth.rate = 2`. It is also recommended to use spaces around operators, in order to enhance readibility, so `y <- 2 + 4` is preferred over `y<-2+4`. And, of course, always annotate your code - for others and for your future self!

In the links below you can find some thoughts and guidelines about styling:
https://google.github.io/styleguide/Rguide.xml
https://www.r-bloggers.com/%F0%9F%96%8A-r-coding-style-guide/

Exercise 6

1. Open the script you created in exercise 5.
2. Check the style and annotation and improve this where you see fit.

Additional practicing

Becoming a proficient user of R requires basically three things: practice, practice and practice. This may sound a little bland, but just to make clear that it's like learning any other new laguage: as you have to internalize the words (vocabulary) and structure (syntax), you need to work with the language. For additional practicing, we have arranged access to the online course material by DataCamp, which consists of interactive exercises for different programming languages. We recommend you to sign up, which you can do via the link below. NOTE: use your student mail address from RU, and register within two weeks (no later than 17 September).

https://www.datacamp.com/groups/shared_links/22cecd4b5a944506d1e074e421bb78400741db61

From the DataCamp material, the first three chapters from the introduction course link up pretty well with this learning unit ('1. Intro to basics', '2. Vectors' and '3. Matrices').

LU 2: The Basics: Part 2

With exercises

Melinda de Jonge

m.dejonge@fnwi.ru.nl

Contents

1 Learning goals

By the end of the class you will be able to:

1. use different object types, identify them and convert between them
2. use different data types and convert between them
3. generate sequences and sort/order data
4. create and manipulate strings

2 Functions we are going to use

- class(), typeof()
- as.numeric(), as.character()
- paste(), sprintf(), strsplit(), sub()
- factor(), levels()
- seq()
- sort(), order()
- rownames(), colnames()
- data.frame(), list(), array()
- subset()

3 The basics

3.1 Object types

During the morning lectures you already learned to work with numeric objects. Besides numeric objects, you can also make and use character, logical and compex objects. **Character objects** are created by including any value within quotation marks `' '` or double quotation marks `" "`.

```
a <- "a"
```

Logical objects can only have two values: `TRUE` and `FALSE` (in some other programming languages these are represented by 1 and 0 respectively or shortened to T and F). Their meaning is obvious. These objects will become very important in the next learning units!

```
b <- TRUE #Or FALSE
```

Lastly, you can create `complex` objects in R. You probably will not need them for now, but it's good to know they exists. **Complex objects** constist of a 'real' part (a) and an 'imaginary' part (b) and are given as: a + bi.

```
c <- 2 + 3i
```

3.2 Checking the type of an object

When you are working in Rstudio, you can check the class of an object in the 'global environment' panel in the top right panel of the screen. You can also check the class of objects in the console by using `typeof()`.

```
typeof(a)
```

```
## [1] "character"
```

```
typeof(c)
```

```
## [1] "complex"
```

You can also check if an object belongs to a certain class. In this case, the response you will get will be logical (TRUE or FALSE). For example:

```
is.character(a)
```

```
## [1] TRUE
```

```
is.character(c)
```

```
## [1] FALSE
```

3.3 Converting between object types

We can coerce any object type into a character object using `as.character()`.

```
c.character <- as.character(c)
typeof(c.character)
```

```
## [1] "character"
```

```
b.character <- as.character(b)
typeof(b.character)
```

```
## [1] "character"
```

You can also convert complex and logical object to numerical ones. However, in this case, their value will change accordingly. TRUE becomes 1, FALSE becomes 0 and all complex values will retain only their real part (notice that R will give you a warning when coercing complex values into numerical ones).

```
b.numeric <- as.numeric(b)
b.numeric
```

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

```
c.numeric <- as.numeric(c)
```

```
## Warning: imaginary parts discarded in coercion
```

```
c.numeric
```

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

You can also convert character to numerics, but only when the character object has a value that can logically be converted.

```
d <- "2"
d.numeric <- as.numeric(d)
d.numeric
```

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

```
typeof(d)
```

```
## [1] "character"
```

```
a.numeric <- as.numeric("a")
```

```
## Warning: NAs introduced by coercion
```

```
a.numeric
```

```
## [1] NA
```

3.4 Making vectors with multiple object types

If you make a vector or matrix with only one type, the resulting object will be assinged that type as well. However, when you mix different types in one object, everything will be coerced into characters.

```
vec <- c('1', 2, TRUE)
class(vec)
```

```
## [1] "character"
```

3.5 Logical operations

Using R, you can do the following logical operations which will come in very handy during the next learning units. These operations will return a logical object (`TRUE` or `FALSE`).

<	less than
<=	less than or equal to
>	larger than
>=	larger than or equal to
==	equal to
!=	not equal to
&	and
|	or

A few examples of how they work:

```
1 == 1
```

```
## [1] TRUE
```

```
1 != 1
```

```
## [1] FALSE
```

```
1 != 2
```

```
## [1] TRUE
```

```
1 == '1'
```

```
## [1] TRUE
```

```
1 == 1 | 1 == 2
```

```
## [1] TRUE
```

```
1 == 1 & 1 == 2
```

```
## [1] FALSE
```

```
1 == 1 & 1 == 2 | 1 < 2
```

```
## [1] TRUE
```

```
CapitalLetters <- c("A", "B", "C")
CapitalLetters == c("A", "BB", "C")
```

```
## [1]  TRUE FALSE  TRUE
```

3.6 Strings

Strings in R are represented by **character objects**

```
mystring <- "Zoe is a very sweet cat."
class(mystring)
```

```
## [1] "character"
```

You can combine two or more strings using `paste()`.

```
mystring2 <- "But she does not like other cats."
paste(mystring, mystring2)
```

```
## [1] "Zoe is a very sweet cat. But she does not like other cats."
```

If you include any non character object in the paste function, it will be coerced to character. The same can be done using the `sprintf()` function.

```
mystring3 <- "She was born in"
year <- 2013
month <- "May"
paste(mystring3, month,year)
```

```
## [1] "She was born in May 2013"
```

```
sprintf("She was born in %s %s","May",2013)
```

```
## [1] "She was born in May 2013"
```

You can split strings based on some separator using `strsplit(string, split=' ')`

```
strsplit(mystring, split = ' ')
```

```
## [[1]]
## [1] "Zoe"   "is"    "a"     "very"  "sweet" "cat."
```

```
strsplit(mystring, split = '')
```

```
## [[1]]
##  [1] "Z" "o" "e" " " "i" "s" " " "a" " " "v" "e" "r" "y" " " "s" "w" "e"
## [18] "e" "t" " " "c" "a" "t" "."
```

It's also possible to replace any word or character with another character using the sub() function.

```
sub("May", "April", "She was born in May 2013")
```

```
## [1] "She was born in April 2013"
```

```
sub("3", "4", "She was born in May 2013")
```

```
## [1] "She was born in May 2014"
```

3.7 Factors

Categorical data can be represented as characters, however when you have a lot of data with many classes, it may be beneficial to store these as factors instead. Factors are stored as integers (1, 2, 3 etc.) where each integer has a label associated with it.

```
sex <- factor(c("male", "female", "female", "male"))
sex
```

```
## [1] male   female female male
## Levels: female male
```

Factors are ordered alphabetically by default. So even though the vector above starts with 'male', 'female' will be the first level of the factor.

```
levels(sex)
```

```
## [1] "female" "male"
```

If you don't want to your factor to be ordered alphabetically you can specify the order when making the factor. This is especially usefully when the factor represents some ordered categories.

```
score <- factor(c("medium", "high", "low"))
levels(score)
```

```
## [1] "high"   "low"    "medium"
```

```
score <- factor(c("medium", "high", "low"),levels=c("low","medium","high"))
levels(score)
```

```
## [1] "low"    "medium" "high"
```

Using you levels argument you specify what the order of the classes in the factor should be. Additionally, you can specify tell R that this order is meaningfull which allows you to compare levels using, for example, logical operators.

```
score[1] > score[2] #Does not work
```

```
## Warning in Ops.factor(score[1], score[2]): '>' not meaningful for factors
```

```
## [1] NA
```

```
score <- factor(c("medium", "high", "low"),
                levels=c("low","medium","high"),ordered=TRUE)
score[1] > score[2]
```

```
## [1] FALSE
```

Because factors are represented as numbers in R's memory, they are more efficient than characters and because they have labels, they are more informative than numbers.

Working with factors can sometimes be confusing because they look like characters but behave differently. This can be confusing when you want to convert your factor to numerical values for example.

```
constants = factor(c(3.4,12.1,9.3))
as.numeric(constants)
```

```
## [1] 1 3 2
```

```
as.numeric(levels(constants))
```

```
## [1]  3.4  9.3 12.1
```

```
#Alternatively
constants = factor(c(3.4,12.1,9.3,3.4))
as.numeric(levels(constants))
```

```
## [1]  3.4  9.3 12.1
```

```
as.numeric(as.character(constants))
```

```
## [1]  3.4 12.1  9.3  3.4
```

3.8 Data structures 2

3.8.1 Indexing by name

This morning you were introduced to scalars, vectors and matrices and saw how you can do some basic arithmetics with these. You also learned how to index vector and matrices using numbers. However, you can also give each element of a vector a name using `names()` for vectors, and `colnames()` and `rownames()` for matrices. For example:

```
weight <- c(60,80,76,72,95)
weight
```

```
## [1] 60 80 76 72 95
```

```
people = c('Mary','John','Tim','Tracy','Ben')
names(weight) = people
weight['John']
```

```
## John
##   80
```

This is particulary helpfull when these names have some meaning. Let's consider an example where we have the weight and height of 5 imaginary people stored in a matrix.

```r
height = c(160,182,183,NA,201)
Data = matrix(c(weight, height), ncol=2, nrow=5)
colnames(Data) = c('Weight','Height')
rownames(Data) = people
```

If we now want to know how tall John is, we do not have to look up which row represents John and which column represent height, we can just use the names.

```r
Data['John','Height']
```

```
## [1] 182
```

3.8.2 Data frames

Usually when we want to store some data in R we use data frames instead of matrixes. One of the advantages of using data frames is that their columns can have different object types, while vectors and matrices can hold only one object type. Similarly to matrices, data frames have two dimensions: the vertical dimension is constituted by the rows and the horizontal dimension by the columns.

```r
eyecolors = c('blue','brown','brown','green','grey')
DataMat = matrix(c(eyecolors, height), ncol=2, nrow=5)
typeof(DataMat)
```

```
## [1] "character"
```

```r
class(DataMat)
```

```
## [1] "matrix"
```

```r
DataFrame = data.frame(eyecolors, height)
rownames(DataFrame) = people
str(DataFrame)
```

```
## 'data.frame':    5 obs. of  2 variables:
##  $ eyecolors: Factor w/ 4 levels "blue","brown",..: 1 2 2 3 4
##  $ height   : num  160 182 183 NA 201
```

```
class(DataFrame)
```

```
## [1] "data.frame"
```

```
rownames(DataFrame) = people
DataFrame
```

```
##       eyecolors height
## Mary       blue    160
## John      brown    182
## Tim       brown    183
## Tracy     green     NA
## Ben        grey    201
```

Here you see that the character vector of eyecolors is coerces into a factor. This is the standard behaviour for the `data.frame()` function. If you do not want this, you can specificy that you want it to stay a character by including `stringsAsFactors = FALSE`.

```
DataFrame2 = data.frame(eyecolors, height, stringsAsFactors = FALSE)
str(DataFrame2)
```

```
## 'data.frame':    5 obs. of  2 variables:
##  $ eyecolors: chr  "blue" "brown" "brown" "green" ...
##  $ height   : num  160 182 183 NA 201
```

You can extract the values from a dataframe in a similar way what you did with matrices. You can also use the `$` operator to extract columns.

```
# extract the eyecolors column with the $ operator
DataFrame$eyecolors
```

```
## [1] blue  brown brown green grey
## Levels: blue brown green grey
```

```
# extract the eyecolors column with the [,] notation
DataFrame[,"eyecolors"]
```

```
## [1] blue  brown brown green grey
## Levels: blue brown green grey
```

```r
# extract the first row of a data frame by reference
DataFrame[1,]
```

```
##      eyecolors height
## Mary      blue    160
```

```r
# extract the first column of a data frame by reference
DataFrame[, 1]
```

```
## [1] blue  brown brown green grey
## Levels: blue brown green grey
```

```r
# extract the 1st and 2nd columns and the 3rd and 5th rows
DataFrame[c(3, 5), 1:2]
```

```
##     eyecolors height
## Tim     brown    183
## Ben      grey    201
```

It's also possible to convert a matrix into a data frame and vice versa.

```r
as.matrix(DataFrame)
```

```
##       eyecolors height
## Mary  "blue"    "160"
## John  "brown"   "182"
## Tim   "brown"   "183"
## Tracy "green"   NA
## Ben   "grey"    "201"
```

```r
data.frame(DataMat)
```

```
##      X1   X2
## 1  blue  160
## 2 brown  182
## 3 brown  183
## 4 green <NA>
## 5  grey  201
```

3.8.3 Arrays

All the object we have discussed so far can store data in 1 or 2 dimensions (rows and columns). Arrays are objects that can store data in more than 2 dimensions. In this manual we only show examples of arrays with 3 dimensions, however, arrays with or more dimensions are also possible and have the same functionalities. Arrays are created using the `array()` function.

```
#Lets pretend we have the height and weight of 5 persons in 2 different years.
height1 <- c(160,182,183,NA,201)
weight1 <- c(60,80,76,72,95)
height2 <- c(160,182,183,NA,201)
weight2 <- c(63,78,77,71,90)
dataset <- array(c(height1,height2,weight1,weight2),dim=c(5,2,2))
dataset
```

```
## , , 1
##
##      [,1] [,2]
## [1,]  160  160
## [2,]  182  182
## [3,]  183  183
## [4,]   NA   NA
## [5,]  201  201
##
## , , 2
##
##      [,1] [,2]
## [1,]   60   63
## [2,]   80   78
## [3,]   76   77
## [4,]   72   71
## [5,]   95   90
```

Like with matrices and dataframes, we can name the rows, columns and matrices.

```
column.names <- c("2017","2018")
row.names <- people
matrix.names <- c("Height","Weight")
dimnames(dataset) <- list(row.names,column.names,matrix.names)
dataset
```

```
## , , Height
##
##       2017 2018
## Mary    160  160
## John    182  182
## Tim     183  183
## Tracy    NA   NA
## Ben     201  201
##
## , , Weight
##
##         2017 2018
## Mary      60   63
## John      80   78
## Tim       76   77
## Tracy     72   71
## Ben       95   90
```

Indexing arrays works the same as matrices and vectors, except that you now need 3 dimensions.

```
#Show all height data
dataset[,,'Height']
```

```
##         2017 2018
## Mary     160  160
## John     182  182
## Tim      183  183
## Tracy     NA   NA
## Ben      201  201
```

```
#Show all data for John
dataset['John',,]
```

```
##        Height Weight
## 2017      182     80
## 2018      182     78
```

Like matrices and vectors, arrays can only contain elements on one type.

3.8.4 Lists

Lists are R objects which can contain elements of different types and size inside it. In the data frame all of the columns included must have the same lenght. In lists you can combine vectors, matrices, data frames and arrays of different dimensions. Lists can even contain other lists. Lets look at an example with vectors of different length.

```
shortvector <- c(1,2,3)
somelist <- list(shortvector, weight)
somelist
```

```
## [[1]]
## [1] 1 2 3
##
## [[2]]
##  Mary  John   Tim Tracy   Ben
##    60    80    76    72    95
```

You can make named lists by giving the name of the object when making the list. These names can then be used to reference objects in the lists.

```
somelist <- list(sv = shortvector, height = height)
# extraxt the height
somelist$height[1]
```

```
## [1] 160
```

```
# similarly you can use the [[ ]] notation and the name of the object
somelist[["height"]][1]
```

```
## [1] 160
```

```
# similarly you can use [[ ]] notation and the reference to the position
somelist[[2]][1]
```

```
## [1] 160
```

```
# Note that [1] only shows the first element in the vector of heights

# Note that the class changes, and that somelist$height is a vector
class(somelist)
```

```
## [1] "list"
```

```
class(somelist$height)
```

```
## [1] "numeric"
```

Lists can be converted to vectors using the `unlist()` function.

```
unlist(somelist)
```

```
##     sv1     sv2     sv3 height1 height2 height3 height4 height5
##       1       2       3     160     182     183      NA     201
```

3.9 Making subsets of your data

We already saw how to show and manipulate data in objects using numeric or named indexes. We can use the same strategy to subset data into new objects.

```
Data2017 <- dataset[,'2017',]
Data2017
```

```
##       Height Weight
## Mary     160     60
## John     182     80
## Tim      183     76
## Tracy     NA     72
## Ben      201     95
```

We can also use logical operation to subset data. For example, we can select everyone who is at least 180 cm tall from the previous example

```
TallPeople <- Data2017[Data2017[,'Height'] >= 180,]
TallPeople
```

```
##      Height Weight
## John    182     80
## Tim     183     76
## <NA>     NA     NA
## Ben     201     95
```

We can split this example in steps to better understand what is happening.

```
indexes <- Data2017[,'Height'] >= 180
indexes
```

```
##  Mary  John   Tim Tracy   Ben
## FALSE  TRUE  TRUE    NA  TRUE
```

```
TallPeople <- Data2017[indexes,]
TallPeople
```

```
##      Height Weight
## John    182     80
## Tim     183     76
## <NA>     NA     NA
## Ben     201     95
```

We can use the exclamation mark (`!`) to negate a statement. For example, if we want to select everyone who is shorter than 180 cm we can use `Data2017[,'Height'] < 180` or we can use the exclamation mark.

```
ShortPeople <- Data2017[Data2017[,'Height'] < 180,]
ShortPeople
```

```
##      Height Weight
## Mary    160     60
## <NA>     NA     NA
```

```
ShortPeople <- Data2017[!Data2017[,'Height'] >= 180,]
ShortPeople
```

```
##      Height Weight
## Mary    160     60
## <NA>     NA     NA
```

Alternatively, we can use the function `subset()` to make logical subsets of our data.

```
TallPeople <- subset(Data2017, Data2017[,'Height'] > 180)
TallPeople
```

```
##      Height Weight
## John    182     80
## Tim     183     76
## Ben     201     95
```

As you can see, the `subset()` function filters out the NA value.

For special values such as 0, NA, and Inf, you can use the `is.null()`, `in.NA()` and `is.infinite()` functions to select or subset these.

```
#Select only the persons who did report their height.
WithHeight <- Data2017[!is.na(Data2017[,'Height']),]
WithHeight
```

```
##      Height Weight
## Mary    160     60
## John    182     80
## Tim     183     76
## Ben     201     95
```

3.10 Sequences

In the previous learning unit you already saw that you can create sequence of numbers using ':'.

```
1:10
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
10:1
```

```
##  [1] 10  9  8  7  6  5  4  3  2  1
```

Using the `seq()` function you can also create sequences with increments different from 1.

```
seq(from=1, to=20, by=2)
```

```
##  [1]  1  3  5  7  9 11 13 15 17 19
```

```
seq(from=20, to=10, by=-2) #Note the sign of the 'by' argument!
```

```
## [1] 20 18 16 14 12 10
```

```
seq(from=1, to=10, length.out=15)
```

```
##  [1]  1.000000  1.642857  2.285714  2.928571  3.571429  4.214286  4.857143
##  [8]  5.500000  6.142857  6.785714  7.428571  8.071429  8.714286  9.357143
## [15] 10.000000
```

3.11 Sorting and ordering

You can sort a vector using the `sort()` function. This will return all the elements in the variable in and ordered fashion. For characters, this means alphabetical order.

```
weight <- c(60,80,76,72,95)
people <- c('Mary','John','Tim','Tracy','Ben')
sort(weight)
```

```
## [1] 60 72 76 80 95
```

```
sort(weight, decreasing = TRUE)
```

```
## [1] 95 80 76 72 60
```

```
sort(people)
```

```
## [1] "Ben"   "John"  "Mary"  "Tim"   "Tracy"
```

It's also possible to use the `order()` function. This will return the ordered indexes of the elements in the vector. We can use this to sort the vector (similar to using the `sort() function`) but also to sort another vector.

```
order(weight) #This returns the indexes that can be used to sort weight
```

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

```
weight[order(weight)] #Same as sort(weight)
```

```
## [1] 60 72 76 80 95
```

```
people[order(weight)] #Sort the people vector based on the values in weight
```

```
## [1] "Mary"  "Tracy" "Tim"   "John"  "Ben"
```

4 Exercises

1. Create a matrix that contains the weights (in 2017), heights, and eyecolors for the people as given in the manual for this learning unit. Make sure that you name the columns and rows of the matrix.
2. Check the class and type of the matrix you just created. Why is the type character?
3. Convert the matrix created in the previous exercise to a data frame and convert weight and height back to numerical objects.
4. Calculate the BMI by dividing the weight by the square of height in m and store the results in a new column in the data frame called BMI.
5. For one person, the height is not known, this is indicated by a NA value. Remove this person from the dataframe.
6. Make a new dataframe called 'browneyes' which contains the weight and height of people with brown eyes only.
7. Sort the rows of the original dataframe by eyecolor first and by weight second.
8. Create a piece of code that gives all characteristics for a person as a sentence. For example 'John has blue eyes, is 190 cm tall and weight 80 kg.' Make sure that you can use the piece of code for all persons in the data frame in such a way that you only need to adjust the name.
9. In the Array example, the data is separated by measurement. Make a new array in wich the data is separated by year (this means you will get an array of 5 2 by 2 matrices).
10. From the array generated in exercise 9, select only the people who have lost weight.

All the following exercizes are with some new data from some made up plants.

```
pot.diameter = c(10,8,11,40,22,15,23)
height = c(15,13,20,175,110,40,90)
stage = c('seedling','seedling','sapling','mature','mature',
          'sapling','mature')
```

11. Create a data frame of this data called 'PlantData'

Of course, we also have the names of these plants. Luckily, they are in already in the right order.

```
names = c('Sansiviera zeylanica','Sansiviera trifasciata',
          'Pachira Aquatica','Heteropanax chinensis',
          'Sterlizia Nicolai','Monstera deliciosa',
          'Howea fosteriana')
```

12. Give the rows of the PlantData the name of the corresponding species.
13. Of course there is a natural order in the life stages of a plant. Create an ordered factor from the stage variable.
14. Order the dataframe based on the life stage of the plants
15. Do taller plants have a larger pot diameter
16. Split the dataset into seperate objects for each life stage

Because plant grow we also have some data from a while ago which is slightly different.

```
old.pot.diameter = c(5,5,15,35,21,15,20)
old.height = c(0,0,18,165,100,40,85)
```

17. Combine the data from the two time periods in one object
18. Order the data by the growth rate of the plants
19. Which of the Sansiviera species has grown the most?

LU3: Data in R

With exercises, but no solutions

Michela Busana

michela.busana@ru.nl

Contents

1 Learning unit goals

By the end of the class we will be able to:

1. *Operate with data files in R*
2. *i.e. read/write datasets in R, clean a dataset, subset a dataset, merge multiple datasets*

2 The workspace and working directory

DEFINITION

workspace = current R working environment including all user-defined objects
working directory = the directory on the computer where R points to. Check it with getwd()

When we create a new object in R it will be saved within the **workspace** of the current R session.

Try typing

```r
ls()
```

```
## character(0)
```

```r
newObject <- c(2 , 4 , 7)
newObject
```

```
## [1] 2 4 7
```

```r
ls()
```

```
## [1] "newObject"
```

`ls()` is a *base R function* that returns the list of objects present in the workspace.

Each R session is linked to a directory in our computer, the *working directory*. This directory is the default path where R accesses hard files that are saved in our machines. This is handy when we want to analyze data that have been collected and kept in a spreadsheet or a text file, for example. Pay attention to syntax differences in Windows versus Linux/OS and commonly related error messages.

> ☆ BONUS: Extra food for brain
>
> A function is a unit of code that takes an input, operates on that input and returns an output. We will learn more about functions in the LU7. For now note that “ls()“, for example, is a function that is built-in in the R environment. These type of functions usually are called *base R function*.

To see which working directory is linked to the current R session type:

```r
getwd() # getwd stands for get working directory
```

The path returned will differ between computers and R session.

Pay attention to syntax differences between Windows and Linux/OS machines.

In a Windows machine your path will look like something:

- `c:\\user\\temp` or
- `c:/user/temp`

In Linux and OS, the path will look like `/home/user/temp` or `~/temp`.

It is possible to set the working directory to a specific path with the function `setwd()`. For example, try modifying your working directory. On my Windows PC I can use something like:

```r
setwd("c:/Users/michela/Documents")
```

However, this path will differ among machines. For example, if I make a typo I will receive an error that R cannot change the working directory. See for example:

```
setwd("c:/Users/michela/Doc")
```

```
## Error in setwd("c:/Users/michela/Doc"): cannot change working directory
```

If we try to change the working directory with `setwd()` and receive an error message, this is because there is an error in the way we typed the path or the path does not exists in the computer.

☆ BONUS: Tip

Workaround!
A good workaround to avoid problems with manually setting the working directory is to:
1. keep R scripts and data files together in the same folder of a PC
2. navigate to the Rscript via your computer folder view
3. open the Rscript with Rstudio
4. double-check your working directory with "getwd()". Is it the correct path to the Rscript file? If the answer is no, go back to point 1.
5. if some files are saved within subfolders of the current working directory call them with the relative path, see below
The steps above are handy to make code reproducible across multiple machines! The moment we share our code with peers, or we get a new PC we will be able to run the same script without the necessity to set the path where the scripts and files are saved in the hard drive of the PC.

2.1 Adding & removing objects

As we have seen above, it is possible to add objects to the current workspace by merely creating a new object, e.g.

```
newObject <- 3
newObject1 <- 1:10
newObject
```

```
## [1] 3
```

```
newObject1
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Similarly, it is possible to remove an object or objects from the workspace with the function `rm()`.

For example:

```
ls()
```

```
## [1] "newObject"  "newObject1"
```

```r
# let's remove the newObject we just created
rm(newObject)
ls()
```

```
## [1] "newObject1"
```

```r
# it is also possible to remove all objects currently present in the workspace
rm(list = ls())
ls()
```

```
## character(0)
```

BONUS: Warning

When reading new files into the workspace do **not** override existing functions or variables! i.e., avoid giving objects or functions the same name as existing R objects or functions.

2.2 Saving the workspace and object(s)

It is possible to save objects from the workspace to an .RData (or .Rdata or .rdata) file. More specifically it is possible to save:

- a single object
- a list of objects
- the whole workspace

Note that the definition of object includes vectors, matrices, arrays, data frames, and lists.

When closing RStudio, the program will ask if we want to save the workspace to an RData file. It is recommended **not** to if we're going to avoid clogging the memory of our PCs. It's better to save files manually when needed.

```r
A <- 1:10
B <- A^0.5
C <- A^1.5

# save a single object
save(A, file = "objectA.RData")

# save a list of objects by providing a list which includes a vector of
# character strings naming the objects we want to save
save(list = c("A", "B"), file = "objectsAB.RData")
# or similarly use
save(A, B, file = "AB.RData")
```

```r
# save the whole workspace
save.image(file = "wholeWS.RData")
```

After typing the above code, try closing Rstudio. Try to navigate to your current working directory. You will see the new .RData file there. Pick one of them and open it with Rstudio and type `ls()` in the Console. Depending on the file opened, `ls()` will return something different. The objects saved in the .RData file are now available to use in the new R session.

Alternatively, we can open the .RData files from the console with the `load()` function. For example, try:

```r
# clean the working directory
rm(list = ls())
# load the .RData file
load("wholeWS.RData")
```

3 Reading data files into R

R can read diffent formats of two-dimensional data files. The most commonly used are:

- .csv or comma separated values, `read.csv()`
- .txt or text files, `read.table()`
- .rds or R data serialized, which is a binary file type specific to R, `readRDS()`

The object returned in the workspace is a data frame. The advantage of the .csv and .txt formats is that they are universal formats that can be opened by a multitude of software. The advantage of the .rds format is that rds files are way faster to read, but they are exclusively used in R.

To read the file "myFile.csv" from your current working directory type:

```r
data <- read.csv("myFile.csv", header = TRUE, sep = ",")
# the option header = TRUE declares that the 1st row contains the column names
# the argument sep = "," defines that the cells are separated by commas
head(data)
```

```
##   X A B
## 1 1 1 1
## 2 2 2 4
## 3 3 3 9
```

To read a txt file:

```r
data <- read.table("myFile.txt")
```

To read a rds file:

```r
data <- readRDS("myFile.rds")
```

It can be useful to check whether a filename exists in the current working directory with:

```r
file.exists("myFile.txt")
```

```
## [1] TRUE
```

Note that a dataset stored in a Excel format (`.xlsx`) can be easily converted to `.csv`. Open the `.xlsx` file in Excel and click on `File -> Save As`. Change the `Save as Type:` value to `CSV (Comma delimited) (*.csv)`, click `Save`. Afterwards a couple of warning messages will appear. Confirm by clicking `OK` and `Yes`. More detailed instructions can be found here.

3.1 R converts characters into factors

R's default behavior when reading data frames is to convert all character columns into factors. This is something to be aware of, since characters and factors have different properties.

Let's look at an example:

```r
# read a data frame
data <- read.csv("example.csv")
data
```

```
##   X x y
## 1 1 1 a
## 2 2 2 b
## 3 3 3 c
## 4 4 4 d
## 5 5 5 e
```

```r
# the class of the column y is factor
class(data$y)
```

```
## [1] "factor"
```

```r
# if we want the column to be a character, we can use the argument
# stringsAsFactors and set it to FALSE
data <- read.csv("example.csv", stringsAsFactors = FALSE)
# the class of the column is now a character
class(data$y)
```

```
## [1] "character"
```

In general, it is preferred to use factors rather than characters because factors use less memory and computations with factors are faster than computations with characters.

3.2 Reading Dates and Times from Files

Dates and times need to be handled carefully.

Typically, it is better to read dates and times as character strings and then convert them into dates or times within R.

Understanding how to handle dates in R is beyond the scope of this lecture. If you are interested in learning more check this blog.

4 Writing files to a directory

It is also possible to write to file a matrix or data frame created in R with the functions:

- `write.csv()` to a .csv file
- `write.table()` to a .txt file
- `writeRDS()` to a .rds file

```
dat <- data.frame(x = 1:5, y = 1:5/4, z = (1:5)^1.2)
# write a csv file
write.csv(dat, file = "writeToFile.txt")

# write a txt file
write.table(dat, file = "writeToFile.txt")
```

The file is saved by default in the working directory.

BONUS: Recap

R can read and write data from/to a file stored on the hardware of your PC. The most common used formats for external files are .csv, .txt, and .rds.
Use the functions read.csv(), read.table(), readRDS() to read from a file; and the correspective write.csv, write.table(), writeRDS() to write to a file.

5 Folder structure and paths

Keeping a folder structure within the working directory is useful. Ideally, we would want three or four subfolders:

1. **RawData** = in this directory store all the original datasets as collected in the field or lad, or downloaded from an open-source website
2. **DerivedData** = store here all the derived datasets that are obtained from the raw datasets after applying some transformation, such as cleaning errors, or reorganizing the columns

3. **Metadata** = store here the documentation of the datasets
4. **Scripts** = additionally, we might want to keep our scripts in a separate directory

This structure is ideal for keeping your files in order! However, other structures are possible and you can make your own.

It is possible to access files in R in slightly different ways. For example, assume that my current working directory is `c:/user/currentWD/` and that the folder currentWD contains a subfolder called RawData, which then includes a file called `dataset.csv`. We need to read the `dataset.csv` into R. There are at least three equivalent ways of achieving this:

- change the working directory `setwd("c:/user/currentWD/RawData/")` and read the file `read.csv("dataset.csv")` (the option we used so far)
- `read.csv(file = "c:/user/currentWD/RawData/dataset.csv")` = use the absolute path of the file (i.e., the path to the file followed by the name of the file)
- `read.csv(file = "./RawData/dataset.csv")` = use the relative path with respect to the current working directory (i.e., which folder I have to look into starting from the current working directory, `c:/user/currentWD/`). Note that the dot `.` at the beginning of the relative path stands for **here**, which is equal to the current working directory.

Similarly, if we wanted to write a file called dataset.txt to the `c:/user/currentWD/DerivedData` directory, and the current working directory is `"c:/user/currentWD/"` we could:

- change the working directory `setwd("c:/user/currentWD/DerivedData/")` and write the file `write.table("dataset.txt")`
- `write.table(file = "c:/user/currentWD/DerivedData/dataset.txt")` = use its absolute path
- `write.table(file = "./DerivedData/dataset.txt")` = use the relative path from the current working directory (`c:/user/currentWD/`)

Note that the use of a **relative path is preferred** because it will contribute to making your code **transferrable among computers**! If you copy paste all the files in your working directory to a new location, the relative path with respect to the current working directory will always be the same, while the absolute path will change.

BONUS: recap

Keep your files organized! It will save you a lot of headaches when looking for the files you need. It is recommended to use the relative paths to access files to make your code more transferrable and reproducible.

5.1 Common errors when reading files

If we try to read a file that does not exist, or we misspell the name of a file or we call a file from a non-existing/wrong working directory, we will receive an error message. For example, if we try to read a file from the wrong working directory, we get an error:

```
df <- read.csv("example1.csv")
```

```
## Warning in file(file, "rt"): cannot open file 'example1.csv': No such file
## or directory
```

```
## Error in file(file, "rt"): cannot open the connection
```

To fix the error look for the file in your computer ask yourself the following questions:

- Does the file exist? If not, create the file.
- If the answer to the previous answer is correct, did I write the name of the file correctly? If not, correct the typo.
- If the answer to the previous answer is correct, did I call the file from the correct working directory? If not, either change the working directory, or mention the path to the file, or a combination of the two.

5.2 Common issues when writing files

Did you ever write a file from R, but cannot find the file in the PC where you expected it to be? If the answer is yes, you most likely wrote the file to a different directory!

Be aware that R writes a file to the working directory, unless it is otherwise specified (i.e. the absolute or relative pasth is added in front of the file name).

6 Recap from LU2: Class = data.frame

In the LU2 we learned the basics of data frames. As a recap, a data frame in R is an object of a specific class: `data.frame`

```
# what is the class of the object?
class(dat)
```

```
## [1] "data.frame"
```

The `data.frame` class stores data in a two-dimensional format. A `matrix` object is also two-dimensional, but data frames differ from matrices because they can hold multiple data types, e.g., some columns can be factors and some columns can be numeric.

For a recap about data structures in R go back to LU2 and watch this short video.

It is also possible to check the object type of each column in the data frame:

```
# check the class for each column
sapply(dat, class)
```

```
##         x         y         z
## "integer" "numeric" "numeric"
```

6.1 Columns & rows

We can explore the names of the columns and rows of a data frame:

```
# check the column names
colnames(dat)
```

```
## [1] "x" "y" "z"
```

```
# colnames() is equivalent to names()
names(dat)
```

```
## [1] "x" "y" "z"
```

```
# check the row names
row.names(dat)
```

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

6.2 Sorting

You can also sort the data frame according to one or more columns.

```
df <- data.frame(x = c(7, 6, 6, 5), y = c("a", "z", "c", "m"))
# sort the dataframe according to the column x
df <- df[order(df$x), ]
df
```

```
##   x y
## 4 5 m
## 2 6 z
## 3 6 c
## 1 7 a
```

```
# sort the dataframe by x and then y
df <- df[order(df$x, df$y), ]
df
```

```
##   x y
## 4 5 m
## 3 6 c
## 2 6 z
## 1 7 a
```

7 Dimensions

A `data.frame` has two dimenstions. The first dimension refers to the rows, while the second dimension refers to the columns. Each row corresponds to the values corresponding to a specific observation. Each column contains the values for a variable. This means that a column contains a single data type, while rows can contain multiple data types (e.g., both factors and numeric values).

Multiple functions can be used to check the dimensions of a data frame: `dim()`, `nrow()`, and `ncol()`.

```
# what are the dimension of the data frame?
dim(dat)
```

```
## [1] 5 3
```

```
# how many rows?
nrow(dat)
```

```
## [1] 5
```

```
# how many columns?
ncol(dat)
```

```
## [1] 3
```

```
# note that the number of rows is equivalent to
dim(dat)[1]
```

```
## [1] 5
```

```
# and the number of columns id equivalent to:
dim(dat)[2]
```

```
## [1] 3
```

8 Adding a new column to a data frame

It is possible to add a new column to a data frame. For example:

```
colnames(dat)
```

```
## [1] "x" "y" "z"
```

```
# add a new column containing the squared values of x
dat$x2 <- dat$x^2
colnames(dat)
```

```
## [1] "x"  "y"  "z"  "x2"
```

```
# multiply the x and z columns and save the result in a column xz
dat$xz <- dat$x * dat$z
```

9 Rename the columns of a data frame

We can rename the columns and rows of a data frame, such as:

```
colnames(dat)
```

```
## [1] "x"  "y"  "z"  "x2" "xz"
```

```
# rename the columns
colnames(dat) <- c("a", "b", "c", "d")
# rename the rows
row.names(dat) <- letters[1:nrow(dat)]
# if you are not familiar with the function letters type ?letters in the console

# check the modified data frame
colnames(dat)
```

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

```
row.names(dat)
```

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

We can also rename a single specific column:

```
# rename the column by reference, the first column
colnames(dat)[1] <- "A"
# check the modified data frame
colnames(dat)
```

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

```
# similarly we can rename the column by name
# here rename the column calles b
colnames(dat)[colnames(dat) == "b"] <- "B"
# check the modified data frame
colnames(dat)
```

```
## [1] "A" "B" "c" "d" NA
```

10 Checking the top and bottom

Large data frames cannot be visualized entirely in the R console, simply because it does not fit!

The Nijmegen_trees.csv dataset is a survey of the trees present in the municipality of Nijmegen. It was downloaded from the webpage of the municipality.

For each tree we know the species name (species_name), the horticultural variety (BOOMSOORT), the postcode (postcode_nummer), the neighborhood (wijk), the year in which the tree was planted (PLANTJAAR), a unique identifier (ID), the location in longitude (x) and latitude (y).

```r
trees <- read.csv("./RawData/Nijmegen_trees.csv")
```

```r
# type trees in the console. What do you see?
trees
```

It is possible to visualize only the top and the bottom of the data frame by using the functions `head()` and `tail()`.

```r
# to see the first six lines of the data type:
head(trees)
```

```
##   BOOMSOORT postcode_nummer    wijk PLANTJAAR    ID        x        y
## 1     Abies            6663    Lent         0 72044 188589.2 430846.9
## 2     Abies            6663    Lent         0 72045 188593.2 430833.8
## 3     Abies            6663    Lent         0 72039 188638.4 430851.8
## 4     Abies            6663    Lent         0 71971 188535.2 430885.3
## 5     Abies            6532 Goffert      2010 12848 186246.1 426743.9
## 6     Abies            6532 Goffert      2010 59400 186323.2 426174.0
##   species_name
## 1        Abies
## 2        Abies
## 3        Abies
## 4        Abies
## 5        Abies
## 6        Abies
```

```r
# to visualize the last six lines of the dataset type:
tail(trees)
```

```
##       BOOMSOORT postcode_nummer       wijk PLANTJAAR    ID        x
## 61987      <NA>            6541     Biezen      2017 71573 186692.8
## 61988      <NA>            6546   't Acker      2017 65936 182992.4
## 61989      <NA>            6546   't Acker      2017 65976 182846.7
## 61990      <NA>            6541     Biezen      2017 71581 186699.7
## 61991      <NA>            6515 Oosterhout      2017 66963 186301.8
## 61992      <NA>            6531  Hazenkamp      2017 67245 186766.8
##                y species_name
## 61987 428566.2         <NA>
## 61988 427337.7         <NA>
## 61989 426722.1         <NA>
## 61990 428517.9         <NA>
## 61991 433158.4         <NA>
## 61992 427855.0         <NA>
```

```
# in RStudio we can use the View() to open the data in a viewer
View(trees)
```

11 Subsetting

By subsetting a data frame we retrieve a portion of the data frame. Subsetting is a handy tool in data management that allows carrying further analyses on the portion of interest in the data frame. We will see some examples later.

11.1 Specific columns & rows

☆ BONUS: To clarify and recap

The [,] and $ are alternative notations to access specific columns of a data frame or, in other words, to subset it. With [,] we can subset both rows and columns, while with $ we can access rows only. You learned about [,] notation in the LU2.
A column of a data frame can accessed **by reference or by name**. The **refence** indicates the position of the columns. For example, the first column has position 1, the second column has position 2 and so on. The **name** refers to the name of the column.

We already learned in LU2 that we can access the columns of a data frame from the data frame itself:

```
# we can access the first column of a data.frame by reference
dat[, 1]
```

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

```
# or similarly by column name
dat[, "x"]
```

```
## Error in `[.data.frame`(dat, , "x"): undefined columns selected
```

```
# we can also use the $ sign
dat$x
```

```
## NULL
```

```r
# However, the columns do NOT exist in the global environment.
# We get an error if we try to access the column a from the global environment:
x
```

```
## Error in eval(expr, envir, enclos): object 'x' not found
```

BONUS: Tip

In a dataset, the columns names and rows are locally recognizable within the dataset, but they do **not** belong to the global environment.

To access two or more columns at the same time use:

```r
dat <- data.frame(x = 1:5, y = 1:5/4, z = (1:5)^1.2, x2 = (1:5)^2)
```

```r
# access columns 1 and 3 by reference
dat[, c(1, 3)]
# access columns 1 and 3 by name
dat[, c("x", "z")]
```

We can also access a row or multiple rows simultaneously:

```r
# access a row by reference
dat[1, ]
# access multiple rows
dat[2:4, ]
# or similarly you can use
dat[c(1, 3, 4), ]
```

We can also access a mixture of rows and columns, e.g.

```r
dat[1:2, c("y", "z")]
```

BONUS: Tip

Be careful when subsetting just one column! The output will be a vector, **not** a data.frame, unless you tell R by adding the argument drop = FALSE

For example:

```r
# this code returns a new data frame
class(dat[, c("x", "z")])
```

```
## [1] "data.frame"
```

```r
# this code returns a vector
class(dat[, c("x")])
```

```
## [1] "integer"
```

```
# Fix this by adding the argument drop = FALSE
# this code returns a data.frame
class(dat[, c("x"), drop = FALSE])
```

```
## [1] "data.frame"
```

Omitting spefic rows and columns by reference is possible.

```
# omit the first row
dat[-1, ]
```

```
##   x    y        z x2
## 2 2 0.50 2.297397  4
## 3 3 0.75 3.737193  9
## 4 4 1.00 5.278032 16
## 5 5 1.25 6.898648 25
```

```
# omit the second row and first column
dat[-2, -1]
```

```
##      y        z x2
## 1 0.25 1.000000  1
## 3 0.75 3.737193  9
## 4 1.00 5.278032 16
## 5 1.25 6.898648 25
```

```
dat
```

```
##   x    y        z x2
## 1 1 0.25 1.000000  1
## 2 2 0.50 2.297397  4
## 3 3 0.75 3.737193  9
## 4 4 1.00 5.278032 16
## 5 5 1.25 6.898648 25
```

```
# omit a mixture of rows and columns
dat[-c(1, 3), -c(1, 3), drop = FALSE]
```

```
##      y x2
## 2 0.50  4
## 4 1.00 16
## 5 1.25 25
```

11.2 Subsetting using logical operations

We might be interested in creating a subset of the data where a particular condition applies. To subset the data, we can create a logical vector indicating whether or not this condition applies and pass it within the [,] notation.

```
# create a logical vector that meets the condition where
# the x variable is >=2
dat$x >= 2
```

```
## [1] FALSE  TRUE  TRUE  TRUE  TRUE
```

```
# pass the logical vector within the [,] notation
dat[dat$x >= 2, ]
```

```
##   x    y        z x2
## 2 2 0.50 2.297397  4
## 3 3 0.75 3.737193  9
## 4 4 1.00 5.278032 16
## 5 5 1.25 6.898648 25
```

```
# we only see the observations when the condition is TRUE

# note that the logical vector can be defined by a multitude of comparisons
log_vec <- c(dat$x >= 2 & dat$y < 1)
dat[log_vec, ]
```

```
##   x    y        z x2
## 2 2 0.50 2.297397  4
## 3 3 0.75 3.737193  9
```

Subsetting using logical operations can be tricky when the data frame includes missing values (NA).

See an example:

```
newDat <- data.frame(x = 1:5, y = c(NA, NA, 6, 9, NA))
# set a logical vector with the condition y > 7
newDat$y > 6
```

```
## [1]    NA    NA FALSE  TRUE    NA
```

```
# How does output look like?

# If we supply this logical vector within [,] we get a messy output
newDat[newDat$y > 6, ]
```

```
##       x  y
## NA   NA NA
## NA.1 NA NA
## 4     4  9
## NA.2 NA NA
```

```r
# this can be solved with the function which
newDat[which(newDat$y > 6), ]
```

```
##   x y
## 4 4 9
```

The function `which()` removes the missing values from a logical vector.

> ⭐ BONUS: Recap
>
> Logical vectors which return a Boolean vector (TRUE or FALSE) can be used to subset a dataset when a condition in TRUE.
> Remember to use which() if the logical vector contains NA values.

11.3 Subsetting with `subset()`

`subset()` is a *base R function* that returns subsets of vectors, matrices or data frames which meet specific conditions as specified by a logical vector. The function `subset` omits NA values by default.

```r
df <- data.frame(x = 1:20, y = (1:20) ^ 2)
# subset the data where x equals y
subset(df, x == y)
```

```
##   x y
## 1 1 1
```

This function can be sometimes problematic because the logical expression used to subset the data can refer to objects specified in the global environment.

For example:

```r
df <- data.frame(A = 1:20, B = (1:20) ^ 2)
th <- 300
# in the following example, B is a column of df, while th is an object
# of the global environment
subset(df, B > th)
```

```
##     A   B
## 18 18 324
## 19 19 361
## 20 20 400
```

BONUS: Recap

The function subset() is similar to the use of logical vectors to subset a data frame.

12 Summary statistics by column

The function `summary()` provides some summary statistics for each column of the data frame. More specifically:

- **if the column class is integer or numeric**:
  - min and max values
  - the mean and median
  - 1st and 2nd quartiles
  - when missing values (NA and NaN) are present, how many they are
- **if the column class is a factor**:
  - the number of occurrences per factor level
  - the number of incidences of missing values (NA and NaN)

```
# let's look again at the trees dataset
summary(trees)
```

```
##                BOOMSOORT     postcode_nummer            wijk
##  Quercus robur      : 7070   Min.   :6511    Meijhorst   : 6071
##  Tilia x europaea   : 3833   1st Qu.:6525    't Acker    : 4765
##  Fraxinus excelsior: 3546   Median :6536    Zwanenveld  : 4617
##  Fagus sylvatica    : 2363   Mean   :6541    Goffert     : 4187
##  Quercus rubra      : 2292   3rd Qu.:6542    Lent        : 3630
##  (Other)            :42853   Max.   :6663    Brakkenstein: 3472
##  NA's               :   35                   (Other)     :35250
##    PLANTJAAR          ID               x                y
##  Min.   :   0   Min.   :    1   Min.   :180574   Min.   :422747
##  1st Qu.:1970   1st Qu.:17517   1st Qu.:183793   1st Qu.:425443
##  Median :1985   Median :35604   Median :185882   Median :426896
##  Mean   :1979   Mean   :36036   Mean   :185708   Mean   :427069
##  3rd Qu.:2002   3rd Qu.:54250   3rd Qu.:187447   3rd Qu.:428096
##  Max.   :2019   Max.   :73355   Max.   :190868   Max.   :434031
##
##               species_name
##  Quercus robur      : 8367
##  Tilia europaea     : 4477
##  Fraxinus excelsior: 4184
##  Carpinus betulus   : 2842
##  Fagus sylvatica   : 2787
##  (Other)           :39300
##  NA's              :   35
```

13 Contingency tables

BONUS: Definition

The function table() creates tabular results of factor levels in categorical variables; in other words, the function counts the occurrence of factor levels.

It is handy both when exploring a dataset interactively and both to make summary tables for a manuscript.

For example, if we want to know if the trees data are balanced and check if the number of observations in different neighborhood differ, we can type:

```
table(trees$wijk)
```

```
## 
##         Altrade          Biezen      Bottendaal    Brakkenstein         De Kamp 
##            1115            2938             959            3472            3165 
##      Galgenveld         Goffert      Grootstal          Hatert        Hazenkamp 
##            1610            4187            2882            3450            2397 
##       Hengstdal       Heseveld      Hunnerberg            Lent        Meijhorst 
##            2875            1846            1481            3630            6071 
## Neerbosch-Oost     Oosterhout    Stadscentrum        't Acker       Weezenhof 
##            1621            2958            2124            4765            1750 
##       Wolfskuil      Zwanenveld 
##            2079            4617
```

```
# what does the table tell?
```

To check how many trees have been planted after 2017 we can create a logical vector that returns `TRUE` if the PLANTJAAR size is `> 2017` and `FALSE` otherwise and pass the logical vector to the function `table()`.

```
table(trees$PLANTJAAR > 2017)
```

```
## 
## FALSE  TRUE 
## 60785  1207
```

```
# to check missing values we can set the useNA argument to always
table(trees$PLANTJAAR > 2017, useNA = "always")
```

```
##
## FALSE  TRUE  <NA>
## 60785  1207     0
```

To check the trees planted after 2017 in each neighborhood of Nijmegen we can supply both variables to `table()`:

```
table(trees$wijk, trees$PLANTJAAR > 2017)
```

```
##
##                  FALSE TRUE
##   Altrade         1097   18
##   Biezen          2841   97
##   Bottendaal       947   12
##   Brakkenstein    3456   16
##   De Kamp         3102   63
##   Galgenveld      1570   40
##   Goffert         4156   31
##   Grootstal       2780  102
##   Hatert          3409   41
##   Hazenkamp       2364   33
##   Hengstdal       2846   29
##   Heseveld        1827   19
##   Hunnerberg      1443   38
##   Lent            3376  254
##   Meijhorst       6039   32
##   Neerbosch-Oost  1605   16
##   Oosterhout      2786  172
##   Stadscentrum    2050   74
##   't Acker        4734   31
##   Weezenhof       1736   14
##   Wolfskuil       2033   46
##   Zwanenveld      4588   29
```

14 Duplicates and missing values

Sometimes datasets are messy.

For example, some rows might be duplicated. The *base R functions* `duplicated()`, and `unique()` can help us dealing with that.

```
dat <- data.frame(A = c(1, 2, 3, 1), B = c(4, 8, 9, 4))
# duplicated() tells us if some rows in a data frame are duplicates
duplicated(dat)
```

```
## [1] FALSE FALSE FALSE  TRUE
```

```
# to remove duplicated rows use the function unique()
unique_dat <- unique(dat)
# check the new data
unique_dat
```

```
##   A B
## 1 1 4
## 2 2 8
## 3 3 9
```

Sometimes field biologists can record only part of the variables for a specific observation. The variable(s) that cannot be recorded will be missing in the dataset. A missing value is typically identified as NA or NaN. The *base R functions* `any()`, `is.na()` and `complete.cases()` can help us dealing with that.

Let's look at an example with a missing value:

```
dat <- data.frame(A = c(1, 2, 3, NA), B = c(4, NA, 9, 7))
# to check if there are missing values type
any(is.na(dat))
```

```
## [1] TRUE
```

```
# to remove missing values type from the column A
dat[complete.cases(dat$A), ]
```

```
##   A  B
## 1 1  4
## 2 2 NA
## 3 3  9
```

```
# to remove missing values from all columns type
dat[complete.cases(dat), ]
```

```
##   A B
## 1 1 4
## 3 3 9
```

BONUS: Recap

Remember to check a data frame carefully when starting to work with it. Often dataset can be messy and contain mistakes.
Many *base R functions* can help us with dealing with messy datasets: duplicated(), unique(), any(), is.na(), and complete.cases()

15 Join multiple datasets

Often the data we need to answer a research question comes from multiple datasets and in order to answer the question we need to join them. Not to worry! R provides multiple *base R functions* to achieve that.

15.1 Concatenate datasets horizontally or vertically

BONUS: Definition

Concatenating datasets means that we add rows of columns from one dataset to another.

It is possible to join two data frames:

- **vertically** with the *base R function* `rbind()` (r here stands for row) when the two datasets have the same number of columns and same column names
- **horizontally** with the *base R function* `cbind()` (c here stands for column) when the two datasets have the same number of rows

An example with `rbind()`:

```r
teachersLU1_3 <- data.frame(name = c("Aafke", "Melinda", "Michela"),
                            LU = c(1, 2, 3))
teachersLU4_5 <- data.frame(name = c("Coline", "Juan"), LU = c(4, 5))

rbind(teachersLU1_3, teachersLU4_5)
```

```
##      name LU
## 1   Aafke  1
## 2 Melinda  2
## 3 Michela  3
## 4  Coline  4
## 5    Juan  5
```

An example with `cbind()`:

```r
teachersLU1_5 <- data.frame(name = c("Aafke", "Melinda", "Michela", "Coline",
                                     "Juan"), LU = 1:5)
dates <- data.frame(LU = 5:1, date = c("09-18", "09-11", "09-11", "09-04",
                                     "09-04"))

# note that this is messy! The dates are mixed up
cbind(teachersLU1_5, dates)
```

```
##      name LU LU  date
## 1   Aafke  1  5 09-18
## 2 Melinda  2  4 09-11
## 3 Michela  3  3 09-11
## 4  Coline  4  2 09-04
## 5    Juan  5  1 09-04
```

```
# one workaround here is to order the data according to the same column:
cbind(teachersLU1_5[order(teachersLU1_5$LU), ], dates[order(dates$LU), ])
```

```
##      name LU LU  date
## 1   Aafke  1  1 09-04
## 2 Melinda  2  2 09-04
## 3 Michela  3  3 09-11
## 4  Coline  4  4 09-11
## 5    Juan  5  5 09-18
```

```
# but it recommended to use the function merge() instead, see below
```

15.2 Merging datasets by column(s)

BONUS: Definition

It is possible to merge datasets that contain one or a set of columns that uniquely identify the observations and that can be used as keys to link the two datasets.

For example

```
teachersLU1_5 <- data.frame(name = c("Aafke", "Melinda", "Michela", "Coline",
                                     "Juan"), LU = 1:5)
teachersLU1_5
```

```
##      name LU
## 1   Aafke  1
## 2 Melinda  2
## 3 Michela  3
## 4  Coline  4
## 5    Juan  5
```

```
dates <- data.frame(LU = 5:1, date = c("09-18", "09-11", "09-11", "09-04",
                                       "09-04"))
dates
```

```
##   LU  date
## 1  5 09-18
## 2  4 09-11
## 3  3 09-11
## 4  2 09-04
## 5  1 09-04
```

```
# contain the same column LU which can be used to link the two datasets
# additionally each data.frame contains unique values

merge(teachersLU1_5, dates)
```

```
##   LU    name  date
## 1  1   Aafke 09-04
## 2  2 Melinda 09-04
## 3  3 Michela 09-11
## 4  4  Coline 09-11
## 5  5    Juan 09-18
```

```
# this new merged dataset is clean!
```

15.3 Setting the key column

The function `merge()` by default uses the column(s) with common names between the two data frames as keys to join them. It is also possible to specify the column names manually using the arguments `by.x` and `by.y`, where x refers to the first dataset and y to the second one. For example:

```
teachersLU1_5 <- data.frame(name = c("Aafke", "Melinda", "Michela", "Coline",
                                     "Juan"), LU = 1:5)
teachersLU1_5
```

```
##      name LU
## 1   Aafke  1
## 2 Melinda  2
## 3 Michela  3
## 4  Coline  4
## 5    Juan  5
```

```
dates <- data.frame(LearningUnit = 5:1, date = c("09-18", "09-11", "09-11",
                                                 "09-04", "09-04"))
dates
```

```
##   LearningUnit  date
## 1            5 09-18
## 2            4 09-11
## 3            3 09-11
## 4            2 09-04
## 5            1 09-04
```

```
merge(teachersLU1_5, dates, by.x = "LU", by.y = "LearningUnit")
```

```
##   LU    name  date
## 1  1   Aafke 09-04
## 2  2 Melinda 09-04
## 3  3 Michela 09-11
## 4  4  Coline 09-11
## 5  5    Juan 09-18
```

15.4 Merging with duplicates

BONUS: Tip

If there are multiple matches between x and y, all combinations of the matches are returned. This can be messy and it's crucial to make sure that there are **not** duplicated values in the two data frames before merging them.

Let's see what happens when we merge data frames with duplicated rows.

```
# let's see w
teachersLU1_5 <- data.frame(name = c("Aafke", "Melinda", "Michela", "Coline",
                                     "Juan", "Melinda"), LU = c(1:5, 2))
teachersLU1_5
```

```
##      name LU
## 1   Aafke  1
## 2 Melinda  2
## 3 Michela  3
## 4  Coline  4
## 5    Juan  5
## 6 Melinda  2
```

```
dates <- data.frame(LU = 5:1, date = c("09-18", "09-11", "09-11", "09-04",
                                       "09-04"))
dates
```

```
##   LU  date
## 1  5 09-18
## 2  4 09-11
## 3  3 09-11
## 4  2 09-04
## 5  1 09-04
```

```
merge(teachersLU1_5, dates)
```

```
##   LU    name  date
## 1  1   Aafke 09-04
## 2  2 Melinda 09-04
## 3  2 Melinda 09-04
## 4  3 Michela 09-11
## 5  4  Coline 09-11
## 6  5    Juan 09-18
```

```
# the duplicate row carries out in the newly merged data frame

# the recommended practice is to remove the duplicates first
merge(unique(teachersLU1_5), unique(dates))
```

```
##   LU    name  date
## 1  1   Aafke 09-04
## 2  2 Melinda 09-04
## 3  3 Michela 09-11
## 4  4  Coline 09-11
## 5  5    Juan 09-18
```

15.5 Merging with NA values

BONUS: Tip

If there are NA values in the key column(s) used for merging, the returned data frame is very messy and will contain extra rows. Set “incomparables = NA“ to avoid this problem

Let’s see what happens when we merge data frames with NA values in the key column used to join the data sets.

```
# the LU column contains some duplicates
teachersLU1_5 <- data.frame(name = c("Aafke", "Melinda", "Michela", "Coline",
                                     "Juan"), LU = c(1, 2, NA, NA, 5))
teachersLU1_5
```

```
##      name LU
## 1   Aafke  1
## 2 Melinda  2
## 3 Michela NA
## 4  Coline NA
## 5    Juan  5
```

```
dates <- data.frame(LU = c(5, NA, 3, 2, NA), date = c("09-18", "09-11", "09-11",
                                                      "09-04", "09-04"))
dates
```

```
##   LU  date
## 1  5 09-18
## 2 NA 09-11
## 3  3 09-11
## 4  2 09-04
## 5 NA 09-04
```

```
merge(teachersLU1_5, dates)
```

```
##   LU    name  date
## 1  2 Melinda 09-04
## 2  5    Juan 09-18
## 3 NA Michela 09-11
## 4 NA Michela 09-04
## 5 NA  Coline 09-11
## 6 NA  Coline 09-04
```

```
# can you spot the problem?

# the recommended practice is to use incomparables = NA
merge(teachersLU1_5, dates, incomparables = NA)
```

```
##   LU    name  date
## 1  2 Melinda 09-04
## 2  5    Juan 09-18
```

15.6 Types of join

When merging datasets by column name, we can choose among different options or joins (see also the figure):

- **natural join** or `all = FALSE`: returns only the rows from x and y with matching cases
- **full outer join** or `all = TRUE`: returns all rows from both x and y. Not matching values are represented by NA values
- **left outer join** or `all.x = TRUE`: returns all rows of x and all rows of y that are matching. Rows in x with no match in y will have NA values in the new columns
- **right outer join** or `all.y = TRUE`: returns all rows of y and all the matching rows of x. Rows in y with no match in x will have NA values in the new columns

```
teachersLU1_5 <- data.frame(name = c("Aafke", "Melinda", "Michela", "Coline",
                                     "Juan"), LU = 1:5)
teachersLU1_5
```

```
##      name LU
## 1   Aafke  1
## 2 Melinda  2
## 3 Michela  3
```

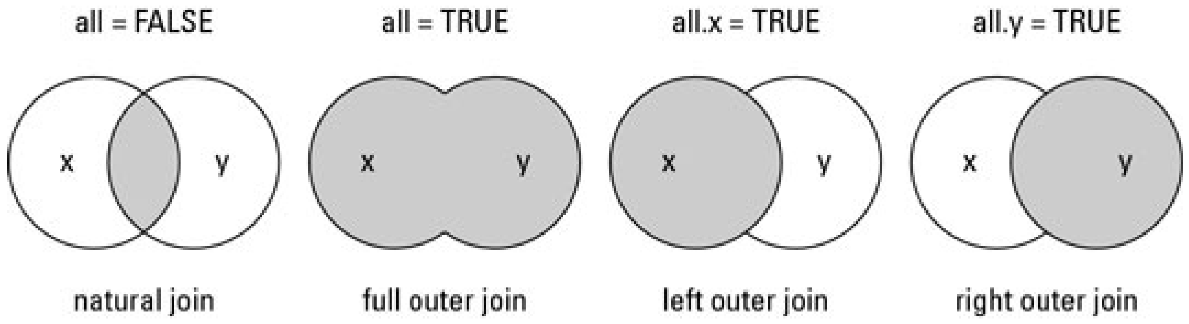


Figure 1 A graphical visualization of the different types of joins. The graph was downloaded from datascienceplus.com.

```
## 4  Coline  4
## 5    Juan  5
```

```
dates <- data.frame(LU = c(7, 5, 3, 2, 1), date = c("09-18", "09-11", "09-11",
                                                     "09-04", "09-04"))
dates
```

```
##   LU  date
## 1  7 09-18
## 2  5 09-11
## 3  3 09-11
## 4  2 09-04
## 5  1 09-04
```

```
# natural join
nat_join <- merge(teachersLU1_5, dates) # equivalent to merge(owl, ids, all = FALSE)
nat_join
```

```
##   LU    name  date
## 1  1   Aafke 09-04
## 2  2 Melinda 09-04
## 3  3 Michela 09-11
## 4  5    Juan 09-11
```

```
# full outer join
full_join <- merge(teachersLU1_5, dates, all = TRUE)
full_join
```

```
##   LU    name  date
## 1  1   Aafke 09-04
## 2  2 Melinda 09-04
## 3  3 Michela 09-11
## 4  4  Coline  <NA>
## 5  5    Juan 09-11
## 6  7    <NA> 09-18
```

```
# left outer join
left_join <- merge(teachersLU1_5, dates, all.x = TRUE)
left_join
```

```
##   LU    name  date
## 1  1   Aafke 09-04
## 2  2 Melinda 09-04
## 3  3 Michela 09-11
## 4  4  Coline  <NA>
## 5  5    Juan 09-11
```

```
# right outer join
right_join <- merge(teachersLU1_5, dates, all.y = TRUE)
right_join
```

```
##   LU    name  date
## 1  1   Aafke 09-04
## 2  2 Melinda 09-04
## 3  3 Michela 09-11
## 4  5    Juan 09-11
## 5  7    <NA> 09-18
```

BONUS: Tip

If uncertain about the type of joint to use, check the dimensions of both data frames before merging and after merging. Do the dimensions correspond to our expectations?

16 Recap

List of the functions learned

- getwd()
- setwd()
- ls()
- rm(),
- save(), save.image()
- load(),
- read.table(), read.csv(), readRDS()
- write.table(), write.csv(), writeRDS()
- ncol(), nrow(), dim()
- names(), colnames(), row.names()
- order()
- which()
- summary(), table()
- head(), tail(), View()
- duplicated(), unique()
- any(), is.na(), complete.cases()
- cbind(), rbind(), merge()

17 Interactive exercise

Open your file view on Windows and locate where you saved the file Nijmegen_trees.csv. Right click on the file, select properties and check its location or path Start RStudio and check your working directory with getwd() Does the working directory match the location of the file? If the answer is not, modify the working directory with setwd()

```r
# Read the file in
data <- read.csv("./RawData/Nijmegen_trees.csv")
# note that we modified the data and added an extra column with clean species
# names so that we can merge with phylogenetic tree later on

# explore the data and the structure
head(data)
tail(data)
data[234:237, ]
dim(data)
nrow(data)
ncol(data)
# order your data by species_nam and postcode_nummer
```

```r
data <- data[order(data$species_name, data$postcode_nummer), ]

# explore the data further
summary(data)
unique(data[data$postcode_nummer == 6524, "species_name"])
# use drop = FALSE to be type consistent, e.g.
unique(data[data$postcode_nummer == 6524,  "species_name", drop = FALSE])
data[data$species_name == "Acer", c("species_name", "wijk")]
# we see a lot NAs at the end. why?
# to avoid it use which
data[which(data$species_name == "Acer"), c("species_name", "wijk")]
subset(data, species_name == "Acer")[, c("species_name", "wijk")]
# What informations can you retrieve from each of
# these functions?

# check what is in your workspace
ls()

#########################################################################
# Question: how do the trees in Nijmegen fit in the tree of life?
# To answer we need additinal data. There are many phylogenies out there,
# we downloaded infos from the Open tree of life
phylogeny <- read.table("./RawData/tree_phylogeny.txt")

# explore the data with the functions summary(),  head()
summary(phylogeny)
head(phylogeny)

# how can we merge the databases? with the species names as key to match them.
colnames(data)
names(data)
colnames(phylogeny)
```

```r
# we can try to merge the datasets:
merge(data, phylogeny)
# ERROR: the process will hang forever, because R does not know how to merge
# the data stop the process with Esc or by pressing the stop button
```

```r
# let's find out why the issue occurred:
colnames(data)
colnames(phylogeny)
# the columns are named differently!

# we could use the argument by
res <- merge(data, phylogeny, by.x = "species_name", by.y = "species")
```

```
# OR # we could change the column names to match:
colnames(data)[colnames(data) == "species_name"] <- "species"
colnames(data)
res <- merge(data, phylogeny)

# we just implemented a natural join

# did the merging do what we was expecting
head(res)
dim(res)
dim(data)
# why not?
# any duplicates?
any(duplicated(res))
any(duplicated(data))
any(duplicated(phylogeny))
# remove duplicates for good:
phylogeny <- unique(phylogeny)
# any missing values in the keys?
any(is.na(data$species))
any(is.na(phylogeny$species))
# how come some trees are not classified?
# likely the data were not typed correctly. If it's your data it's worth to
# go back to your notebooks and check them
# we have no way to change it as we did not collect the data, so we delete
# the data where we miss the species name.
# remove missing values:
data <- data[complete.cases(data$species), ]


# show different results with different all options
# full outer join
res_all <- merge(data, phylogeny, all = TRUE)
head(res_all)
dim(res_all)
dim(data)

# let's merge with the option all.x = T, left outer join
res_x <- merge(data, phylogeny, all.x = TRUE)
# checking:
head(res_x)
dim(res_x)
dim(data)
```

```r
# right outer join
res_y <- merge(data, phylogeny, all.y = TRUE)
head(res_y)
dim(res_y)
dim(data)

# show the picture with the differences between types of joins

# To answer the question  how do the trees in Nijmegen fit in the tree of life
# we need the first option: the left outer join
res_x
write.table(res_x, "./DerivedData/Nijmegen_tree_phylogeny.txt")


# make some summaries with table function:
table(res_x$family)
# two-way table
table_genusXfamily <- table(res_x$genus, res_x$family)
table_familyXgenus <- table(res_x$family, res_x$genus)

# save an Rdata object
save(table_genusXfamily, file = "./DerivedData/table_genusXfamily.Rdata")
# save the whole workspace
save.image(file = "./DerivedData/example_lecture3.Rdata")

# let's clean up our workspace
rm(list = ls())
ls()
# now if we wanted to retriev the data we could simply:
load("./DerivedData/table_genusXfamily.Rdata")
ls()

load("./DerivedData/example_lecture3.Rdata")
ls()
```

18 Exercises

18.1 Introduction

Create an R file called exercises_LU3.R in the LU3 folder. Try opening RStudio from the program's list and the R file. In both cases check the working directory and, if needed, set it manually to the location where you keep the materials for the LU3 Which difference do you notice when opening RStudio from program's list and the R file? Compare your results with those of your neighbour.

18.2 The Lizard dataset

The lizards.txt dataset in the RawData folder includes real-world data about the perching behavior of two species of lizards in South Bimini Island (Shoener, 1968).

The lizards' data set contains the following variables:

- Species (the species of the lizard): a two-level factor with levels Sagrei and Distichus
- Height (perch height): a two-level factor with levels high (greater than 4.75 feet) and low (lesser or equal to 4.75 feet)
- Diameter (perch diameter): a two-level factor with levels narrow (greater than 4 inches) and wide (lesser or equal to 4 inches)

Steps:

1. Read the dataset in R and save it as an object in the workspace
2. Check the class of the object and the class of each column
3. Check the dimensions of the dataset
4. In Rstudio explore the data with the function View()
5. Visualize the top and bottom of the dataset. What information can you retrieve?
6. Make a summary of the dataset. What does the summary tell us in this case?
7. Change the column names from Species, Diameter, and Height to species, diameter and height
8. Add a new column to the dataset (HxD) which combines the Height and Diameter columns (e.g. `low_narrow`). HINT: use the function `paste()` or `pasete0()`
9. Subset the data and show only rows for the Sagrei species
10. How many narrow and wide perches do the two species of lizards use?
11. How many high versus low perches do the two species use?

18.3 The Owl dataset

The owl data (Zuur et al. (2009), Roulin and Bersier (2007)) quantify the number of vocalizations by owl chicks (NegPerChick) when parents are absent in different nests as a function of food treatment (deprived or satiated), the sex of the parent, arrival time of the parent at the nest, and brood size. More information about the data collection can be found in the metadata subfolder.

The owl dataset has been split into two parts: owl_part1.csv and the owl_part2.csv (see the RawData folder).

To solve this exercise:

1. read the two files, owl_part1.csv, and owl_part2.csv from the RawData subfolder

2. how do the two datasets differ?
3. Combine them in a single dataset. Pay attention to the order of the column names before!
4. Order the data by the column Nest and NegPerChick
5. Save the newly combined dataset as a .csv into the DerivedData folder

18.4 The Owl and Parent IDs datasets

Let's work further with the owl data. We want to merge the owl dataset with an additional dataset which includes the identifiers (id) of the parent for each nest, so that we can create a new dataset that includes the ids of the parent in addition to the available data. The owl dataset (owls.csv) and the ids dataset (ParentIds.csv) are ready for use in the RawData folder.

To solve this exercise:

1. Read the two datasets in the console
2. Explore both datasets quickly
3. How can we merge them?
4. Before joining the datasets pay attention to potential problems with duplicates or NA values and the identification of the key column for merging
5. Try the four types of merging and check possible differences
6. Explore the newly created dataset (pay attention to the dimensions)
7. What type of merging is more appropriate?
8. Save the dataset in the DerivedData subfolder as a .txt file

18.5 The Rikz dataset

The rikz dataset includes "*marine benthic data from nine inter-tidal areas along the Dutch coast. The data were collected by the Dutch institute RIKZ in the summer of 2002. In each inter-tidal area (denoted by 'beach'), five samples were taken, and the macro-fauna and abiotic variables were measured.*" See link

- Sample: sample number
- Richness: species richness
- Exposure: index composed of the surf zone, slope, grain size, and depth of anaerobic layer
- NAP: height of sampling station compared to mean tidal level
- Beach: beach identifier or id

To solve this exercise:

1. read the rikz.csv file from the RawData folder
2. Explore the data
3. Modify the column Exposure as a two-levels factor: 10 and 11

4. Modify the column Richness by setting the maximum value to 10
5. How is the exposure distributed among Beach ids?
6. How is the Richness distributed considering the exposure?
7. How is the richness distributed among Beach ids?
8. Subset the dataset to include only rows where the Richness is below 5. Which Beach ids do they correspond to?
9. How many samples have Richness values below 5 per Beach?

18.6 Optional

If you finished all the above exercises and have some time left you could load a dataset you need to use for your thesis, explore it and clean it from within R.

Learning unit 4 - Controlling the flow

Without solutions

Juan Gallego-Zamorano

Contents

1 Learning Unit Goals

By the end of the class you will be able to:

1. Control the flow of your script using "conditional statements"
2. Iterate over a code several times using loops
3. Exit loops to catch potential errors inside them

2 Functions we are going to use

- If, If/else and ifelse statements
- While loops, for loops, nested loops, break, next and try
- Integrating ifelse statements inside loops

3 Conditional statements

Conditional statements are used to specify the execution of your code. They are really useful if you want to run only some parts of your code if a condition is met, or for running a code several times. In this learning unit you will learn the use of **“if statements”** (conditional statements) and **“loops”**.

4 IF statement

Everyday we do hundreds of **IF** statements in our life. Every decision we make has some implications and therefore we use **IF** statements in our head to decide what to do. For example:
-**IF** I eat, **THEN** I will gain energy, **ELSE** I will be tired...
-**IF** I pay attention now, **THEN** I will understand this lecture, **ELSE** I will fail the course straight away!
Another really useful example is to think in a traffic light as below (We will talk about the yellow light later...)

The Traffic Light: An Everyday IF Statement
IF light is red, THEN stop!
IF light is yellow, THEN what???
IF light is green, THEN go!

In R, the structure of an IF statement is:

```
if(condition) {
  statement
}
```

This is read as: If the “condition” is TRUE, THEN it will do the “statement”. The condition can be either a relational or logical statement, which returns a Boolean value, such that the condition is either TRUE or FALSE. The condition must be of length equal to one. If the condition is, for example, a vector the program will return a weird output with a warning message “the condition has length > 1 and only the first element will be used”.

As a quick recap relational operators are: < **less than**; > **greater than**; <= **less than or equal to** ; >= **greater than or equal to**; == **equal to**; and != **not equal to**. While logical operators are: ! **not**; & **and**; && **logical and**; | **or**; || **logical or**.

For example:

```
if(1==1) {
  print("Is the same number")
}
```

```
## [1] "Is the same number"
```

If the condition (expression) is FALSE, THEN it won’t do the statement (so nothing will be outputted in the console)

```
if(1==2) {
  print("Is the same number")
}
```

If we want the code to output something when the condition is FALSE, we have to include the **ELSE** clause:

```
if(1==2) { # Because this is FALSE
  print("Is the same number")
} else { # THEN do the next statement
  print("Different numbers")
}
```

```
## [1] "Different numbers"
```

As you can imagine, we can repeat this structure to check more complex conditions. For example:

```
x <- 1 # We create a variable call "x" with the value 1

if (x < 0) {#IF X is a number below 0 THEN print "Negative number"
  print("Negative number")
} else if (x > 0) {#ELSE check if it is above 0 and print "Positive number"
  print("Positive number")
} else {# IF it is not below nor above, then it is 0!
  print("The number is zero")
}
```

```
## [1] "Positive number"
```

```
x <- 0 # We create a variable call "x" with the value 0

if (x < 0) {#IF X is a number below 0 THEN print "Negative number"
  print("Negative number")
} else if (x > 0) {#ELSE check if it is above 0 and print "Positive number"
  print("Positive number")
} else {#IF it is not below nor above, then it is 0!
  print("The number is zero")
}
```

```
## [1] "The number is zero"
```

Moreover, if statements can also be coded in one line with ifelse:

```
# The structure of the ifelse is as follows
ifelse(condition, is TRUE, is FALSE)
```

`ifelse(condition, yes, no)` where test is a logical obejct that can be either TRUE or FALSE. Where the test is TRUE, ifelse() replaces **TRUE** with whatever operation is in **yes**. Similarly, where the test is **FALSE**, ifelse() replaces FALSE with whatever operation is in **no**. The cool thing about ifelse() is that it can be applied to vectors, so the condition can be a vector. This would not be possible with an if/else statement, which requires the condition to have the length of one.

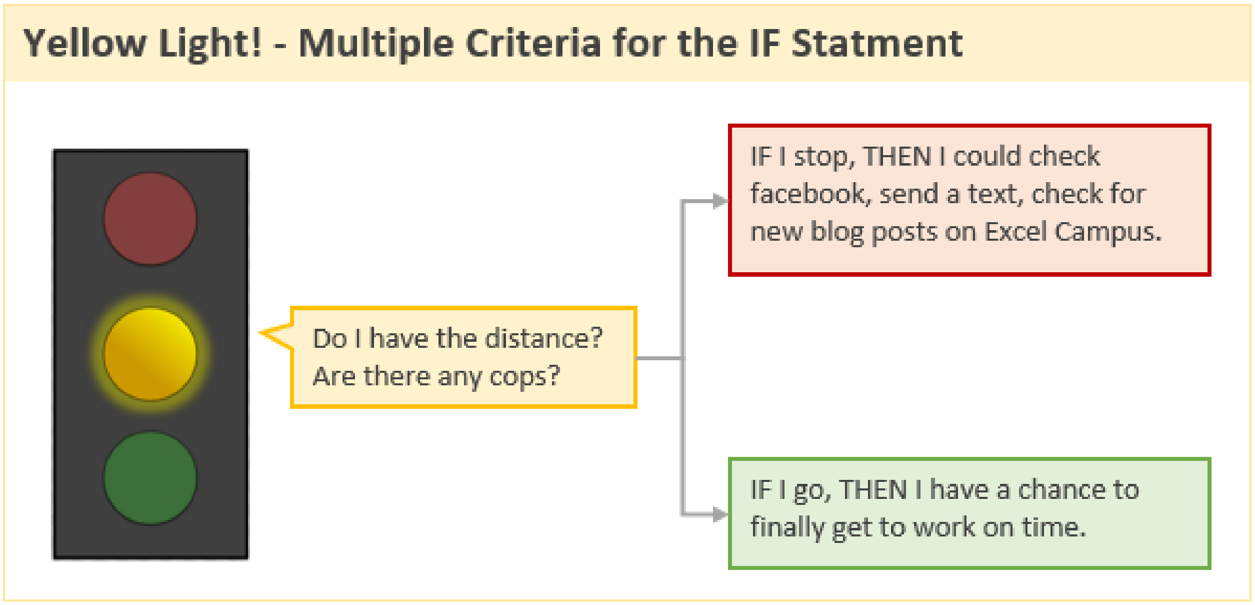
```
x <- 1
ifelse(x < 0, "Negative number", "Positive number")
```

```
## [1] "Positive number"
```

```r
ifelse(x > 0, "Positive number", "Negative number")
```

```
## [1] "Positive number"
```

Do you remember the traffic light?
Now that we know how to apply an "if/else" nested statement we can take a more complex decision.
Let's assume that the traffic light is yellow. We will go if we are at less than 20m from the traffic light, otherwise we will stay and wait for the green light.


Yellow Light! - Multiple Criteria for the IF Statment
Do I have the distance? Are there any cops?
IF I stop, THEN I could check facebook, send a text, check for new blog posts on Excel Campus.
IF I go, THEN I have a chance to finally get to work on time.

Lets code it:

```r
light <- "yellow" # Let's assign a color to our traffic light
distance <- 25 # And a distance to the traffic light

if(light == "red"){

  print("Is RED! STOP!")

} else if(light == "green"){

  print("Is GREEN! GO!!")

} else if (light == "yellow") {

  print("Whatch out! Is YELLOW!")

  if(distance < 20){

    print("RUN YOU CAN MAKE IT!!")

  } else{
    print("STOP YOU WILL NOT MAKE IT!!")
  }
}
```

```
## [1] "Whatch out! Is YELLOW!"
## [1] "STOP YOU WILL NOT MAKE IT!!"
```

Biologists often use **IF** statements with conditions based on comparisons between data. By means of IF statements they can check wheter their code works or not, exit your code if a condition is meet, or assign new variables based on logical conditions.
For example with our data of the trees in Nijmegen, we can use the if statements to:

1. Compare the age between different trees
2. Assign an age for the different trees

Let's read the data:

```
# Change the relative path or use read.csv(file.choose()) to find the data
trees <- read.csv("./RawData/Nijmegen_trees.csv")
names(trees)
```

```
## [1] "postcode_nummer" "wijk"            "BOOMSOORT"       "PLANTJAAR"
## [5] "ID"              "x"               "y"
```

Explanation of each column of the database:

- Postoce_number: The postcode of the district where a tree is planted
- Wijk: The name of the district where each tree is planted
- Boomsoort: The species of each tree
- Plantjaar: The age of each tree
- ID: A unique identifier of each tree
- x: Longitude where each tree is planted
- y: Latitude where each tree is planted

Let's get two random trees

```
tree1 <- trees[12670,] # A Random tree
tree1
```

```
##       postcode_nummer    wijk       BOOMSOORT PLANTJAAR    ID        x
## 12670            6532 Goffert Fagus sylvatica      1890 14568 185949.6
##              y
## 12670 425736.2
```

```
tree2 <- trees[19762,] # Another random tree
tree2
```

```
##       postcode_nummer   wijk     BOOMSOORT PLANTJAAR    ID        x
## 19762            6535 Hatert Quercus robur      1980 26987 184922.1
##              y
## 19762 423579.1
```

1. Compare the age between different trees: Which of the two trees is older?

```
# Which tree is older?
if (tree1$PLANTJAAR < tree2$PLANTJAAR){
  print("Tree 1 is older than Tree 2")
} else {
  print("Tree 2 is older than Tree 1")
}
```

```
## [1] "Tree 1 is older than Tree 2"
```

2. Assign an age for the different trees: If we assume that a tree from the 1900 or earlier is "old", then we can assign an age to the different trees:

```
# We can create a dichotomous variable which can take only two values:
# Old or Young.
# The value Old is assigned if the tree was planted before or in 1900,
# the value Young if the tree was planted after 1900

if(tree1$PLANTJAAR <= 1900) {
  tree1$Age <- "Old"
  print(tree1$Age)

} else {
  tree1$Age <- "Young"
  print(tree1$Age)

}
```

```
## [1] "Old"
```

```
# See that Tree 1 is old
tree1
```

```
##       postcode_nummer    wijk       BOOMSOORT PLANTJAAR    ID        x
## 12670            6532 Goffert Fagus sylvatica      1890 14568 185949.6
##              y Age
## 12670 425736.2 Old
```

```
if(tree2$PLANTJAAR <= 1900) {
  tree2$Age <- "Old"
  print(tree2$Age)

} else {
  tree2$Age <- "Young"
  print(tree2$Age)

}
```

```
## [1] "Young"
```

```
# See that Tree 2 is young
tree2
```

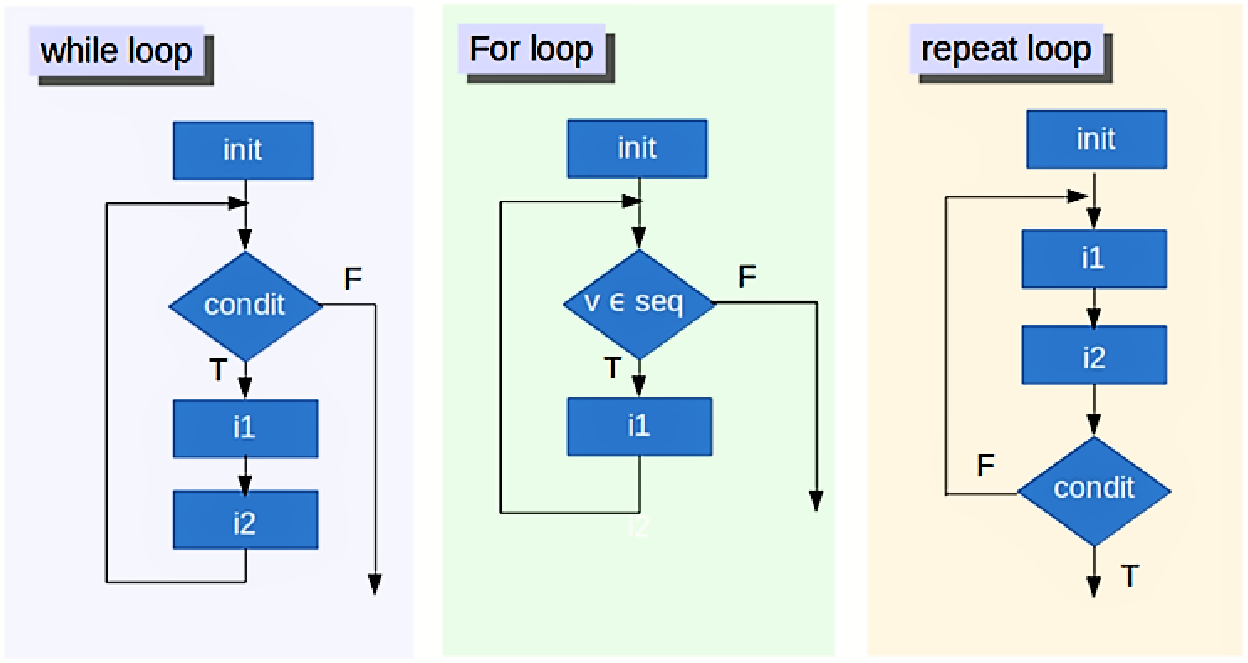
```
##       postcode_nummer   wijk     BOOMSOORT PLANTJAAR    ID        x
## 19762            6535 Hatert Quercus robur      1980 26987 184922.1
##              y   Age
## 19762 423579.1 Young
```

However, what if we want to assign an age **for each of the 61992** trees in Nijmegen **???** This is almost a impossible mission to do it one by one, should be possible thought, or we could use the so called **LOOPS** for it.

5 LOOPS

Loops are a way to iterate over a code several times. There are three main loops:

1. While loops
2. For loops
3. Repeat loops



5.1 While loops

While loops are somewhat similar to if statement, they execute the expression inside them as long as the condition (expression to test) is TRUE. The difference with the if statements is that while loops will continue to execute the code as long as the condition is TRUE. It is crucial that that the condition part of a while loop becomes FALSE at some point during the execution. Otherwise, the while loop will run for ever.

```
# If the condition is TRUE then it will execute the expression for ever
while(condition) {
  expression
}
```

Imagine that you drive a car and that you are accelerating. While you are accelerating, the speed meter tells you that you increase your speed every 5km/h. In programming, this can be translated into a while loop. However, be careful! If you never break or set a maximum speed then the while loop will keep increasing the speed and will last forever. In the example below we set that the while loop will run until the speed reaches the maximum of 50 km/h.

```r
speed <- 0

while(speed <= 50){ # it will continue until the speed is over 50
  print(paste("Speed is", speed, "km/h"))
  speed <- speed + 5
}
```

```
## [1] "Speed is 0 km/h"
## [1] "Speed is 5 km/h"
## [1] "Speed is 10 km/h"
## [1] "Speed is 15 km/h"
## [1] "Speed is 20 km/h"
## [1] "Speed is 25 km/h"
## [1] "Speed is 30 km/h"
## [1] "Speed is 35 km/h"
## [1] "Speed is 40 km/h"
## [1] "Speed is 45 km/h"
## [1] "Speed is 50 km/h"
```

```r
# The computer prints your speed until 50km but now the speed is higher than
# that so the While loop stoped. This is because the condition speed <= 50
# becomes false for values of speed of 51+
print(speed)
```

```
## [1] 55
```

If we remove the last line inside the While loop, it will continue for ever! This is because the speed is always less than 50! **ALWAYS make sure the while loop finishes at some point!!** i.e. the condition should become FALSE at some point.

```r
speed <- 0

while(speed <= 50){
  print(paste0("Speed is set to ", speed))

}
```

Let's combine this while loop with the "if statements" that we learnt before.
Imagine that you are driving and you enter in a town road. The speed limit is 30km/h but you are going at 60km/h. You will slowly decelerate and the computer of the car will send you a message to reduce your speed until you are safe from the cops.

```r
speed <- 60

while (speed > 30) {
  print(paste("Your speed is", speed,"km/h"))

  if (speed > 40) {

    print("Slow down!!!")
    speed <- speed - 5

  } else if(speed <= 40){

    print("Slow down a bit more...")
    speed <- speed - 2

  }
}
```

```
## [1] "Your speed is 60 km/h"
## [1] "Slow down!!!"
## [1] "Your speed is 55 km/h"
## [1] "Slow down!!!"
## [1] "Your speed is 50 km/h"
## [1] "Slow down!!!"
## [1] "Your speed is 45 km/h"
## [1] "Slow down!!!"
## [1] "Your speed is 40 km/h"
## [1] "Slow down a bit more..."
## [1] "Your speed is 38 km/h"
## [1] "Slow down a bit more..."
## [1] "Your speed is 36 km/h"
## [1] "Slow down a bit more..."
## [1] "Your speed is 34 km/h"
## [1] "Slow down a bit more..."
## [1] "Your speed is 32 km/h"
## [1] "Slow down a bit more..."
```

```
 # Now we should be safe, lets check our speed
print(paste("Your speed is", speed,"km/h"))
```

```
## [1] "Your speed is 30 km/h"
```

While loops are useful if we do not know the number of iterations, i.e. when the number of iterations are not predictable beforehand. For example: count the number of clicks on a web page banner within the next two days, or the number of birds migrating through a specific point...In these cases, we do not know the exact number of iterations that we need to count those numbers.

However, we do normally know the number of iterations and therefore the most common type of loop is the **For loop**.

5.2 For loops

For loops are the most used loop. They run an expression (code) a pre-defined number of times. See its structure:

```
#The structure is read as: for each variable, in a sequence...
#execute the expression
for(variable in sequence) {
  expression
}
```

A variable can be for example an index, or a position, or an element in a sequence. The variable will take each value of the sequence and will do an iteration.
Imagine that we want to print every letter in the English alphabet, ABC. Because we know that the English alphabet ABC has 26 letters, we know the number of times that we want to iterate in our loop, therefore 26 is our sequence.
See how is coded and read it: For each **variable (one_letter)** in the **sequence (ABC)**, do the **expression (print(one_letter))**.
In this case, "one_letter" will adopt each value in the ABC and do the expression the number of times pre-defined in the sequence (26).

```
ABC <- letters # letters the sequence of letters in the abc in lowercase

for(one_letter in ABC) {
  print(one_letter)
}
```

```
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
## [1] "e"
## [1] "f"
## [1] "g"
## [1] "h"
## [1] "i"
## [1] "j"
## [1] "k"
## [1] "l"
## [1] "m"
## [1] "n"
## [1] "o"
## [1] "p"
## [1] "q"
## [1] "r"
## [1] "s"
## [1] "t"
## [1] "u"
## [1] "v"
## [1] "w"
## [1] "x"
## [1] "y"
## [1] "z"
```

Below there are two more examples of how the for loops works and they are read.
For each number in the sequence of ten, print the number…
In this case, "number" will take each value in the sequence of 10 (it will go from 1 to 10), and print everytime a "new_number" that in this case is the same.

```
ten <- 1:10 # or seq(from = 1, to = 10, by = 1)

for(number in ten) {
  new_number <- number
  print(new_number)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

For each number in the sequence of ten, take number and sum 0.1 and print the decimal number…

```
for(number in ten) {
  decimal <- number + 0.1
  print(decimal)
}
```

```
## [1] 1.1
## [1] 2.1
## [1] 3.1
## [1] 4.1
## [1] 5.1
## [1] 6.1
## [1] 7.1
## [1] 8.1
## [1] 9.1
## [1] 10.1
```

As before with the while loop, we can integrate "if statements" inside the for loops.
In the example below, each number will be summed 0.1 and printed. In addition, if the number is equals to 6, then it will print "The number is 6" before summing 0.1.

```
for(number in ten) {

  if(number == 6) {
    print("The number is 6")
  }

  decimal <- number+0.1
  print(decimal)
}
```

```
## [1] 1.1
## [1] 2.1
## [1] 3.1
## [1] 4.1
## [1] 5.1
## [1] "The number is 6"
## [1] 6.1
## [1] 7.1
## [1] 8.1
## [1] 9.1
## [1] 10.1
```

Two main **control statements** can be used to control the behaivour of the for loop. **Break** and **Next**.
In the example below, if the number is 6 then the loop will **break**, it will stop!

```
for(number in ten) {

  if(number == 6) {
    print("The number is 6 so I won't continue printing")
    break
  }

  decimal <- number+0.1
  print(decimal)
}
```

```
## [1] 1.1
## [1] 2.1
## [1] 3.1
## [1] 4.1
## [1] 5.1
## [1] "The number is 6 so I won't continue printing"
```

Next will jump to next iteration of the loop.

```
for(number in ten) {

  if(number == 6) {
    print("The number is 6 so I won't add a decimal but I continue in 7")
    next
  }

  decimal <- number+0.1
  print(decimal)
}
```

```
## [1] 1.1
## [1] 2.1
## [1] 3.1
## [1] 4.1
## [1] 5.1
## [1] "The number is 6 so I won't add a decimal but I continue in 7"
## [1] 7.1
## [1] 8.1
## [1] 9.1
## [1] 10.1
```

Sometimes you might want to run your loop enterily and test if there are any errors, in other words, you want to try the loop. That is possible with the **TRY** argument. If there is any error, **try** skips over the error-causing input and continues on with the rest of the lopp.
In the example below, the loop will stop and throw us an error when trying to do the log of “oops” (because it’s impossible!):

```
inputs = list(1, 2, 4, 'oops', 0, 10)

for(number in inputs) {
  print(paste("log of", number, "=", log(number)))
}
```

If we use the **try** function, the output of the loop will show that there is an error when trying to do the log of “oops” but it will skip it and continue with the next value of the input.

```
inputs = list(1, 2, 4, 'oops', 0, 10)

for(number in inputs) {
  try(print(paste("log of", number, "=", log(number))))
}
```

```
## [1] "log of 1 = 0"
## [1] "log of 2 = 0.693147180559945"
## [1] "log of 4 = 1.38629436111989"
## Error in log(number) : non-numeric argument to mathematical function
## [1] "log of 0 = -Inf"
## [1] "log of 10 = 2.30258509299405"
```

A common way to write loops is to use **indexes**. Previously we told you that “one_letter” or “number” was our index. However, it is more common to use a letter e.g. i,j,k.
See the examples below:

```r
# i is the looping index and it can be used to access info of the sequence
for(i in 1:length(ten)) {
  print(ten[i])
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

```r
# i is the looping index and it can be used to access info of the sequence
for(i in 1:length(ten)) {
  print(paste0(ten[i], " is the ", i, "th number of the sequence"))
}
```

```
## [1] "1 is the 1th number of the sequence"
## [1] "2 is the 2th number of the sequence"
## [1] "3 is the 3th number of the sequence"
## [1] "4 is the 4th number of the sequence"
## [1] "5 is the 5th number of the sequence"
## [1] "6 is the 6th number of the sequence"
## [1] "7 is the 7th number of the sequence"
## [1] "8 is the 8th number of the sequence"
## [1] "9 is the 9th number of the sequence"
## [1] "10 is the 10th number of the sequence"
```

To improve the result of the previous loop, we can use some “if else” statements.

```r
# i is the looping index and it can be used to access info of the sequence
for(i in 1:length(ten)) {
  if(i == 1){
    print(paste0(ten[i], " is the ", i, "st number of the sequence"))
  } else if(i == 2){
    print(paste0(ten[i], " is the ", i, "nd number of the sequence"))
  } else if(i == 3){
    print(paste0(ten[i], " is the ", i, "rd number of the sequence"))
  } else{
    print(paste0(ten[i], " is the ", i, "th number of the sequence"))
  }

}
```

```
## [1] "1 is the 1st number of the sequence"
## [1] "2 is the 2nd number of the sequence"
## [1] "3 is the 3rd number of the sequence"
## [1] "4 is the 4th number of the sequence"
## [1] "5 is the 5th number of the sequence"
## [1] "6 is the 6th number of the sequence"
## [1] "7 is the 7th number of the sequence"
## [1] "8 is the 8th number of the sequence"
## [1] "9 is the 9th number of the sequence"
## [1] "10 is the 10th number of the sequence"
```

Tip: To test and ease write a loop is useful to first set the index to 1 (i.e. i in 1 OR i <- 1) and run the snipped of code for that specific value. When it works, you set the index to 2 (i.e. i in 2 OR i <- 2) and run it again, and so on for a couple of times. Then, when the code looks like it’s working fine, you create the final loop (i.e. i in 1:seq). In this way you can better understand which line of your code is working and which one is causing issues. When programming **always** try to make the problem smaller.

Do you remember our problem with the trees? We wanted to assign an age **for** each tree. First we need to ask: How many trees do we have?

```
str(trees)
```

```
## 'data.frame':    61992 obs. of  7 variables:
##  $ postcode_nummer: int  6524 6524 6524 6524 6524 6524 6524 6524 6524 6524 ...
##  $ wijk           : Factor w/ 22 levels "'t Acker","Altrade",..: 7 7 7 7 7 7 7 7 7 7
##  $ BOOMSOORT      : Factor w/ 524 levels "Abies","Abies nordmanniana",..: 66 66 66 66
##  $ PLANTJAAR      : int  1930 1930 1930 1950 1960 1950 1950 1930 1930 1930 ...
##  $ ID             : num  256 257 258 259 260 261 262 263 264 265 ...
##  $ x              : num  188114 188064 188018 188032 188204 ...
##  $ y              : num  427421 427416 427411 427311 427419 ...
```

```
length(trees) # Is this correct?? - Why??
```

```
## [1] 7
```

```
# This is the correct number that we want, our sequence
length(trees$ID)
```

```
## [1] 61992
```

```
# We can assign a variable for the sequence, number of trees
n_trees <- length(trees$ID)
```

Now that we know our sequence, we can create the for loop:

```
for(i in 1:n_trees) { #for the index i in sequence of 1 to the number of trees

  if(trees$PLANTJAAR[i] <= 1900) { #if the tree ith was planted before 1900
    trees$Age[i] <- "Old" #then assign the Age of Old
  } else {                #else assign the Age of Young
    trees$Age[i] <- "Young"
  }

}
```

Moreover, we check how many old and young trees we have using the table function:

```r
table(trees$Age)
```

```
##
##   Old Young
##   561 61431
```

5.3 Nested loops

Imagine that now we want to know for each district, the number of trees of the different ages. The previous problem had only one dimension (age), but now we have two dimensions (district, age). To solve this, we need to iterate through each district, and through each tree to know its age. Therefore, we need to create two individual sequences and two individual indexes. This is know as **nested for loop**.

Objective: create a database per district and with the number of old and young trees per district.
Note: `%in%` is a binary operator, which returns a logical vector indicating if there is a match or not for its left operand. It is useful to subset datasets based on a condition that we know.

```
# the number of districts
n_districts <- length(unique(trees$wijk))

# Empty data frame with number of districts
trees_district <- data.frame(District = rep(NA, n_districts),
                             Old = rep(NA, n_districts),
                             Young = rep(NA, n_districts))

# i will go from 1 to the final number of districts
# so for i in the sequence of 1 to the total number of districts...
for(i in 1:n_districts){

  # we get the i district (one district)
  one_district <- as.character(unique(trees$wijk)[i])

  # we used the %in% to subset the dataset and
  # select all the trees in one_district
  trees_one_district <- trees[trees$wijk %in% one_district,]

  # j will go from 1 to the total number of trees in a district,
  # they have a unique identifier so we can get the total number
  # of trees with lenght(length(trees_one_district$ID))
  # So the for loop will be: for j in the sequence of 1 to number of trees in a distri
  for(j in 1:length(trees_one_district$ID)){

    # Check if the tree ith was planted before 1900
    if(trees_one_district$PLANTJAAR[j] <= 1900) {

      # then assign the Age of Old
      trees_one_district$Age[j] <- "Old"

    } else {                # else assign the Age of Young
      trees_one_district$Age[j] <- "Young"
    }
  }

  # Import the name of the district to the empty database
  trees_district$District[i] <- as.character(unique(trees_one_district$wijk))

  # Check if there is any Young tree and fill the number of young trees
  # or put 0

  # This is read as: Is there "Young" %in% all the list of ages?
  if("Young" %in% trees_one_district$Age){
    trees_district$Young[i] <- length(which(trees_one_district$Age == "Young"))
  } else {
    trees_district$Young[i] <- 0
  }

  # Check if there is any Old tree and fill the number of young trees
  # or put 0
  if("Old" %in% trees_one_district$Age){
    trees_district$Old[i] <- length(which(trees_one_district$Age == "Old"))
  } else {
    trees_district$Old[i] <- 0
  }

}
```

Lets check the result:

```
trees_district
```

```
##          District Old Young
## 1      Galgenveld   0  1610
## 2         Altrade  26  1089
## 3    Brakkenstein 157  3315
## 4        Hengstdal  64  2811
## 5      Hunnerberg  13  1468
## 6         Goffert  67  4120
## 7       Grootstal  19  2863
## 8          Hatert   2  3448
## 9    Stadscentrum  26  2098
## 10     Bottendaal   1   958
## 11      Hazenkamp   7  2390
## 12      Meijhorst   7  6064
## 13     Zwanenveld   8  4609
## 14      Weezenhof   0  1750
## 15        De Kamp   2  3163
## 16       't Acker   5  4760
## 17         Biezen   3  2935
## 18       Heseveld  44  1802
## 19      Wolfskuil   5  2074
## 20           Lent  80  3550
## 21     Oosterhout   0  2958
## 22 Neerbosch-Oost  25  1596
```

5.4 Repeat loops

A repeat loop is used to iterate over a block of code multiple number of times without any condition to exit. However, they are beyond the scope of this lecture so in case you want to know more about them here is a link:
https://www.datamentor.io/r-programming/repeat-loop/

6 Exercises

Recently Galán-Acedo et al. (2019) published a database of ecological traits of the world's primates. This database has broad applicability in primatological studies, and can potentially be used to address many research questions at all spatial scales, from local to global. However, this database is provided in different csv files (i.e. one for body mass sizes, one for habitats, and one for species-specific home ranges). In these exercises you will create a new database out of Galán-Acedo's one with new useful parameters and you will be able to answer some ecological questions about primates. You can find the full dataset in: 10.5281/zenodo.1342458

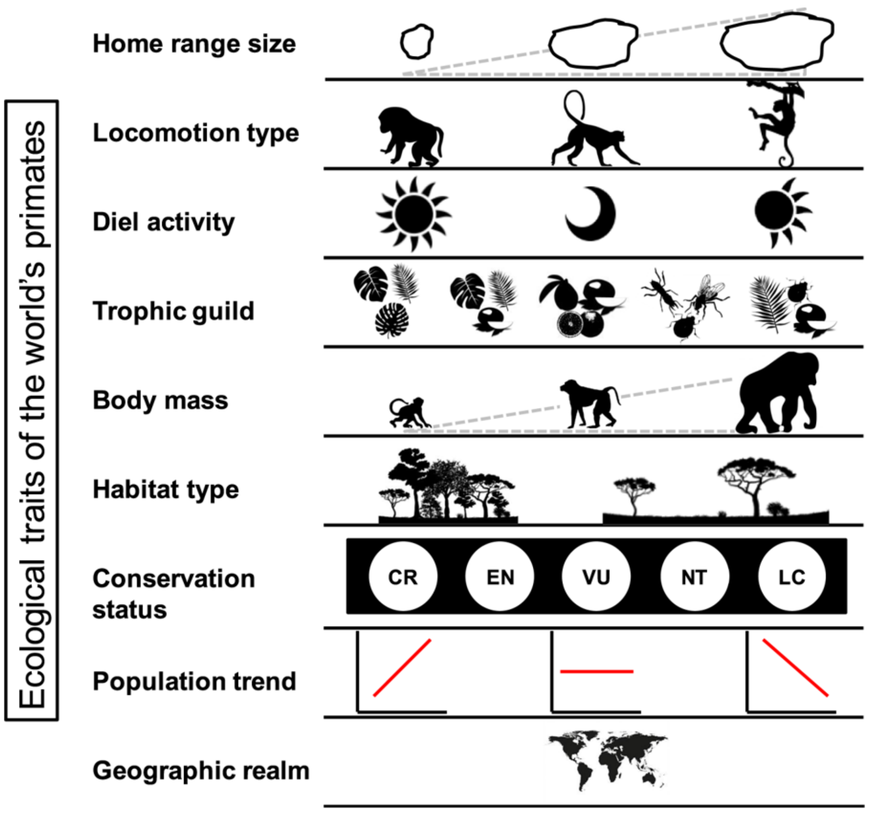


Fig. 1: Summary of the ecological traits of the world's primates included in the database. From left to right pictures represent: (1) home range size gradient from small to large; (2) locomotion types are terrestrial, both locomotion types, and arboreal; (3) diel activity includes diurnal, nocturnal and cathemeral; (4) trophic guild includes folivore, folivore-frugivore, frugivore, insectivore, omnivore, and gummivore (the latter not depicted); (5) body mass gradient from small to large; (6) habitat type includes seven categories (see text) but only two are depicted as examples (forest and savannah); (7) IUCN conservation status includes seven categories, with five depicted here (CR critically endangered, EN endangered, VU vulnerable, NT near threatened and LC least concern); (8) population trend is represented by three graphs indicating increasing, stable and decreasing populations; and (9) geographic realm is represented by a global map.

6.1 Difficulty level = Basic

First you need to read all the different datasets: - Primates_BodyMass.csv: With information per species about body size - Primates_Habitat.csv: With information about the type of habitat used per species - Primates_HomeRange.csv: With information about the size of home range per species

Remove the duplicates using the duplicated function (see LU3) and:

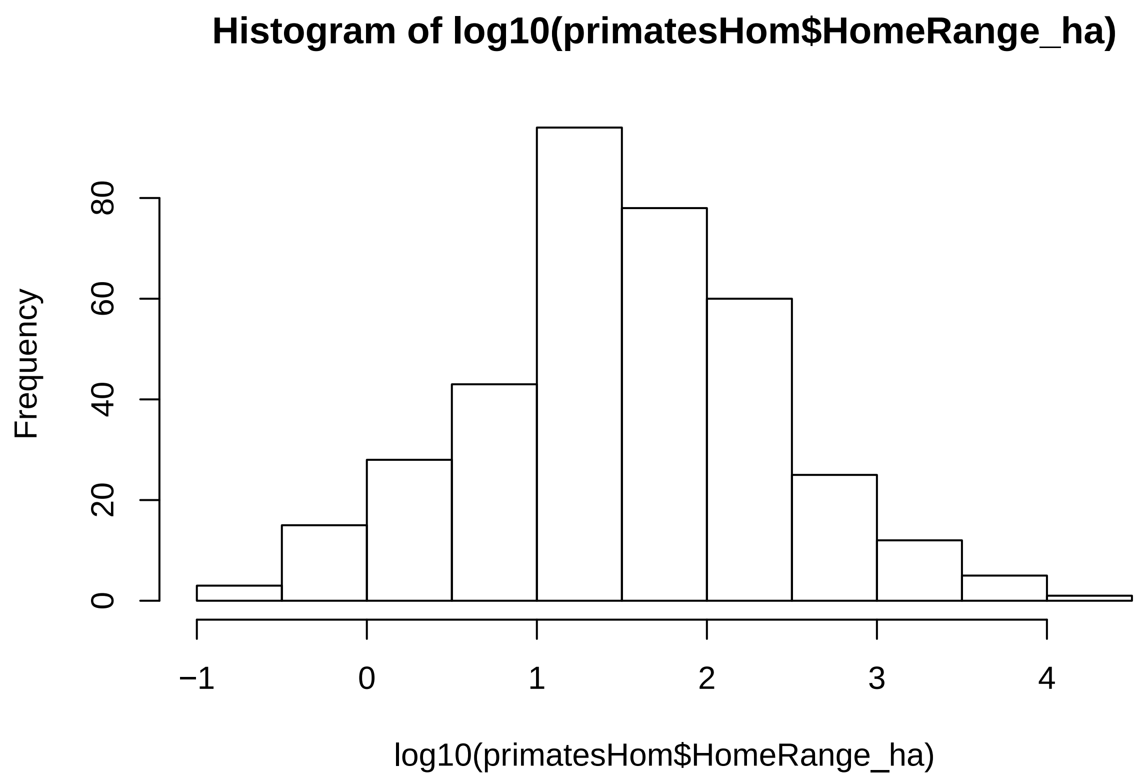
1. Create a conditional statement that outputs which primate is bigger, smaller or the same size between: *Alouatta belzebul* or *Trachypithecus barbei*.

2. Considering that species occuring in one single habitat are specialist of that habitat. Create a variable using ifelse(), that sais if a species is a "Habitat specialist" or a "Habitat generalist". How many specialist and generalist species are in the dataset?

6.2 Difficulty level = Medium

3. I wanted to create a For loop to assign for each primate species if they have a small, medium or big home range.
   I decided to do that based on the statistics of the value of their home range. For that, I checked the histogram of the log10 value of the home range, and the summary statistics, and I found that the 1$^{\text{st}}$ quantile is 1 which corresponds to 10ha (10^1ha) and the 3$^{\text{rd}}$quantile is around 2 which corresponds to 100ha (10^2).
   These values seems perfect for my purpose, however, I was not able to create the For loop because is giving an error. Can you help me??
   For guidence, someone told me that there should be around 89 small-sized range species, 172 medium-sized range, and 103 big-sized range species.

```
hist(log10(primatesHom$HomeRange_ha))
```



```
summary(log10(primatesHom$HomeRange_ha))
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's 
## -0.5229  1.0212  1.4983  1.5408  2.0969  4.0909     141
```

This is my trial of For loop that didn't work out…

```
# I created a variable with the log10 values of the home range
primatesHom$logHomerange <- log10(primatesHom$HomeRange_ha)

# I created a variable with NA that will contain if the range is small, medium or big
primatesHom$HomerangeSize <- NA

# And now I start the for loop with the index =) but it doesn't work!
for(i in length(primatesHom$Species)){

  sp_i <- primatesHom[i]

  if(sp_i$logHomerange < 1){

    primatesHom[i]$HomerangeSize <- "small"

  } else if(sp_i$logHomerange > 1){

    primatesHom[i]$HomerangeSize <- "medium"

  } else {

    primatesHom[i]$HomerangeSize <- "large"

  }

}
```

The solution is:

```
# To be given
```

6.3 Difficulty level = Difficult

4. Create a similar For loop as in 3. but with body mass sizes. Some experts consider mammals below 1kg small-sized, between 1kg and 10kg medium-sized, between 10kg and 20kg large-sized, and above 20kg very-large-sized.

We have created different variables in the different datasets, however to answer some ecological questions, we need to merge them and do some more processing…

5. Merge all the datasets and answer the below questions:

- What is the percentage of species per body size that are generalists?
- Can we say that primates are a generalist taxonomic group? (interpret your results)
- Does the range size of the species increase with increasing the body size?

LU 5: The ‘apply’ family, aggregate and debugging

With exercises and solution

Maarten Broekman & Melinda de Jonge

m.broekman@science.ru.nl, *m.dejonge@fnwi.ru.nl*

Contents

1 Learning goals

By the end of the class you will be able to:

1. use functions in the apply() family to avoid the use of for loops
2. use the aggregate function to calculate summary statistics of subsets of a data set
3. solve the most common R errors and find solutions to other less common errors

2 Functions we are going to use

- for loops
- subset()
- apply()
- lapply()
- sapply()
- tapply()
- aggregate()

3 In this Learning Unit

- The "apply" family consists of several functions that can be used to apply a function to e.g. all rows or columns of a matrix/dataframe or all elements of a list. In this learning unit, we discuss the following members of the apply family: apply(), lapply(), sapply() and tapply(). These functions often return results that can also be obtained with loops (discussed in LU4), but are much faster.
- The aggregate() function can be used to calculate summary statistics of subsets of a data.
- R will return error messages when a code does not function, these messages can help you locate the problem in the code
- R has a very active user base, and many problems have solutions on online forums such as StackOverflow.

4 The basics

4.1 For loops and apply

To show how loops and functions of the "apply" family can produce the same results, we use randomly generated fish data (fish.weight.csv). This dataset contains the body weights of three fish species. For each fish species we want to know the mean and maximum body weight.

```
## First set the working directory
#setwd("")

## Load the data frame fish.weight
fish.weight <- read.csv("./RawData/fish.weight.csv",sep=";")

## Look at the first six rows of the data frame
head(fish.weight)
```

```
##         cod  mackerel   herring
## 1 0.2616791 0.2695651 0.3131698
## 2 0.2536846 0.2929308 0.3349552
## 3 0.2932540 0.3020822 0.3539116
## 4 0.3188049 0.3022859 0.3339471
## 5 0.2628168 0.2267598 0.2744634
## 6 0.2950176 0.3473954 0.3007222
```

```
## Create empty vectors meanFishWeights and maxFishWeights
meanFishWeights <- c()
maxFishWeights <- c()

## Use a for loop to calculate the mean and maximum weight of each fish
## species
## Store the results in the vector meanFishWeights and maxFishWeights
for(i in 1:ncol(fish.weight)){
  meanFishWeights[i] <- mean(fish.weight[,i])
  maxFishWeights[i] <- max(fish.weight[,i])
}

meanFishWeights
```

```
## [1] 0.2928995 0.2849864 0.3330912
```

```
maxFishWeights
```

```
## [1] 0.3687932 0.3473954 0.3952527
```

```
## Use summary() to check results
summary(fish.weight)
```

```
##       cod             mackerel          herring
##  Min.   :0.2537   Min.   :0.2268   Min.   :0.2745
##  1st Qu.:0.2645   1st Qu.:0.2488   1st Qu.:0.3100
##  Median :0.2941   Median :0.2975   Median :0.3345
##  Mean   :0.2929   Mean   :0.2850   Mean   :0.3331
##  3rd Qu.:0.3037   3rd Qu.:0.3041   3rd Qu.:0.3511
##  Max.   :0.3688   Max.   :0.3474   Max.   :0.3953
```

apply() has the following form:

apply(X, MARGIN, FUN)

where X is the data, a matrix or a dataframe. MARGIN indicates whether the function is applied over rows (MARGIN=1), columns (MARGIN=2), or both (MARGIN=c(1,2)). FUN is the function that is applied, this can either be functions already available in R (e.g. mean, sum, max, etc.) our your own created function (discussed in LU7). The output of Apply is a vector.

```
## Use apply() to generate the same results
meanFishWeights <- apply(fish.weight,2,mean)
maxFishWeights <- apply(fish.weight,2,max)

meanFishWeights
```

```
##       cod  mackerel   herring
## 0.2928995 0.2849864 0.3330912
```

```
maxFishWeights
```

```
##       cod  mackerel   herring
## 0.3687932 0.3473954 0.3952527
```

4.2 Lapply and sapply

Lapply works similar to apply() but performs a function to every element of a list (instead of every row or column from a matrix or dataframe) and also returns a list (instead of a vector). Lapply() has the following form:

lapply(X, FUN)

In which X is a list and FUN in a function, similar as in apply(). This function can be "[", a selection operator. The value after the first comma specifies which row is selected, the value after the second comma specifies which column is selected.

sapply() works similar as lapply, but simplifies the output if possible (e.g. returns a vector instead of a list).

To illustrate how lapply and sapply work we will load fishData.csv, which contains both the body weights and lengths of three different fish species, make subsets for each fish species and store these subsets in a list

```
## Read the file fishData.csv
fish.data <- read.csv("./RawData/fishData.csv")

## Look at the first six rows of fish.data
head(fish.data)
```

```
##   species.list species.mass species.length
## 1          cod    0.3598479      0.3171510
## 2          cod    0.3505284      0.3382661
## 3          cod    0.2288044      0.3525479
## 4          cod    0.3126473      0.3707348
## 5          cod    0.3379635      0.3739625
## 6          cod    0.3077175      0.3466604
```

```
## Make a list with subsets of the data for each fish species
fish.data.list <- list()
fish.data.list[[1]] <- subset(fish.data,species.list=="cod")
fish.data.list[[2]] <- subset(fish.data,species.list=="mackerel")
fish.data.list[[3]] <- subset(fish.data,species.list=="herring")
```

We can now use apply and sapply to extract from the dataset of each fish species only the colum with the fish length and calculate the mean fish length

```
## Use lapply to extract from each subset the last column
## Because we select all rows, the entry after the first comma after "["
## is empty
## Save the results as fish.length
fish.length <- lapply(fish.data.list,"[",,3)
fish.length
```

```
## [[1]]
##  [1] 0.3171510 0.3382661 0.3525479 0.3707348 0.3739625 0.3466604 0.2897138
##  [8] 0.3575440 0.3666980 0.3534814
##
## [[2]]
##  [1] 0.2856037 0.3479374 0.3214358 0.3335567 0.2922849 0.3164287 0.3385158
##  [8] 0.2951104 0.3746411 0.4041166
##
## [[3]]
##  [1] 0.3190827 0.3460843 0.3068681 0.3566739 0.3315917 0.3494571 0.3420924
##  [8] 0.3304976 0.3046518 0.3638973
```

```
## Use lapply to calculate the mean fish length for each fish species,
## using the data in fish.length
## The output is a list
lapply(fish.length,mean)
```

```
## [[1]]
## [1] 0.346676
```

```
##
## [[2]]
## [1] 0.3309631
##
## [[3]]
## [1] 0.3350897
```

```
## Use sapply to calculate the mean fish length for each fish species,
## using the data in fish.length
## The output is a vector
sapply(fish.length,mean)
```

```
## [1] 0.3466760 0.3309631 0.3350897
```

4.3 Tapply and aggregate

Tapply() applies a function to a vector (for example a column of a dataframe) for each unique value given by INDEX. It has the following form:

tapply(X, INDEX, FUN)

The output of tapply() is an array.

We can use tapply to calculate for each fish species the average body weight, using the fish.data dataset

```
## Use tapply to calculate the average body weight for each fish species
tapply(fish.data$species.mass,fish.data$species.list,mean)
```

```
##       cod   herring  mackerel
## 0.3250896 0.3023467 0.3049527
```

aggregate() works comparable tapply(), it applies a function to each subset of a dataframe. It has the following form:

aggregate(x, by, FUN)

In which x is an R object, for example a dataframe, by is a list of variables by which the elements in x are grouped (it should have the same length as x) and FUN indicates the function that should be applied. The output of aggregate() is a dataframe.

We can also use aggregate to calculate the average body weight for each fish species. In addition, with aggregate we can calculate the average body weight and length at the same time, using again the data fish.data.

```
## Use aggregate to calculate the average species mass for each fish species
## We have to convert the column fish.data$species.list to a list, as
## aggregate requires a list for the by argument.
aggregate(fish.data$species.mass,list(fish.data$species.list),mean)
```

```
##    Group.1         x
## 1      cod 0.3250896
## 2  herring 0.3023467
## 3 mackerel 0.3049527
```

```
## Use aggregate to calculate the average species mass and average species
## length for each fish species at the same time
## Now we do not specify a column, but use the whole dataframe for x.
aggregate(fish.data,list(fish.data$species.list),mean)
```

```
##    Group.1 species.list species.mass species.length
## 1      cod           NA    0.3250896      0.3466760
## 2  herring           NA    0.3023467      0.3350897
## 3 mackerel           NA    0.3049527      0.3309631
```

4.4 Debugging

Sometimes when you run a piece of code, either written by yourself or written by others, it does not run as expected. Often, R will trow some error message at you. Here's a silly example where we try to make a character object which holds the word 'dog':

```
a <- dog
```

```
## Error in eval(expr, envir, enclos): object 'dog' not found
```

In this case, the error message gives you a pretty good idea of what's going wrong, when we forget to add the quotation marks when we want to make a character object, R thinks we want to assign the value stored in the variable dog.

Sometimes, the error message thrown by R is not so obvious. Let's consider another silly example where we have an empty matrix of 3 rows and 2 columns and try to fill this with number from 1 to 6 using a for loop. Of course, this can be done much more efficiently, but for the sake of this example, we are going to do it using a for loop.

```
someMatrix = matrix(,nrow=3,ncol=2)
for (i in 1:6){
  someMatrix[i,i] <- i
}
```

```
## Error in `[<-`(`*tmp*`, i, i, value = i): subscript out of bounds
```

As you can see, the error message we get here is less obvious than the previous one. In cases like this, and while debugging scripts in general, it can be a good idea to use the print() function to keep track of what is going on.

```
for (i in 1:6){
  print(i)
  someMatrix[i,i] <- i
}
```

```
## [1] 1
## [1] 2
## [1] 3
```

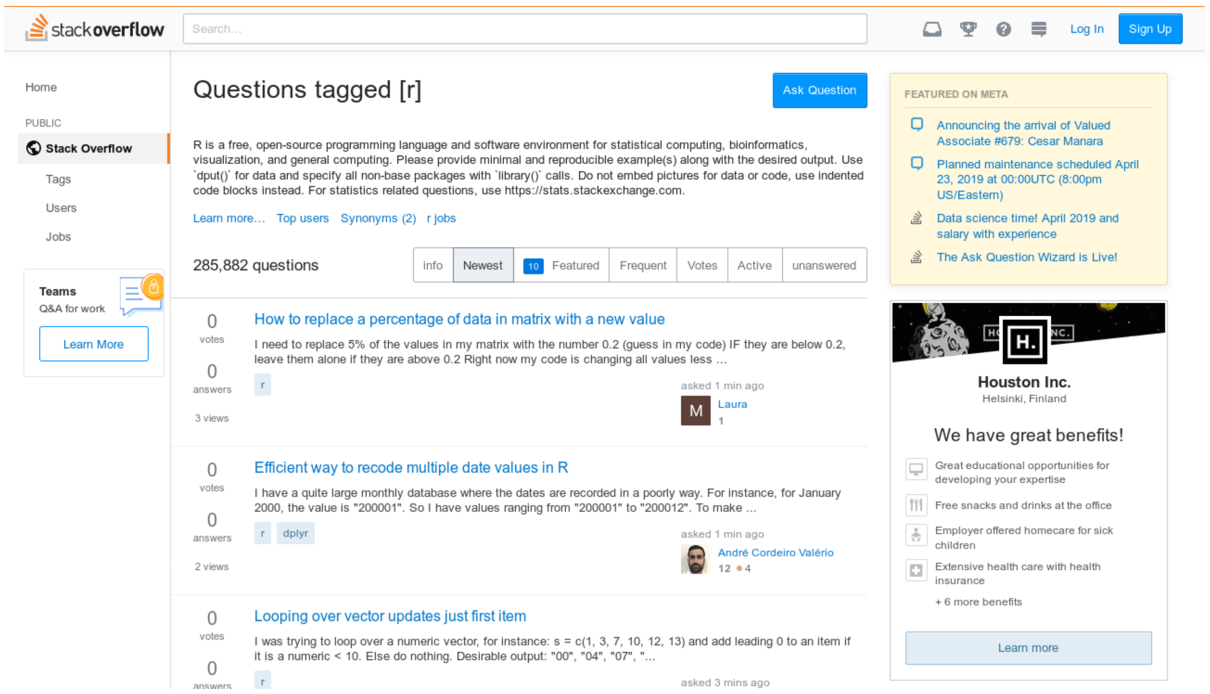
```
## Error in `[<-`(`*tmp*`, i, i, value = i): subscript out of bounds
```

This shows that error occurs when i reaches 3. So the error occurs when we try to assing a value to someMatrix in row 3 and column 3, however, the matrix only has 2 columns, obviously we cannot assign a value to a column that does not exist.

An additional tool that R offers to help debug code is the traceback() function. However, this is mainly usefull when the error is hidden in functional calls.

4.4.1 Where to look for help

One of the benefits of working with R is that it has a pretty active user base. These users come together on various internet forums where they ask each other for help to solve a certain problem and share their knowledge on all kinds of R related things. The most well known example of such a forum is 'StackOverflow'.



Questions tagged [r]

R is a free, open-source programming language and software environment for statistical computing, bioinformatics, visualization, and general computing. Please provide minimal and reproducible example(s) along with the desired output. Use `dput()` for data and specify all non-base packages with `library()` calls. Do not embed pictures for data or code, use indented code blocks instead. For statistics related questions, use https://stats.stackexchange.com.

Learn more… Top users Synonyms (2) r jobs

285,882 questions

info | Newest | 10 Featured | Frequent | Votes | Active | unanswered

0 votes 0 answers 3 views
How to replace a percentage of data in matrix with a new value
I need to replace 5% of the values in my matrix with the number 0.2 (guess in my code) IF they are below 0.2, leave them alone if they are above 0.2 Right now my code is changing all values less …
r
asked 1 min ago Laura 1

0 votes 0 answers 2 views
Efficient way to recode multiple date values in R
I have a quite large monthly database where the dates are recorded in a poorly way. For instance, for January 2000, the value is "200001". So I have values ranging from "200001" to "200012". To make …
r dplyr
asked 1 min ago André Cordeiro Valério 12 ● 4

0 votes 0 answers
Looping over vector updates just first item
I was trying to loop over a numeric vector, for instance: s = c(1, 3, 7, 10, 12, 13) and add leading 0 to an item if it is a numeric < 10. Else do nothing. Desirable output: "00", "04", "07", "…
r
asked 3 mins ago

FEATURED ON META
Announcing the arrival of Valued Associate #679: Cesar Manara
Planned maintenance scheduled April 23, 2019 at 00:00UTC (8:00pm US/Eastern)
Data science time! April 2019 and salary with experience
The Ask Question Wizard is Live!

Houston Inc.
Helsinki, Finland
We have great benefits!
Great educational opportunities for developing your expertise
Free snacks and drinks at the office
Employer offered homecare for sick children
Extensive health care with health insurance
+ 6 more benefits

Keep in mind that StackOverflow is a forum for all kinds of programming languages, so alway speficy that you are looking for solutions in R. If you cannot directly find what you are looking for on StackOverflow, you can always try to do a simple search in DuckDuckGo (or any other search engine).

4.4.2 Most common errors

According to Naom Ross, who analyzed posts on StackOverflow, these are the most common R errors.

1. Subscript out of bound: We saw an example of this in the previous section. These errors generally mean that you are trying to assign a value to a place outside of the outer bounds of the matrix or vector.
2. Could not find a function:

```
g1 <- sumary(fish.data)
```

```
## Error in sumary(fish.data): could not find function "sumary"
```

This often happens when you have a typo in your function name, or when you want to use a function from a package that is not yet loaded into R.

3. Error in if:

```
a <- c(1,2,NA,4)
for (i in 1:length(a)){
  if (a[i] > 2){
    print(a[i])
  }
}
```

```
## Error in if (a[i] > 2) {: missing value where TRUE/FALSE needed
```

This happens when you try to pass a non-logical or a missing value to the logical operator inside an if statement.

4. Cannot open:

```
a <- read.csv('a.txt')
```

```
## Warning in file(file, "rt"): cannot open file 'a.txt': No such file or
## directory
```

```
## Error in file(file, "rt"): cannot open the connection
```

This error often pops up when you try to load in some data from a file. It means that the file you try to open does not exists (maybe you are in the wrong directory, or maybe there is a typo in the file name), or that it cannot be accessed (maybe you do not have read rights).

5. No applicable method:

```
droplevels(1:10)
```

```
## Error in UseMethod("droplevels"): no applicable method for 'droplevels' applied to an
```

This can occur when you are trying to apply a function to an inappropriate data type or object class.

4.4.3 An example

Lets go line-by-line through this piece of code somebody made to print the average length and standard deviation thereof for each species ordered from light to heavy.

```r
# Fist clear our global enviroment so we all start with a clean slate
rm(list=ls())

# Set the directory and load the fish data from earlier.
fishData <- read.csv("./RawData/fishData.csv")

#Calculate the mean and standard deviation of the length of the species
size.per.species <- aggregate(fishData,by=list(fishData$speciesList),mean)
std.per.species <- aggregate(fishData,by=list(fishData$speciesList),std)

# Order species based on their length
sizes.sorted.by.length <- size.per.species[
  order(size.per.species$species.lenght),]
std.sorted.by.length <- std.per.species[
  order(size.per.species$species.length),]

print('The species sorted by mass:')
for(i in 1:nrow(sizes.sorted.by.length){
  print(paste(i,
              ': ',
              sizes.sorted.by.length$Group.1[i,],
              ', average length: ',
              sizes.sorted.by.length$species.length[i],
              ', sd: ',
              std.sorted.by.length$species.length[i],
              ,sep=''))
}
```

And here is the final script after sorting through the errors

```r
# Fist clear our global enviroment so we all start with a clean slate
rm(list=ls())

# Set the directory and load the fish data from earlier.
fishData <- read.csv("./RawData/fishData.csv")

#Calculate the mean and standard deviation of the length of the species
size.per.species <- aggregate(fishData,by=list(fishData$species.list),mean)
std.per.species <- aggregate(fishData,by=list(fishData$species.list),sd)

# Order species based on their length
sizes.sorted.by.length <- size.per.species[
  order(size.per.species$species.length),]
std.sorted.by.length <- std.per.species[
  order(size.per.species$species.length),]

print('The species sorted by mass:')
```

```
## [1] "The species sorted by mass:"
```

```r
for(i in 1:nrow(sizes.sorted.by.length)){
  print(paste(i,
              ': ',
              sizes.sorted.by.length$Group.1[i],
              ', average length: ',
              sizes.sorted.by.length$species.length[i],
              ', sd: ',
              std.sorted.by.length$species.length[i],
              sep=''))
}
```

```
## [1] "1: mackerel, average length: 0.330963108923873, sd: 0.0376415851644494"
## [1] "2: herring, average length: 0.335089683570251, sd: 0.0202335581458669"
## [1] "3: cod, average length: 0.346675980567795, sd: 0.0260561623063289"
```

5 Exercises

In the interactive exercises we simulated the masses of three fish species. These data can be found as a csv file "fish.weight.csv". Assume that the fish weights are obtained from fishes caught by fishermen. Every day they caught one fish of each species. The weights of the fishes caught on day 1 are in row 1, the weights of the fishes caught on day 2 are in row 2, etc.

1. What is the mean mass of fish caught each day? Calculate with both a for loop and one of the apply functions
2. What is the mass of the largest fish caught each day? Calculate with both a for loop and one of the apply functions
3. What is the total mass of fish caught each day? Calculate with both a for loop and one of the apply functions

The next exercizes are done using "fishdata.csv".

4. Make a subset of the data for each fish species and store these subsets as different elements of a list. Extract for each subset the column with the masses of the species and calculate the average mass for each species. Use both lapply and sapply.

5. Make a histogram of the distribution of species masses for each fish species

The next exercizes are done using “Nijmegen_trees_LU5.csv”.

6. Below, you’ll find a script that was originally made to calculate the mean and standard deviation of the age of the trees in Lent separated to genus. In it’s current form, the script does not work, it’s up to you to fix it.

```r
rm(list(ls()))

trees <- read.csv("./RawData/Nijmegen_tees_LU5.csv"))

# Let's first remove the trees for which the planting year is 0
trees <- trees[trees$planting.year != 0]

# Now select only trees in Lent
Lent <- trees[trees[,neighbourhood] = 'Lent',]

# Now calculate the age
Lent$age <- 2019-Lent$planting.year

# Lets add a new column to the data frame which contains the genus of the species
speciessplit <- strsplit(Lent$species,split=' ')
Lent$genus <- unlist(lapply(speciessplit,function(x) head(x)))

# Now let's calculate the mean age of the trees in Lent and standard deviation thereof
aggregate(Lent$age,by=Lent$genus,mean)
aggregate(Lent$age,by=Lent$genus,std)
```

7. Calculate the number of trees in each neighbourhood. Use both tapply and aggregate
8. Calculate the number of trees of each species. Use both tapply and aggregate

The next exercizes are done using “owls.csv”. from LU3

9. Calculate the number of deprived and satiated owls in each nest
10. Calculate the mean brood size and arrival time for each nest

The last exercizes are done using “Primates_Habitat.csv”. from LU4.

11. Correct the following R script. This excercise is extra difficult, because there are errors that do not lead to an error message.

Some hints

- There are several typo's in the script
- Is the "_" in the species column substituted for a blank space
- There are 3 errors in the apply function
- The sum_habitat for all generalist species is zero, is this correct?
- Does the tapply function gives realistic values for the average body masses?

```r
rm(list=(ls()))

## Read the Primates_Habitat data
Primates_Habitat <- read.csv("Primates_Habitat.csv")

## Summary of the dataset
Summary(Primates_Habitat)

## Remove rows with NA's in the species column
Primates_Habitat <- Primates_Habitat[complete.cases(Primates_Habitat$species),]

## In the column "Species", there is a _ in each species name, substitute this with a bl
gsub("_"," ",Primates_Habitat$Species)

## Make a column with the sum of the values in the columns of the different habitat type
## When this sum is 1, the species is a habitat specialist
## When the sum is higher than 1, the species is a habitat generalist
Primates_habitat$sum_habitat <- apply(Primates_Habitat[,c("Habitat_Forest","Habitat_Sava

## Add a column which indicates for each species whether the species is a habitat genera
## or on which habitat the species specializes
for(monkey in 1:Primates_Habitat){

  ## If the sum in the column sum_habitat is larger than 1, the species is a generalist
  if(Primates_Habitat$sum_habitat[monkey]<1){
    Primates_Habitat$HabStrategy[monkey] <- "Generalist"
  } else if(Primates_Habitat$Habitat_Forest[monkey]==1) {
    Primates_Habitat$HabStrategy[monkey] <- "Forest specialist"
  } else if(Primates_Habitat$Habitat_savanna[monkey]==1) {
    Primates_Habitat$HabStrategy[monkey] <- "Savanna specialist"
  } else if(Primates_Habitat$Habitat_Shrubland[monkey]==1) {
    Primates_Habitat$HabStrategy[monkey] <- "Shrubland specialist"
  } else if(Primates_Habitat$Habitat_Grassland[monkey]==1) {
    Primates_Habitat$HabStrategy[monkey] <- "Grassland specialist"
  } else if(Primates_Habitat$Habitat_Wetlands[monkey]==1) {
    Primates_Habitat$HabStrategy[monky] <- "Wetlands specialist"
  } else if(Primates_Habitat$Habitat_Rocky.areas[monkey]==1) {
    Primates_Habitat$HabStrategy[monkey] <- "Rocky areas specialist"
  } else if(Primates_Habitat$Habitat_Desert[monkey]==1) {
    Primates_Habitat$HabStrategy[monkey] <- "Desert specialist"
  }
  print(monkey)
}

## Check whether all generalist species have a sum_habitat larger than one
Primates_Habitat$sum_habitat[Primates_Habitat$HabStrategy=="Generalist"&is.na(Primates_H

## Also read the data with body mass data of the primates
Primates_BodyMass <- read.csv("Primates_BodyMass_LU5.csv")

## Remove rows with NA's in the species column
Primates_Habitat <- Primates_Habitat[complete.cases(Primates_Habitat$Species),]

## In this dataset, also substitute "_" for a blank space in the Species column
Primates_BodyMass$Species<-gsub("_"," ",Primates_BodyMass$Species)

## Merge the habitat and body mass dataset
Primates<-merge(Primates_Habitat,Primates_BodyMass,by="Species")

## The class of the column BodyMass_kg is a factor, convert to numeric
Primates$BodyMass_kg<-as.numeric(Primates$BodyMass_kg)

## Calculate the average body mass for species in the different HabStrategy classes
tapply(Primates$BodyMass_kg,Primates$HabStrategy,mean)
```

12. In the R script of exercise 11, species that do not occur in any of the habitats are also called habitat generalists. Can you add extra lines to the for loop that assigns NA's in the column HabStrategy for these species.

13. Imagine you want to have the species names in all capital letters, search on the internet to find a function that can do this for you.

Learning Unit 6 - Making attractive and informative graphs

Coline Boonman

Contents

1 Learning goals

After this learning unit is completed, you should be able to:

- produce the following plot types: histogram, scatterplot, line plot, dot chart, bar plot, and boxplot.
- change colors and size of text, points, lines, bars.
- to add a legend, a main title, axis labels, and text to a plot.
- plot multiple plots in 1 figure.
- add lines to a plot.

2 Graphs

2.1 What type of graphs are there? And when to use which?

Depending on the question you want to answer, different type of graphs are used.

Histograms are used to explore the data you have. It visualizes the distribution of a numeric variable in your dataset. On the X-axis, the range of values in your variable is depicted, and on the Y-axis the frequency is shown. This frequency is how often that specific x-value is present in that variable in your dataset (unless specified differenty).

Scatter plots are used to visualise variation in your data. You can plot two numeric variables from your dataset against each other, where one is depicted on the X-axis and the other on the Y-axis. Again, the range of values on both axes is the range of values present in that variable in your dataset (unless specified differenty).

Line plots can help visualise trends in your data. You can plot two numeric variables from your dataset against each other, where one is depicted on the X-axis and the other on the Y-axis. Different from the scatterplot, a line plot is used only when one value of Y exists for one value of X. An example for this is plotting the amount of babies born in The Netherlands (on the y-axis) per year (on the x-axis). Also in line plots, the range of values on both axes is the range of values present in that variable in your dataset (unless specified differenty).

Boxplots help you to explore the variation within categories. You can plot a categorical variable against a numeric variable from your dataset, where the different categories in the categorical variable are depicted on the X-axis and the numeric variable is shown on the Y-axis. A box is comprised of the median (thick horizontal bar), first and third quartile (lower and upper horizontal bar of the box, respectively), and the whiskers (minimum and maximum value). Also in boxplots, the range of values on the Y-axis is the range of values present in that variable in your dataset, and the categories plotted on the X-axis are all the categories that exist in that variable in your dataset (unless specified differenty).

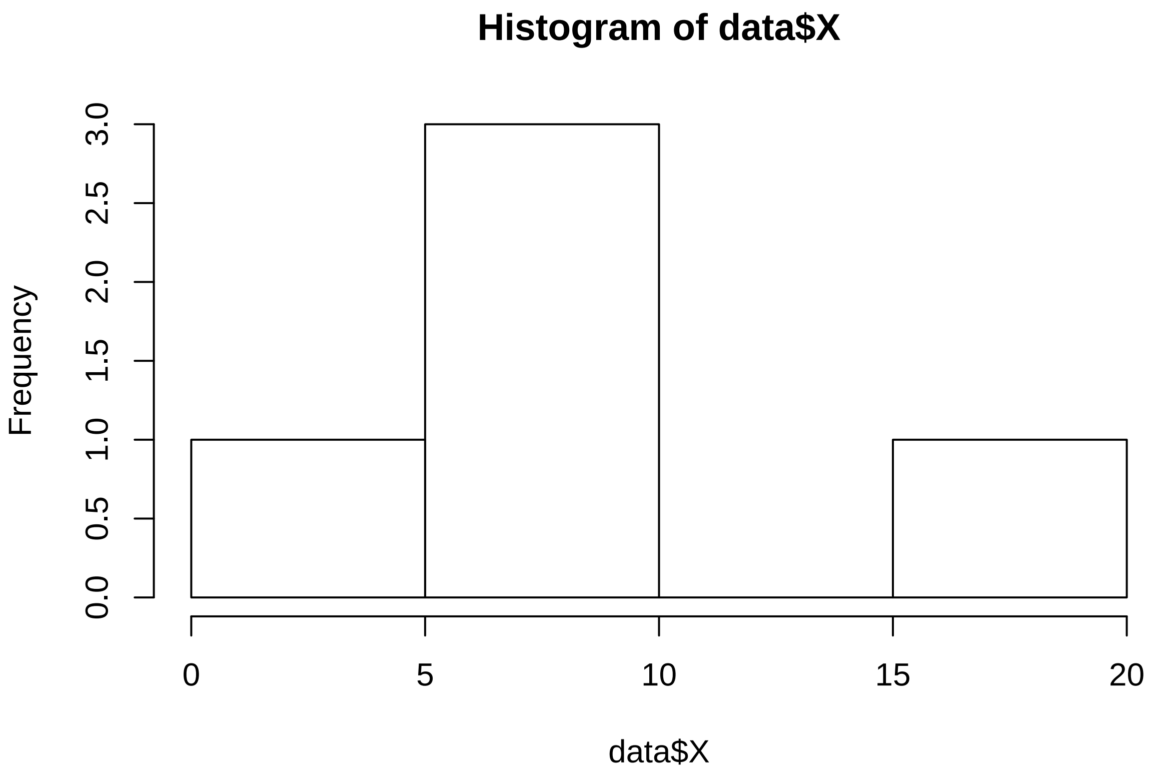
Dot charts helps you to explore differences between categories. You can plot a categorical variable against a numeric variable from your dataset, where the different categories in the categorical variable are depicted on the Y-axis and the numeric variable is shown on the

X-axis. Please note that only one value of X should exist per category, which makes the dot chart different from the box plot. In dot charts, the range of values on the X-axis is the range of values present in that variable in your dataset, and the categories plotted on the Y-axis are all the categories that exist in that variable in your dataset (unless specified differenty).

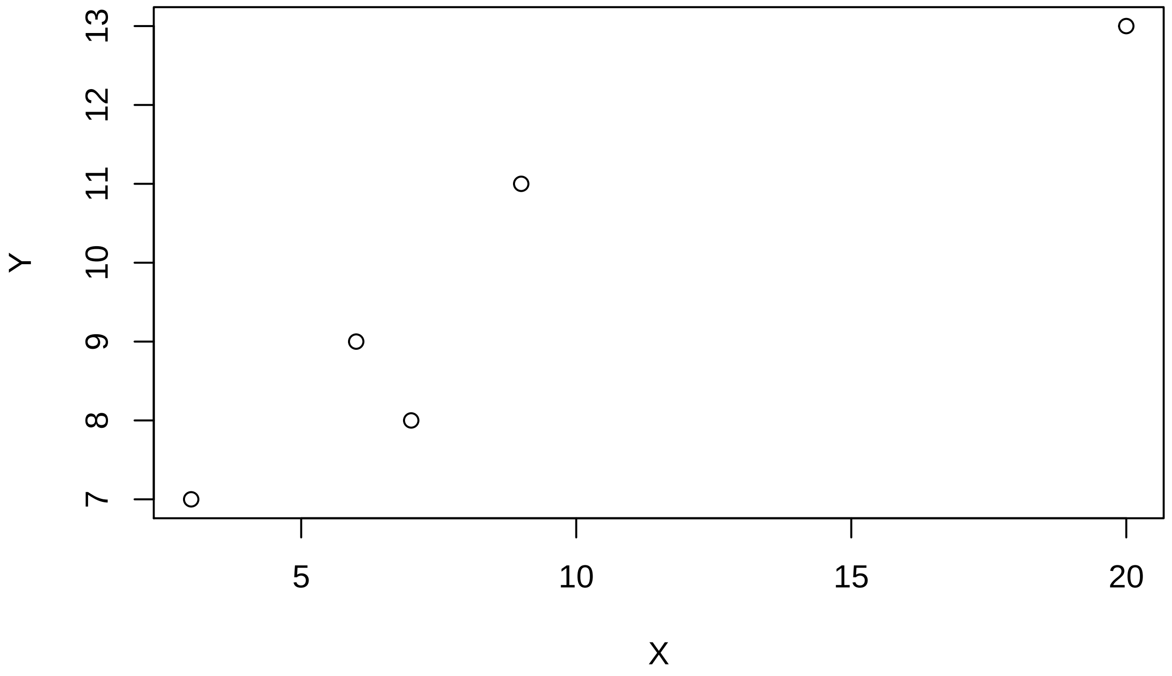
Bar plots are used to show variation between categories. You can plot a categorical variable against a numeric variable from your dataset, where the different categories in the categorical variable are depicted on the X-axis and the numeric variable is shown on the Y-axis. As a standard, mean values of the numeric variable are depicted, as opposed to boxplots, and you can add error bars that show the variation within a category (standard deviation or standard error). The range of values on the Y-axis is the range of values present in that variable in your dataset, and the categories plotted on the X-axis are all the categories that exist in that variable in your dataset (unless specified differenty).

2.2 Generating informative graphs

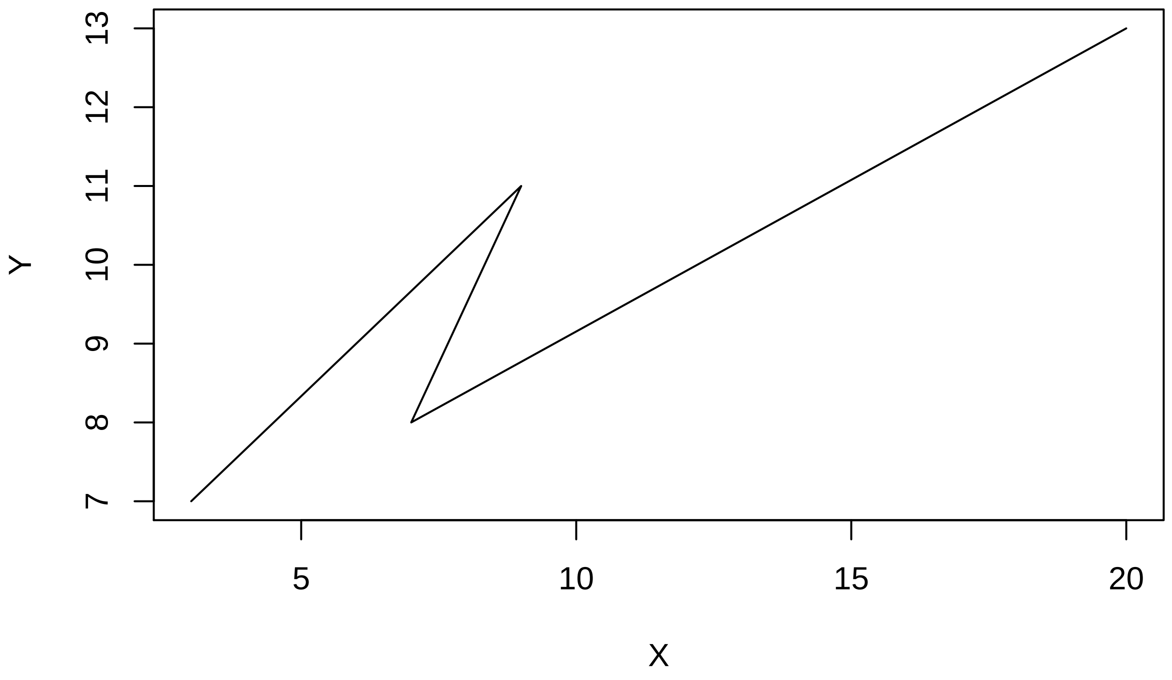
```r
data = data.frame(X=c(3,6,9,7,20),
                  Y=c(7,9,11,8,13),
                  Z=c("cat","cat","dog","dog","dog"))


# Histogram
# numeric variable.
# Between brackets include: a column of a dataset or include a vector
hist(data$X)
```



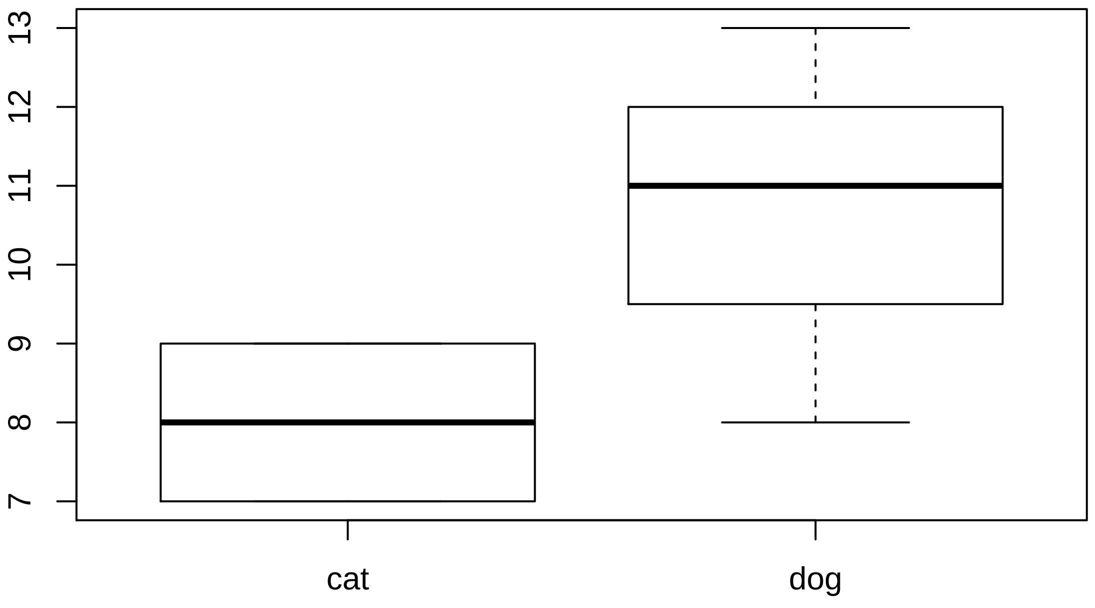
```r
# Scatter plot
# X and Y need to be numeric. p informs R to make points.
plot(Y ~ X, data, type='p') #equal to plot(data$X, data$Y, type='p')
```



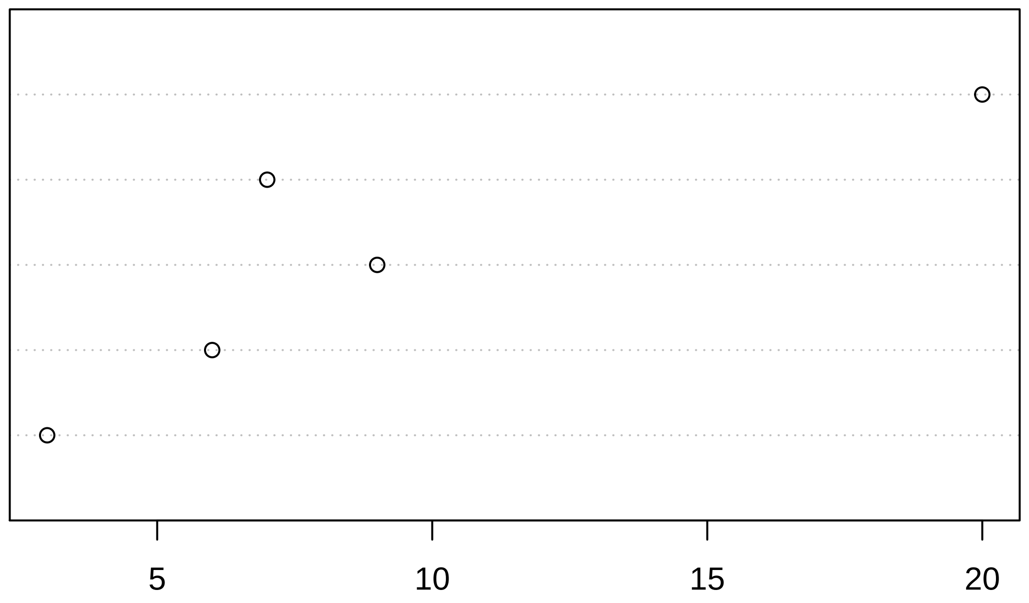
```r
# Line plot
# X and Y need to be numeric. l informs R to make a line
plot(Y ~ X, data, type='l') #equal to plot(data$X, data$Y, type='l')
```



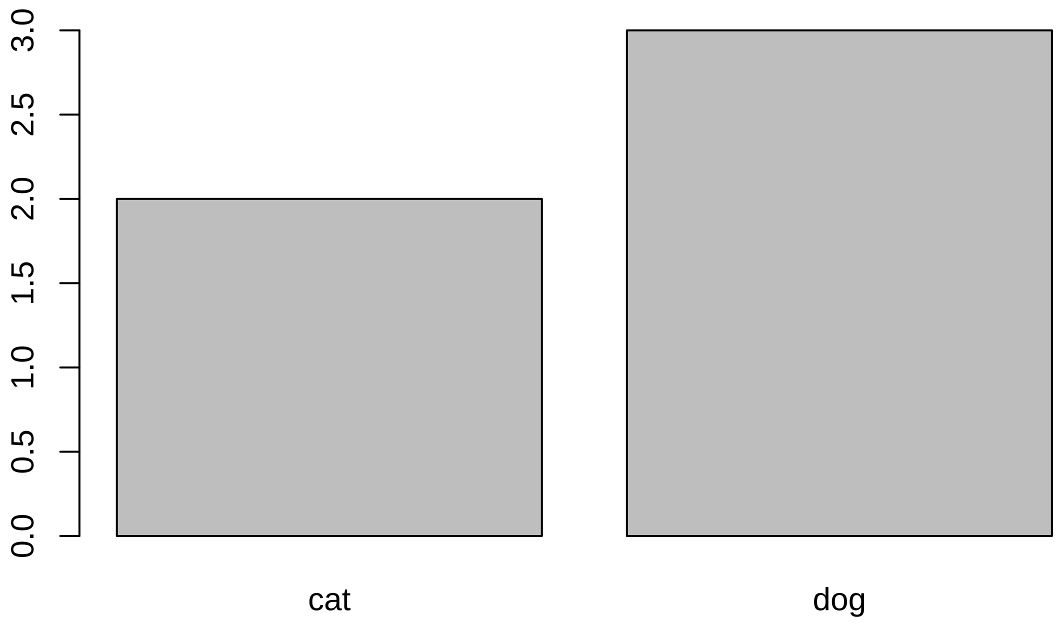
```r
# Box plot
# Z needs to be categorical, and Y needs to be numeric.
boxplot(Y ~ Z, data)
```



```r
# Dot chart
# X needs to be numeric
dotchart(data$X)
```



```r
# Bar plot
# Between brackets include: name of a table with categories and their frequency
barplot(table(data$Z))
```



2.3 Making graphs more attractive

The graphs made in the previous section are very informative and they can answer quick questions about your dataset. However, when you want to use the graph in an article, thesis, report, etc. there might be some things you want to change. In this section, we will provide you with the basic tools to improve the estatics of the graphs to increase interpretability and readability.

2.3.1 *Title*

Adding a title to your graph.

```
main='Title'
```

2.3.2 *Axis label*

Adding a clear label to the axes of your graph.

```
# For numeric variables
xlab='name X-axis'
ylab='name Y-axis'

# For categoric variables
# step 1. Check what categories there are, and in what order they are listed.
unique(df$variable)
# step 2. Rename the catagories.
names.arg=c('X','Y') # for bar plots
labels=c('X','Y')    # for dot charts
```

2.3.3 *Axis range*

Changing the limit of the axis that is plotted for numeric variables.

```
xlim=c(min,max)
ylim=c(min,max)
```

It is also usefull to have be able to remove the 'additional margin' at the lower and upper end of an axis.

```
# Add this in your plot() lines of code
xaxs="i"
yaxs="i"
```

2.3.4 *Size*

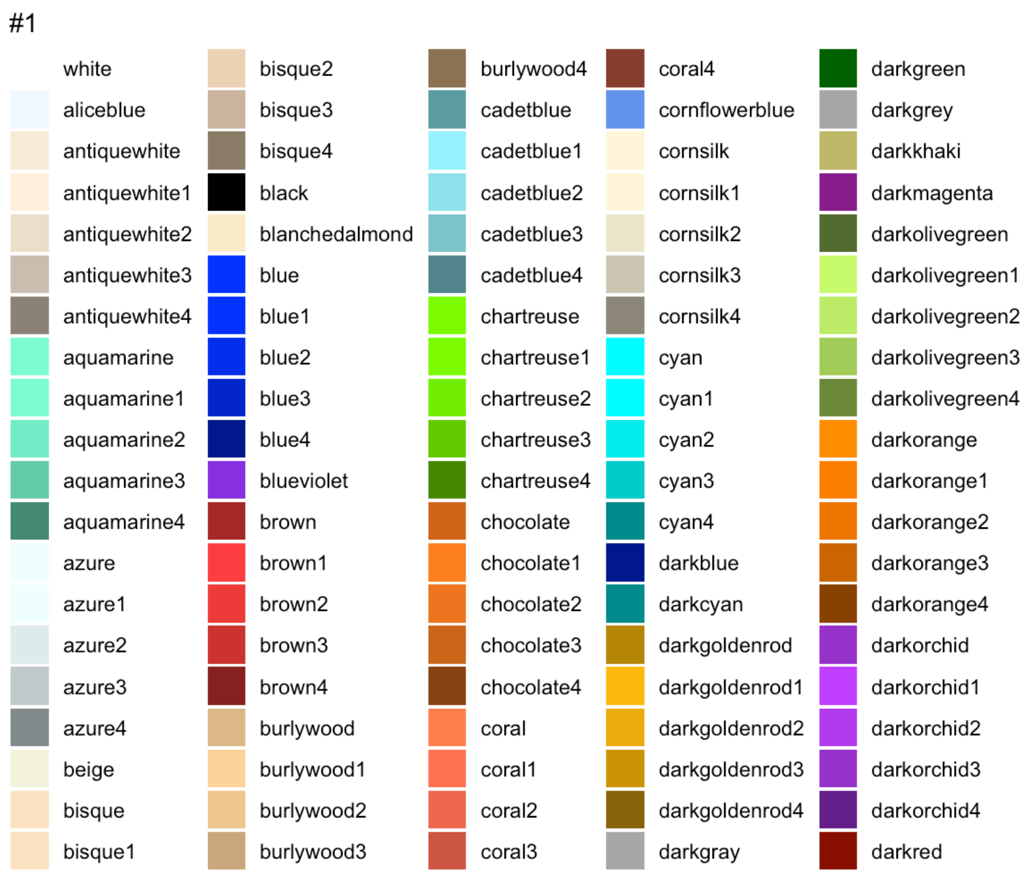
Changing the size of text or symbols is done with the word ‘cex’. This is a numerical indicator of size change relative to the default.

```
cex.axis  # continuous axis
cex.names # categorical axis
cex.lab   # labels (axis title)

cex       # symbol size
lwd       # line width

#example
cex=1.5   # increased with half the original size
```

2.3.5 *Color*

Adding colors to your graph: text, symbols, lines, bars.

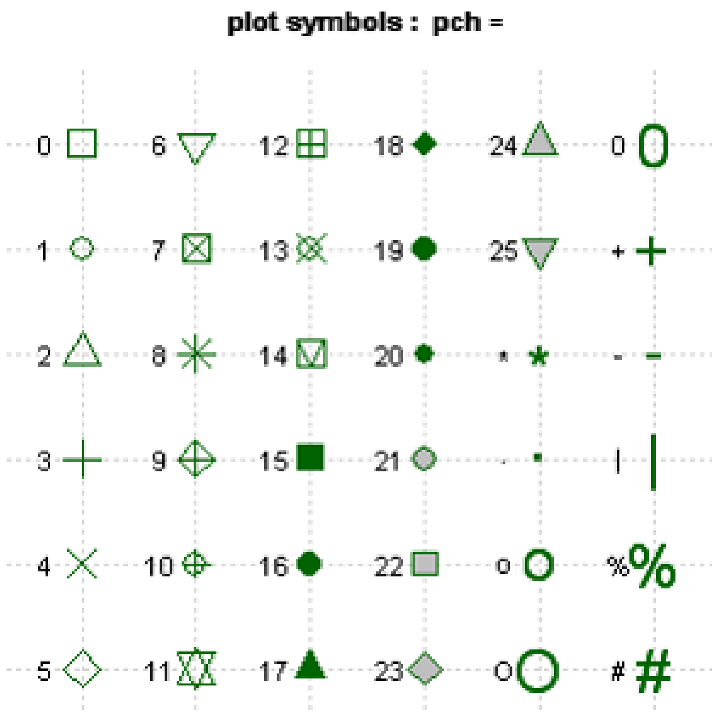
```
col.axis  # color axis
col.lab   # color label
col.main  # color title

col       # symbols or lines

#example
col='red' # color is red
```

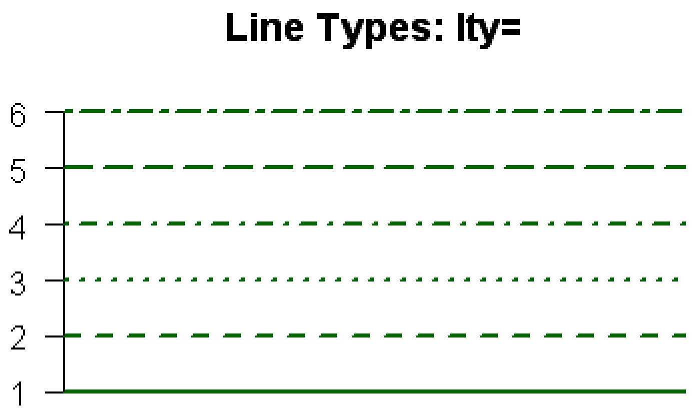


2.3.6 *Symbols and lines*

Changing the type of symbol and/or line using a numerical indicator.

```
pch        # symbol
lty        # line type

#example
pch=2      # triangle
lty=3      # dotted line
```



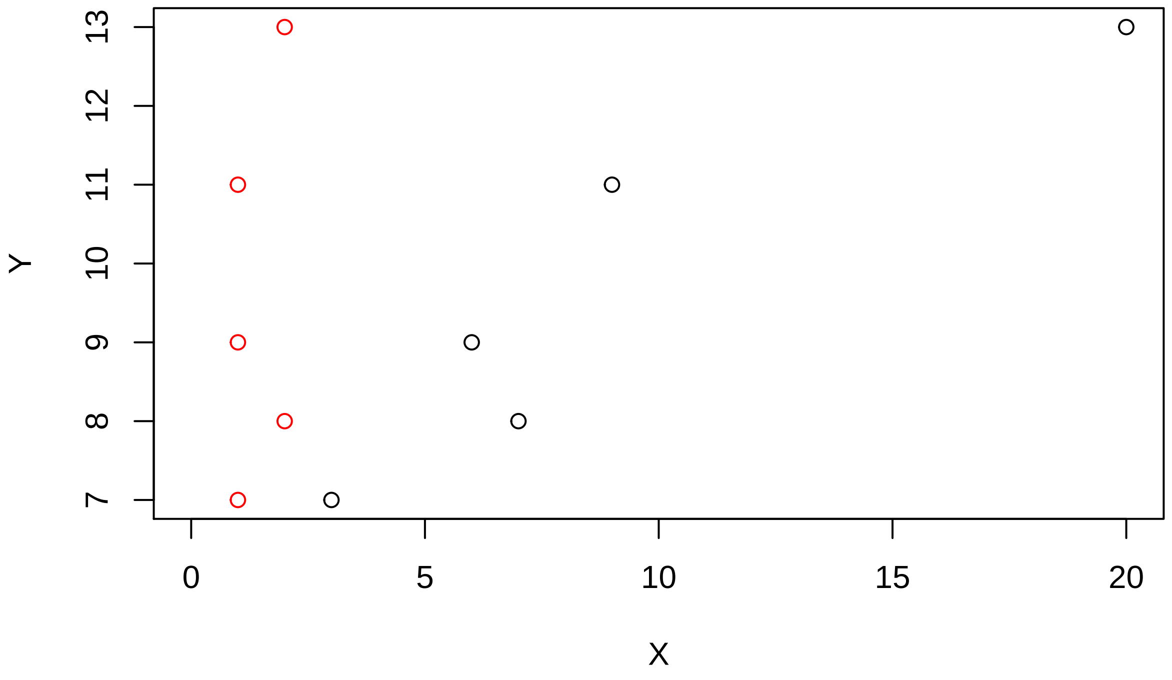


2.3.7 *Adding additional points or lines*

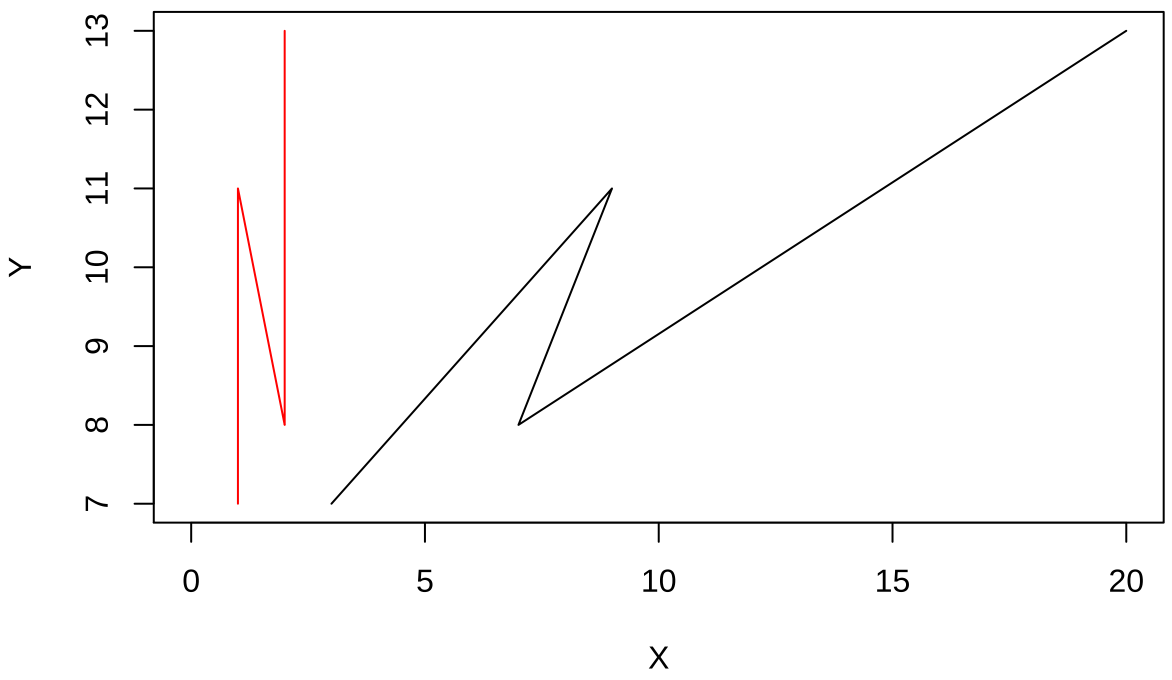
You might want to create a figure and add additional points or lines from other datasets, or from the same dataset but with different subsets.

```
lines()        # add lines to the same graph
points()       # add points to the same graph

# NOTE Make sure to adjust the x and y limits in the first plot
# in order to make all additional points and lines visible
```

```
#example
data = data.frame(X=c(3,6,9,7,20),
                  Y=c(7,9,11,8,13),
                  Z=c(1,1,1,2,2))

plot(Y ~ X, data, type='p', xlim=c(0,20)) # Scatterplot
points(Y ~ Z, data, col="red") # Add points
```



```
plot(Y ~ X, data, type='l',xlim=c(0,20)) # Line plot
lines(Y ~ Z, data, col="red") # Add a line
```



2.3.8 *Legend*

Adding a legend to your graph is usefull when you use colors that are not indicated in text of axes or labels.

```r
legend(x = ,          # number of x-coordinate where legend needs to be placed
       y = ,          # number of y-coordinate where legend needs to be placed
       legend = c(), # vector with unique categories that go into the legend
       title="text", # add a legend title
       col = c(),     # vector with the colors, as defined in the plot
       cex = ,        # increase the size of the legend
       pch = )        # indicate which point type you want.
                      # change pch= into lty= (and lwd=) if you want a line.

# or

legend("position",    # "top", "topright", "bottomleft", "left", etc
       legend = c()) # vector with unique categories that go into the legend)

#example
legend(x=1, y=1, legend=c(unique(df$cat)), fill=df$cat, cex=1.5)

# The categoric variable in the legend should be a character!
# Otherwise, the legend is not plotted correctly.
# You can check this with str(df)
# You can change this with df$cat<-as.character(df$cat)
```

2.3.9 *Adding lines*

Adding a line to your graph.

```
abline(a, b)    # a is the intercept and b is the slope

#example
abline(h = 1)   # horizontal line crossing the Y-axis at Y=1
abline(v = 1)   # vertical line crossing the X-axis at X=1
abline(0, 1)    # tilted line crossing Y-axis at Y=0 with a slope of 1
```

2.3.10 *Additional text*

Adding additional text to your graph can help interpret the results.

```
text(x = ,         # number of x-coordinate where text needs to be placed
     y = ,         # number of y-coordinate where text needs to be placed
     labels = c()) # text to be written

#example
text(x=1, y=1, 'hello')
```

2.3.11 *Multi-plot figure*

Having mupliple graphs that belong to each other into one figure is usefull; it save time (no need to save each graph seperately', and it makes estetically pleasing figures (graphs are all placed with the same distance from each other).

```
par()     # general call for plots

mfrow=c(rows,columns)
# the amount of rows and columns you want in your figure
# each graph is placed in one cell.

mar=c(b,l,t,r)
# marging around the graph. b=below, l=left, t=top, r=right of the plot

oma=c(b,l,t,r)
# outer margin area (space between graphs in a multi-plot figure).
# Abbreviations as in mar().

#example
par(mfrow=c(1,2),
    oma=c(0,0,0,2))
```

If you want to create a title for the complete multi-plot figure, use **mtext()**, similar to **text()** in *Additional text*.

2.3.12 *Saving graphs*

Once you have created a nice graph, you want to save it in order to put it into your report. There are two ways of doing this:

1. By clicking *Plots* in the lower right quadrant of your RStudio window. Then click *Export* and select the format you want to save your graph in.
2. By coding: (faster - less actions to be taken)

```
jpeg()
png()
pdf()
# close off with: dev.off()
# This indicates the end of the code that needs to be saved as a figure.

#example
jpeg("hello.jpeg") # name of the image
plot()             # add the code of the graph you want to save
abline()           # this can also include lines and text
dev.off()          # close of with dev.off()

# NOTE if the saved figure is not in the correct width - height ratio,
# you can adjust this by adding width and height (in pixels for images,
# and in inch for pdf)
jpeg("hello.jpeg", width=100, height=80)
```

3 Live example: Trees in Nijmegen

3.1 Basic graphs

First, we will run through some standard code for each graph type.

We will use the dataset containing data of trees in Nijmegen. This dataset includes the following variables:

- ID - Number for each individual
- species - Species name
- planting.year - Year the individual is planted
- zip.code - Zip code of the location of the individual
- neighbourhood - Neighbourhood of the location of the individual
- height - Height of the individual

```r
# First set the working directory
setwd(" ")
```

```r
# Read in the data
my.df<-read.table("Nijmegen_trees_LU6.csv", header=T, sep=" ")
```

Now, we want to explore the data in our dataset.

```r
# Check what data is in this dataset
str(my.df)
```

```
## 'data.frame':    44276 obs. of  6 variables:
##  $ ID           : int  256 257 258 259 260 261 262 263 264 265 ...
##  $ species      : Factor w/ 470 levels "","Abies","Abies nordmanniana",..: 59 59 59 59 59 5
##  $ planting.year: int  1930 1930 1930 1950 1960 1950 1950 1930 1930 1930 ...
##  $ zip.code     : int  6524 6524 6524 6524 6524 6524 6524 6524 6524 6524 ...
##  $ neighbourhood: Factor w/ 2404 levels "Altrade","Biezen",..: 6 6 6 6 6 6 6 6 6 6 ...
##  $ height       : num  8.89 4.23 8.16 8.95 10.93 ...
```

```r
head(my.df)
```

```
##    ID                          species planting.year zip.code
## 1 256 Aesculus hippocastanum Baumannii          1930     6524
## 2 257 Aesculus hippocastanum Baumannii          1930     6524
## 3 258 Aesculus hippocastanum Baumannii          1930     6524
## 4 259 Aesculus hippocastanum Baumannii          1950     6524
## 5 260 Aesculus hippocastanum Baumannii          1960     6524
## 6 261 Aesculus hippocastanum Baumannii          1950     6524
##   neighbourhood    height
## 1    Galgenveld  8.891099
## 2    Galgenveld  4.229524
## 3    Galgenveld  8.155946
## 4    Galgenveld  8.952107
## 5    Galgenveld 10.930223
## 6    Galgenveld  6.220409
```

```
summary(my.df)
```

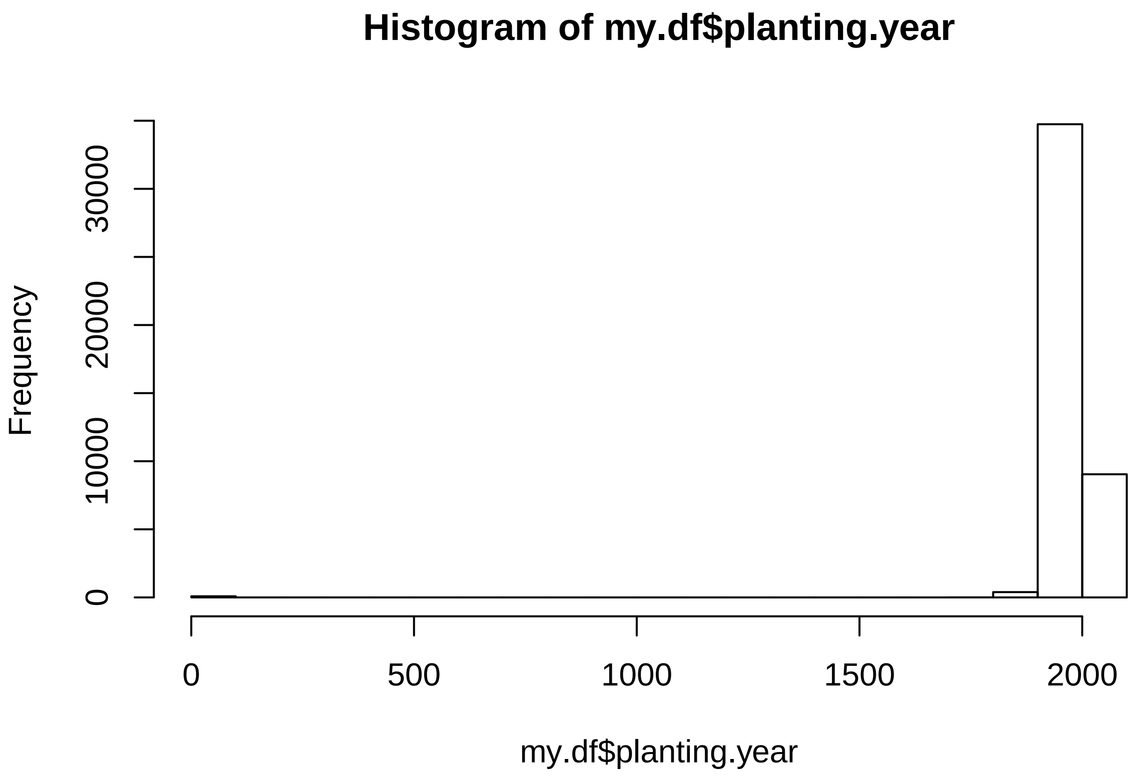
```
##        ID                          species      planting.year     zip.code
##  Min.   :    1   Quercus robur      : 4592   Min.   :   0   Min.   :6511
##  1st Qu.:13112   Tilia x europaea   : 3245   1st Qu.:1970   1st Qu.:6525
##  Median :26483   Fraxinus excelsior: 2447   Median :1980   Median :6534
##  Mean   :30679   Fagus sylvatica    : 1969   Mean   :1976   Mean   :6538
##  3rd Qu.:47386   Quercus rubra      : 1935   3rd Qu.:1995   3rd Qu.:6541
##  Max.   :73303   (Other)            :30078   Max.   :2019   Max.   :6663
##  NA's   :1       NA's               :   10   NA's   :1
##       neighbourhood       height
##  Meijhorst   : 4725   Min.   : 1.735
##  Goffert     : 3469   1st Qu.: 6.997
##  Zwanenveld  : 3340   Median : 8.007
##  Hatert      : 2885   Mean   : 8.009
##  Brakkenstein: 2629   3rd Qu.: 9.026
##  Hengstdal   : 2503   Max.   :14.539
##  (Other)     :24725
```

```
# Remove NA from dataset
my.df<-na.omit(my.df)
```

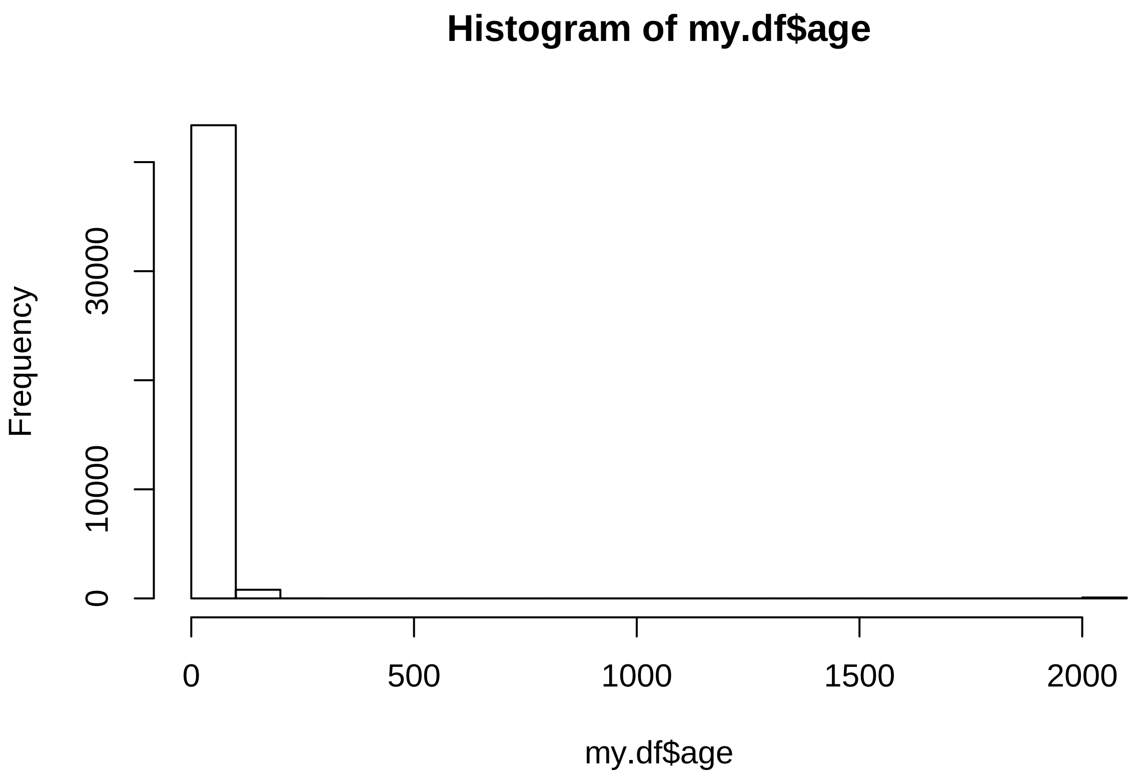
3.1.1 *Histogram*

Now we have seen which variables are numeric, we can investigate how this data is distributed:

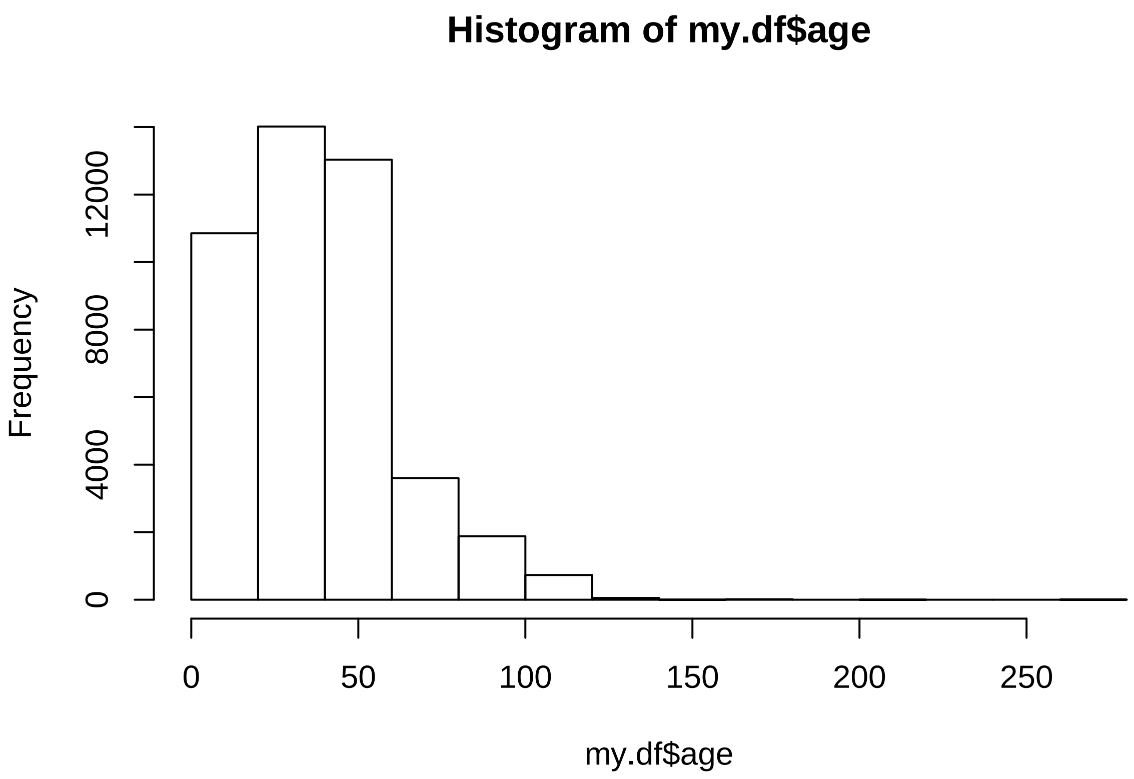
```
hist(my.df$planting.year)
```



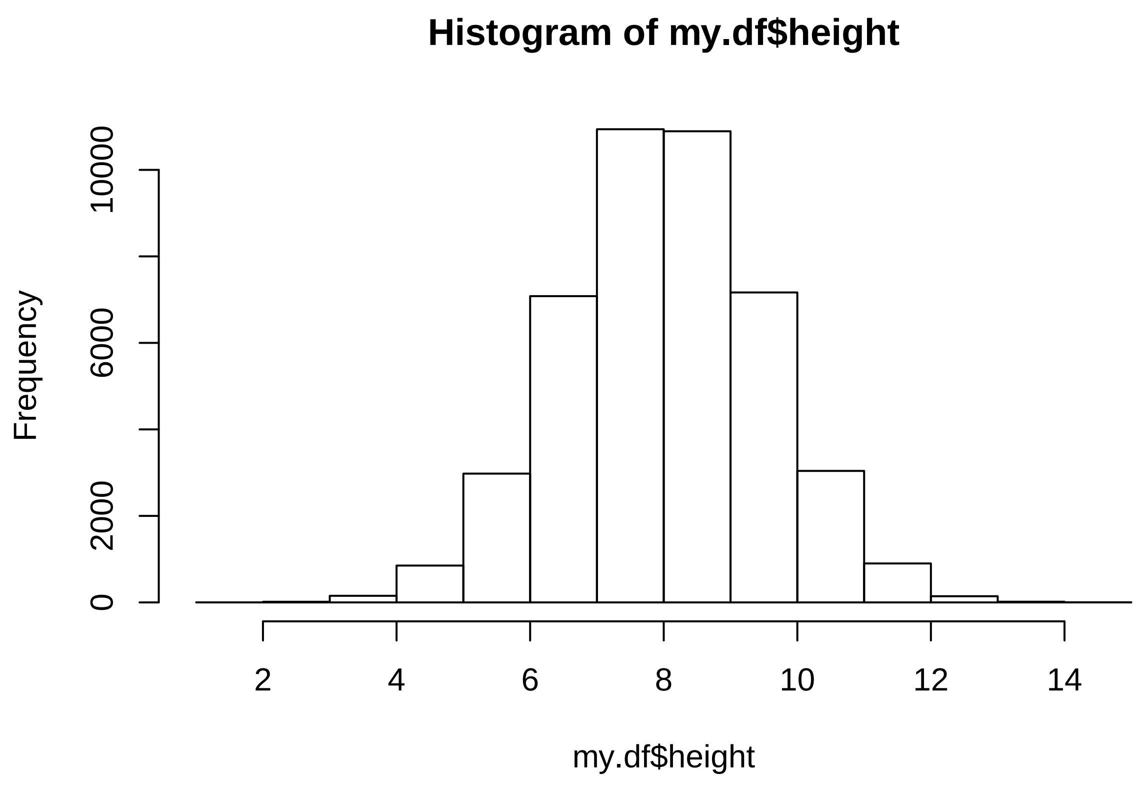
```r
# Isn't it nicer to look at the age?
my.df$age<-2019-my.df$planting.year
hist(my.df$age)
```



```
# A tree of 1000 years old seems like there has been a mistake.
#Remove this individual from the dataset.
my.df<-my.df[my.df$age<1000,]
hist(my.df$age)
```



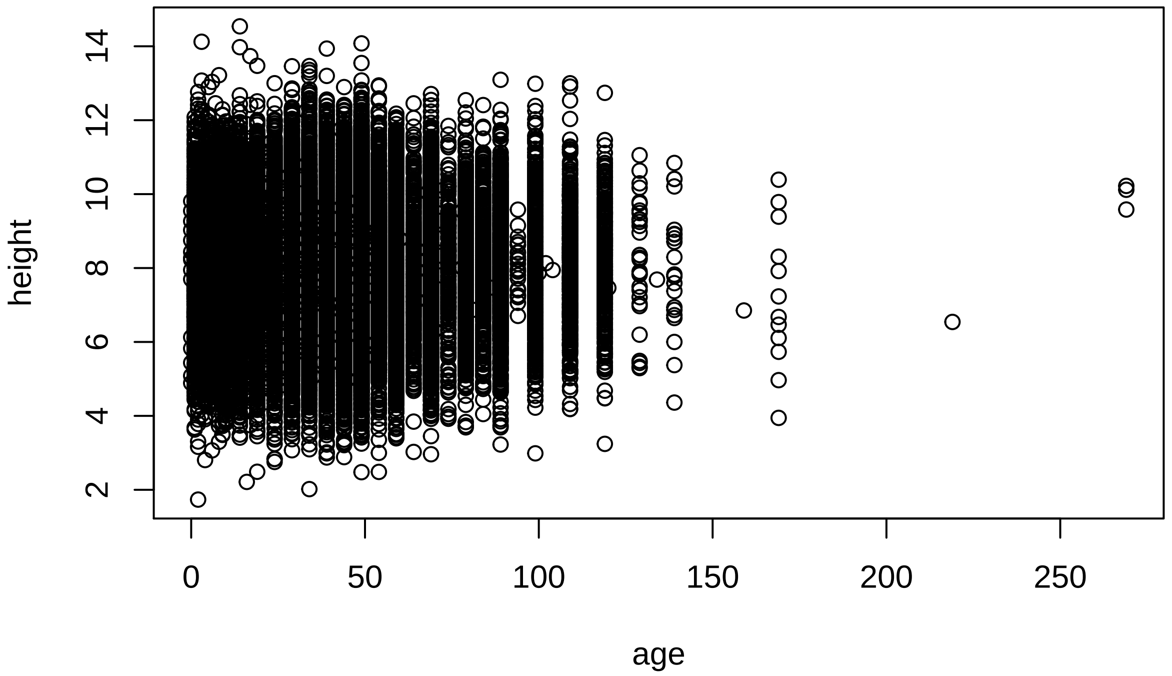
```
# Now we can actually see how the distribution of tree age in Nijmegen

# The other numeric variable in the dataset is height.
hist(my.df$height) # this trait is normally distributed.
```



3.1.2 *Scatter plot*

Another interesting question we can look at is ‘are all trees of the same age of the same height?’ In other words, how much variation in height is there with different ages?

```r
plot(height ~ age, data=my.df, type="p") # points
```

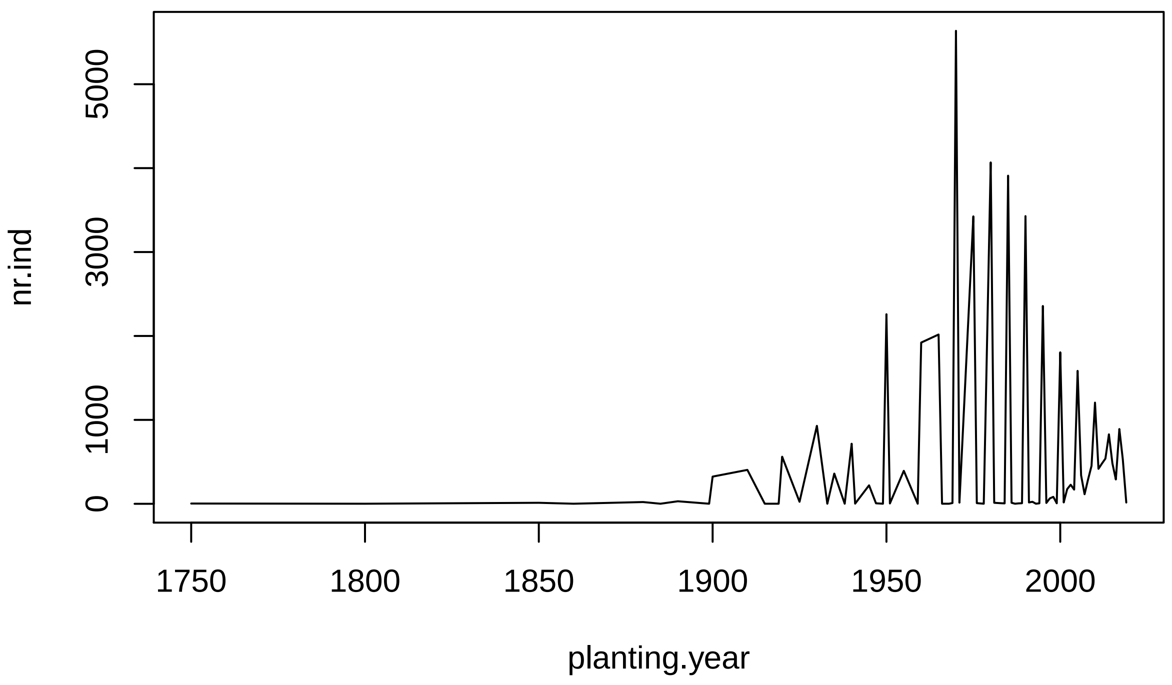


Question Can you see the vertical lines in the plot? What could this mean? *Trees have been planted in 'programs' (many trees at the same time) and those trees have different heights*

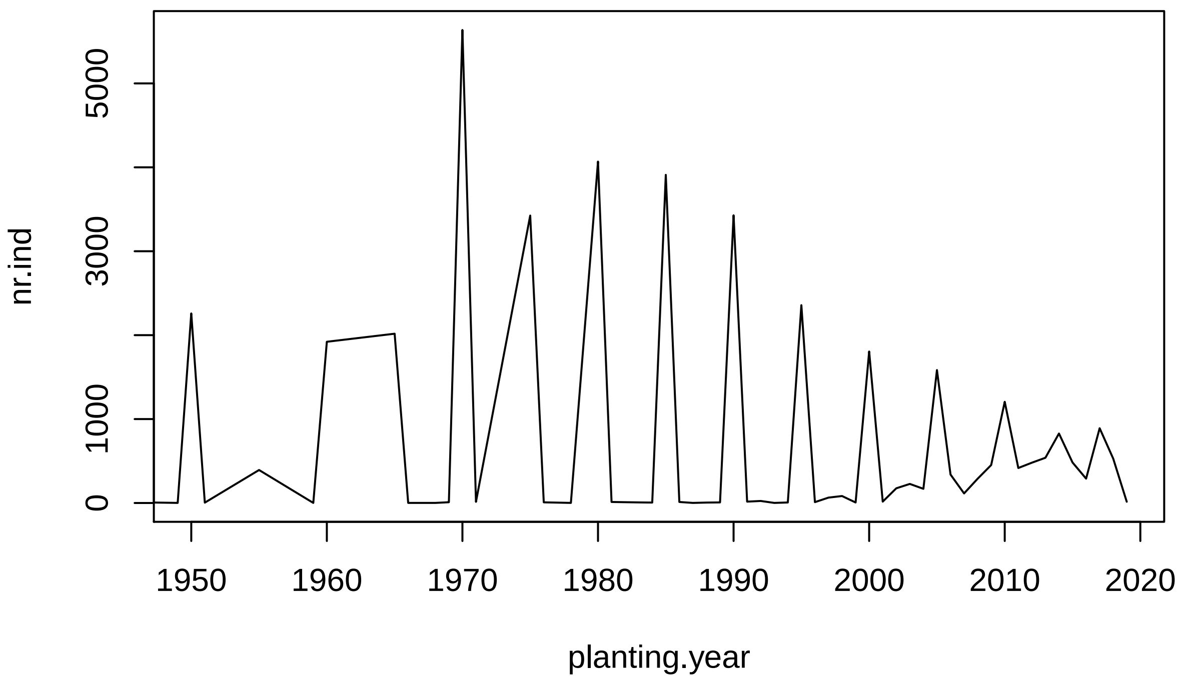
3.1.3 *Line plot*

Now we have seen that trees have been planted by 'program', we can plot the actual number of trees that have been planted in each year:

```r
# step 1. create a new dataset: the sum of all individuals planted per year.
my.df$nr.ind<-1 # create a column with the number 1 (1 row = 1 individual)
new.df<-aggregate(nr.ind ~ planting.year, data=my.df, FUN = "sum")

# step 2 make the plot
plot(nr.ind ~ planting.year, data=new.df, type="l") #lines
```



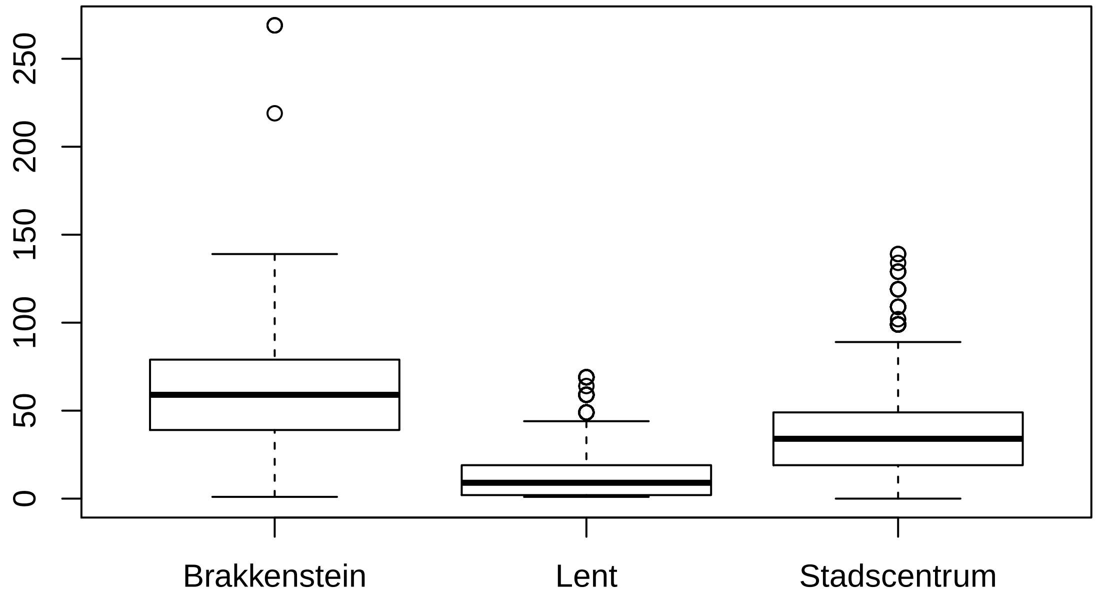
```
# step 3. let's zoom in a bit
plot(nr.ind ~ planting.year, data=new.df, type="l", xlim=c(1950,2019)) #lines
```



3.1.4 *Box plot*

These planting programs have probably not occured throughout the whole city each year. What is the age of trees in different neighbourhoods? More specifically, what is the variation in tree age for the different neighbourhoods?

```r
# Let's look at the following neighbourhoods:
# city center, Lent, Brakkenstein (old and new parts of the city)
new.df<-my.df[my.df$neighbourhood=="Stadscentrum"|
                my.df$neighbourhood=="Lent"|
                my.df$neighbourhood=="Brakkenstein",]
# Since many more neighbourhoods are originally in this dataset,
# we actually need to 'drop levels':
# remove all the categories in this variable we are not interested in
new.df$neighbourhood<-droplevels(new.df$neighbourhood)
boxplot(age ~ neighbourhood, data=new.df)
```



```r
# To remove outliers from the plot, you can add outline=FALSE
boxplot(age ~ neighbourhood, data=new.df, outline=FALSE)
```

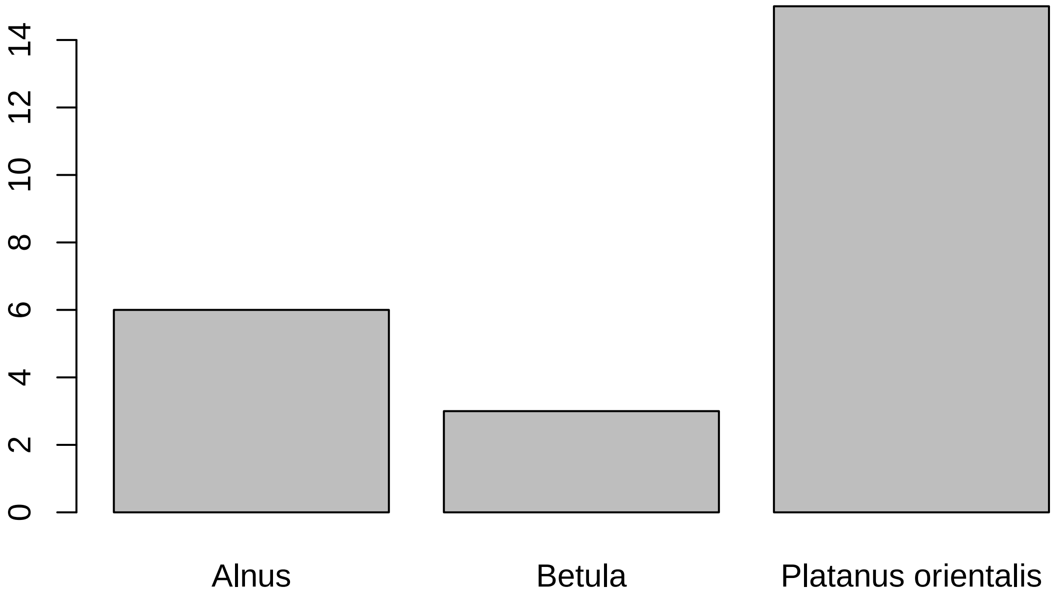
Question What do the boxes mean? *median, 25th and 75th quantile, and minimum an maximum value*

3.1.5 *Bar plot*

At this point, we might wonder about the species that actually have been planted throughout all those years.

```r
# create a table with all the species and the frequency of each species
species.table<-table(my.df$species) #There are too many species for one plot

# Let's select some common species and see how many there are in Nijmegen
new.df<-my.df[my.df$species=="Betula"|                  # in Dutch: Berk
```

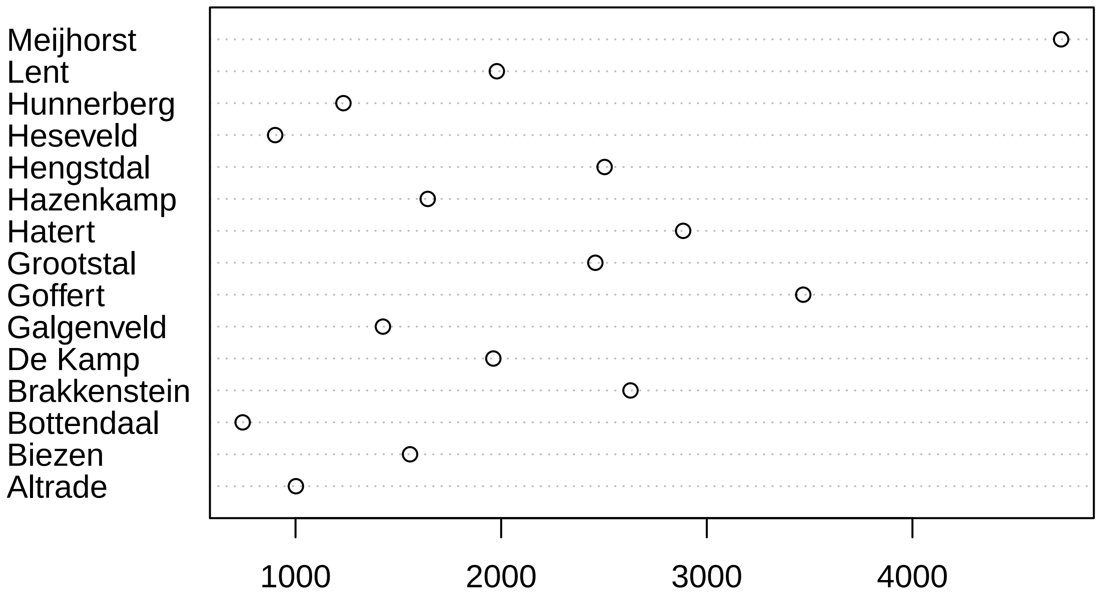
```
                my.df$species=="Alnus"|                  # in Dutch: Els
                my.df$species=="Platanus orientalis",] # in Dutch: Plantaan
new.df$species<-droplevels(new.df$species) # remove all the other categories
species.table<-table(new.df$species)
barplot(species.table)
```



3.1.6 *Dot chart*

What is the greenest neighbourhood of Nijmegen (in terms of trees)? In other words, how many trees are there per neighbourhood?

```
# Create a new dataframe with the number of trees per neighbourhood and
# the names of the neighbourhood as row names
my.df$nr.ind<-1
new.df<-aggregate(nr.ind ~ neighbourhood, data=my.df, FUN = "sum")
row.names(new.df)<-new.df$neighbourhood
# Since there are way too many neighbourhoods.
# Let's organise and only look at 15 neighbourhoods.
new.df2<-new.df[1:15,]
dotchart(new.df2$nr.ind, labels=row.names(new.df2))
```

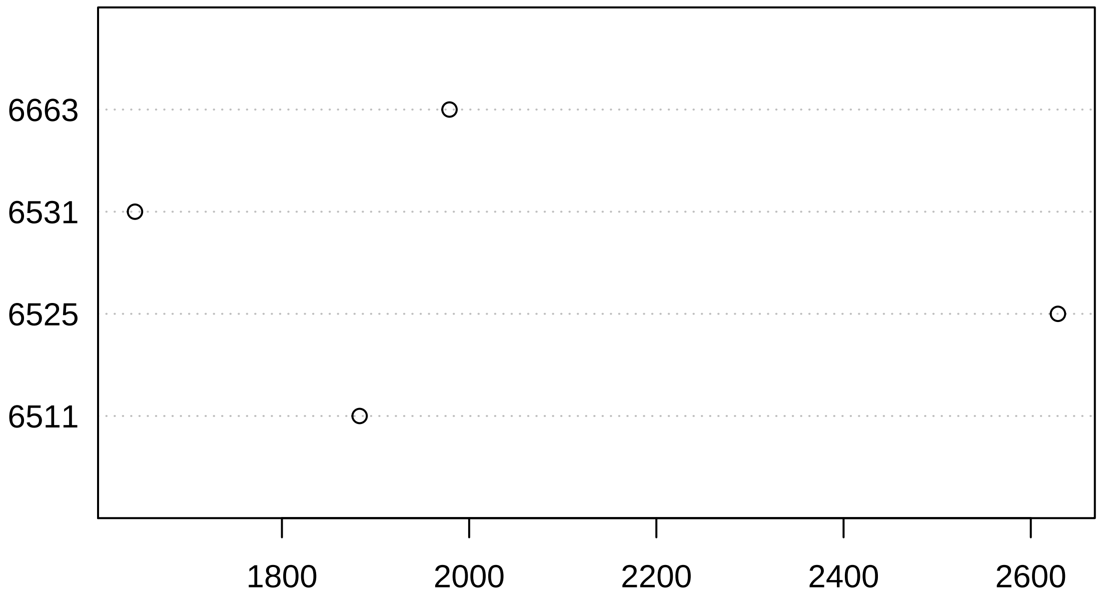


3.2 Making graphs more attractive

In section 2.3 a few hints are given to improve the visuals of graphs. Let's use them in this practical. We will do this by answering the question **Which neighbourhood is the greenest?**

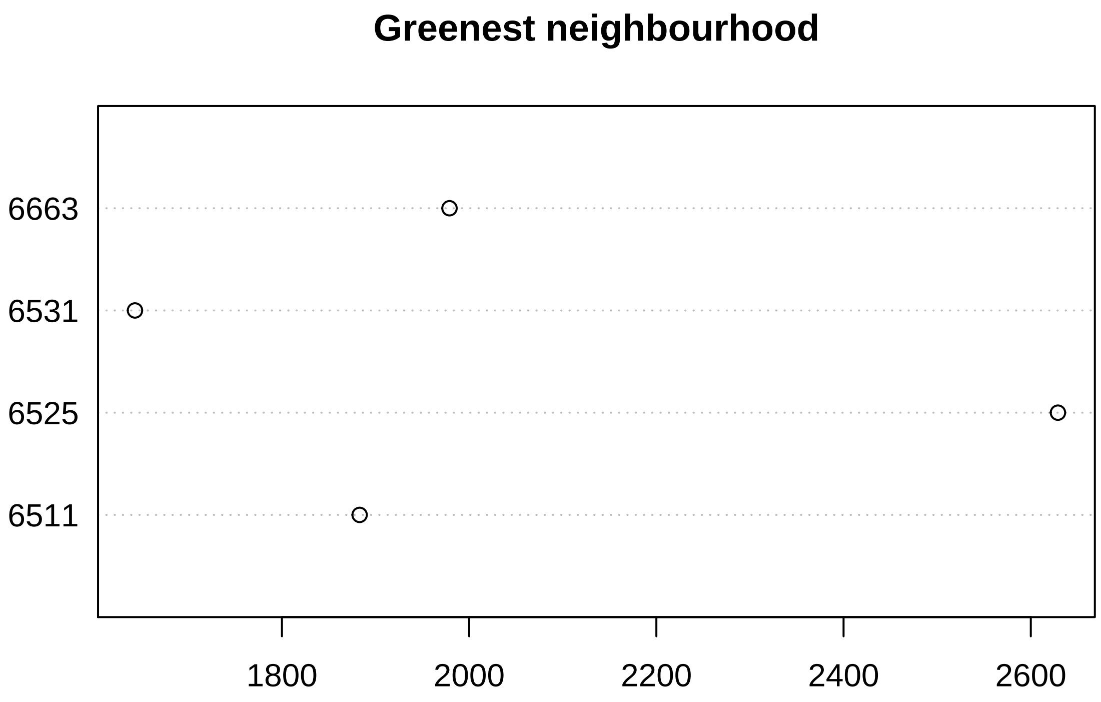
First, we create a dataframe that can be used for dot charts with the amount of trees in a neighbourhood.

```
new.df<-aggregate(nr.ind ~ zip.code, data=my.df, FUN = "sum")
new.df2<-  new.df[new.df$zip.code==6511| # city center
                  new.df$zip.code==6525| # university campus (Brakkenstein)
                  new.df$zip.code==6663| # Lent
                  new.df$zip.code==6531,] # Coline
row.names(new.df2)<-new.df2$zip.code
dotchart(new.df2$nr.ind, labels=row.names(new.df2))
```



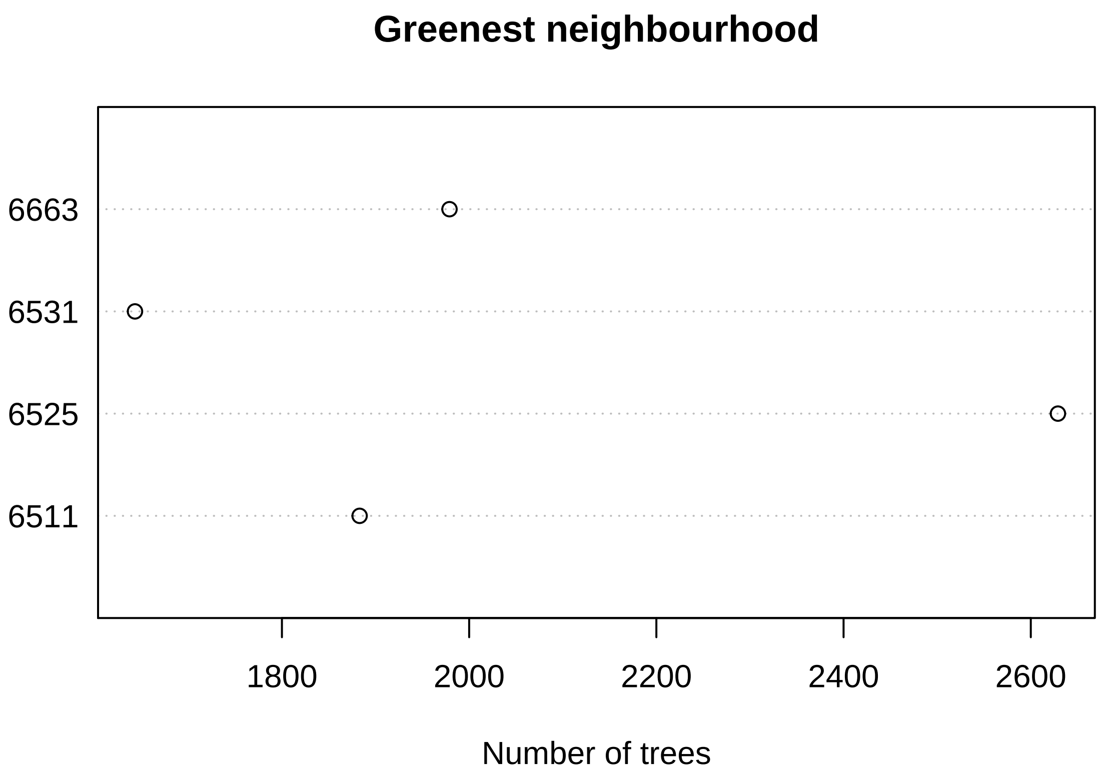
3.2.1 *Title*

```r
dotchart(new.df2$nr.ind, labels=row.names(new.df2),
         main="Greenest neighbourhood")
```



3.2.2 *Axis label*

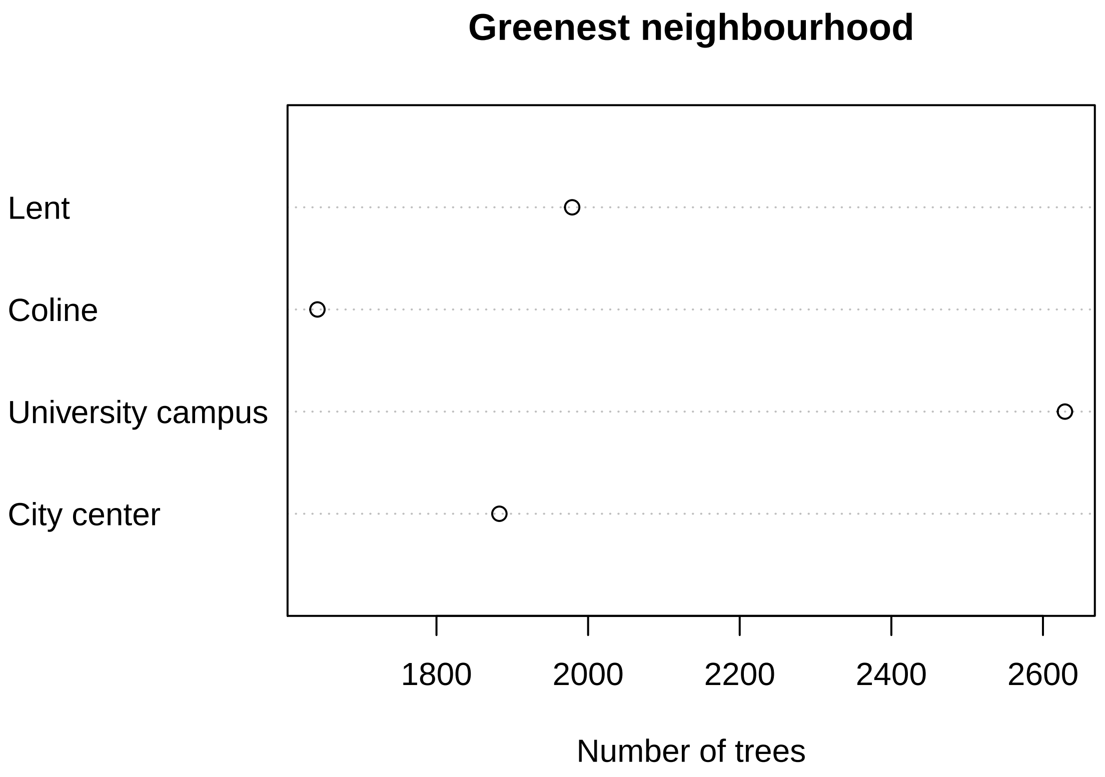
```
# include label for x-axis
dotchart(new.df2$nr.ind, labels=row.names(new.df2),
         main="Greenest neighbourhood",
         xlab="Number of trees")
```



```
# change Y axis --> categorical
# step 1. check what categories we have and in what order
new.df2
```

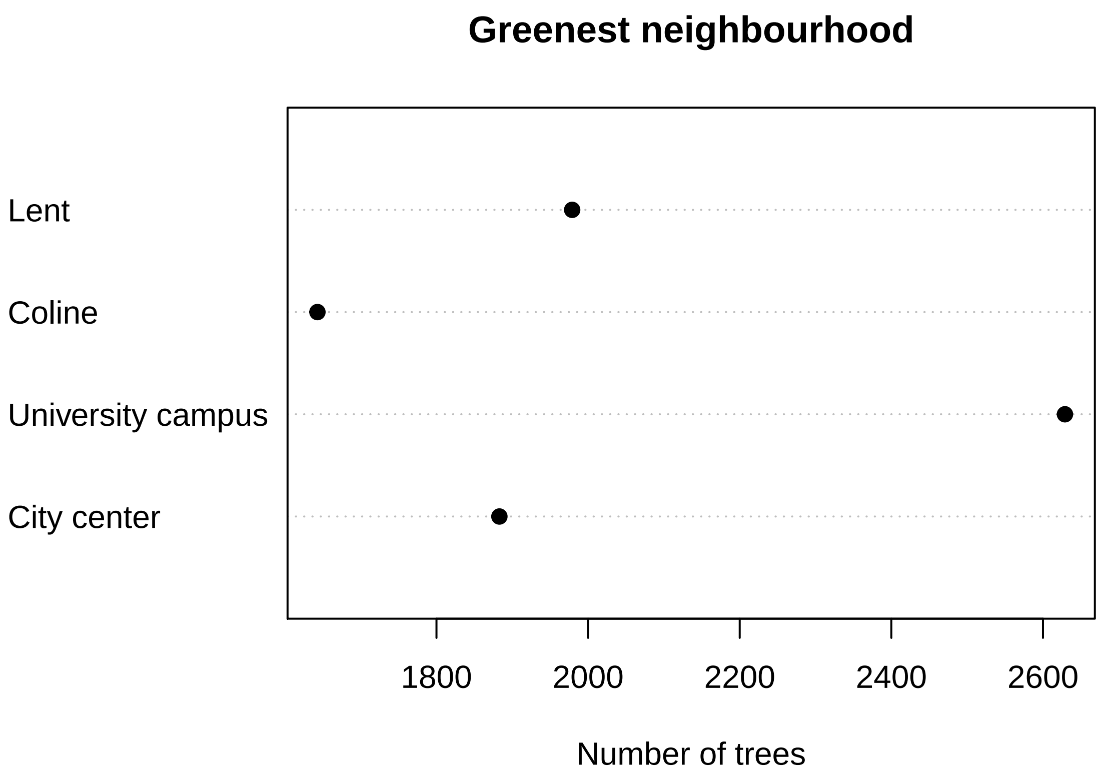
```
##      zip.code nr.ind
## 6511     6511   1883
## 6525     6525   2629
## 6531     6531   1643
## 6663     6663   1979
```

```
# y axis is ordered on zip code
dotchart(new.df2$nr.ind,
         main="Greenest neighbourhood",
         xlab="Number of trees",
         labels=c("City center",#6511
                  "University campus",#6525
                  "Coline",#6531
                  "Lent"#6663
                  ))
```



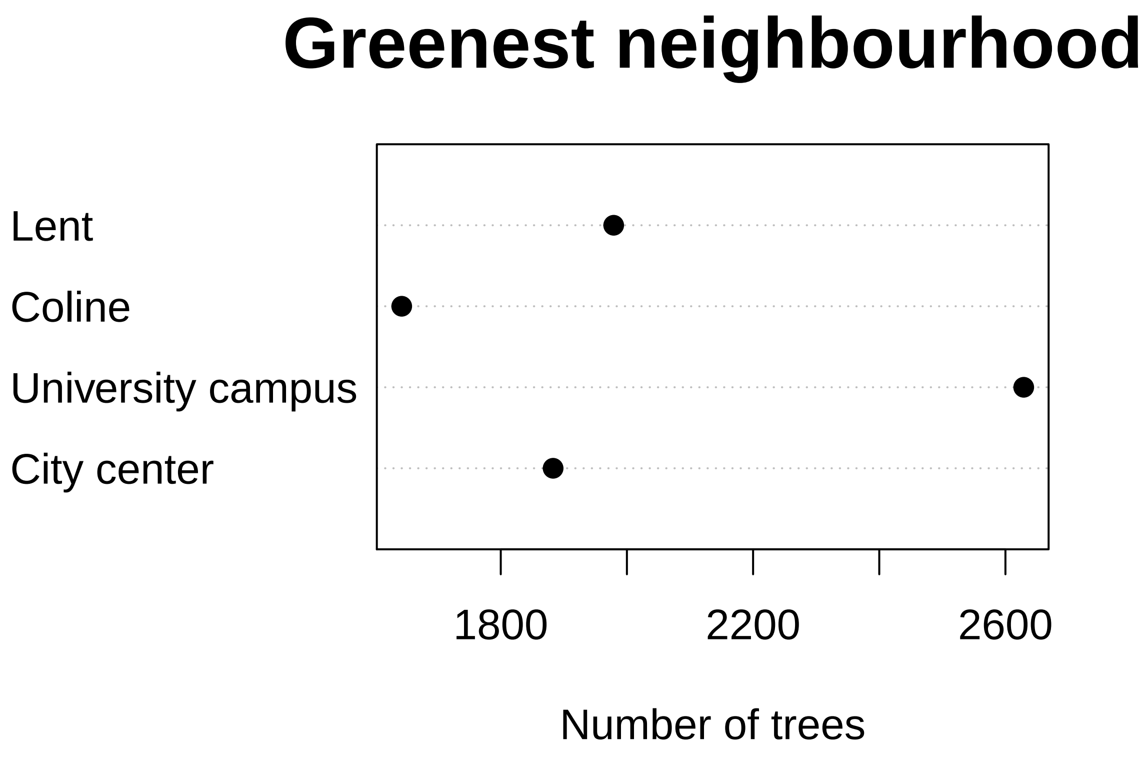
3.2.3 *Change point type*

```
dotchart(new.df2$nr.ind,
         main="Greenest neighbourhood",
         xlab="Number of trees",
         labels=c("City center",#6511
                  "University campus",#6525
                  "Coline",#6531
                  "Lent"#6663
                  ),
         pch=19)
```



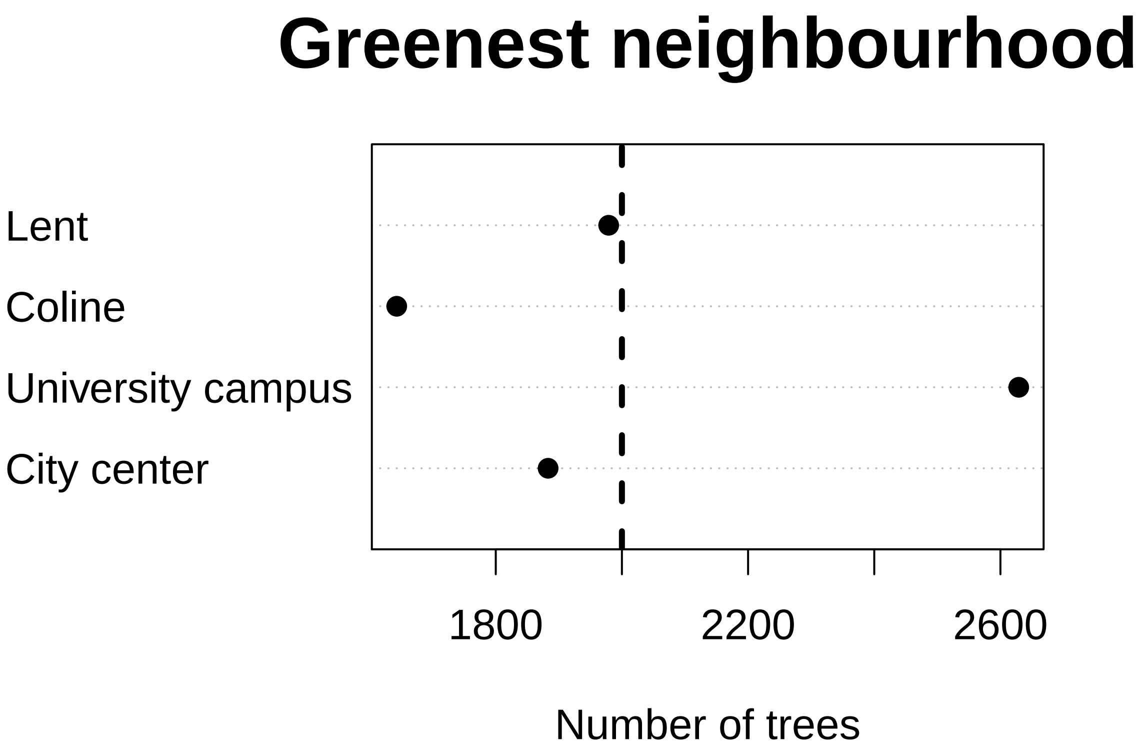
3.2.4 *Change size*

```
dotchart(new.df2$nr.ind,
         main="Greenest neighbourhood",
         xlab="Number of trees",
         labels=c("City center",#6511
                  "University campus",#6525
                  "Coline",#6531
                  "Lent"#6663
                  ),
         pch=19,
         cex=1.3, cex.main=1.7)
```



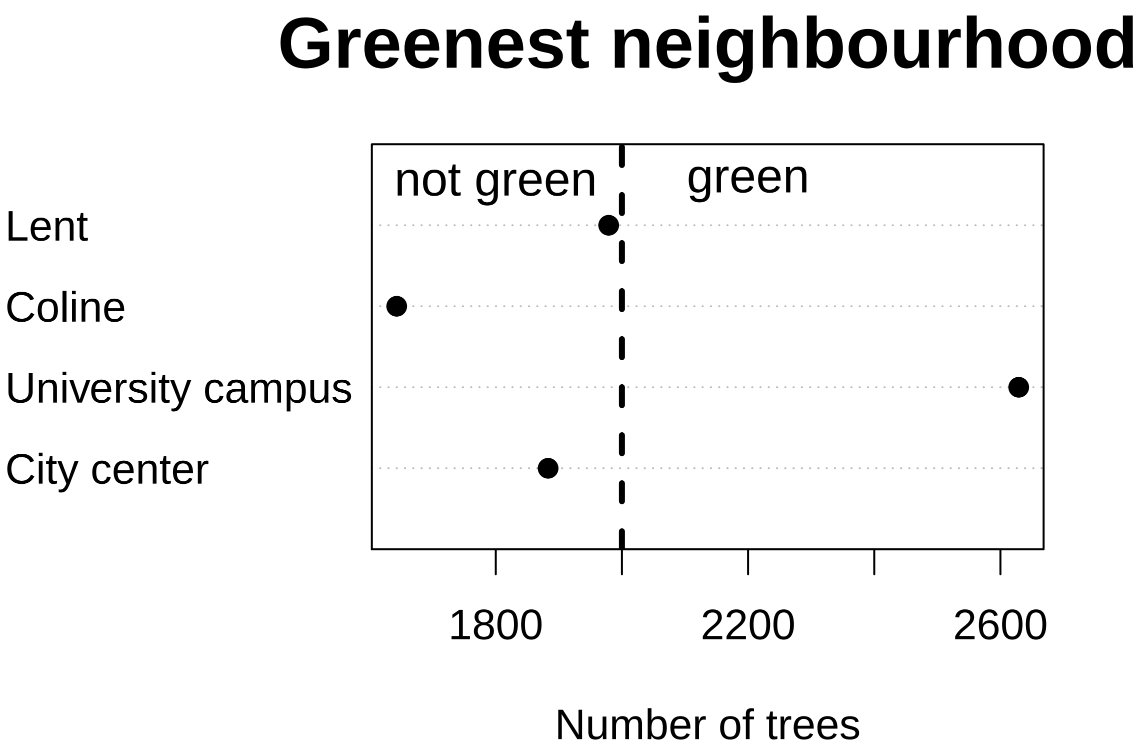
3.2.5 *Add lines*

```r
dotchart(new.df2$nr.ind,
         main="Greenest neighbourhood",
         xlab="Number of trees",
         labels=c("City center",#6511
                  "University campus",#6525
                  "Coline",#6531
                  "Lent"#6663
                  ),
         pch=19,
         cex=1.3, cex.main=1.7)

# Let's say, we consider a neighbourhood green if it has more than 2000 trees.
abline(v=2000, lwd=3, lty=2)
```



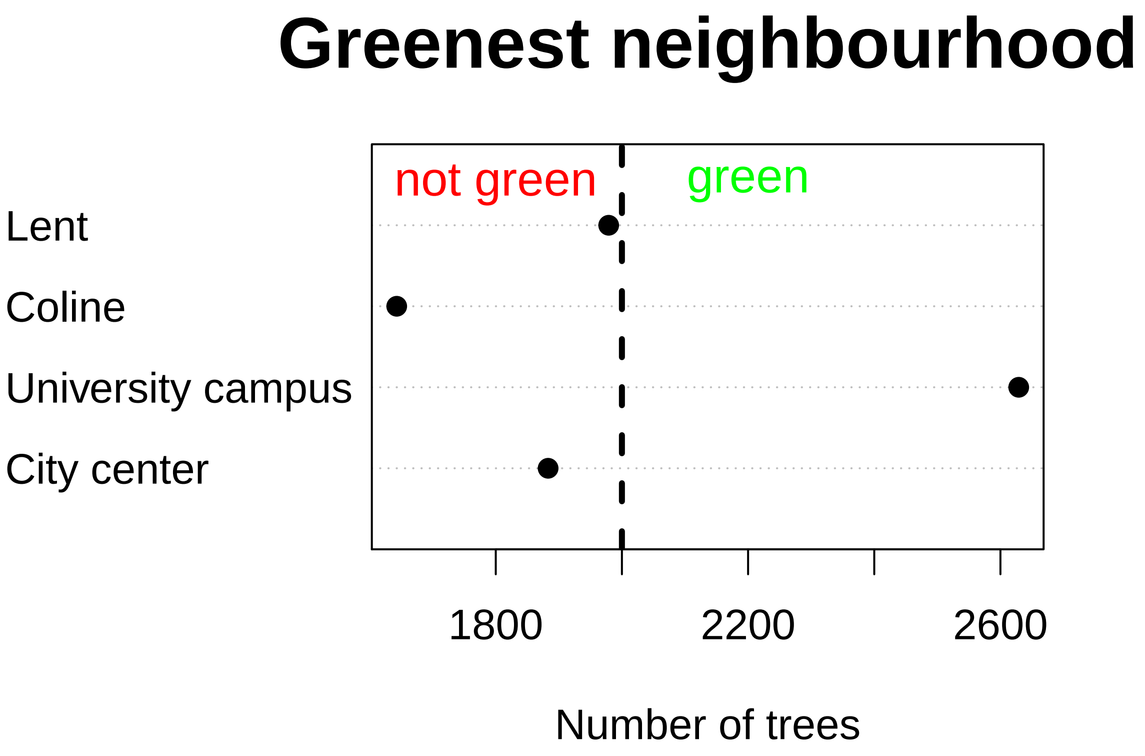
3.2.6 *Additional text*

```
dotchart(new.df2$nr.ind,
         main="Greenest neighbourhood",
         xlab="Number of trees",
         labels=c("City center",#6511
                  "University campus",#6525
                  "Coline",#6531
                  "Lent"#6663
                  ),
         pch=19,
         cex=1.3, cex.main=1.7)

abline(v=2000, lwd=3, lty=2)

text(x=2200,y=4.5,"green",cex=1.5)
text(x=1800,y=4.5,"not green",cex=1.5)
```



3.2.7 *Color*

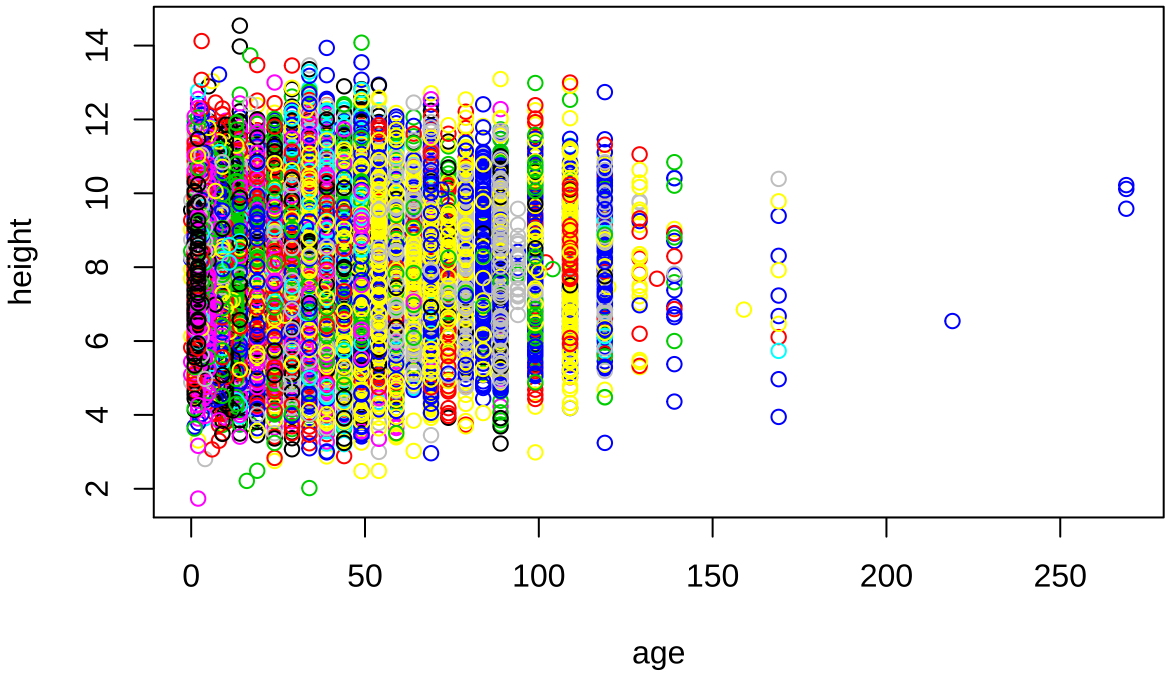
```
dotchart(new.df2$nr.ind,
         main="Greenest neighbourhood",
         xlab="Number of trees",
         labels=c("City center",#6511
                  "University campus",#6525
                  "Coline",#6531
                  "Lent"#6663
                  ),
         pch=19,
         cex=1.3, cex.main=1.7)

abline(v=2000, lwd=3, lty=2)

text(x=2200,y=4.5,col='green',"green",cex=1.5)
text(x=1800,y=4.5,col='red',"not green",cex=1.5)
```



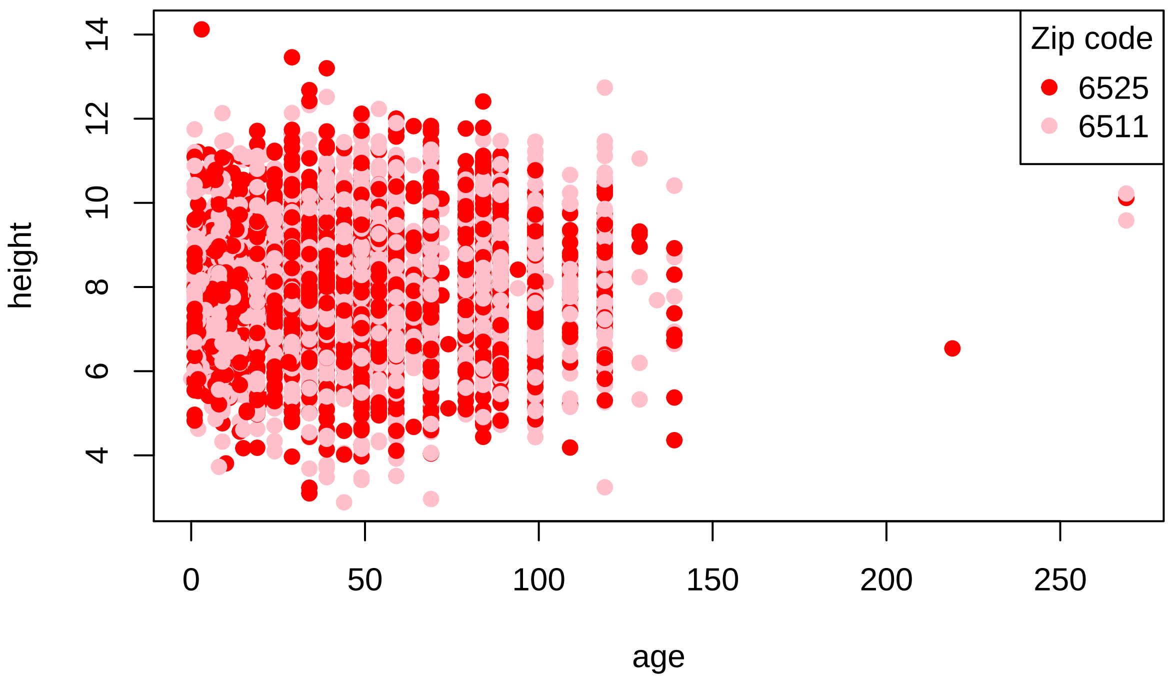
3.2.8 *Legend*

Now we want to see how the height of the tree relates to it's age, and if this differs between neighbourhoods.

```r
plot(height ~ age, col=neighbourhood, data=my.df, type="p") # points
```



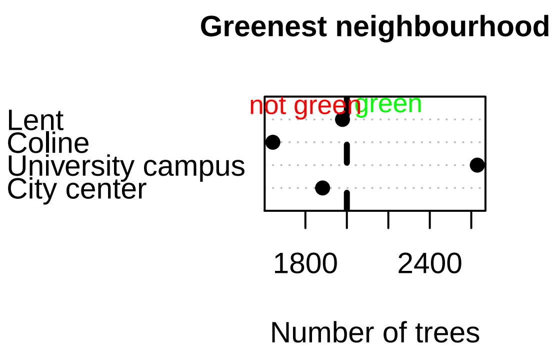
```
# There are way too many neighbourhoods to make sense out of the color scheme.

#Let's make it simple and look at two negihbourhoods only.
my.df2<-  my.df[my.df$zip.code==6511| # city center
                my.df$zip.code==6525,] # university campus (Brakkenstein)

# Set a color scheme
cols<-c("red","pink")
plot(height ~ age, col=cols, data=my.df2, type="p", pch=19) # points

# To know which color represents which neighbourhood, we should include a legend.
legend("topright", title="Zip code",
       legend=c(unique(my.df2$zip.code)),col=cols,pch=19) #vector with the labels
```

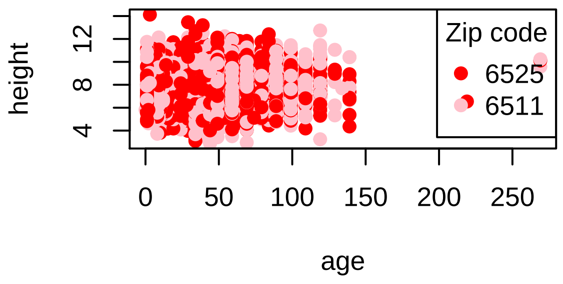


3.2.9 *Multi-plot*

```
par(mfrow=c(2,2))

# Graph 1
dotchart(new.df2$nr.ind,
         main="Greenest neighbourhood",
         xlab="Number of trees",
         labels=c("City center",#6511
                  "University campus",#6525
```

```
                   "Coline",#6531
                   "Lent"#6663
                   ),
         pch=19,
         cex=0.9, cex.main=1)

abline(v=2000, lwd=3, lty=2)

text(x=2200,y=4.5,col='green',"green",cex=1)
text(x=1800,y=4.5,col='red',"not green",cex=1)

# This creates an empty plot, thereby skipping a place in the par() settings
plot.new()

# Graph 2
plot(height ~ age, col=cols, data=my.df2, type="p", pch=19) # points
legend("topright", title="Zip code",
       legend=c(unique(my.df2$zip.code)),col=cols,pch=19) #vector with the labels

# Also fill the last quarter by an empty plot
plot.new()
```





3.2.10 *Saving the image*

The image above doesn't look good at all. In order to save the image propperly, we need to change some settings.

```
jpeg("Save.live.example.jpeg")

    par(mfrow=c(2,1))

# Graph 1
dotchart(new.df2$nr.ind,
         main="Greenest neighbourhood",
         xlab="Number of trees",
         labels=c("City center",#6511
                  "University campus",#6525
                  "Coline",#6531
                  "Lent"#6663
                  ),
         pch=19,
         cex=1.3, cex.main=2)

abline(v=2000, lwd=3, lty=2)

text(x=2200,y=4.5,col='green',"green",cex=2)
text(x=1800,y=4.5,col='red',"not green",cex=2)


# Graph 2
plot(height ~ age, col=cols, data=my.df2, type="p", pch=19) # points
legend("topright", title="Zip code",
       legend=c(unique(my.df2$zip.code)),col=cols,pch=19) #vector with the labels

dev.off()
```

```
# Not everything is visible in the saved image as in the figure under 3.2.9.
# We need to set the size of the image in order to have everthing in there.
jpeg("Save.live.example2.jpeg",width=500,height=500)

par(mfrow=c(2,1))

# Graph 1
dotchart(new.df2$nr.ind,
         main="Greenest neighbourhood",
         xlab="Number of trees",
         labels=c("City center",#6511
                  "University campus",#6525
                  "Coline",#6531
                  "Lent"#6663
                  ),
         pch=19,
         cex=1.3, cex.main=2)

abline(v=2000, lwd=3, lty=2)

text(x=2200,y=4.5,col='green',"green",cex=2)
text(x=1800,y=4.5,col='red',"not green",cex=2)


# Graph 2
plot(height ~ age, col=cols, data=my.df2, type="p", pch=19) # points
legend("topright", title="Zip code",
       legend=c(unique(my.df2$zip.code)),col=cols,pch=19) #vector with the labels

dev.off()
```

4 Individual exercises

For the individual exercise we will use a different dataset. This dataset contains plant trait data of Chinese savanna tree species. Samples are taken in the dry valley near Yuanjian, Yunnan, China. These savanna systems are being grazed by herbivores. For trees to survive, they need to have a form of defense. In general, we can classify species into two types of defense: structural and non-structural. The structural defenders are more bush-shaped and have many branches and/or spines that prevent leaves on the inside of this structure to be eaten. Non-structural defenders have the tendancy to grow tall, so that herbivores simply cannot reach their leaves anymore. The data in the dataset is sampled along a gradient of herbivore pressure.

The dataset contains the following data:

- OBS - observation number (unique number for each sampled tree) *integer*
- SPP - species name *categoric*
- GRAZING - indication if the area is grazed or not (Yes/No) *categoric*
- H - tree height (meter) *numeric*
- CA - canopy area (square meter) *numeric*
- LS - leaf size (square mili meter) *numeric*
- SD - is this species a structural defender (Yes/No) *categoric*
- SPINE - does this individual have spines (Yes/No) *categoric*

```r
data<-read.table("Chinese.savanna.trees_trait.data.txt",header=TRUE,dec=".")
```

4.1 Questions

To get you started, here are some instructions:

- Step 1. What traits are there?
- Step 2. How are these traits distributed? Do we need to transform any?
- Step 3. Make a graphs that best answers the question. Note, you might need to create a data table that is different from the dataset in order to make the graph.

Question 1 - Do traits differ between species of the two defense mechanisms?

Question 2 - Do traits change plastically with herbivore pressure? Is this different for the two different defense mechanisms?

Question 3 - Do traits scale allometrically with height? Well, the answer is right below, but we want you to plot the relationships anyway. Leaf size relates with height with the following relationship: LS=0.18*H+2.69 with an R-squared of 0.08 Canopy area relates with height with the following relationship: CA=0.93*H+0.13 with an R-squared of 0.7 Don't forget to add this information to each graph.

LU7: Pseudocode and functions

With exercises

Michela Busana & Konrad Mielke

michela.busana@ru.nl & k.mielke@science.ru.nl

Contents

1 Learning goals

By the end of the class we will be able to:

1. Apply fundamental tools to plan and organize code
2. Write functions

2 Pseudocode

2.1 What it is and why it is useful

⭐ BONUS: Definition

Coding is essentially taking a difficult problem and breaking it down into simple steps that a computer can understand.
The pseudocode is a simplified and clear way of expressing an algorithm in a spoken language.

When we code, we need to take a small amount of time to plan what exactly we want to do. This plan is called pseudocode.

Pseudocoding is similar to making an outline or structure before starting to write a manuscript. This structure is the foundation on which to develop both a manuscript or programme.

Planning ahead can save a lot of time, e.g., through the anticipation of potential problems. If we write proper pseudocode, we will be prepared for potential issues and prevent them. While, if we did not anticipate the issues, they would occur and we would have to do a lot of searching before we can find out where the bugs/errors are.

The pseudocode can be kept at the top of a script or integrated into the code in the form of comments. Both strategies will help us and also other users to understand our scripts.

Writing proper pseudocode always starts by asking what the problem is and what our goal is. If at the beginning it takes some practice to write efficient pseudocode, hang on! After the learning curve is over, we will be addicted to good organization. Writing smart pseudocode is also a critical transferrable skill that we can implement in any other programming language.

To better explain the concept, we will do an interactive example in class.

2.2 Making Dutch sandwiches

Let's write down the pseudocode for an algorithm to prepare a traditional Dutch sandwich with butter and hagelslag.

```
1. take the bag of bread
2. if the bag is closed
      i. open the bag
3. take a slice of bread
4. if the slide of bread is not covered with butter
      i. pick a knife carefully from the handle
     ii. take the butter
    iii. if the butter is closed
            • open the butter
     iv. scratch about a spoon of butter
      v. spread the butter uniformely on the bread
5. if the slice of bread is covered with butter
      i. take the box of hagelslag
     ii. if the box of hagelslag is closed
            1. open the box
    iii. gently sprinkle the sandwich with hagelslag
6. place the slice of bread on the plate
```

2.3 An example: the life cycle of *Boloria eunomia*

The *Boloria eunomia* (or *Proclossiana eunomia*) is an endangered butterfly typical of the Northern Hemisphere (see Nève G. et al. 2008 and Figure 1). The species has seen a sharp reduction due to habitat fragmentation.

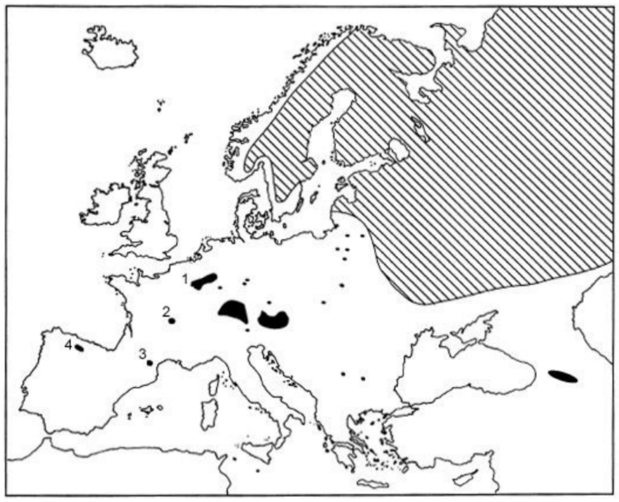


Figure 1 European distribution of Boloria eunomia as reported by Nève G. et al. 2008. The numbers 1, 2, and 3 represent the locations where the authors collected the data.

Butterflies are insects and undergo complete metamorphosis, in which there are four distinct stages: egg, larva (caterpillar), pupa (chrysalis), and adult. The *Boloria eunomia* is very sensitive to external temperatures and is expected to decline even further due to climate change. Previous studies showed that some life-history stages are more sensitive to rising temperature than others. More specifically, pupae are the most sensitive and a high mortality rate is associated with high temperatures. Figure 2 represents a drawing of the life cycle of *Boloria eunomia*.

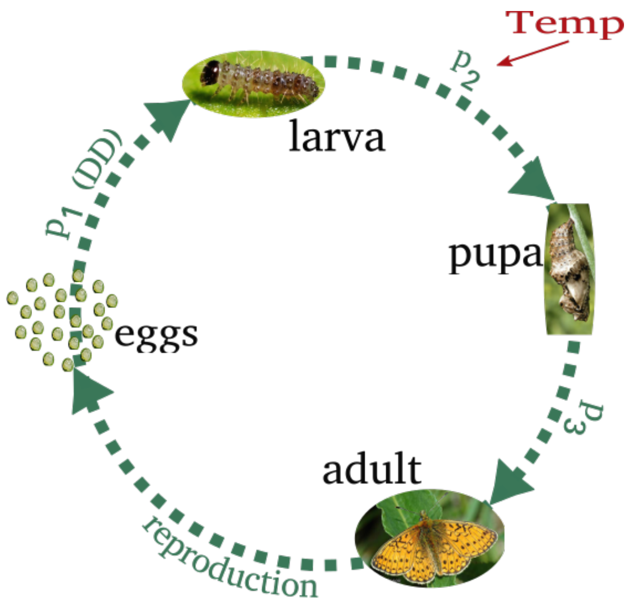


Figure 2 Simplified life cycle of Boloria eunomia. Eggs survive to transition to the larval stage with probability p_1. Egg survival depends on the total number of eggs present (density dependence or DD), such that when the number of eggs increases their survival probability decreases. Surviving larvae transition to become pupae with probability p_2. This probability is highly affected by the temperature of the environment. Pupae become adults with probability p_3. Adults produce eggs.

We want to predict if the species will go extinct under a future climate scenario of $+2°C$ by developing an individual-based model (IBM). IBMs are simulations of individuals in a population or ecosystem that mimics the interactions between individuals and the environment. Individuals and their properties are followed through their life, and relevant trends are summarized at the individual, population and community levels. They are used often to inform management and conservation practices or to investigate theoretical questions. They are useful to understand mechanisms driving patterns because it is easy to manipulate the models and observe their dynamics. We will learn further in the next lecture how we can use R to generate data.

Solution

How can we build an IBM for the Boloria eunomia?

We start from the life cycle and the biological processes that we need to include which are:

1. the survival probabilities/transitions from one stage to the next
2. reproduction
3. the climate as an element affecting the survival of the larvae
4. density dependence as a variable affecting egg survival

With all this information, we can go ahead and

1. write down the equations together with the life cycle of the insects
2. create the desired output: population sizes through time and a plot

The Pseudocode:

```
1. set initial population parameters: initial number of eggs, simulation time ($m$), and
   the values of the probabilities according to the equations written in the life-cycle
2. create an empty vector to store results, namely p_size
3. for each time step t
     1. individual eggs can survive to become larvae as a Bernoulli trial with probability
        p1, which includes density dependence. A Bernoulli trial is a random experiment
        that can return either success (1) or failure (0)
     2. surviving eggs become larvae. Eggs that die are removed from the population
     3. Larvae survive to pupae with probability p2. This probability is not fixed, but it
        is a function of the average temperature.
     4. Surviving larvae become pupae, dead larvae are removed
     5. Pupae survive to become adults with probability p3.
     6. Surviving pupae become adults, dead adults are removed
     7. if the number of adults is zero,
          • quit the program with an error telling the population went extinct after t
            time steps
     8. else if the number of adults size is > 0,
          • save the population size and let adults reproduce and lay eggs according to
            a poisson distribution
     9. then go back to step 4
    10. repeat from step 4 to 10
    11. when t equals m the simulation terminates and returns the population sizes
```

The words in ***green*** are operative verbs or actions. We can call them functions in a programming language.

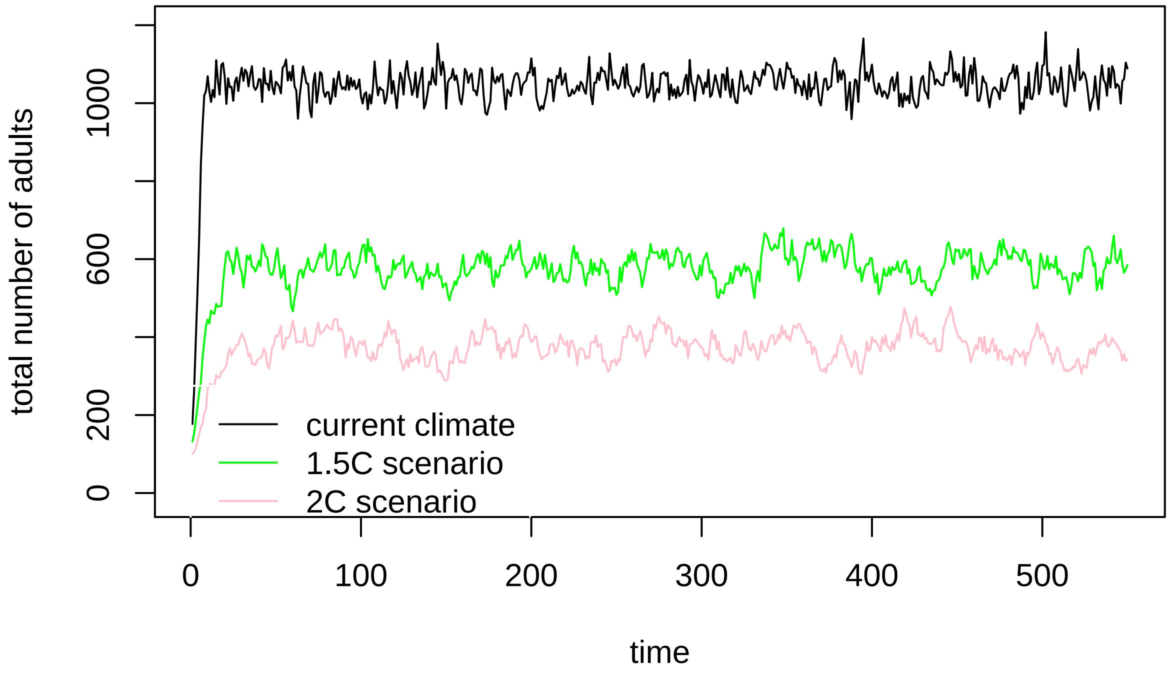
The words in ***red*** are common key-words used in many programming languages. They represent forks in the programme, as they determine which different action will be applied. Each decision is based on the answer to a question. The answer to an if/else statement is a Boolean expression (True or False).

The words in ***blue*** introduce a cycle, a loop. The word for is also a common key-word as you have already learned in the learning unit on loops.

The corresponding real code for *Boloria eunomia* is written below. At the end of the script, you will see a plot representing variations in the total number of adults through time in the three scenarios: (1) current climate, (2) $+1.5°C$ increase in global temperatures, and (3) $+2°C$ increase in global temperatures.

Please, note that the script is likely too advanced for you at this stage. It is only meant to be used as a reference. Do not worry if there are some lines that you do not understand!

```
# survival probabilities can be modelled with a logit transformation
logit <- function(x) y = exp(x) / (1 + exp(x))

# the function returns the population size of adult butterflies at
# each time step
boloria_dyn <- function(
  initial_p_size = 1000, # initial number of eggs #<<
  simulation_time = 550, # total simulation time
  Temp = 0, # current temperature difference experienced by the species
  p1 = 0.74, # probability of transitioning from egg to larva
  DD = -0.0001, # parameter of density dependence in egg surival
  iT = 0.1, # intercept of the stage transition from larva to pupa
  sT = -0.3, # slope for avg temperature of the transition from larva to pupa
  p3 = 0.45, # stage transition from pupa to adult
  lambda = 10 # reproduction, lambda of poisson distribution
  ){
  # create an empty vector to store results, p_size
  p_size <- rep(NA, simulation_time)
  eggs <- initial_p_size
  for (t in 1:simulation_time){
    # individual eggs can survive to become larvae as a bernoulli trial
    # with probability p1
    # surviving individuals become larvae
    pel = logit(p1 + DD * eggs)
    larvae <- sum(rbinom(eggs, 1, pel))
    # surviving larvae become pupae
    tmp = iT + sT * Temp
    p2 = logit(tmp)
    pupae <- sum(rbinom(larvae, 1, p2))
    adults <- sum(rbinom(pupae,1, p3))
    # calculate the  number of adults and store it
    p_size[t] <- adults
    if(p_size[t] <= 0){
      # if the population goes extinct interrupt the computation
      paste0(" population went extinct at time ", t)
      return(list(p_size = p_size[1:t]))
    } else {
      # Adults reproduce and lay eggs according to a poisson distribution
      eggs <- sum(rpois(adults, lambda))
    }
  }
  return(list(p_size = p_size[1:t]))
}

# generate population size of Boloria eunomia for three different climate scenarios
## current climate
res <- boloria_dyn()
## 1.5 C raise in temperature
resT15 <- boloria_dyn(Temp = 1.5)
## 2  C raise in temperature
resT2 <- boloria_dyn(Temp = 2)

# Compare the scenarios
plot(y = res$p_size, x = 1:length(res$p_size), type= "l", ylim = c(-13, 1200),
     ylab = "total number of adults", xlab = "time")
lines(y = resT15$p_size, x = 1:length(resT15$p_size), type= "l", col = "green")
lines(y = resT2$p_size, x = 1:length(resT2$p_size), type= "l", col = "pink")
legend(0, 275, col = c("black", "green", "pink"), lwd = c(1, 1, 1), legend =
         c("current climate", "1.5C scenario", "2C scenario"), box.col = "white")
```



3 Writing functions in R

3.1 What is a function?

BONUS: Definition

A function is a unit of code that takes an input, executes actions related to the input and from that generates an output.

We have already used functions before! The function

```r
mean(c(2,3))
```

```
## [1] 2.5
```

takes the input, which in this case is a vector consisting of the integers 2 and 3, executes its action by calculating the mean of the input and returns it.

R contains many built-in functions, such as `mean()`, `plot()`, `read.csv()`, that can be used right away. These functions are generally refered to as *base R functions*

In this lecture, we will learn how to write our own functions.

3.2 Reasons to write functions

BONUS: Tip

By writing functions we improve the:
- Reusability
- Abstraction
- Readability

of our code

Why would we want to write functions in the first place? As a general rule, we should think about writing a function as soon as we use the same code fragment more than once, and these are the main reasons why:

- **Reusability**: Once we have written our function, we can use it over and over again. Although it might be additional work in the beginning, it will save a lot of time in the long run. Just think about it: How much additional work would be necessary if there were no *base R function*? A lot, that is for sure!
- **Abstraction**: Writing a function can be difficult. However, once we have done so and we are sure that the function does what it is meant to do, we don't need to think about

the inner works anymore. As with other objects in life like telephones or the internet, we can just use it and trust that it works!

- **Readability**: A more extended script without functions would be terrible to read! Absolutely terrible! Imagine a world without functions in which we want to calculate the mean of a numeric vector `x` without using functions. We could use a `for` loop like in the example below. The code is unnecessarily long and hard to read when compared with the *base R function* `mean(x)`.

```
# given a vector x
x <- c(1, 2, 5)

# manually calculate the mean of x with a for loop
summation  <- 0
len <- 0
for (i in 1:3) {
  len <- len + 1
  summation  <- summation + x[i]
}
myMean <- summation / len
myMean
```

```
## [1] 2.666667
```

```
# calculate the mean of x with the R base function mean()
mean(x)
```

```
## [1] 2.666667
```

Because there isn't a function for everything in R, we have to help ourselves and write functions ourselves. Below it is explained how to do it.

3.3 Structure and syntax of a function

BONUS: Syntax

$$\begin{tikzcd}\textbf{Argument} \arrow[r] & \boxed{\textbf{Body}} \arrow[r] & \textbf{Output}\end{tikzcd}$$

A function consists of three parts.

- the **argument**: input that the function processes
- the **body**: the main part of the function in which it does its operations
- the **output**: what the function is returning when called

The syntax of a function is self-explanatory and has to be precisely followed.

```r
myFunction <- function(arguments) {
  body
  return(output)
}
```

3.3.1 The argument

What do we want the function to operate on? This is the question we have to ask when we decide what the argument should be. Arguments can be often classified in:

- Data arguments which supply the data to compute on
- Detail arguments which control the details of how the computation is done.

A natural (and very stupid) function with one data argument would be

```r
plusThree <- function(x) {
  y <- x + 3
  return(y)
}
plusThree(x = 3)
```

```
## [1] 6
```

Sometimes, we want to write functions in which some of the parameters are optional. We want to be able to adjust them, but if we do not have a specific idea on what the parameter should be, we want to use a default value. This is possible by setting them up with default values in the argument. We can then either call the function with only a value for x (in which case y is set to 3 in the example below) or both values for x and y.

```r
plusThreeOrOther <- function(x, y = 3) {
  y <- x + y
  return(y)
}
plusThreeOrOther(x = 3)
```

```
## [1] 6
```

```r
plusThreeOrOther(x = 3, y = 4)
```

```
## [1] 7
```

Please note that some functions do not need any argument. We can also write functions without:

```r
printOne <- function() {
  print("One!")
}
printOne()
```

```
## [1] "One!"
```

Functions without arguments are rare, though. Either they always operate the same, which might be necessary in some cases. Or, which is much worse, they operate on global variables. More on this topic later on.

3.3.2 The body

The body of a function is probably the most important part and for sure the most challenging part to design. In the body, the function has to execute operations to get from the input to the desired output. In functions, we can use all operations that we would also apply outside of a function, including loops and if/else statements.

The body of the function can be visualized calling the function name.

```
plusThree <- function(x) {
  y <- x + 3
  return(y)
}

plusThree
```

```
## function(x) {
##   y <- x + 3
##   return(y)
## }
```

3.3.3 The output

There are two different types of output.

One type is the output that is returned when a function is called. We can explicitly set this with the `return()` *base R function*. It is not mandatory to use‘return(), `but it is recommended. If there is no`return()“ *base R function*, our function will return the last evaluated expression (i.e., the last line of the body).

```
plusThree <- function(x) {
  y <- x + 3
  return(y)
}
# the function above is equivalent to:
plusThree <- function(x) {
  y <- x + 3
}
value <- plusThree(x = 3)
print(value)
```

```
## [1] 6
```

Multiple objects can be combined in a list to be returned simultaneously.

```
multipleCalculations <- function(x) {
  timesTwo <- 2 * x
  squared <- x * x
  returnList <- list("timesTwo" = timesTwo,"squared" = squared)
  return(returnList)
}
values <- multipleCalculations(x = 3)
print(values)
```

```
## $timesTwo
## [1] 6
##
## $squared
## [1] 9
```

```
# each element of the list can be accesses separately.
# For example, access the timesTwo output with
values$timesTwo
```

```
## [1] 6
```

```
# or with
values[[1]]
```

```
## [1] 6
```

In addition to the output explicitly returned by a function with `return()`, there can be additional output. A common example is output that is printed to the screen using the `print` function and output that is generated and saved to a file without returning it. We call this type of output a side effect of the function.

```
sideEffect <- function(x) {
  print(x^2)
  return(x)
}
# this call will print x^2
x <- sideEffect(3)
```

```
## [1] 9
```

```
# and then we can see the returned output by calling the object
x
```

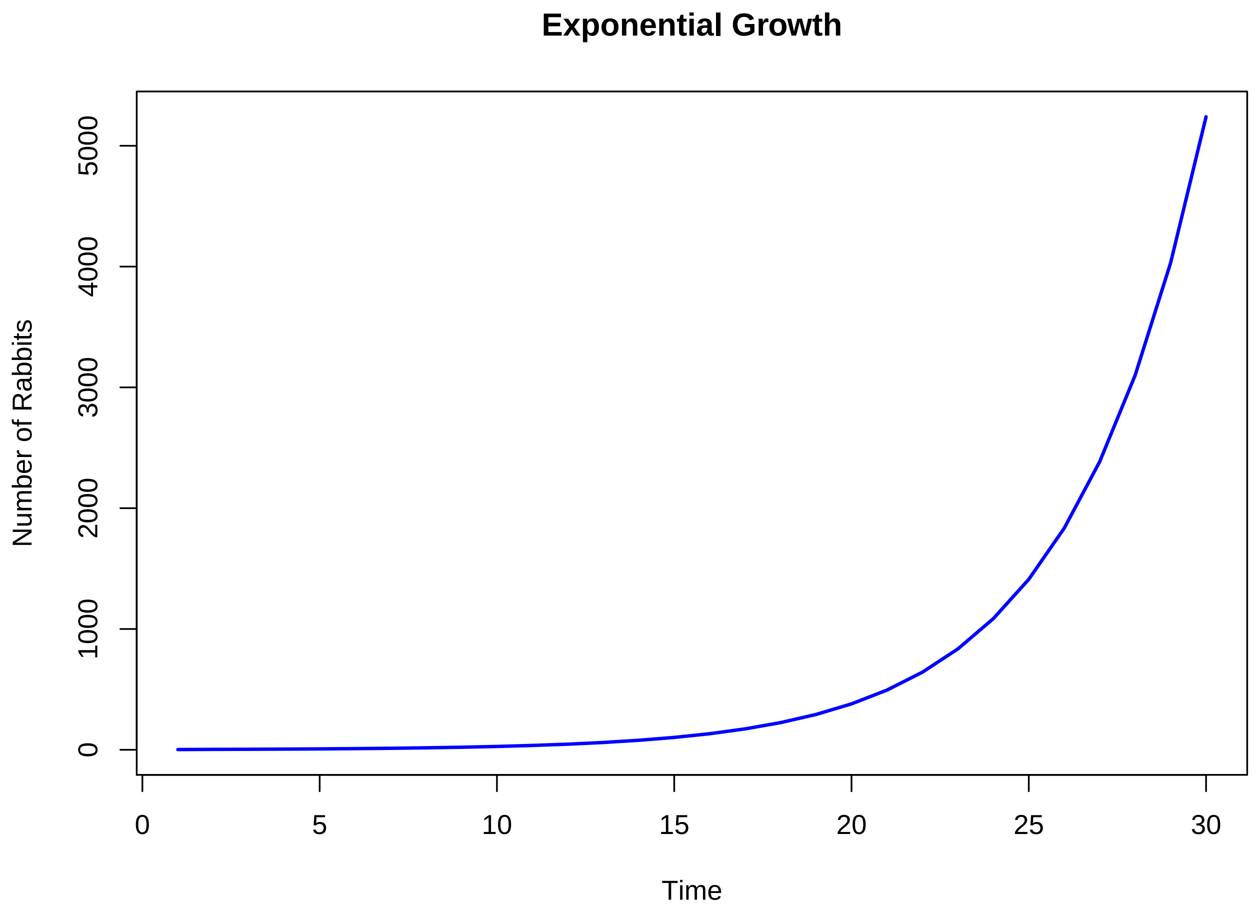
```
## [1] 3
```

3.4 Error messages

It's crucial to integrate informative error messages instead of letting a function return incorrect or surprising results.

Say we want to write a function to calculate the growth of a population of rabbits, which follows the exponential growth model (Lack 1954): $n_t = n_0 \times r^t$, where n_t is the population size at time t, n_0 is the initial population size, and r is the net reproductive rate which represents the number of offspring produced on average by an individual.

The population of rabbit is exemplified in the following plot:



A first good guess would be:

```
expGrowth <- function(tmax, n0, r) {
  nt <- rep(0, tmax) # create an empty vector to store results
  nt[1] <- n0
  nt[2:tmax] <- n0 * r^(2:tmax)
  return (nt)
}
expGrowth(20, 2, -1.1)
```

```
## [1]   2.000000   2.420000  -2.662000   2.928200  -3.221020   3.543122
## [7] -3.897434 4.287178 -4.715895 5.187485 -5.706233 6.276857
## [13] -6.904542 7.594997 -8.354496 9.189946 -10.108941 11.119835
## [19] -12.231818 13.455000
```

```
expGrowth(20, -5, 1.6)
```

```
##  [1]    -5.00000    -12.80000    -20.48000    -32.76800    -52.42880
##  [6]   -83.88608   -134.21773   -214.74836   -343.59738   -549.75581
## [11]  -879.60930  -1407.37488  -2251.79981  -3602.87970  -5764.60752
## [16] -9223.37204 -14757.39526 -23611.83241 -37778.93186 -60446.29098
```

The produced output is incorrect. The problem here is that the arguments are parameters that take unrealistic values. Thus, the function is not working as intended. This is called a **bug**.

BONUS: Definition

A **bug** is an error, flaw, failure or fault that causes a programme to produce an incorrect or unexpected result, or to behave in unintended ways. We learnt about bugs in LU5 and we will learn how to deal with bugs further in the LU10.

For example, the initial population size has to be bigger than one (at least two individuals in the initial population). We can solve the problem quickly by checking whether the value of `n0` is bigger than 1. If not, we want to exit the function. We can do so by throwing an error message condition is not met by using the `stop()` *base R function*.

```
expGrowth <- function(tmax, n0, r) {
  if(n0 <= 1){
    # if the condition applies, print error message
    stop("Error: Initial population size must be bigger than 1")
  }
  nt <- rep(0, tmax) # create an empty vector to store results
  nt[1] <- n0
  nt[2:tmax] <- n0 * r^(2:tmax)
  return (nt)
}
expGrowth(20, 2, -1.1)
```

```
##  [1]   2.000000   2.420000  -2.662000   2.928200  -3.221020   3.543122
##  [7]  -3.897434   4.287178  -4.715895   5.187485  -5.706233   6.276857
## [13]  -6.904542   7.594997  -8.354496   9.189946 -10.108941  11.119835
## [19] -12.231818  13.455000
```

```
expGrowth(20, -5, 1.6)
```

```
## Error in expGrowth(20, -5, 1.6): Error: Initial population size must be bigger than 1
```

Similar to the `stop()` function, there is the `stopifnot()` *base R function* which, as we might have expected, works exactly the opposite way around: If a condition is not met, the function will be stopped. The advantage here is that `stopifnot()` saves one line of code (namely the if statement), but we can't specify our own error messages. Thus, it is better to use `stop()` in most cases.

```
expGrowth <- function(tmax, n0, r) {
  nt <- rep(0, tmax) # create an empty vector to store results
  stopifnot(n0 > 1) # if the condition applies, print error message
  nt <- rep(0, tmax) # create an empty vector to store results
  nt[1] <- n0
  nt[2:tmax] <- n0 * r^(2:tmax)
  return (nt)
}
expGrowth(20, 2, -1.1)
```

```
##  [1]   2.000000   2.420000  -2.662000   2.928200  -3.221020   3.543122
##  [7]  -3.897434   4.287178  -4.715895   5.187485  -5.706233   6.276857
## [13]  -6.904542   7.594997  -8.354496   9.189946 -10.108941  11.119835
## [19] -12.231818  13.455000
```

```
expGrowth(20, -5, 1.6)
```

```
## Error in expGrowth(20, -5, 1.6): n0 > 1 is not TRUE
```

We can warn the user if he/she attempts to use argument values that we suspect to be incorrect by issuing a warning using the `warning()` *base R function*. Note that in this case, the rest of the function will still be executed. For example, we expect the values of r to be positive.

- If the exponent is larger than 0, we do the same as we did before.

```
expGrowth <- function(tmax, n0, r) {
  if(n0 <= 1) stop("Error: Initial population size must be > 1")
  if (r <= 0) warning("Warning: the net growth rate is negative!")
  nt <- rep(0, tmax) # create an empty vector to store results
  nt[1] <- n0
  nt[2:tmax] <- n0 * r^(2:tmax)
  return (nt)
}
expGrowth(20, 2, -1.1)
```

```
## Warning in expGrowth(20, 2, -1.1): Warning: the net growth rate is
## negative!
```

```
##  [1]   2.000000   2.420000  -2.662000   2.928200  -3.221020   3.543122
##  [7]  -3.897434   4.287178  -4.715895   5.187485  -5.706233   6.276857
## [13]  -6.904542   7.594997  -8.354496   9.189946 -10.108941  11.119835
## [19] -12.231818  13.455000
```

3.5 Global and local variables

We shortly talked about global variables before, but here we will go into more detail.

BONUS: Definition

Global variables are those variables that exist inside the main part of your R script. We can understand the concept better when we know the second type of variables: **Local variables**. These are variables that are created inside functions. They only exist inside the function, and once the function is finished, the variables are deleted.

So far, so good. There is a problem, though, which occurs when we use or modify global variables inside a function. To illustrate the problem, we go back to an example from a previous section in which a value of three was added to a variable. Here, however, we remove the arguments of the function.

```r
plusThree <- function() {
  return(x + 3)
}
x <- 5
value <- plusThree()
print(value)
```

```
## [1] 8
```

```r
x <- 10
value <- plusThree()
print(value)
```

```
## [1] 13
```

The problem here is that the output of the function depends on the rest of our program, outside of the function call. If we modify the value of x, the output of the function changes, too. More importantly, other people that try to use our function might get an error at execution if the variable `x` might not be defined in their global environment at all. An input that is not part of the arguments of the function is called **hidden input** or **hidden argument**.

3.6 Calling functions

Calling a custom function works the same as calling a pre-defined function. There are three different options to set the arguments of a function:

- **By position**: When we write a function, we name the different arguments in a specific order. When we call the function, we then can set the arguments in the same order, without anything besides the values that we intend the arguments to receive. This is the easiest and quickest way to do it, but also the most dangerous as you may have misremembered the order of the arguments. In that cae arguments will receive wrong values.

- **By full name**: Alternatively, we can set the arguments by their full name. Thus, we explicitly write out that argument $x = 3$. By doing so, we can neglect the order of the arguments. This is the safest method since we have to think explicitly about the direct association between arguments and values.
- **By partial name**: An uncommon method which we mention here for the sake of completeness is setting arguments by partial name. In case we only have one variable which starts with a specific letter, we can use this letter instead of the full name, e.g. using $b = 3$ instead of $base = 3$. As this does not have any significant advantages over setting arguments by their full name, we should not use this method in practice.

Below, see examples for calling the function `expGrowth()` with all three different methods.

```
expGrowth(20, 10, 1.4) # setting arguments by position
```

```
##  [1]   10.00000   19.60000   27.44000   38.41600   53.78240   75.29536
##  [7]  105.41350  147.57891  206.61047  289.25465  404.95652  566.93912
## [13]  793.71477 1111.20068 1555.68096 2177.95334 3049.13467 4268.78854
## [19] 5976.30396 8366.82554
```

```
expGrowth(tmax = 20, n0 = 10, r = 1.4) # setting arguments by full name
```

```
##  [1]   10.00000   19.60000   27.44000   38.41600   53.78240   75.29536
##  [7]  105.41350  147.57891  206.61047  289.25465  404.95652  566.93912
## [13]  793.71477 1111.20068 1555.68096 2177.95334 3049.13467 4268.78854
## [19] 5976.30396 8366.82554
```

```
expGrowth(t = 20, n = 10, r = 1.4) # setting arguments by partial name
```

```
##  [1]   10.00000   19.60000   27.44000   38.41600   53.78240   75.29536
##  [7]  105.41350  147.57891  206.61047  289.25465  404.95652  566.93912
## [13]  793.71477 1111.20068 1555.68096 2177.95334 3049.13467 4268.78854
## [19] 5976.30396 8366.82554
```

4 Cooking recipe: How to write a function?

Follow these steps when you want to write a function:

- Think about what you need to do: You will have to think about your function in any case, so it is best to do it before you start. At this point, it does not have to be too formalized. In particular, ask the following questions:
  - What is the aim of the function?
  - What is the input?

  - What is the output?
  - How to get from the input to the output?

- Write down the pseudocode: Now is the time to concretize and sort your thoughts from the previous step. Try to be as clear as possible.
- Start with a simpler version of your problem: Identify the key components and the details of your problem. Then remove the details for now; you can still add those later on. In addition, you need to be able to check whether your solution will be correct. So make sure that you know the right answer to your problem.
- Write a script that solves your problem: Don't think of functions or anything yet. Just answer the problem correctly by comparing it to what you know is the correct solution.
- Rewrite your script to use variables: Set all the variables in the first few lines of the code. After that, you should only use these variables for all operations. Do not set values in-between.
- Clean-up your example: Which lines are unnecessary? Which ones would be better of at different positions? Try to improve the readability and cleanness of your code. If it already a mess now, it will not get nicer later on. Make sure that your script still works from time to time!
- Turn your script into a function: You started with the aim of creating a function. Now is the time to go there. Instead of setting the variables at the beginning of the script, turn them into arguments of your function. Then test your function and make sure that it works correctly.
- Include the details: You sparred all the details in the beginning. Work them in one by one, regularly checking that your function works correctly. An excellent way to do this is by looking at exceptional cases of your problem; your function should ideally work for all of them.

5 Bonus for the brain: Efficiency and tips

This section is meant to propose extra information we thought will come handy for you in the future. We invite you to do/think about them after the course has finished and you have started a research project with the help of R.

Some tips:

- An useful function solves a problem correctly, and it's understandable to others.
- Start writing a function by thinking about the problem, the goal and write down the pseudocode!
- Give function and objects meaningful names! Be consistent! There is no right or wrong in this as long as your convention is clear. As a starting point, it's nice to use verbs for function names and nouns for argument names.

- Sometimes functions can take as arguments objects of different types (e.g., a matrix and a vector). If the output type depends on the input type, then the output type can change. Be careful as this can cause issues!

5.1 Functions can be arguments too

We can call a function as an argument in another function.

```r
# summarize_col applies a function fun to a data frame, df
summarize_col <- function(df, fun){
  output <- rep(0, length(df))
  for (i in 1:length(df)){
    output[i] <- fun(df[[i]])
  }
  return(output)
}

df <- data.frame(A = 1:20, B = (1:20)^3)

summarize_col(df, fun = median)
```

```
## [1]   10.5 1165.5
```

```r
summarize_col(df, fun = mean)
```

```
## [1]   10.5 2205.0
```

```r
summarize_col(df, fun = sd)
```

```
## [1]    5.91608 2504.89321
```

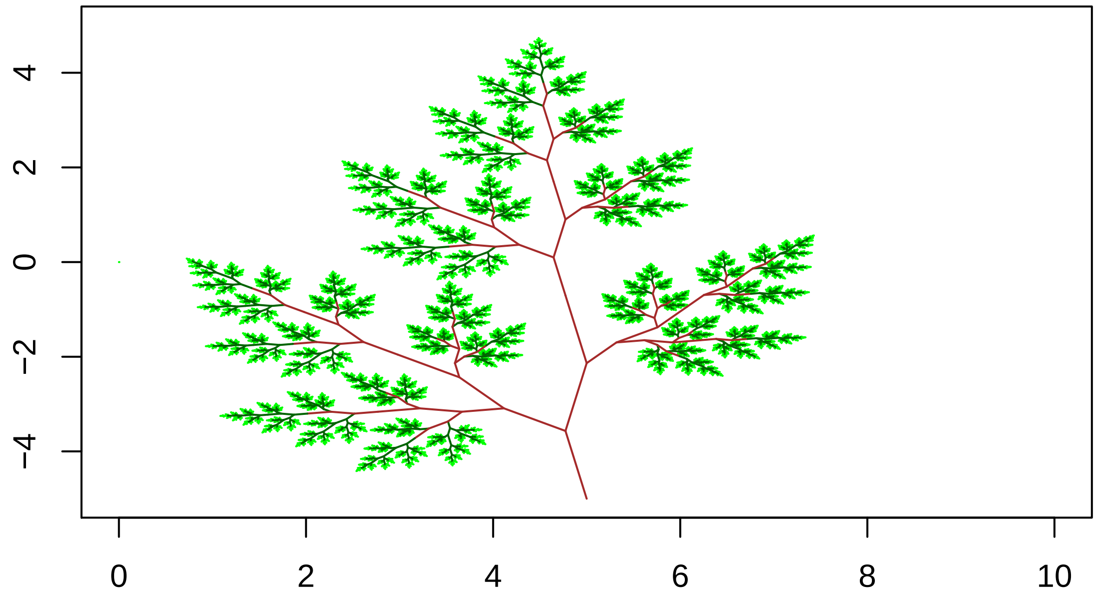
5.2 Calling functions in the body

Another robust operation is to call other functions from within the body of a function. We have done that in many functions already! For example, look back at the `summarize_col()` above: from within `summarize_col()`, we called the `length()` function. We can do the same thing with our own functions! This is a mighty operation to break down complex functions to smaller parts!

Recursive functions

A recursive function is a function that calls itself in the body during its execution. Recursive functions are commonly used to reproduce structures that repeat themselves, such as leaves, dunes, etc.

```r
# the following code plots a leaf
turtle <- function(vec,dir,l,colour){
    vec2 <- c(vec[1] + l * cos(dir), vec[2] + l * sin(dir))
    lines(c(vec[1], vec2[1]), c(vec[2], vec2[2]), type="l", col=colour)
    return(vec2)
}
# linedraw is a recursive function that calls itself
linedraw <- function(vec,direction,len,n,dir2){
    if (n >0){
        if (n >6 ){colour = "brown"}
        else{
            if (n > 2){colour = "dark green"}
            else{colour = "green"}
        }
        vec <- turtle(vec, direction, len, colour)
        linedraw(vec, direction + dir2 * pi / 4, len * 1.12 / 2, n - 1, dir2)
        vec <- turtle(vec, direction - pi / 10, len, colour)
        linedraw(vec, direction - dir2 * pi / 4, len * 1.12 / 3, n - 1, -dir2)
        vec <- turtle(vec, direction, len, colour)
        linedraw(vec, direction, len * 1.12 / 2, n - 1, dir2)
    }
}
plot(c(0, 0), c(0, 0), type = "l", col = "green", ylim = c(-5, 5), xlim = c(0,
                                   10), xlab = "", ylab = "")
linedraw(c(5, -5), pi / 2 + pi / 20, 1.45, 10, 1)
```



When we write a recursive function, the first thing to think about is the exit condition. At some point, we need the function to stop calling itself. If this condition is never met or met too late, our program will either run forever or crash.

5.3 Apply user-defined functions

In the LU6 we learnt to use the *base R function* `apply()`, `sapply()`, `tapply()` and `lapply()` to apply a function to different R objects. As is the case for almost everything, these functions are not limited to objects but can be applied to user-defined functions, too (see the example below).

```
# the function expGrowth was defined in the beginning of this lecture
# sapply applies the function expGrowth and returns a vector
sapply(c(0.9, 1.1, 1.3), FUN = expGrowth, n0 = 2, tmax = 20)
```

```
##             [,1]      [,2]       [,3]
##  [1,] 2.0000000  2.000000   2.000000
##  [2,] 1.6200000  2.420000   3.380000
##  [3,] 1.4580000  2.662000   4.394000
##  [4,] 1.3122000  2.928200   5.712200
##  [5,] 1.1809800  3.221020   7.425860
##  [6,] 1.0628820  3.543122   9.653618
##  [7,] 0.9565938  3.897434  12.549703
##  [8,] 0.8609344  4.287178  16.314614
##  [9,] 0.7748410  4.715895  21.208999
## [10,] 0.6973569  5.187485  27.571698
## [11,] 0.6276212  5.706233  35.843208
## [12,] 0.5648591  6.276857  46.596170
## [13,] 0.5083732  6.904542  60.575021
## [14,] 0.4575358  7.594997  78.747528
## [15,] 0.4117823  8.354496 102.371786
## [16,] 0.3706040  9.189946 133.083322
## [17,] 0.3335436 10.108941 173.008318
## [18,] 0.3001893 11.119835 224.910814
## [19,] 0.2701703 12.231818 292.384058
## [20,] 0.2431533 13.455000 380.099275
```

5.4 Writing stable functions

R can be used both to do interactive analysis and to do programming.

When working interactively, we want to iterate as quickly as possible and check each result as we go. When we do programming, we have to be more careful because the same code can be reused with different inputs and on different environments. We have to be careful because unexpected errors might arise.

For example, unstable type functions return different types of objects. e.g., with an input vector, they return a vector, but with an input data frame, they return a data frame. If we are working interactively, we know what input object we are passing. This might be less obvious within a long program. In programming, it's better to avoid writing functions of inconsistent type. A way to achieve that is to write error messages and stop the execution if an object is of a different type than expected.

Finally, we have to be aware that R has global options which can affect the operation of certain functions (see `options()`). These global options can affect the behavior of our function and thus cause issues when you move from one computer to another. For example, the option `stringsAsFactors = TRUE` sets R to treat strings as factors in a data frame.

6 The interactive part of the class

```r
# Basics of functions

#remember the syntax of a function
my_fun <-function(args1, args2) {
   body
}

# division function
division <- function(a, b){ # b takes a default value
   return(a / b)
}

# you can can call the function division in multiple ways:
division(1, 2)
```

```
## [1] 0.5
```

```r
division(a = 1, b = 2)
```

```
## [1] 0.5
```

```r
division(20, 0)
```

```
## [1] Inf
```

```r
division(0, 0)
```

```
## [1] NaN
```

```r
# default values for variables
division <- function(a, b = 10){ # b takes a default value
   return(a / b)
}

division(1)
```

```
## [1] 0.1
```

```r
division(1, 10) #both are the same now
```

```
## [1] 0.1
```

Logistic growth is a popular population model for populations that are limited by a carrying capacity. According to the model, the population P at time t is

$P(t) = \frac{L}{1+\exp(-kt)}$,

with L being the carrying capacity and k being the reproduction rate.

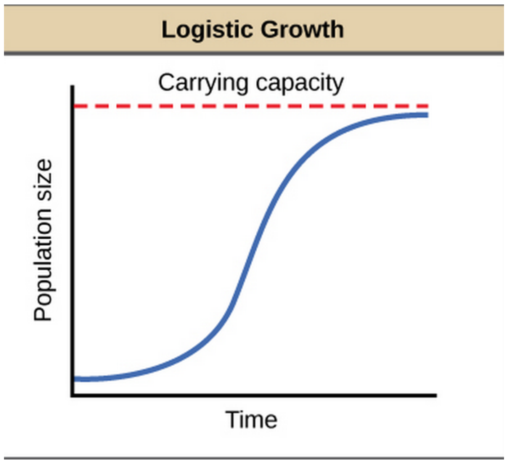
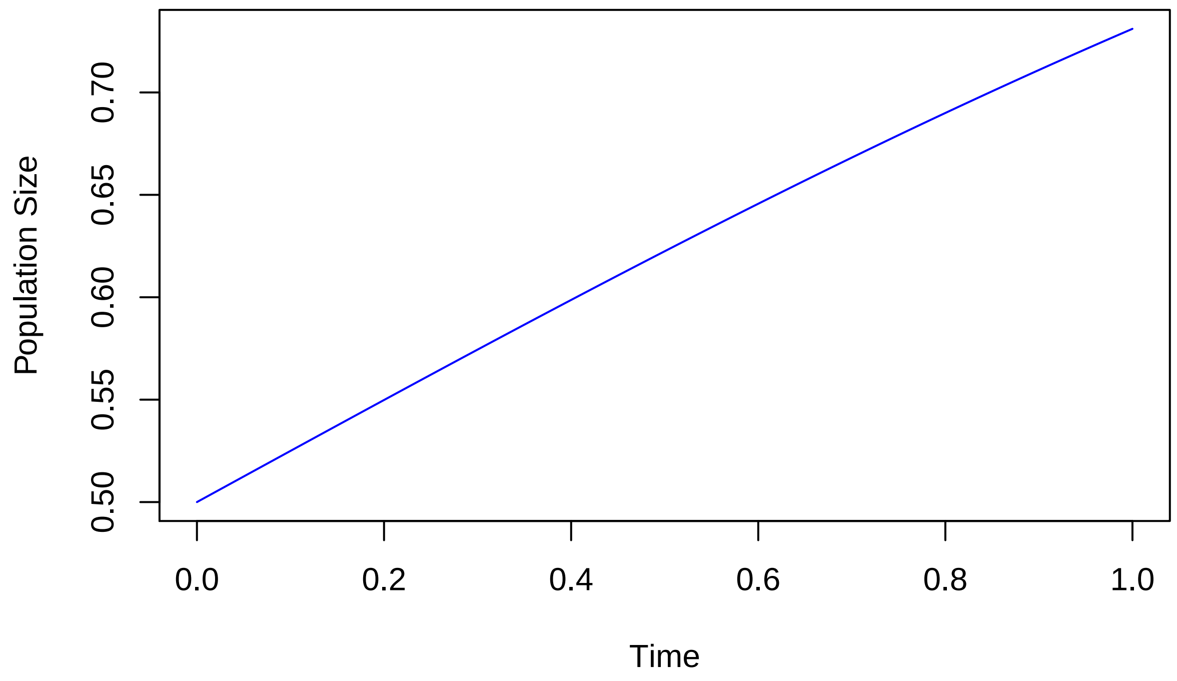


Figure 3 Depiction of logistic growth. Image credit: "Environmental limits to population growth: Figure 1," by OpenStax College, Biology

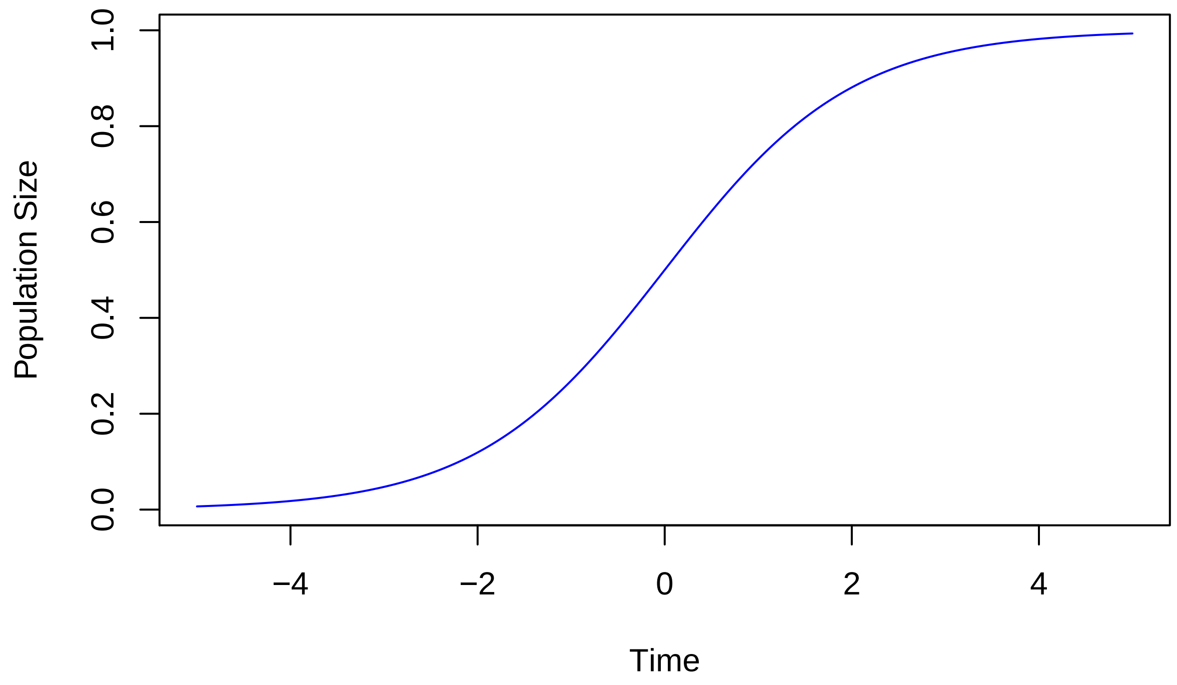
```r
# Development of functions

# discuss with the students how to approach this and do it together
logisticV1 <- function(t, L, k) {
  v <- exp(-k * t)
  y <- L/(1 + v)
  return (y)
}

# k should be positive in logistic growth
# we can ensure it is by using an if statement

logisticV2 <- function(t, L, k) {
  if (k < 0) {
    stop("error: k  has to be positive")
  } else {
  v <- exp(-k * t)
  y <- L/(1 + v)
  return (y)
  }
}

logisticV2(t = 1, L = 5, k = -1)
```

```
## Error in logisticV2(t = 1, L = 5, k = -1): error: k  has to be positive
```

```r
# we want to visualize logistic growth
# first step is to calculate the values of the function for different times t

evalLogistic <- function(steps = 100, minV = 0, maxV = 1, L, k) {
  x <- rep(0, steps)
  y <- rep(0, steps)
  for (step in 0:steps) {
    xVal <- (maxV - minV) * step/steps + minV
    x[step+1] <- xVal
    y[step+1] <- logisticV2(t = xVal, L = L, k = k)
  }
  return(list = c(x = x, y = y))
}

# calling the function gives x- and y-values of logistic growth
# we can now call the following function to plot it

plotLogistic <- function(steps = 100, minV = 0, maxV = 1, t, L, k) {
  values <- evalLogistic(steps = steps, minV = minV, maxV = maxV, L = L, k = k)
  plot(values[1:(steps+1)],values[(steps+2):((steps+1)*2)], type="l",col="blue", ylab="P
}

# using a high number of steps gives better accuracy but is a little bit slower

plotLogistic(steps = 10000, min = 0, max = 1, L = 1, k = 1)
```



```
# if we "zoom out" the shape of the function becomes visible

plotLogistic(steps = 10000, min = -5, max = 5, L = 1, k = 1)
```


Population Size
0.0
0.2
0.4
0.6
0.8
1.0
−4
−2
0
2
4
Time

7 Exercises

7.1 (Difficulty: easy)

The function **circle_fun()** calculates the circumference and the area of a circle given the value of the diameter. Run the code provided below. Modify the code to apply the function to a vector of diameters: c(1, 3.5, 0.1). Extract the values of the circumference and the area separately.

```
circle_fun <- function(diameter = 1.2){
  circumference = pi * diameter
  area = pi * (diameter / 2) ^ 2
  return(list(circumference = circumference, area = area))
}
circle_fun()
```

```
## $circumference
## [1] 3.769911
##
## $area
## [1] 1.130973
```

7.2 (Difficulty: easy)

Measuring the height of trees with a meter is very difficult. To escape the difficulties that come with climbing, the height is normally estimated from measurements that can be carried out on the ground.

7.2.1 Part 1

Complete the given function to estimate the height of a tree. Our default setting is to observe trees from a distance of 40 m. We measure the angle to the top, θ, to estimate the height.

R uses radians instead of degrees to estimate the tangent. Therefore, we need to convert the degrees into radiants by multiplying the value of the angle by π (which is called `pi` in R) and divide it by 180. This has been done for you in the exercise.

Test the function for an angle of $\theta = 37°$.

Hint: Remember that $height = distance \times tan(\theta)$.

```
# calculate the height of a tree based on the distance and the angle
# as measured by an observer on the ground. The formula is:
treeHeight <- function(angle, distance = ___){
height <-  ___ * ___ ( ___ * pi / 180 )
```

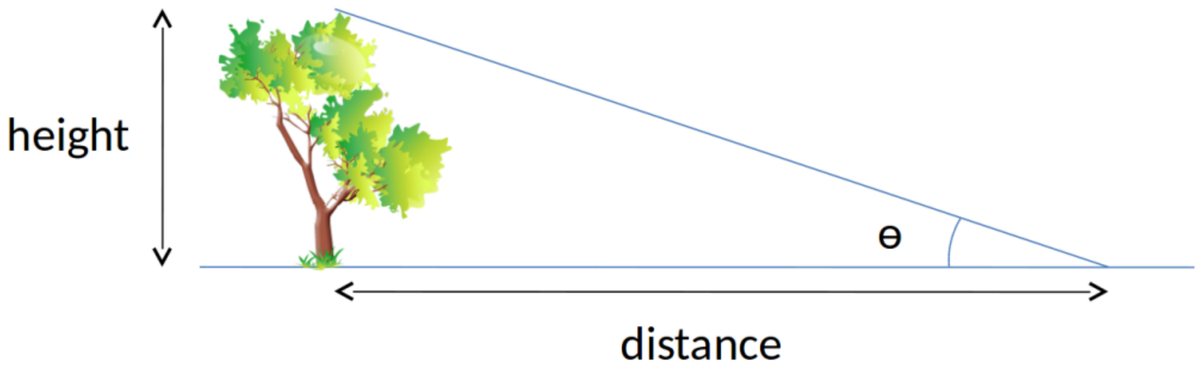


Figure 4 Representation on how to calculate the height of a tree based on observed angle and distance of the observer.

```r
return(___)
}

treeHeight()
```

7.2.2 Part 2

We have deviated from our default setting and observed a tree from a distance of 60 m with an angle of 10° to the top. Apply the `treeHeight()` function to this setting to calculate the height?

7.3 (Difficulty: easy)

7.3.1 Part 1

Complete the following function that converts measurements taken in miles to kilometers. The formula is $kilometers = miles \times 1.609$. Then apply the function to the values 2, 6, and 25 miles.

```r
# convert miles to kilometers

convertMilesToKMeters <- function(____ = 10){
  return(_____)
}
```

7.3.2 Part 2

Write a similar function that converts kilometers to miles, called **convertKMeter-sToMiles()** and apply it to the values 3, 9.6, and 40 kilometers.

7.4 (Difficulty: intermediate)

The Ricker model is a discrete population model that mimics the number of individuals in a population over multiple generations. The formula of the model is:

$$N_{t+1} = N_t \times e^{r(1-\frac{N_t}{k})}$$

where N_{t+1} and N_t are respectively the population sizes at time $t+1$ and t, k is the carrying capacity of the environment, and r is the intrinsic growth rate. The model is often used to predict the number of fish in a fishery.

```
# the functions calculates the expected population size after the n times.
# the calculation is based on the ricker model, which takes as arguments:
# r = intrinsic growth rate
# k = carrying capacity
# Nt = initial population size
ricker <- function(r, k, Nt, times){
  Nt1 <- c(Nt, rep(0, times - 1))

  for(i in 2:times){
    Nt1[i] <- Nt1[i - 1] * exp(r * (1 - (Nt1[i - 1]/k)))
  }
  return(Nt1)
}
ricker(2, 1000, 1500, 30)
```

```
##  [1] 1500.0000  551.8192 1352.3270  668.4276 1297.3420  715.7914 1263.7086
##  [8]  745.7488 1240.0304  767.2635 1222.0722  783.8038 1207.7943  797.0859
## [15] 1196.0631  808.0828 1186.1866  817.3976 1177.7138  825.4286 1170.3353
## [22]  832.4515 1163.8305  838.6647 1158.0369  844.2154 1152.8321  849.2153
## [29] 1148.1212  853.7511
```

7.4.1 Part 1

The Ricker model output might be considered unrealistic because the population size is expressed as a real number with decimal degrees (and we know that, for example, 0.3 of an individual does not exist in nature). Change the function so that the calculations will be computed in integers. The `round()` *base R function* can be used to address this weakness. Check the help file with `?round()`.

Moreover, the argument values are also bounded. For example:

- r is bigger than 0
- k, Nt and times are all integers and bigger than 0.

Introduce some checks within the function body to make sure the user set the arguments correctly and stop the execution if the conditions do not apply. Note that you can easily check if a number is an integer with the `%%` operator. Try for example to type in your console `4%%2` and `5%%2`. Test your implementation by calling the function with the same arguments as above.

7.4.2 Part 2

Improve the model further to stop the function when the population goes extinct. We know that the population goes extinct for the set of arguments: $r = 2$, $k = 10$, $Nt = 1500$, and $times = 30$. Test the improved function with these arguments.

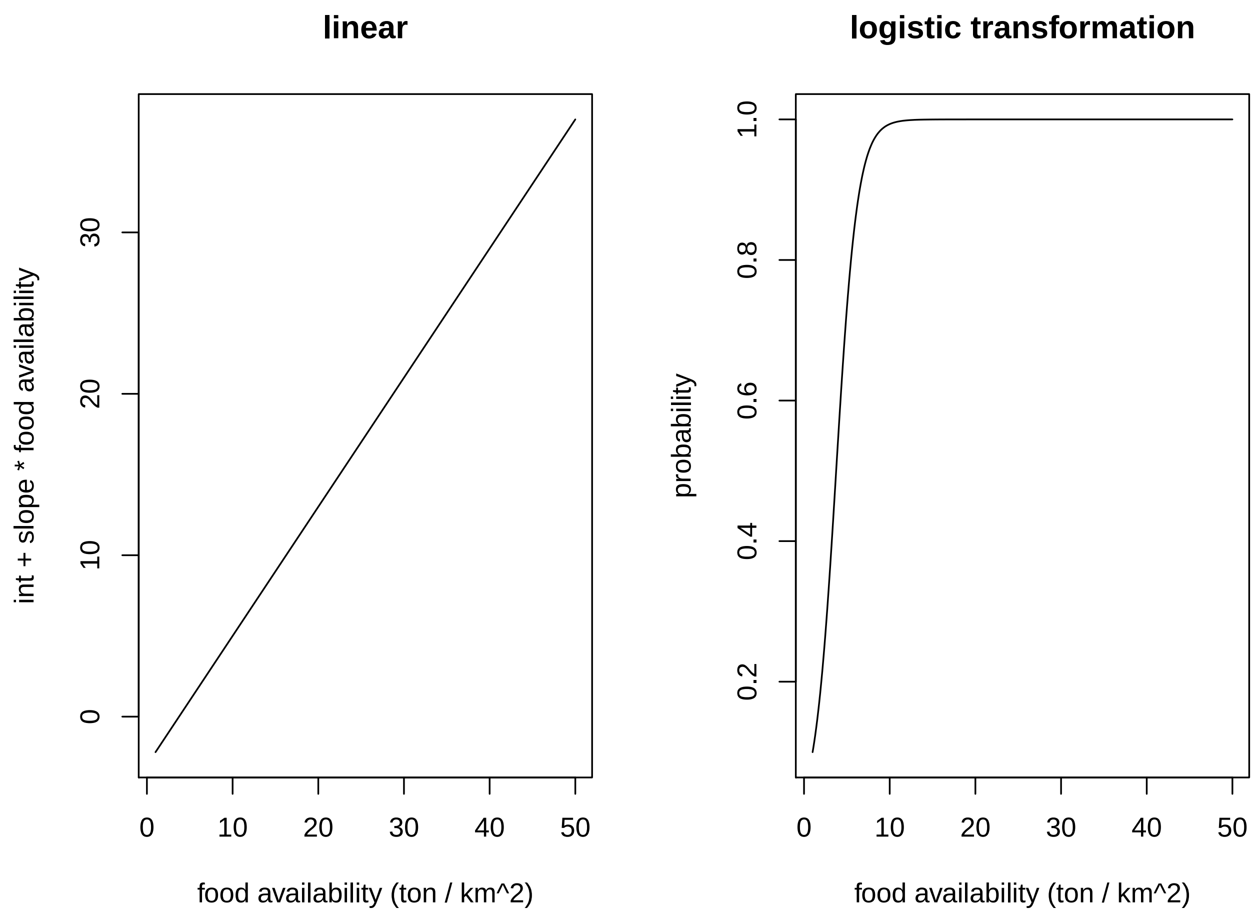
7.5 (Difficulty: intermediate)

The survival of lambs to adult is affected by the food availability (which is measured in *ton* of food per km^2) in the area where they are born. The more food is available, the higher the survival probability. In this exercise, we want to write a function which calculates the survival probability.

First, we need to decide on a model to estimate the survival probability. We know that a probability is by definition a number between 0 and 1. The logistic function is an excellent approach to model the survival probability that dependends on independent variables, such as food availability. This is because the logistic transformation converts arbitrary values to probabilities bounded between 0 and 1, exactly what we want. We want to use the logistic function to model the survival probability, such that:

$$logistic(int + slope \times food) = \frac{exp(int + slope \times food)}{1 + exp(int + slope \times food)}$$

The integer *int* and the slope *slope* are parameters of the function which we have to set based on previous knowledge. For the lambs, $int = -3$ and $slope = 0.8$ are realistic values. The figure below shows the logistic transformation of the values. On the left-hand side, we see the values of $int + slope \times food$ before the transformation and on the right-hand side the values after the transformation.



Check wikipedia if you need a refresher on the logistic function.

With all of this in mind, go ahead and write a function that calculates the survival probability of a lamb from juvenile to adult. Use an intercept of -3 and a slope for food availability of 0.8. How much does the survival probability change if we double the amount of food available from 2 ton to 4 ton?

Before writing your function, remember to plan and write down your pseudocode.

7.6 (Difficulty: hard)

We weren't aware of the most basic R functions (a big mistake to start with, but nothing we should worry about here) and have written a function which calculates either the mean and the minimum of a vector `x`. Set `meanFlag` to `TRUE` if you want to calculate the mean of `x`, or set `minFlag` to `TRUE` if you want to calculate the minimum.

```
meanMinFunction <- function(x, meanFlag = FALSE, minFlag = TRUE) {
  mean <- 0
  min <- x[1]
  if (meanFlag == TRUE) {
    for (i in 1:length(x)) {
      mean <- mean + x[i]
    }
    mean <- mean/length(x)
    return(mean)
  } else if (minFlag == TRUE) {
    for (i in 1:length(x)) {
      if (x[i] < min) {
        min <- x[i]
      }
    }
    return(min)
  }
}
```

We are not happy with our result, though. Help us out by adjusting for the following three weaknesses:

7.6.1 Part 1

When we set both flags to FALSE, we do not get anything from our function. In that case, we probably did a mistake in the function call which we obviously want to be notified of. Add a third case to the if/else statement that gives a warning when we call the function with both flags being FALSE. Test the case with a vector consisting of the values 1, 2 and 3.

7.6.2 Part 2

When we set both flags to TRUE, the function does not return the correct output. It only returns the mean, not both mean and minimum. Change the if/else structure and include a fourth case for when both flags are TRUE. Call the function with the same vector as in part 1, but this time set both flags to TRUE.

7.6.3 Part 3

Now the function does what it should do! However, other users might be confused as we did not include any comments at all which makes it hard to read and understand what is done here. Add comments to explain the four cases that are covered by the if/else structure. Then add further comments to indicate which part of code is dedicated to calculate the mean and which part is dedicated to calculate the minimum.

7.7 (Difficulty: hard)

We want to calculate the tree density in the city of Nijmegen. The density is defined as the quotient of the number of trees and the surface area. We don't have any reliable information on the surface area of Nijmegen, though. So naturally, we want to estimate it.

An intuitive way to do so is to calculate for each tree the Euclidean distance to the next one. To intuition goes as follows: If there are lots of trees, then the next tree will be very close. If there are only a few of them, then the next tree will be far away. In fact, if we had infinitely many data points, we could calculate the exact tree density from the Euclidean distances of all trees to their nearest neighbors. We don't have that but we have a lot of data points in our tree dataset.

The Euclidean distance is the length of a straight line between the trees i and j and is defined as

$d_{i,j} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$

with x and y being the coordinates of the trees The smallest distance for tree i to the next tree then is

$d_{\text{min},i} = \min(d_{i,j})$,

with index j checking for all trees except for i (as this would be the distance from a tree to itself which is 0 by definition). Finally, we want to average over all smallest distances

$d_{\text{avg, min}} = \text{mean}(d_{\text{min},i})$,

with index i running over all trees. In this exercise, we will write a function that not only calculates the mean Euclidean distance for the tree dataset but also gives meaningful output. The function will be developed over multiple exercise parts.

7.7.1 Part 1

We already prepared pseudocode for a program that calculates the average of smallest Euclidean distances in our dataset. Read the code, understand it and then color the steps the way we did in the interactive example (that is: green for actions, red for key-words and blue for loops). If you don't know how to color text, simply think about the differences between the three categories.

```
1. load the tree dataset
2. create an empty vector $d_{\text{min}}$ to store the smallest distances for all trees
3. for each tree $i$
    1. calculate the distances of tree $i$ to all trees $j$
    2. replace the distances of tree $i$ to itself with 10000 (hint: the exact value does not
       matter. What matters is that it is larger than the smallest distance)
    3. calculate the minimum of the distances using the base R function min()
    4. store the smallest distance in $d_{\text{min}}$
```

4. calculate $d_{\text{avg, min}}$ from d_{min} using the base R function mean()

7.7.2 Part 2

We translated our pseudocode into a R script that calculates the average of smallest Euclidean distances. Read the script and indicate which steps of the pseudocode correspond to which line of the R code.

```
treeDataset <- read.csv(file = "additional_material/Nijmegen_trees.csv")[1:1000,]
d_min <- replicate(nrow(treeDataset), 0)

x <- treeDataset$x
y <- treeDataset$y

for (i in 1:nrow(treeDataset)) {
  d <- sqrt((x[i] - x) * (x[i] - x) + (y[i] - y) * (y[i] - y))
  d[i] <- 10000
  d_min[i] <- min(d)
}

d_min_avg <- mean(d_min)
print(d_min_avg)
```

```
## [1] 20.65624
```

7.7.3 Part 3

We want to take the next step and analyse the tree density for different sub-parts of Nijmegen. To do so we can either copy and paste our script, but that would get ugly pretty quickly. To avoid that, we want to write a function.

As the input of the function, use the x and y coordinates. As output, return the average of smallest Euclidean distances. To test the function, apply it to the tree dataset. The result should be the same as in part 2.

7.7.4 Part 4

As the last step, we want to adopt our function to give meaningful output. After all, a single number without any context does not mean much. To change this, we want to add a print message which connects: average of smallest Euclidean distances, the unit of measurement (meters), and the neighborhood of the city it is calculated for. Then, we can quickly compare the results of different neighborhoods of Nijmegen.

Adopt the function and apply it to at least three different parts of Nijmegen (which are saved in the variable $wijk$) How substantial are the differences? Do they meet your expectations?

LU 8: Introduction to Simulations in R

With exercises

Luca Santini

l.santini@science.ru.nl

Contents

1 Learning Unit Goal:

Learn how to generate random values, sample from objects, solve problems using a simulation.

2 Functions we are going to use

- 'r' functions for the normal, binomial, poisson and uniform distributions
- 'd' functions for the normal, binomial, poisson and uniform distributions

- ‘p’ functions for the normal, binomial, poisson and uniform distributions
- sample()
- set.seed()
- for loops
- plotting functions

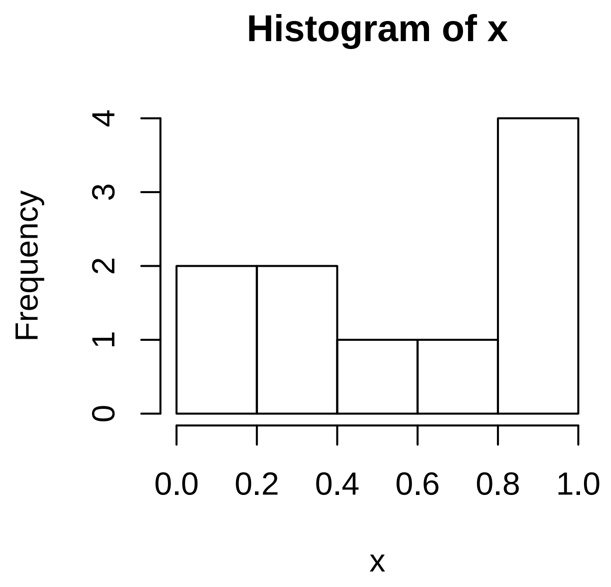
3 The basics

- A simulation is the use of a computer to represent the dynamic of a system.
- Simulations are used to study the dynamic behavior of objects or systems in response to conditions that cannot be easily or safely applied in real life.
- Simulations generally consists of algorithms with random variables
- Basic things to learn to develop simulations are: random number generations, reproducibility, statistical distributions, and algorithms.

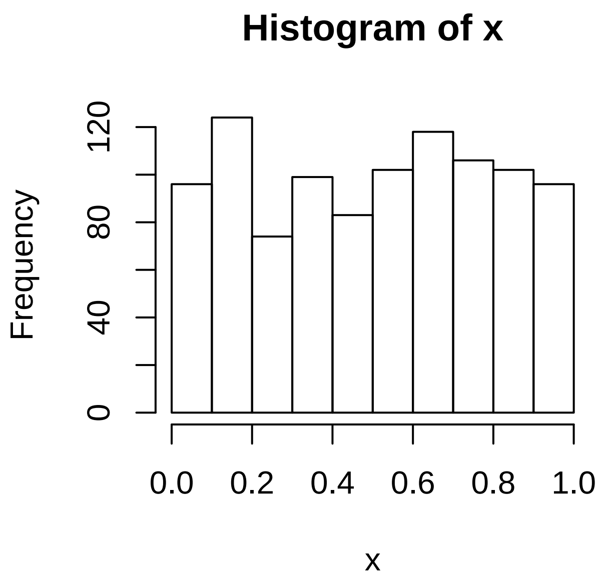
4 Let’s learn the basic functions

4.1 Simulate random values

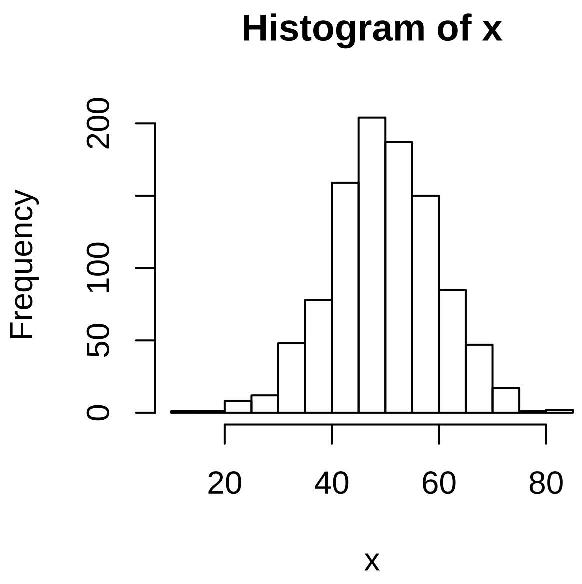
```r
#uniform distribution
#we only sample 10 values, the distribution doesn't look really uniform
x<-runif(n=10, min=0, max=1)
hist(x)
```



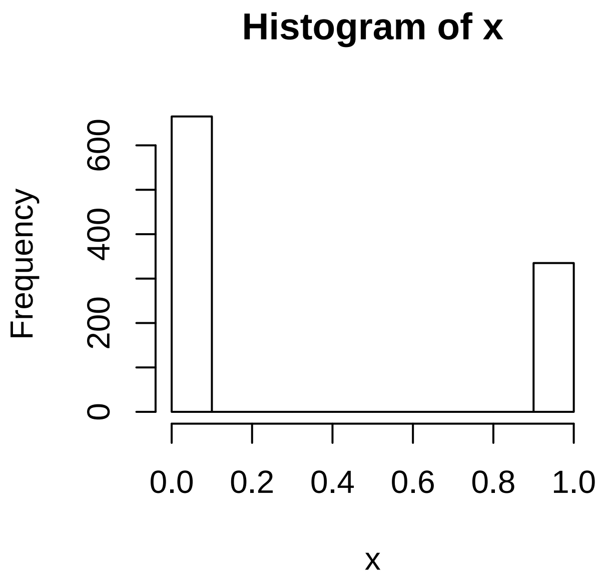
```
#if we increase the sample the distribution looks better
x<-runif(n=1000, min=0, max=1)
hist(x)
```



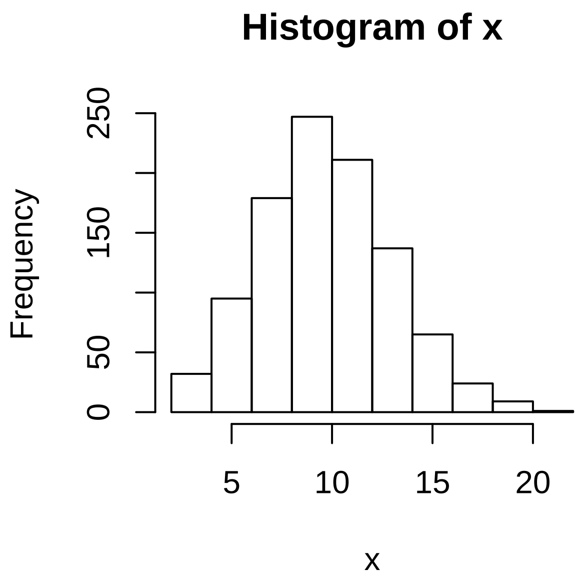
```
#normal distribution
x<-rnorm(n=1000, mean=50, sd=10)
hist(x)
```



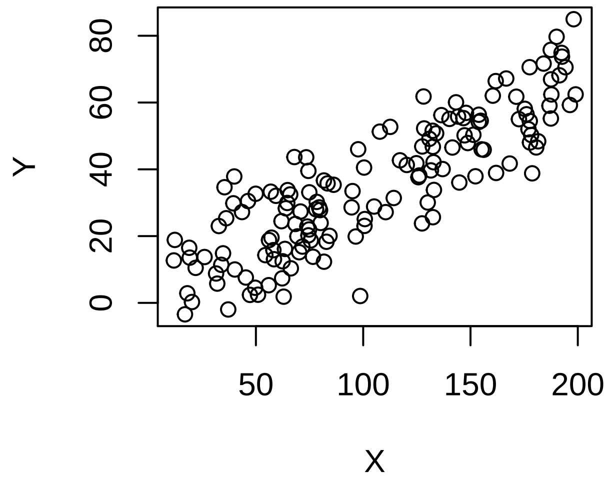
```
#binomial distribution
x<-rbinom(n=1000, size=1, prob=0.3)
hist(x)
```



```
#poisson distribution
x<-rpois(n=1000, lambda=10)
hist(x)
```



```
#Generate random variables
X<-runif(150, 10, 200) #we generate a random variable
#we use the linear model formula a+x*b to generate a second variable that depends on t
#and add an error to the relationship sampling from a normal distribution
Y<- 5 + X * 0.3 + rnorm(length(X), 0, 10)
plot(X, Y)
```



4.2 p functions

Prefixing your function with p calls up the cumulative distribution function (CDF).

Given a number or list of numbers, the CDF function allows us to compute the probability that a random number will be less that that number

```
#What is the probability that a random number from a normal distribution with
#mean 0 and sd 1 is less than 0?
pnorm(95, mean=100, sd=5)
```

```
## [1] 0.1586553
```

```
#We can manually test if this is true by sampling from the distribution
sampledValues<-rnorm(10000, mean=100, sd=5)
table(sampledValues<95)/10000
```

```
##
##  FALSE   TRUE
## 0.8389 0.1611
```

```
#We can set the argument lower.tail=FALSE to obtain the opposite
#(probability of a random number to be larger than)
pnorm(95, 100, 5, lower.tail = FALSE)
```

```
## [1] 0.8413447
```

```
#We have an exam with 10 questions, each having 5 possible answers.
#We know nothing and answer randomly.
#What is the probability that at least 3 questions are answered correctly?
pbinom(3, size=10, p=1/5, lower.tail=FALSE)
```

```
## [1] 0.1208739
```

```
#On average, 5 badgers cross a road every day.
#What is the probability of having 2 or less badgers crossing the road in
#a particular day?
dpois(2, lambda=5)
```

```
## [1] 0.08422434
```

4.3 d functions

The d command is used to calculate the probability density at given point from the probability density function (PDF). In a PDF all probabilities sum up to one.

```
#We have an exam with 10 questions, each having 5 possible answers.
#We know nothing and answer randomly. What is the probability that only 2 are
#answered correctly?
dbinom(2, size=10, prob=1/5)
```

```
## [1] 0.3019899
```

```
#and 6...?
dbinom(6, size=10, prob=1/5)
```

```
## [1] 0.005505024
```

```
#On average, 5 badgers cross a road every day.
#What is the probability of having exactly 5 badgers crossing the road in
#a particular day?
ppois(5, lambda=5)
```

```
## [1] 0.6159607
```

4.4 Sample random values from a vector

```
#Sample 10 random letters of the english alphabet
v<-sample(x=letters, size=10) #letters is an inbuilt vector of letters in R
v
```

```
##  [1] "z" "s" "o" "v" "u" "q" "k" "g" "b" "e"
```

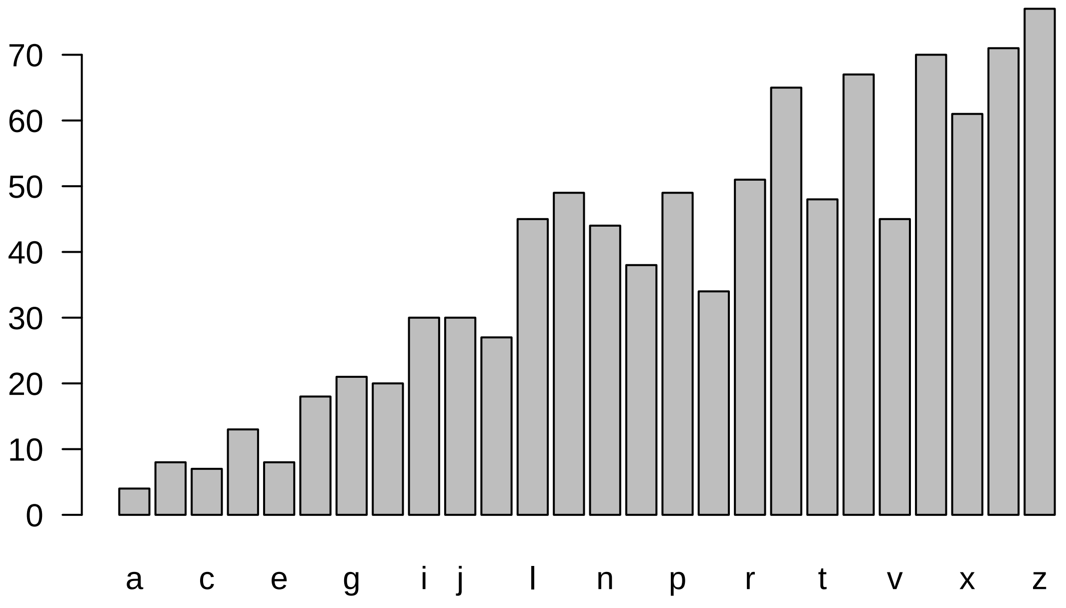
```
#Once we sample one letter we cannot resample the same letter a second time (unless
#specified)
#so if we sample more than the size of a vector it will give us an error
v<-sample(x=letters, size=50)
```

```
## Error in sample.int(length(x), size, replace, prob): cannot take a sample larger than
```

```
#To be able to resample the same element multiple times we have to use
#the argument "replace" (which by default is FALSE)
v<-sample(x=letters, size=50, replace=TRUE)
v
```

```
##  [1] "q" "e" "p" "o" "j" "j" "r" "i" "m" "r" "f" "u" "m" "x" "b" "q" "v"
## [18] "w" "i" "m" "r" "q" "u" "s" "u" "a" "g" "u" "c" "w" "b" "j" "r" "q"
## [35] "k" "y" "a" "c" "v" "s" "z" "n" "j" "o" "h" "j" "k" "c" "o" "a"
```

```
#Sample with uneven probabilities
#Here the letters at the end of the alphabet have higher probabilities to
#be resampled
v<-sample(x=letters, size=1000, replace=TRUE, prob=1:length(letters))
barplot(table(sort(v)), las=1)
```



4.5 Reshuffle a vector

```
#We can use the sample function also to reshuffle a vector
#for example, we can reshuffle of a vector from 1 to 10
sample(x=1:10, size=10)
```

```
##  [1]  3  8  4  5  1 10  9  7  2  6
```

```
#or we can reshuffle the vector of letters
sample(x=letters, size=length(letters))
```

```
##  [1] "y" "q" "w" "e" "l" "o" "r" "n" "i" "h" "j" "d" "f" "v" "t" "x" "b"
## [18] "z" "p" "u" "k" "g" "a" "s" "m" "c"
```

4.6 Reproducibility

The random values that R generates look random but are not. Pseudo-randomness in R is achieved by taking in a seed value, which is calculated as a combination of values (e.g. time of the day, space left on the hard disk, etc.). So to reproduce the output of a function that samples randomly, we can set a "seed". Note that set.seed() function only works once, i.e. if you re-run the function that follows a second time you need to set a new seed.

```
for (i in 1:10) {
set.seed(10) #any seed is fine, you just get different random values
v<-sample(x=letters, size=3)
print(v)
}
```

```
## [1] "n" "h" "k"
## [1] "n" "h" "k"
## [1] "n" "h" "k"
## [1] "n" "h" "k"
## [1] "n" "h" "k"
## [1] "n" "h" "k"
## [1] "n" "h" "k"
## [1] "n" "h" "k"
## [1] "n" "h" "k"
## [1] "n" "h" "k"
```

5 Now let's practice what we learnt

5.1 Estimate the probability of an event occurring using a simulation

Difficulty level = Basic

1) What is the probability of rolling a 7 with two 6-sided dices?
2) What is the probability that if I draw two dices (1 with 10 sides and the other with 5 sides) I get a 10 and a 5 respectively? Solve it using a simulation.

5.2 Create a function that generates a random password

Difficulty level = Medium

Instructions:
The function takes three arguments: the number of characters (nChar), capital letters (nNumbers), and numbers that the password must contain (nCapital)

Hints:
- R has an inbuilt vector called letters, containing all letters of the alphabet
- toupper() function convert a lower case character to upper case
- using the argument "collapse" in the paste() function you can collapse a vector of elements to a character string

5.3 Estimate the distribution of possible population size of a species given a few information

Difficulty level = Medium-Difficult

We have studied the territory size and pack size of wolf population for several years. Question: Given the data we have collected, what is the distribution of possible population sizes within a given area? (estimate the median and 25th and 75th quantiles of the distribution)

Data:
- We estimated the territory size (in km2) for 8 packs: 82, 150, 120, 270, 200, 100, 95, 240
- We were able to count the number of individuals in 5 packs: 4, 5, 3, 6, 7
- We know that on average 10-15% of the population consist in solitary individuals
- The total habitat area available is 1000 km2

Hints:
- The population size can be estimated as (HabitatArea /TerritorySize) * PackSize + solitary individuals
- build a simulation that samples the variables randomly to calculate the population size
- replicate the simulation 1000 times to get 1000 population size values
- the 'median' function allow you to calculate the median of a vector
- the 'quantile' function allow you to calculate the quantiles of a vector
(Quantiles are cut points dividing the range of a probability distribution into continuous intervals with equal probabilities. The median of a distribution corresponds to the 0.5 quantile)

5.4 Simulate the population growth and fluctuations with a given level of exploitation and plot the temporal trend

Difficulty level = Difficult

Instructions: - Simulate the population for 100 years (100 discrete time steps, which assumes the population changes happens at distinct and separate points in time)
- The population starts with 100 individuals
- The growth rate (r) is a random value sampled from a normal distribution with a mean of 0.2 and a standard deviation of 0.1
- The population growth follows a continuous logistic model, i.e. Population * exp(r * (1 - Population/K))

- The carrying capacity (K) is 500 individuals
- Every year 10 individuals are removed from the population
- Plot the temporal trend plot(X=time, Y=population size)
- Set a seed so that it always produces the same result

5.5 The virtual ecologist approach

Difficulty level = Advanced

Question:
How many nights should I leave the traps in the field to be sure (95% confidence) that I catch at least 100 individuals of a rodent species?

Information:
- We have 100 traps
- There are 10 rodent species in the area, but we are only interested in one
- The probability of catching one individual with 1 trap is 0.3 per trapping night.
- All rodent species have the same probability of being trapped.

Instructions:
- Simulate the population for 100 years
- The population starts with 100 individuals
- The growth rate (r) follows a normal distribution with a mean of 0.2 and a standard deviation of 0.1
- The population growth follows a continuous logistic model, i.e. Population * exp(r * (1 - Population/K))
- The carrying capacity (K) is 500 individuals
- Every year 10 individuals are removed from the population

Learning unit 9 - Introduction to tidyverse

With exercises

Mirza Čengić

Contents

1 Learning unit goals

After this learning unit is completed, you should be able to:

- Understand what are R packages, and why are they useful
- Understand what is tidyverse, and how shared design philosophy behind it is useful
- Perform the most common data wrangling operation with tidyverse packages
- Organize and summarize your data to gain new insight
- Create effective visualizations with the ggplot2 package

2 What are R packages

Packages in R are the fundamental units of reproducible code, which can often dramatically expand the usefulness of R. An R package is a collection of functions, data, and documentation that extends the capabilities of base R. In this document we will refer to the functions that you get with a fresh installation of R as *base R functions*, while other functions come from one of many additional packages. There are almost 15 000 packages available on the official R package repository - CRAN (Comprehensive R Archive Network). Besides CRAN, you can find many packages on other repositories, such as Bioconductor or GitHub.

2.1 Where to find R packages

There is a good chance that a package already exists for a problem that you are trying to solve. However, due to thousands of various packages, it is sometimes difficult to find an exact package that fits our needs. Some of the useful tools to find R packages are:

- CRAN task views gives an overview of packages per topic.
- awesome-r.com gives an overview per topic and recommends packages and learning resources.
- Bioconductor has many useful packages for bioinformatics.
- GitHub is a website where code developers and programmers share their code. There are many useful packages here under the different stage of development.
- Learn more about what is git and github from R user perspective.

2.2 How to install and load R package

Before we can use a package, we will need to install it first. The packages are usually installed from online repositories, and to install a package from CRAN, we use the function `install.packages()`. We only need to install a package once (unless we need a different version of it), and next time we need it, we can just load it with the function `library()`. If we want to install a package from GitHub for example, then we need to use package `devtools` that has function `install_github()`.

```r
# Before you can use a package, we have to install it first.
# Package name here has to be in quotes
install.packages("xx")


# To load the package, use the function library()
# Package name can be written with or without quotes here.
library(xx)
```

3 What is tidyverse?

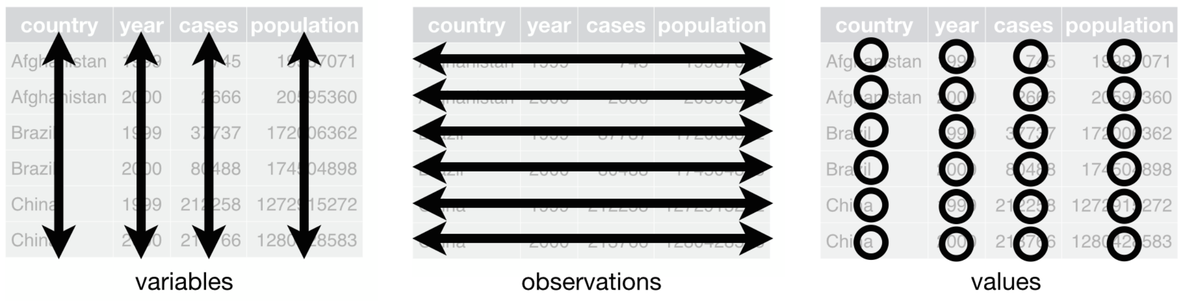
Package `tidyverse` is a collection of multiple R packages that are designed to facilitate working with data in R. All packages contained within it share underlying design philosophy. They and are designed to work together naturally and to make the process of learning new packages easier. There are four basic principles that tidyverse packages use:

- Designed for humans.
- Reuse existing data structures.
- Compose simple functions with the pipe.
- Embrace functional programming.

3.1 What is tidy data?

Tidy data framework makes it easier to clean datasets, since only a small set of tools are needed to deal with a wide range of messy datasets. The organization of tidy data goes as following:

- Each variable forms a column.
- Each observation forms a row.
- Each value has its own cell.



Many tools in the `tidyverse` expect data to be formatted as a tidy dataset. For more information you can read the tidy data paper (Wickham 2014.).

3.2 Packages in the tidyverse meta-package

```r
# If tidyverse package is missing, uncomment the line below and
# install tidyverse.
# RStudio shortcut to comment/uncomment lines - Ctrl + Shift + c

# install.packages("tidyverse")
library(tidyverse)
```

```
## -- Attaching packages ---------------------------------------------------------------------

## √ ggplot2 3.0.0     √ purrr   0.2.5
## √ tibble  1.4.2     √ dplyr   0.7.6
## √ tidyr   0.8.1     √ stringr 1.3.1
## √ readr   1.1.1     √ forcats 0.3.0

## -- Conflicts ------------------------------------------------------------------ tid
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```r
# After tidyverse is loaded for the first time, the message with
# loaded packages and possible conflicts (functions from other packages
# with the same name) will be displayed.

# You can use the function ls() to list the objects you currently have in
# your environment, or you can use it in this way to see
# all available functions from a specific package.

# Let's see what functions are available in tidyverse
ls("package:tidyverse")
```

```
## [1] "tidyverse_conflicts" "tidyverse_deps"      "tidyverse_logo"
## [4] "tidyverse_packages"  "tidyverse_update"
```

As we can see from the output of the `ls()` function above, there are only 5 functions that come with tidyverse. Tidyverse is essentially a quick way to load multiple packages that target solving problems using a shared approach, and we can consider it a meta-package, or a package whose purpose is to load other packages.

```r
# Since tidyverse is just a collection of other packages, there aren't
# many functions in tidyverse itself. However, we can check what packages
# come with tidyverse.
tidyverse_packages()
```

```
##  [1] "broom"         "cli"           "crayon"        "dplyr"         "dbplyr"
##  [6] "forcats"       "ggplot2"       "haven"         "hms"           "httr"
## [11] "jsonlite"      "lubridate"     "magrittr"      "modelr"        "purrr"
## [16] "readr"         "readxl\n(>=" "reprex"        "rlang"         "rstudioapi"
## [21] "rvest"         "stringr"       "tibble"        "tidyr"         "xml2"
## [26] "tidyverse"
```

Right now there are 26 packages contained in tidyverse, and some of them are created to solve different data-related problems. Some of the packages we will use today are:

- `readr` contains functions for efficient reading and writing of data.
- `stringr` is useful for working with strings (characters).
- `dplyr` provides a powerful set of tools for data analysis and data manipulation. `dplyr` is designed with local data in mind. If you need to work with remote databases, a good alternative is `dbplyr`.
- `tidyr` offers tools that help you create tidy data from messy data.
- `ggplot2` is a powerful tool for making data visualizations.
- `magrittr` is included for its pipe operators (most common one is `%>%`), that can be used to chain the operations together.

Other packages that we will not cover today, but that are very useful:

- `lubridate` is useful for working with dates and time-series.
- `purrr` enhances R's functional programming capabilities, and adds to the *apply family of functions, enabling you to replace many for-loops and write more expressive code that is easier to read.
- `modelr` package provides functions for creating modeling pipelines, and it is designed primarily to support teaching the basics of modeling within the tidyverse. There is a set modeling-oriented tools being developed that follow tidyverse design philosophy called `tidymodels`.

For more information, you can visit the `tidyverse` website.

4 Working with `tidyverse`

Let's load Nijmegen trees dataset, and see what's in there. Download the materials from Brightspace, and open the `LU9_tidyverse.RProj` file from the main folder. This is RStudio project file that helps you organize your work better, and it will already take care of the working directory path for you (check this with `getwd()`).

```
# You can use read_csv() instead of read.csv() to load Nijmegen trees data.
# tidyverse::read_csv() is a safer and faster alternative to the
# base R utils::read.csv().

nijmegen_trees <- read_csv("data/nijmegen_trees.csv")
```

```
## Parsed with column specification:
## cols(
##   postcode_nummer = col_integer(),
##   wijk = col_character(),
##   area = col_double(),
##   BOOMSOORT = col_character(),
##   PLANTJAAR = col_integer(),
##   ID = col_double(),
##   x = col_double(),
##   y = col_double()
## )
```

```
# Explore data
head(nijmegen_trees)
```

```
## # A tibble: 6 x 8
##   postcode_nummer wijk    area BOOMSOORT     PLANTJAAR    ID      x      y
##             <int> <chr>  <dbl> <chr>             <int> <dbl>  <dbl>  <dbl>
## 1            6524 Galge~  1.00 Aesculus hip~      1930   256 1.88e5 4.27e5
## 2            6524 Galge~  1.00 Aesculus hip~      1930   257 1.88e5 4.27e5
## 3            6524 Galge~  1.00 Aesculus hip~      1930   258 1.88e5 4.27e5
## 4            6524 Galge~  1.00 Aesculus hip~      1950   259 1.88e5 4.27e5
## 5            6524 Galge~  1.00 Aesculus hip~      1960   260 1.88e5 4.27e5
## 6            6524 Galge~  1.00 Aesculus hip~      1950   261 1.88e5 4.27e5
```

```
# You can use glimpse() to see the structure of your dataframe and
# the data in it.
# It's basically a combination of str() and head()
glimpse(nijmegen_trees)
```

```
## Observations: 61,992
## Variables: 8
## $ postcode_nummer <int> 6524, 6524, 6524, 6524, 6524, 6524, 6524, 6524...
## $ wijk            <chr> "Galgenveld", "Galgenveld", "Galgenveld", "Gal...
## $ area            <dbl> 1.000916, 1.000916, 1.000916, 1.000916, 1.0009...
## $ BOOMSOORT       <chr> "Aesculus hippocastanum Baumannii", "Aesculus ...
## $ PLANTJAAR       <int> 1930, 1930, 1930, 1950, 1960, 1950, 1950, 1930...
## $ ID              <dbl> 256, 257, 258, 259, 260, 261, 262, 263, 264, 2...
## $ x               <dbl> 188114.5, 188064.3, 188017.8, 188031.7, 188204...
## $ y               <dbl> 427420.7, 427415.6, 427410.9, 427311.4, 427418...
```

```
# read_csv function will return a tibble. Tibble is essentially a dataframe,
# whose most obvious advantage is how it is printed in the console.

nijmegen_trees
```

```
## # A tibble: 61,992 x 8
##    postcode_nummer wijk    area BOOMSOORT    PLANTJAAR    ID      x      y
##              <int> <chr>  <dbl> <chr>            <int> <dbl>  <dbl>  <dbl>
##  1            6524 Galge~  1.00 Aesculus hi~      1930   256 1.88e5 4.27e5
##  2            6524 Galge~  1.00 Aesculus hi~      1930   257 1.88e5 4.27e5
##  3            6524 Galge~  1.00 Aesculus hi~      1930   258 1.88e5 4.27e5
##  4            6524 Galge~  1.00 Aesculus hi~      1950   259 1.88e5 4.27e5
##  5            6524 Galge~  1.00 Aesculus hi~      1960   260 1.88e5 4.27e5
##  6            6524 Galge~  1.00 Aesculus hi~      1950   261 1.88e5 4.27e5
##  7            6524 Galge~  1.00 Aesculus hi~      1950   262 1.88e5 4.27e5
##  8            6524 Galge~  1.00 Aesculus hi~      1930   263 1.88e5 4.27e5
##  9            6524 Galge~  1.00 Aesculus hi~      1930   264 1.88e5 4.27e5
## 10            6524 Galge~  1.00 Aesculus hi~      1930   265 1.88e5 4.27e5
## # ... with 61,982 more rows
```

Functions in tidyverse often have their role in the data analysis pipeline in a same way that words play a role in sentences. For this reason, you will often see functions that perform an action referred to as *verbs*. In this section you will learn about some key `tidyverse` functions/verbs with which you can solve the majority of your data manipulation tasks. For more information on data analysis with `tidyverse`, you can refer the free book R for Data Science.

- Choose observations (rows) by their values with `filter()`.
- Choose variables (columns) by their names with `select()`.
- Create new variables using functions and existing variables with `mutate()` and `transmute()`.
- Summarize many observations to a single value with `group_by()` and `summarize()`.
- Reorder the dataframe with `arrange()`.

These functions provide the main verbs for the language of data manipulation. These functions can be used individually, or you can chain them together to perform a complex analysis, which we will try later. They have an expected input and output data type, which is most commonly a dataframe/tibble.

4.1 filter() and select()

Functions `filter()` and `select()` come from the `dplyr` package, and are used to filter specific observations (rows), or to select or remove specific variables (columns). Remember, you **filter rows**, and **select columns**.

With `filter()` you can subset observations according to their values. For example, we can filter all rows from `nijmegen_trees` dataset for given species, or we can filter rows to see which trees were planted in a certain year. The second and subsequent arguments of `filter()` are the expressions that filter the data frame.

With base R you may do something like this:

```r
# Filtering rows where the variable PLANTJAAR is equal to 2019.
# In other words, return a dataset containing trees planted in 2019.
nijmegen_trees[nijmegen_trees$PLANTJAAR == 2019, ]
```

The square brackets `[` syntax (syntax is the way the code is written) to subset specific rows or columns is not very intuitive, especially if you haven't programmed before. After you finish this lesson, you can judge for yourself what will be easier to remember.

```r
# Remember to use equals operator ==, not argument assignment operator =.
filter(nijmegen_trees, PLANTJAAR == 2019)

# Filter species with NA values in the variable BOOMSOORT
filter(nijmegen_trees, is.na(BOOMSOORT))

# You can also use functions that return TRUE/FALSE. Here we will
# find pine species in Nijmegen, using the function str_detect()
# from tidyverse package stringr.
filter(nijmegen_trees, str_detect(BOOMSOORT, "Pinus"))

# Remembering how to do this with base R usually takes some time, because
# the arguments of grep function (equivalent of str_detect) are inverted.
nijmegen_trees[grep("Pinus", nijmegen_trees$BOOMSOORT), ]
```

To use `filter()` function effectively, you can use any of the value comparison operators provided in base R, such as: `>`, `>=`, `<`, `<=`, `!=` (not equal), `==` (equal), or `%in%` (matches) when you want to match multiple values. You can also expand this using logical operators to get multiple conditions (i.e. `filter(value > 8 & value < 16)`). You can also use functions that return TRUE or FALSE to filter the values (such as `str_detect()` function). Notice that you do not necessarily have to use quotes to refer to the columns when using tidyverse function, because these functions use what is called lazy-evaluation.

With `select()` function, you can choose which columns to retain or remove from your dataframe, to get to the variables you are interested in. This is can be particularly useful when working with datasets with many variables. In this case `select()` may not be that useful, but you can get the idea how to use it.

```r
## Selecting columns
## Select columns BOOMSOORT and wijk using base R approach

nijmegen_trees[, c("BOOMSOORT", "wijk")]

## Select a single column with select()
select(nijmegen_trees, BOOMSOORT)

# You can separate column names with comma, or use c()
# Select variables BOOMSOORT and wijk from the dataset
```

```r
select(nijmegen_trees, BOOMSOORT, wijk)
select(nijmegen_trees, c(BOOMSOORT, wijk))

# If we do not want to keep columns, but remove them instead,
# we can do that with the "-" sign

# Remove columns BOOMSOORT and wijk from the dataset
select(nijmegen_trees, -BOOMSOORT, -wijk)
select(nijmegen_trees, -c(BOOMSOORT, wijk))

# Check the function help with ?select for examples of helper functions
# In this case, take all columns that start with a letter "p"
# You can make it case sensitive with the argument ignore.case
select(nijmegen_trees, starts_with("p"))
```

4.2 Chaining operations together with pipes

You will likely run into situations when you need to write many functions to perform a task. During these types of operations, we can either decide to use temporary variable name, or embed the functions within each other (eg. `filter(select(rename()))`), or if we often need to run multiple functions on a dataframe to perform a specific operation we can introduce the concept of chaining the operations together to create a dataframe. To perform this, we can use the pipe operator from the magritrr package `%>%`. Use **Ctrl + Shift + M** keyboard shortcut in RStudio to insert pipe.

The pipe operator will take the whatever is on the left-hand side (LHS), and send it to the function on the right-hand side (RHS). LHS is interpreted as a first argument of the RHS function. If pipes don't "click" with you initially, don't give up and try to understand how pipes can be useful, since piping functions and chaining operations together will be a very useful addition to your toolbox. Piping also makes code more readable for humans, since it follows a logical order of operations. You can think about the pipe `%>%` as "and then".

Let's see how can we write the following sequence of events as R code:

- First **eat breakfast**, and then **bike to the University**, and then **study for an hour**, and then **drink some coffee**.

```r
# This is how you may do it using "temporary" variable
morning_routine_breakfast <- eat(meal = "breakfast")
morning_routine_travel <- bike(morning_routine_breakfast, destination = "university")
morning_routine_study <- study(morning_routine_travel, length = "60")
morning_routine <- drink(morning_routine_study, type = "coffee")

# More realistic
morning_routine <- eat(meal = "breakfast")
```

```
morning_routine1 <- bike(morning_routine_breakfast, destination = "university")
morning_routine2 <- study(morning_routine_travel, length = "60")
morning_routine_final <- drink(morning_routine_study, type = "coffee")


# Or we can nest functions inside out in a single expression. It can be
# difficult to track parentheses or argument positions.
# Since it cannot be fitted on a single line, we will break it across lines.
morning_routine <- drink(study(bike(eat(meal = "breakfast"),
                                    destination = "university"),
                               length = "60"), type = "coffee")

# With pipes.
# The code is easier to read, and writing this code is more fluid
# since it follows human language logic.

morning_routine <- eat(meal = "breakfast") %>%
  bike(destination = "university") %>%
  study(length = "60") %>%
  drink(type = "coffee")
```

Let's start examining this dataset. First we see that it contains data on tree species in Nijmegen, along with the species name, year when it was planted, postcode, and neighborhood in which it is located. There are also x and y coordinates that can be used to convert this dataframe to a spatial points dataframe, however we will not do this today.

```
# You can either do this:
head(nijmegen_trees_cleaned2)

# Or with pipes, you can do it this way:
nijmegen_trees_cleaned2 %>% head()

# NOTE!
# Use Ctrl + Shift + M keyboard shortcut in RStudio to insert pipe.

# Let's change the second argument; this will print out first 10 rows
# of the object, instead of 6, which is the default for head():
head(nijmegen_trees_cleaned2, 10)

# When we use pipes, the piped object ('nijmegen_trees_cleaned2')
# is assumed to be in the first argument place of the function
# that follows, and the second argument (n for function head())
# can be then written in the first place, like this:
nijmegen_trees_cleaned2 %>% head(10)
# This is the reason why all functions in tidyverse have the data object
# as its first argument.


# Let's repeat the example from above using pipes. Let's pay attention to
# writing clean and readable code.
# rename() function is useful to rename columns.
# Left hand side is the new column name, right hand side is the old column
# name, i.e. rename(x, new_name = old_name).
# We will also remove the trees that have 0 value for the year of planting,
# since this is most likely incorrect data input.

nijmegen_trees %>%
  rename(
    postcode = postcode_nummer,
    species = BOOMSOORT,
    year = PLANTJAAR) %>%
  select(species, year, postcode, wijk) %>%
  filter(year > 0)

# What this code does you can read out as:
# Take 'nijmegen_trees' data, and then rename the 3 columns, and then
# select the 4 columns, and then filter rows where variable 'year'
# is larger than zero.
```

4.3 mutate(), transmute(), and arrange()

You will probably use functions `mutate` and `transmute` very often, especially when combined with pipes. With the function `mutate`, you can add new columns or modify existing ones, and you can manipulate everything inside mutate/transmute with functions and mathematical operations.

```
# Let's add a column in which we calculate tree age

# While in base R you might do something like this:

nijmegen_trees$tree_age <- 2019 - nijmegen_trees$PLANTJAAR

# With tidyverse you can use mutate from the dplyr package
nijmegen_trees %>%
  mutate(
    tree_age = 2019 - PLANTJAAR
  )
```

If you want to sort the rows by a column, you can use the `arrange()` function. By default it gives ascending order, but you can nest `desc()` to sort a variable in descending order.

```
# Some of the oldest trees in this database are the beech trees in Heijendaal.
nijmegen_trees %>%
  mutate(
    tree_age = 2019 - PLANTJAAR
  ) %>%
  arrange(desc(tree_age))
```

For workflows that involve data analysis, you will likely want to create a dataframe that fits your exact needs. Function `transmute()` starts from a clean slate, and you can think of it as using `mutate()` that creates columns in an empty dataframe.

```
# Clean up the data
nijmegen_trees_cleaned <- nijmegen_trees %>%
  transmute(
    postcode = postcode_nummer,
    wijk,
    species = BOOMSOORT,
    year = PLANTJAAR,
    age = 2019 - year
    ) %>%
  filter(year > 0)

# Check the contents of species column
unique(nijmegen_trees_cleaned$species)
```

There are some issues with the species names here, so let's use another tidyverse package that is useful for working with strings (character data).

4.4 stringr package

Package `stringr` is a tidyverse tool for manipulating strings (character data). Manipulating strings is a common issue during data analysis, and `stringr` has many useful functions that are more intuitive than base R functions with the same purpose. It has a useful design feature where every function begins with a prefix `str_*()`, so you can use RStudio autocomplete feature to find the function you need.

We can see that there are some hybrid species (with x in the name, referring to the crossing), or different varieties or subspecies. We will try to create a name that could best summarize the species binomial name. You can decide also to create a column referring to whether the species is a hybrid or not.

```
# since we will just change one column, and add another, we'll use mutate()
# Remember, mutate adds columns, so is_hybrid will go to the last place,
# and we only modify column species, since the column name already exists.
```

```r
nijmegen_trees_cleaned %>%
  mutate(
    is_hybrid = if_else(str_detect(species, " x "), TRUE, FALSE),
    species = str_replace(species, " x ", " "),
    species = str_replace(species, "_", " "),
    species = word(species, end = 2))

# If we want to use some information from the LHS object,
# you can use "." for that.

nijmegen_trees_cleaned %>%
  mutate(
    id = 1:(nrow(.))
      )
```

4.5 Combining multiple functions

We will clean this dataset, and prepare it for further analysis. This chunk of code does many things at once, but essentialy it is a single step of creating a cleaner dataframe from the original one. We will create few columns that may not be that useful, but we will create them to demonstrate different ways of creating columns. We will also use here the function `case_when()` from package `dplyr`, which behaves like a nested `if_else()` and it can be very useful to know that this function exists.

```r
# Clean the data
tree_data_cleaned <- nijmegen_trees %>%
  rename(year = PLANTJAAR) %>%
  filter(year > 0) %>% # Remove trees with unrealistic values
  mutate(
    species = str_replace(BOOMSOORT, " x ", " "), # Remove hybrid species
    species = str_replace(BOOMSOORT, "_", " "), # Remove underscore
    species = word(species, end = 2),
    test = "test_column",
    col = 1:nrow(.),
    species2 = .$BOOMSOORT,
    tree_age = 2019 - year
  ) %>%
  transmute(
    species, year, tree_age,
    wijk,
    Postcode = postcode_nummer,
    # Reclassify to two values with if_else()
    age_binary = if_else(tree_age > 40, "Old", "Young"),
    # Reclassify to multiple values with case_when()
    age_categories = case_when(
      tree_age <= 2 ~ "Stem",
      tree_age <= 20 & tree_age > 2 ~ "Young",
      tree_age <= 40 & tree_age > 20 ~ "Adult",
      tree_age <= 80 & tree_age > 40 ~ "Mature",
      tree_age > 80 ~ "Old",
      TRUE ~ "None"
    ))
tree_data_cleaned
```

```
## # A tibble: 61,886 x 7
##    species         year tree_age wijk    Postcode age_binary age_categories
##    <chr>          <int>    <dbl> <chr>      <int> <chr>      <chr>
##  1 Aesculus hip~  1930       89 Galgen~     6524 Old        Old
##  2 Aesculus hip~  1930       89 Galgen~     6524 Old        Old
##  3 Aesculus hip~  1930       89 Galgen~     6524 Old        Old
##  4 Aesculus hip~  1950       69 Galgen~     6524 Old        Mature
##  5 Aesculus hip~  1960       59 Galgen~     6524 Old        Mature
##  6 Aesculus hip~  1950       69 Galgen~     6524 Old        Mature
##  7 Aesculus hip~  1950       69 Galgen~     6524 Old        Mature
##  8 Aesculus hip~  1930       89 Galgen~     6524 Old        Old
##  9 Aesculus hip~  1930       89 Galgen~     6524 Old        Old
## 10 Aesculus hip~  1930       89 Galgen~     6524 Old        Old
## # ... with 61,876 more rows
```

4.6 group_by and summarize

Soon you will want to learn more from your data. You will find that most of those things are related with counting per some group. To group a table, use `group_by()` which will create a grouped tibble. `ungroup()` removes grouping, and should be used after you perform operations on the grouped data. So if we want to know which neighborhood in Nijmegen has the oldest trees, or which is most common tree species in the old city core, you will end up using just few more additional functions.

Let's look more into the data using columns `wijk` and `tree_age`, and analyze the age of trees for Nijmegen neighborhoods. We will calculate for each neighborhood (wijk) average tree age, age of the oldest tree in the neighborhood, as well as the number of trees in each neighborhood.

```
# Calculate mean age per neighbourhood
# We use function summarize() to summarize values per group
tree_data_cleaned %>%
  group_by(wijk) %>% # Create grouped tibble
  summarize(
    mean_age = mean(tree_age), # Use na.rm = TRUE if there are NA values
    max_age = max(tree_age)
  ) %>%
  ungroup() # We need to remove grouping afterwards
```

```
## # A tibble: 21 x 3
##    wijk          mean_age max_age
##    <chr>            <dbl>   <dbl>
##  1 't Acker          33.1     169
##  2 Aldenhof          44.4     119
##  3 Biezen            26.3     169
##  4 Bottendaal        22.3      99
##  5 Galgenveld        32.8      99
##  6 Goffert           49.0     169
##  7 Hatert            34.3     119
##  8 Hatertse Hei      37.9     119
##  9 Hees              29.3     169
## 10 Heijendaal        58.3     269
## # ... with 11 more rows
```

Function `summarize()` will reduce multiple values down to a single value. This means that you can lose lot of information from your data, and keep only the grouped and calculated values. To perserve all of the columns, we can use `mutate()` instead of `summarize()`.

```
# We will also calculate the number of observations per group
# Store this data into a variable for later use
plot_data_age <- tree_data_cleaned %>%
  group_by(wijk) %>% # Create grouped tibble
  mutate(
    mean = mean(tree_age),
    max = max(tree_age),
    n = n() # n() gives the number of observations per group
  ) %>%
  ungroup()

plot_data_age
```

```
## # A tibble: 61,886 x 10
##    species   year tree_age wijk  Postcode age_binary age_categories  mean
##    <chr>    <int>    <dbl> <chr>    <int> <chr>      <chr>          <dbl>
##  1 Aescul~   1930       89 Galg~     6524 Old        Old             32.8
##  2 Aescul~   1930       89 Galg~     6524 Old        Old             32.8
##  3 Aescul~   1930       89 Galg~     6524 Old        Old             32.8
##  4 Aescul~   1950       69 Galg~     6524 Old        Mature          32.8
##  5 Aescul~   1960       59 Galg~     6524 Old        Mature          32.8
##  6 Aescul~  1950       69 Galg~     6524 Old         Mature          32.8
##  7 Aescul~  1950       69 Galg~     6524 Old         Mature          32.8
##  8 Aescul~  1930       89 Galg~     6524 Old         Old             32.8
##  9 Aescul~  1930       89 Galg~     6524 Old         Old             32.8
## 10 Aescul~  1930       89 Galg~     6524 Old         Old             32.8
## # ... with 61,876 more rows, and 2 more variables: max <dbl>, n <int>
```

```
# Calculating the number of trees per neighborhood
# We will take only a few areas of Nijmegen
tree_data_cleaned %>%
  filter(wijk %in% c("Heijendaal","Lent", "Goffert")) %>%
  group_by(wijk) %>%
  summarize(
    trees_number = n()
  ) %>%
  ungroup()
```

```
## # A tibble: 3 x 2
##   wijk       trees_number
##   <chr>             <int>
## 1 Goffert            4187
## 2 Heijendaal         3458
## 3 Lent               3550
```

```
# We can group by multiple variables. Let's check the number of trees per
# neighborhood and per age category
tree_data_cleaned %>%
  filter(wijk %in% c("Heijendaal","Lent", "Goffert")) %>%
  group_by(wijk, age_binary) %>%
  summarize(
    trees_number = n()
  ) %>%
  ungroup() %>%
  arrange(trees_number)
```

```
## # A tibble: 6 x 3
##   wijk       age_binary trees_number
##   <chr>      <chr>             <int>
## 1 Lent       Old                 298
## 2 Heijendaal Young               987
## 3 Goffert    Young              1844
## 4 Goffert    Old                2343
## 5 Heijendaal Old                2471
## 6 Lent       Young              3252
```

```
# It seems like there are many young trees in Lent
```

5 Using the ggplot2 package

Once we prepare our data for analysis, we can use vizualization to get a better insight. Using data vizualization is important during data cleaning and data analysis stages. Main tool for data visualization with R has became the ggplot2 package. It features many options, and it uses a slightly different approach to visualize the data compared to the `plot()` approach from base R. With base R plotting, you need to manually add all of the individual plot components to the canvas, such as points, lines, legends etc. However, `ggplot2` is based on the grammar of graphics paradigm (hence the *gg* in `ggplot2`). Grammar of graphics has few constant elements that you need to have in order to build the plot, and many things are done already automatically for you. With `ggplot2`, we need to initialize the plot and provide the data (with function `ggplot()`), tell it how to map variables to aesthetics (function `aes()`), what graphical primitives/geometries to use (function `geom_*()`), and it will take care of the details. We connect different ggplot elements with `+` symbol (don't confuse `%>%` and `+` when using ggplot2).

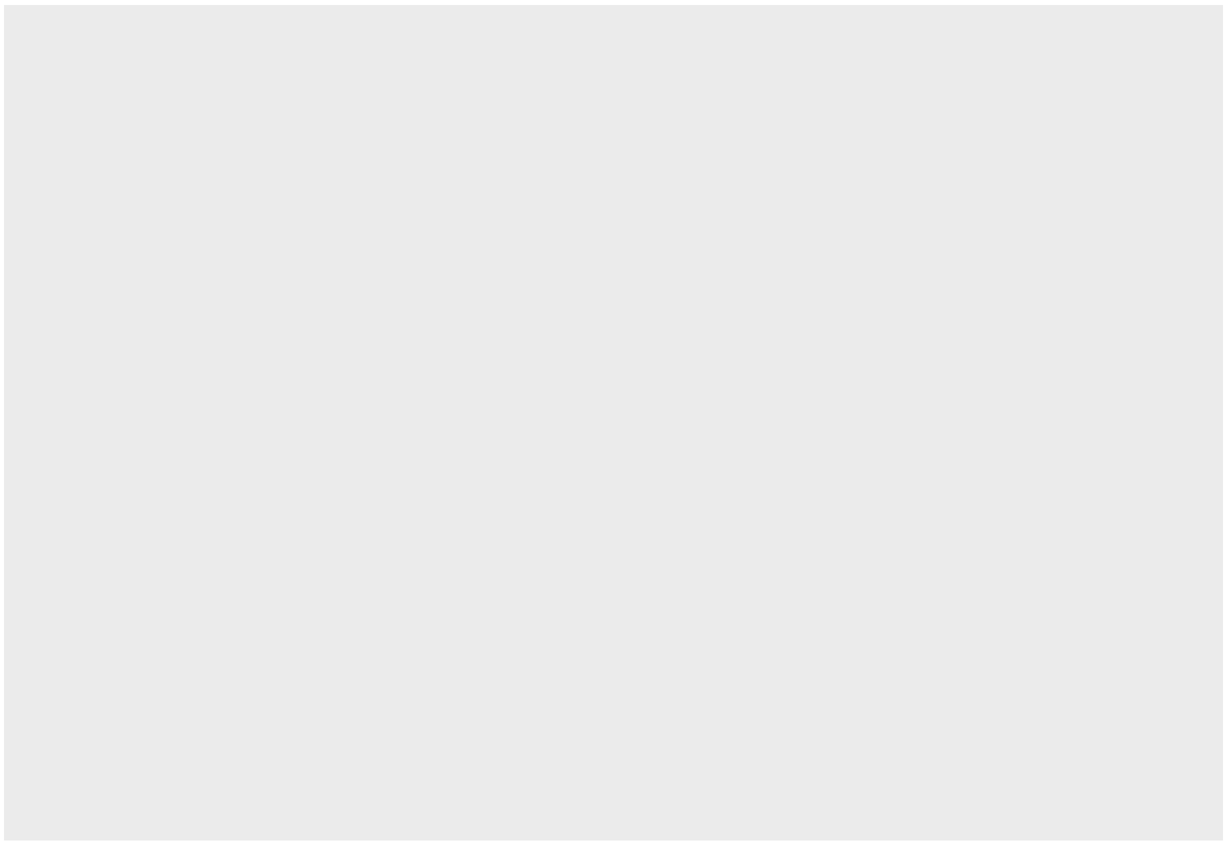
Let's explore the Nijmegen trees dataset with `ggplot2`.

5.1 Basic building blocks

The most basic elements of creating a ggplot2 visualization are the functions `ggplot()`, `aes()` and `geom_*()`:

The plot
Function `ggplot()` initializes a ggplot object. Every plot has to start with this function, and you can use it to declare what data will be shown in the plot.

```
ggplot(tree_data_cleaned)
```

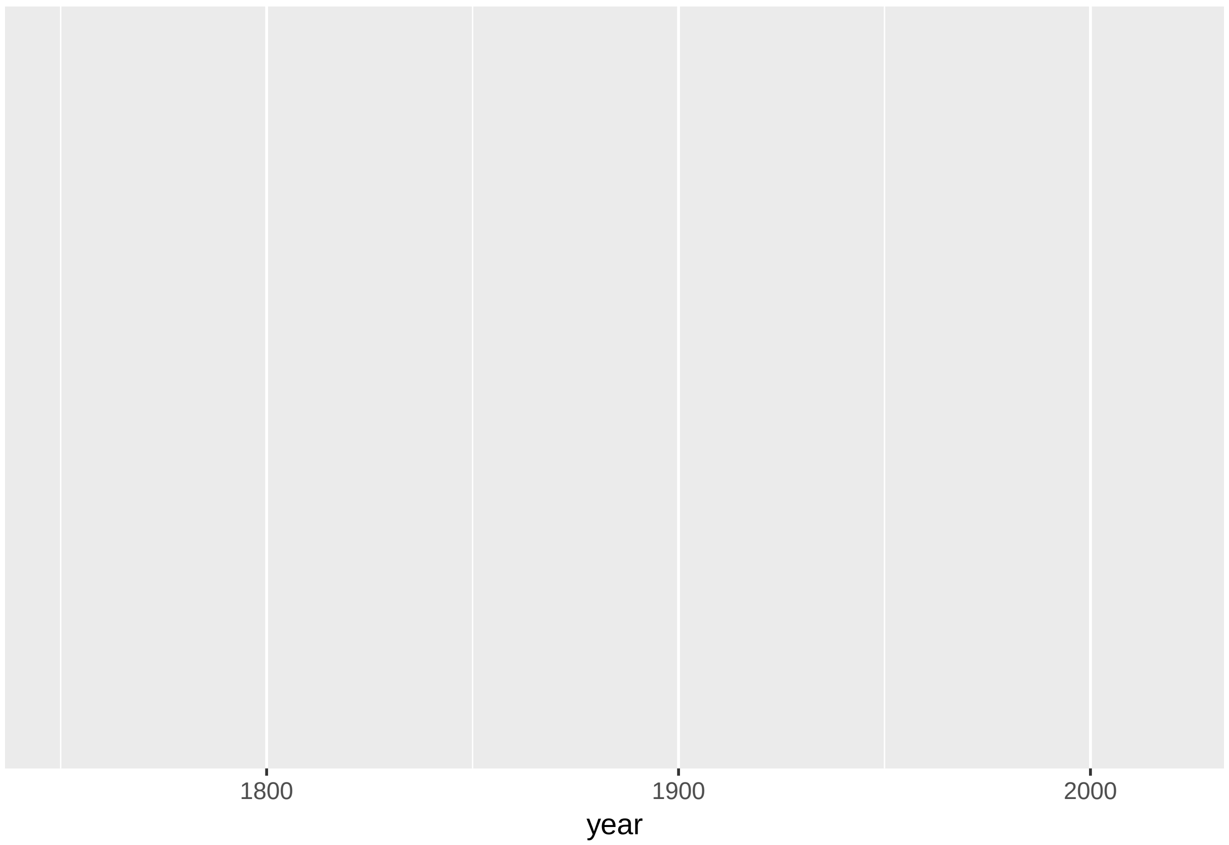


The result of code above is a blank canvas. The data is already mapped to the canvas, but we need to tell ggplot how to map the data to the visual properties (aesthetics) of geometric objects (geoms). Although it may sounds cryptic, it should be more clear with the first example.

In this case, we want to check the column `year` (planting year) to see if everything is ok. Histograms can be a useful tool to inspect numerical data, so we will create our first histogram with `ggplot2`.

The aesthetics
With aesthetics you map your data to the axes, as well as other graphical properties such as color, size, shape. You can have aesthetics for each geom, and control the appearance of each one individually. In cartesian coordinate system, x axis is the horizontal one, and y is the vertical one. When you want to show a variable with a single value per observation, such as in this, common approach is to show the observations on the x axis, an variable value on the y axis. In this case the observation is in the column `year`, and we will map it to the x axis, using the `aes()` function.

```
ggplot(tree_data_cleaned) +
  aes(x = year)
```



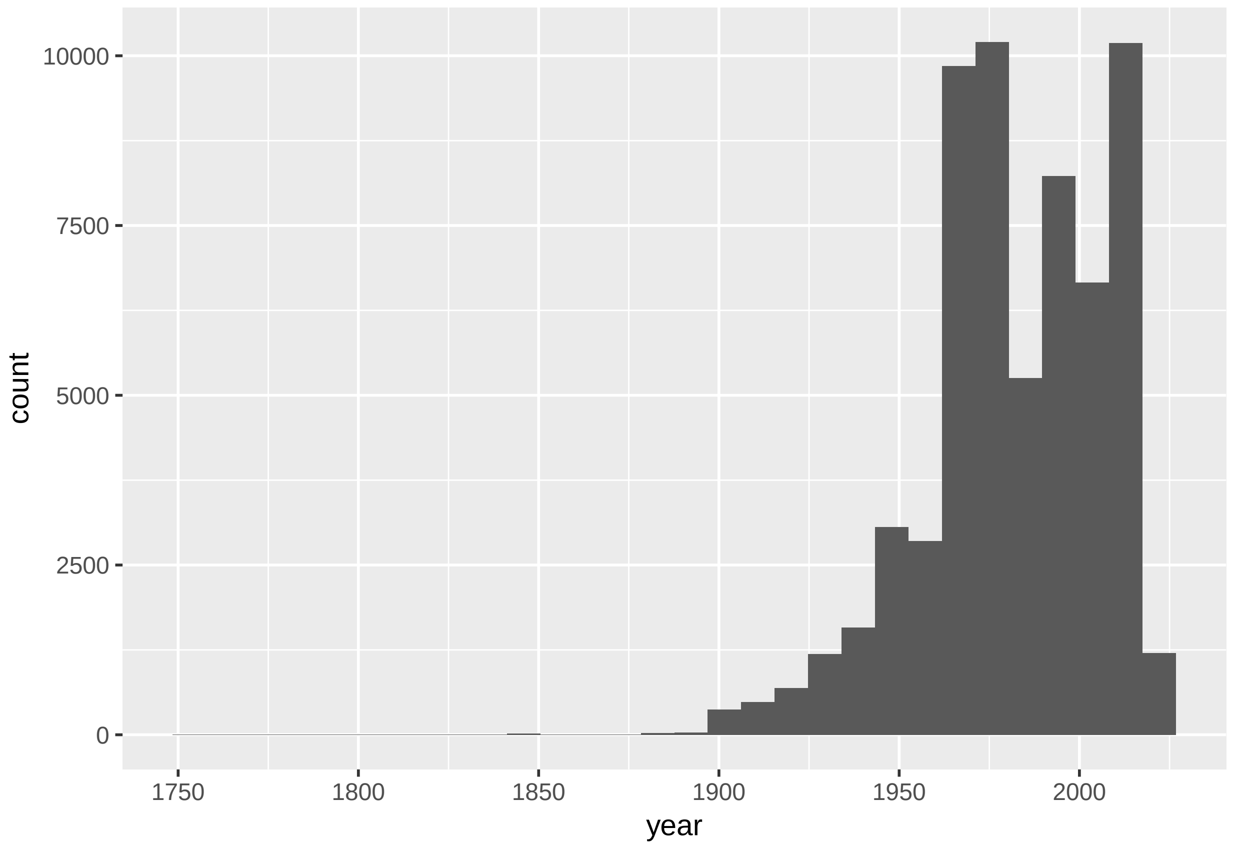
The graphics (geoms)

We can see that there are some more elements in the plot, such as the x axis with its range of values, and the axis name. However, we still have to tell ggplot which geometric objects should it use to represent the data visually. Geometric graphical objects in `ggplot2` have function prefix `geom_*()`. Histograms are made with `geom_histogram()`, and a histogram just need value of one variable for x axis. The values for that variable will be binned, and frequency in those bins will be plotted. For a barchart, where you map variable to y axis as well, use `geom_bar()` or `geom_col()`.

5.2 Putting it together

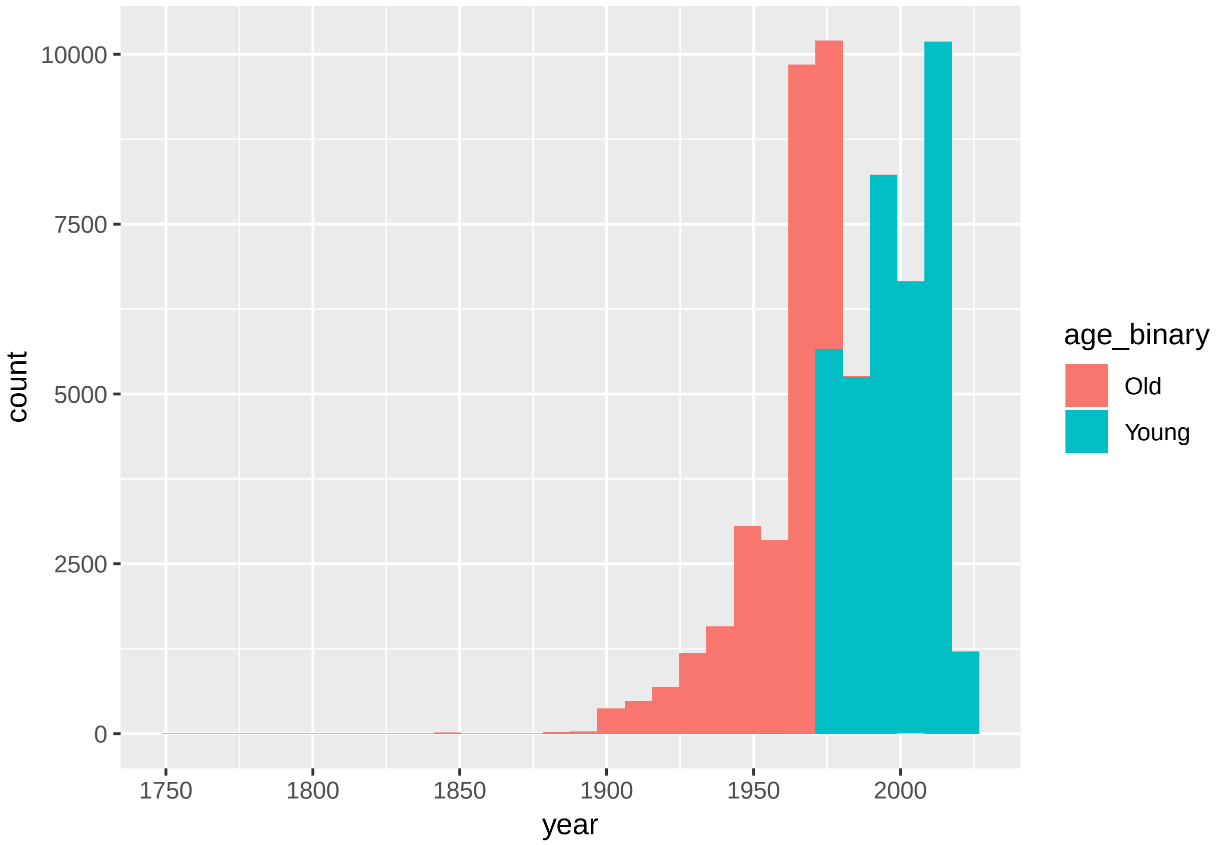
```
ggplot(tree_data_cleaned) +
  aes(x = year) +
  geom_histogram()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



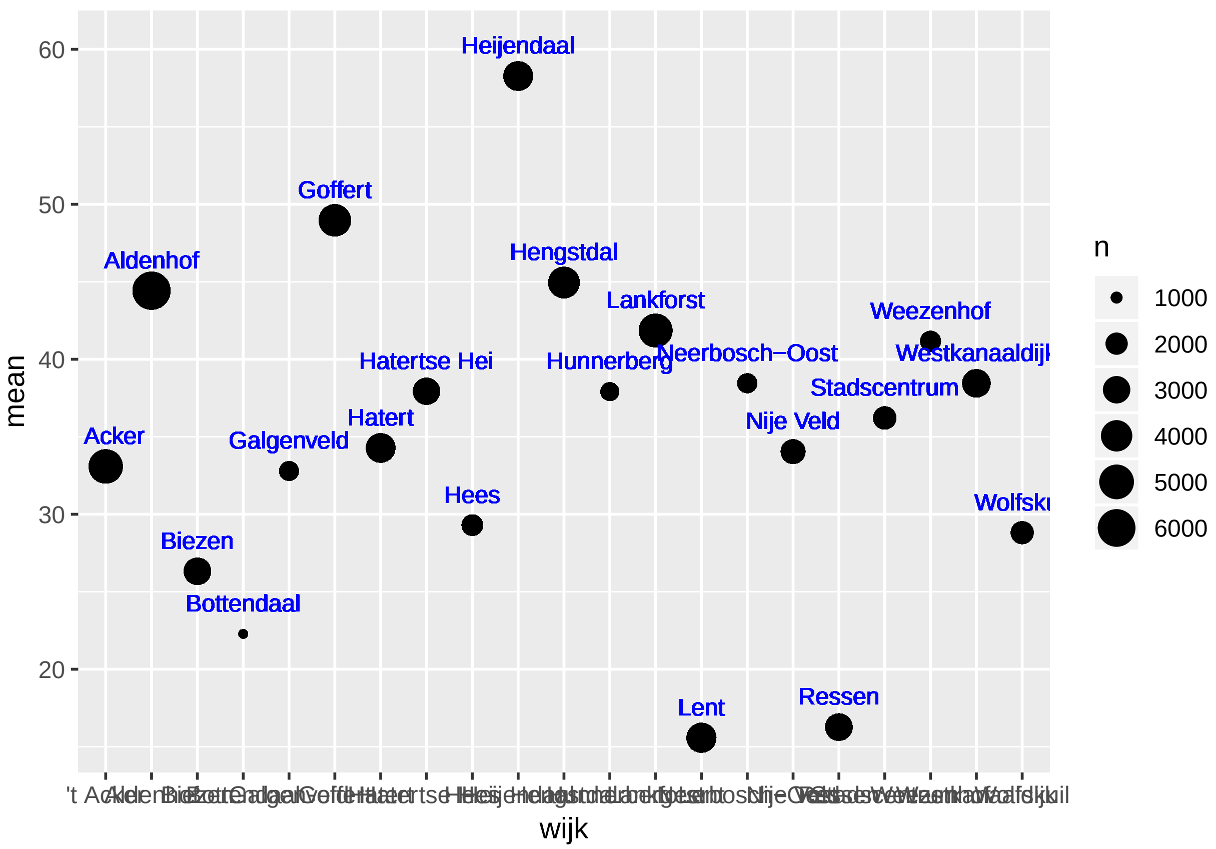
```r
# We can also map data to colors. For this kind of a geom, you can modify
# the outline color and the fill color with arguments color and fill.
# This needs to be in the aesthetics part. Note that you can use multiple geoms,
# and different aesthetics for each geom.
ggplot(tree_data_cleaned) +
  aes(x = year, fill = age_binary) +
  geom_histogram()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Although many things can be modified here, we now know that the majority of trees were planted in the last 100 years, with the increasing amount during the last 70 years. Let's change the data we use for plotting, and see the average age of trees per neighborhood. We will use new geoms, and combine two geoms for a richer plot.

```
#### Let's change the data, and see the average age of trees per neighborhood
ggplot(plot_data_age) +
  # You can put aesthetics inside of geom function.
  geom_point(aes(x = wijk, y = mean, size = n)) +
  geom_text(aes(x = wijk, y = mean + 2, label = wijk),
            color = "blue", size = 3)
```



```
# If you don't map geom graphics to a variable, then the argument should be
# outside of aes(). See how the color and size of text are outside of aes().
# We added here a value of 2 to the y position of the text,
# so that the wijk label is above the point.
```

We have our first graphic created with ggplot2, however there are many things that can be improved. The x axis is very difficult to read, which could be solved in multiple ways (rotating the label, rotating the plot, show subset of data). We have also mapped point size the the data points. If you want to map data to the variable, that needs to be added inside of the `aes()`. If you want to set the color or size manually (color the dots red for example), you need to have it outside of `aes()`. Depending on the geom, sometimes you control the color both with `color` and `fill` arguments. For example, outline of bars is controled with `color`, and color of box with `fill`.

5.3 Building blocks summary

`ggplot2` is a powerful package with many options that you can explore. Here we will cover only the very basics of ggplot, but enough to get you started.

- **plot**
  Function `ggplot()` initializes a ggplot object. Every plot has to start with this function, and you can use it to declare what data will be shown in the plot.

- **aes**
  With aesthetics you map your data to the axes. You can have aesthetics for each geom, and control the appearance of each one individually.
- **geom_**
  - geom_point
  - geom_line
  - geom_col
  - geom_boxplot
  - geom_text

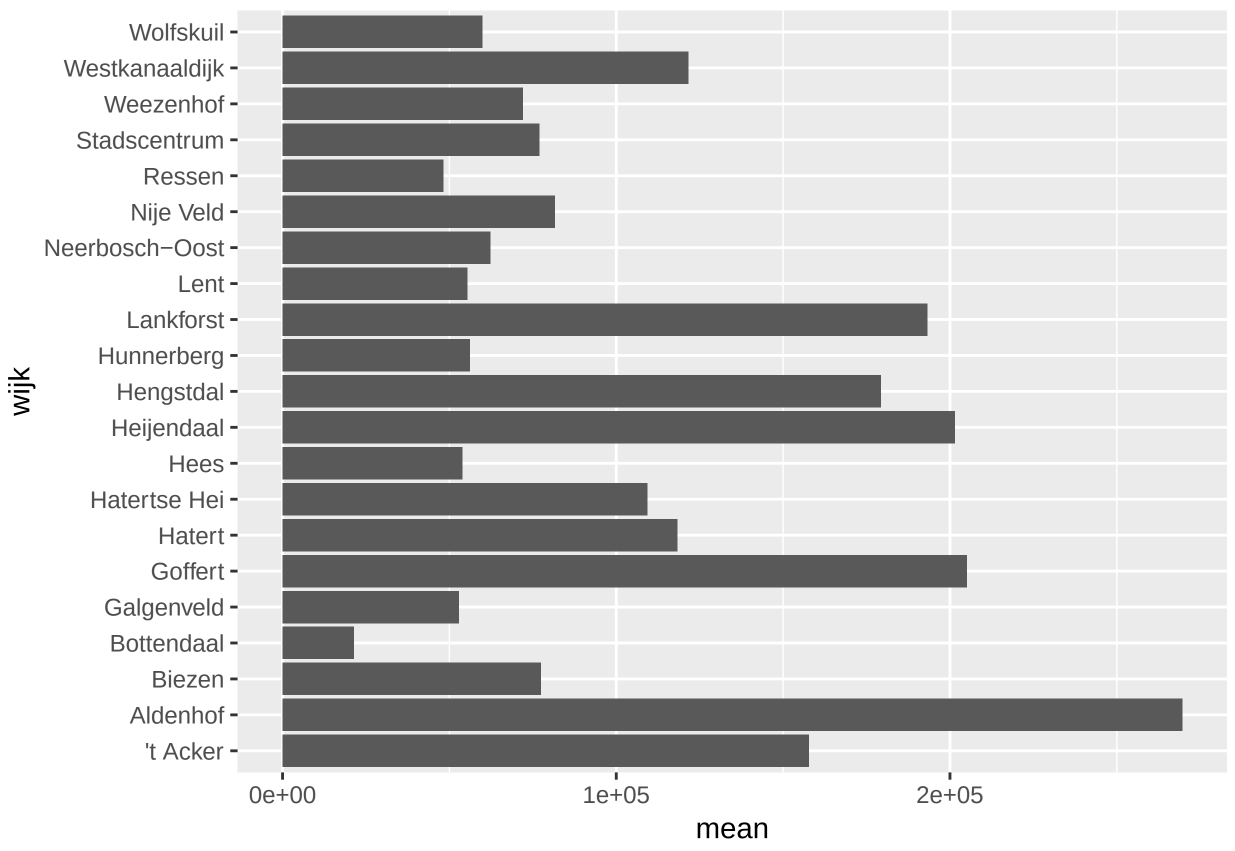
…and many others.

- **scale_**
  Modify the xy scales, color palletes, and add transformations to the scales.
- **stat_**
  Add statistics to the plot. For example, you can easily add a fitted line to point data.
- **coord_**
  Manipulate the plot coordinates.
- **theme**
  Manipulate the visual aspects of the theme. You can modify the appareance of almost all components of the plot. You can also use a different theme that have `theme_*()` prefix.
- **facet_**
  Faceting is a useful way of showing multiple panels in a single graphic, where we use a variable to separate the panels.

5.4 Customizing the graphic

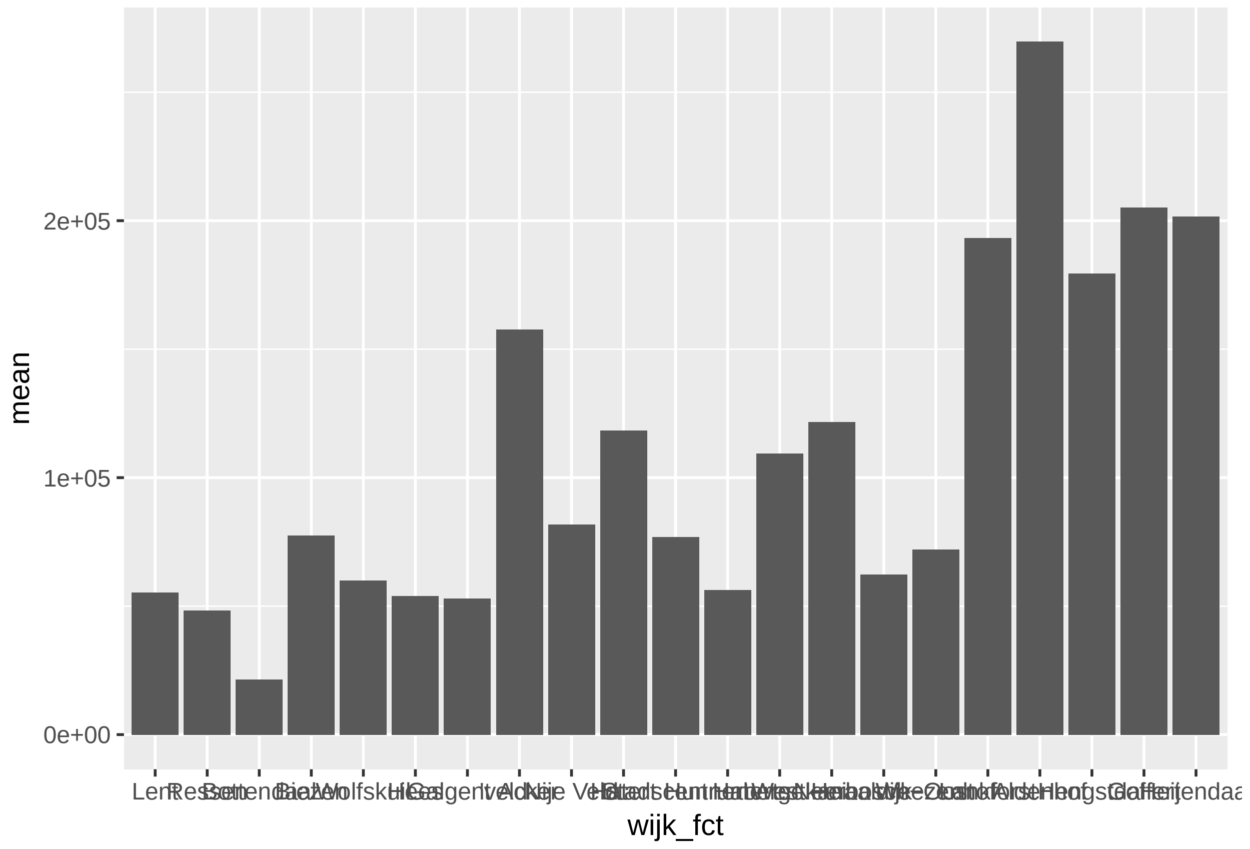
The graphic we made is still difficult to read, and color is not very useful here. Instead of modifying the labels, we can modify plot coordinates so that the labels are horizontal. We will also use `geom_col()` where instead of using points, we shows bars whose heights represent data value.

```r
# Plot mean age using barchart, and rotate the
# plot by adding the coord modifier
plot_data_age %>%
  ggplot() +
  aes(x = wijk, y = mean) +
  geom_col() +
  coord_flip()
```

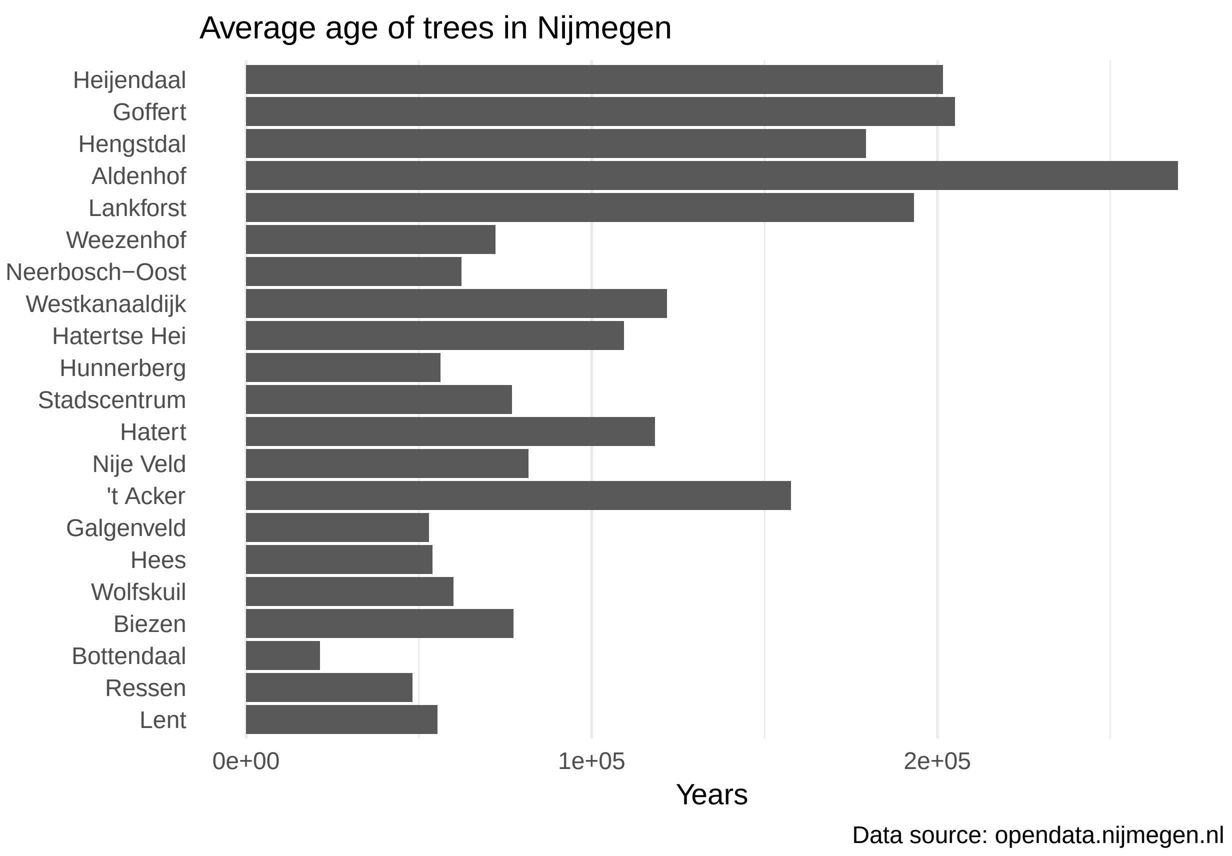


This already looks much better. However, let's try to see how we can improve this graphic further. This graphic would be much more effective if the bars were ordered, which will require manipulating the data before plotting it. Luckily, there is a tidyverse solution to this problem as well. We will also use one of the themes included with ggplot2, and we will label the axes and add a title. It is possible to adjust almost every part of ggplot with function `theme()`. However, there are many packages that extend `ggplot2` capabilities, from new geoms, themes, interactivity, web implementation, and others. Click the links for more detailed list of `ggplot2` building blocks, and the extensions. If you want to see more cool data visualizations made with R and `ggplot2`, visit the R graph gallery.

```r
# Create barplot showing mean tree age per neighbourhood

# Using fct_reorder from forcats package, we will reorder column wijk
# according to the mean value, and store it in new factor column wijk_fct

# Pay attention when do you need to use '%>%', and when do you use '+'.
# The '+' operator is only used to chain ggplot elements and comes after
# the function ggplot().

# For this example, let's break down the plot. Since it can be stored into a
# variable, we will store the basic block, and modify it further

p_avg_age <- plot_data_age %>%
    mutate(
    wijk_fct = fct_reorder(wijk, mean)
  ) %>%
  ggplot() +
  aes(x = wijk_fct, y = mean) +
  geom_col()

p_avg_age
```



```
p_avg_age_themed <- p_avg_age +
  coord_flip() +
  theme_minimal() +
  labs(title = "Average age of trees in Nijmegen",
       y = "Years", x = NULL, caption = "Data source: opendata.nijmegen.nl") +
  theme(
    plot.title = element_text(size = 12),
    panel.grid.major.y = element_blank()
      )

p_avg_age_themed
```

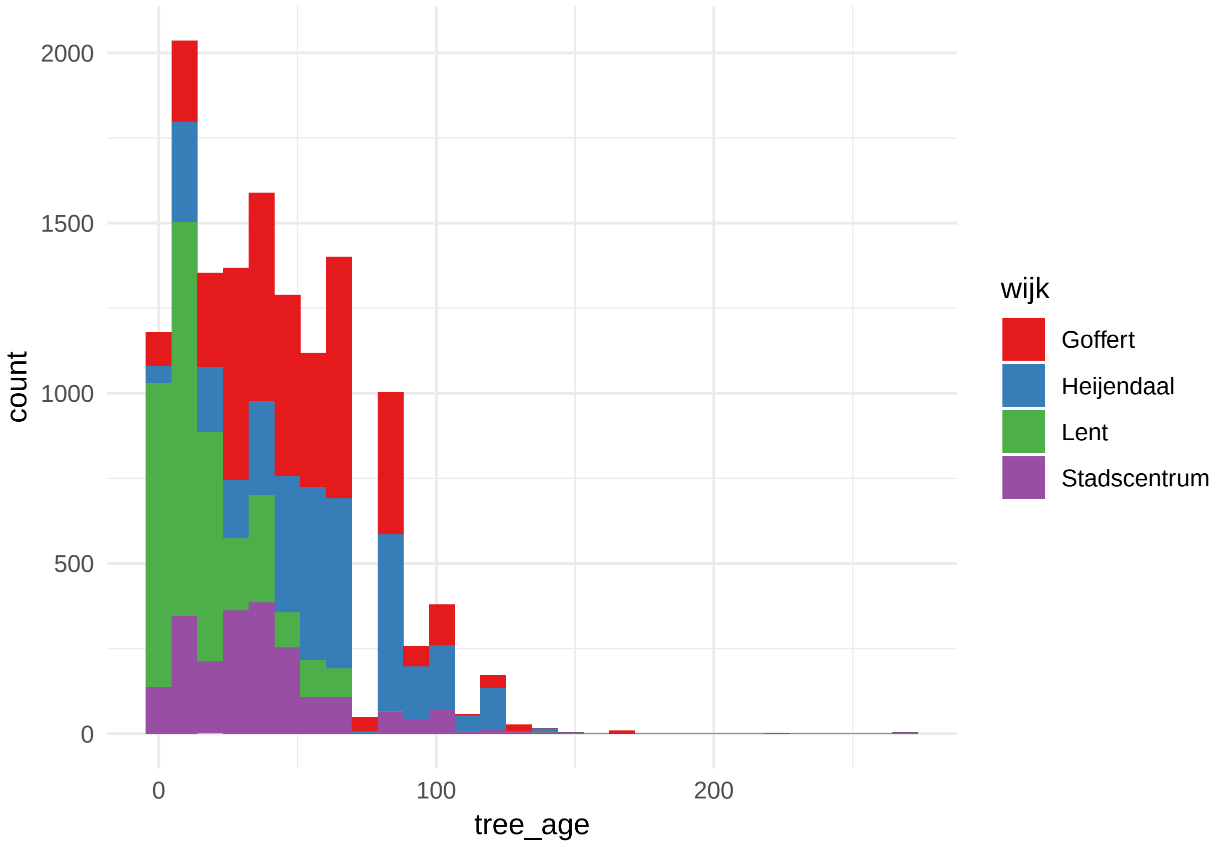


```
# To save the plot, use ggsave function
# ggsave("Output/plot_treeage.png", p_avg_age_themed)
```

Let's use facetting feature to show the distribution of tree age for selected neighborhoods. The easiest way to do this is with histograms. Let's first map the color of bars to the wijk column (neighborhood), and we will use a color scale that works better with qualitative data. In this case we will use ColorBrewer scale, which can be accessed with the function `scale_fill_brewer()`.

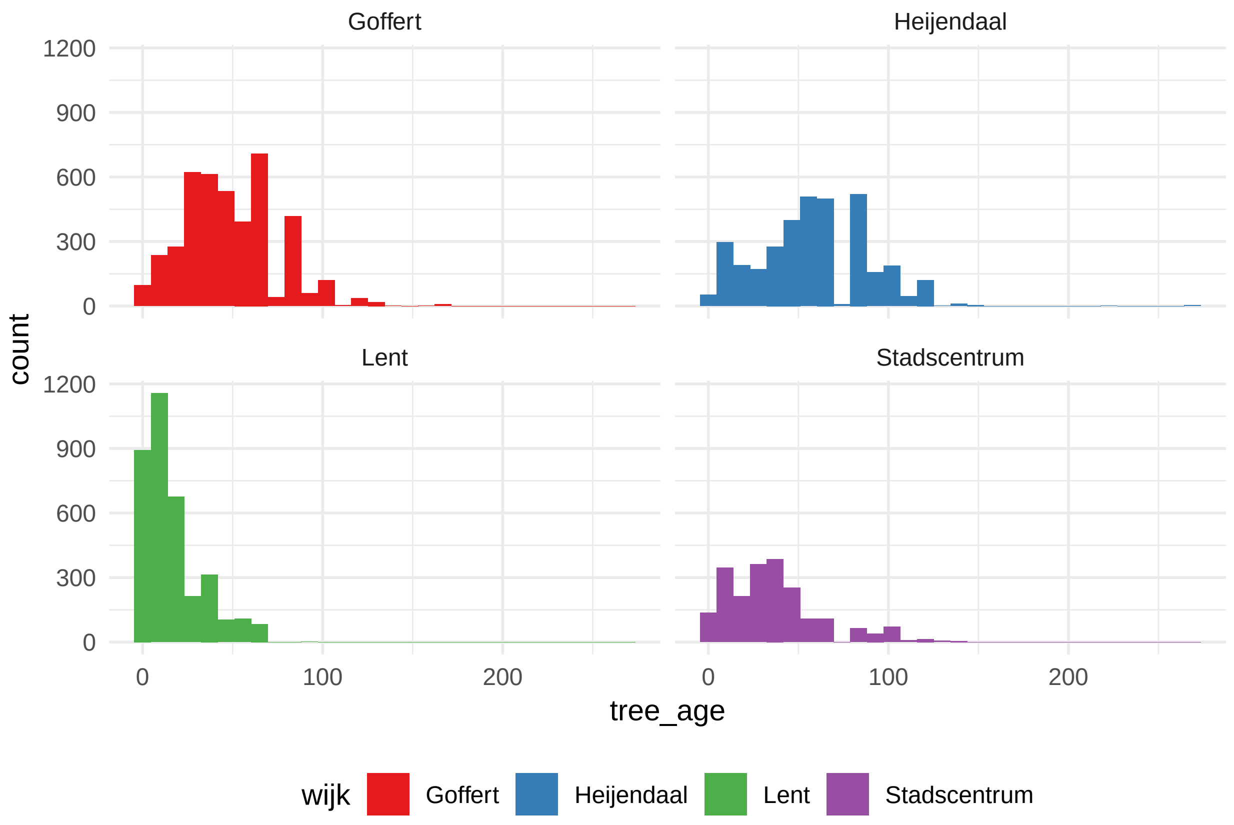
```
# Create a histogram that shows the age frequency of trees for 4 neighborhoods
p_hist_age <- tree_data_cleaned %>%
  filter(wijk %in% c("Goffert", "Lent","Stadscentrum", "Heijendaal")) %>%
  ggplot() +
  aes(x = tree_age, fill = wijk) +
  geom_histogram() +
  scale_fill_brewer(type = "qual", palette = 6) +
  theme_minimal()
p_hist_age
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



```
# This can be visualized in a better way, so let's use faceting to show data in
# multiple panels.
# We need to use the tilde operator here (~), where we say which variable is
# used for faceting. We will modify the theme to move the legend to the bottom
p_hist_age +
  facet_wrap(~ wijk) +
  theme(
    legend.position = "bottom"
  )
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```


Goffert
Heijendaal
Lent
Stadscentrum
count
1200
900
600
300
0
0
100
200
tree_age
wijk
Goffert
Heijendaal
Lent
Stadscentrum

6 Let's practice

6.1 Exercise 1: Most common trees in Nijmegen.

Difficulty level - Easy
What are the most common tree species planted in Nijmegen?

Task:
- What are the **ten** most frequent **species** of trees that are planted in Nijmegen. Minimum things you need to have is the species scientific name and the number of individual trees in **descending** order for top 10 tree species.

Hints:
- Helper function `n()` gives the number of observations per group.
- Read the help of the `arrange()` function to see how to reorder descending.
- This task can be solved with pipes in 5 lines of code or less, but use as much code as needed.

6.2 Exercise 2: Most common tree genera in Nijmegen

Difficulty level - Easy
What are the most common tree genera planted in Nijmegen?

Task:

- Find out what are **ten** most frequent tree **genera** in Nijmegen.

Hints:
- Under the rules of binomial nomenclature adopted by Carl Linnaeus, species names consist always out of two parts (eg. *Quercus robur*). First part of the name ("Quercus") is the species genus (plural of genus is genera), and the second part ("robur") is the specific name of the species within that genus.
- There is a conveniently named function in `stringr` package that you can use to extract words from a sentence.
- Package `dplyr` offers some helper functions that wrap some commonly used function calls together in a more verbose-sounding function. Check the help for functions `tally()` and `count()` and see how they can be useful for you.
- Functions `mutate()` and `transmute()` from `dplyr` package are used to create new columns.
- Check how can function `top_n()` be useful.

6.3 Exercise 3: Where to find a relict tree in Nijmegen?

Difficulty level - Medium

Picea omorika is an relict coniferous species from the tertiary period, and now it survives in few canyons in the Dinaric Alps. However, despite its native range rarity, it is a very popular ornamental tree. Find out which neighborhoods in Nijmegen have the most trees of *Picea omorika* planted. Use data visualization to convey your message!

Task:
- Find out what **neighborhoods** have *Picea omorika* planted, and **how many trees** planted there?
- Visualize your results. Add plot title and label the axes. Add plot subtitle and caption if necessary.
- Bonus task: order the bars in the bar chart.

Hints:
- String value of the species name in the csv is "Picea omorika". - Bar charts are a good way of showing quantify of things per group.
- Depending on your data structure, you can use either `geom_bar()` or `geom_col()` to create a bar chart. Check function help for more information.
- Don't forget that you chain ggplot2 functions with a plus symbol `+`.
- Although making your plot pretty is secondary here, feel free to use a theme or to edit the visual appeal of your plots if you have the time.

6.4 Exercise 4: Age range of trees

Difficulty level - Medium

Task:
For this task visualize the **range of tree age values** of each neighborhood using **boxplots**. Boxplots are useful vizualization tool to see the spread of values. Boxplots center line represents median value of the range. However, to make it more effective, **order the boxes** in the plot by the **median** value, therefore you will need to calculate median tree age and reorder the data before plotting.

Hints:
- To perserve information when summarizing data per group, use `mutate()`.
- Package `forcats` is useful to reorder factors.
- Function name to create boxplots is `geom_boxplot()`.
- Since there are many neighborhoods, the plot will look better if it's rotated.

6.5 Exercise 5: Greenest Nijmegen neighborhoods

Difficulty level - Medium/Difficult

We want to calculate here which neighborhood in Nijmegen is the greenest, i.e. which has the most trees relative to the size of that neighboorhood. However, the data on the neighborhood area is in a separate file, so we will need here to combine two datasets, perform calculations, and create the data visualization. The dataset you need is located in `data/nijmegen_wijk_area.csv`.

You can use `base::merge()` or `dplyr::*_join` to combine the two data frames by common column. It is not mandatory to use a tidyverse function in this section, only to perform the task. Most useful type of a join in this case the inner join (`inner_join()`).

Task:
- Calculate **tree density** for Nijmegen neighborhoods, and present the results graphically to show which ones have the most trees. There are around 20 neighborhoods in this dataset, so you can include all of them in the graph.

Hints:
- Dutch word for 'neighborhood' is 'wijk'.
- Unit for the neighborhood area is km^2.
- To calculate density, you want to divide the population by the area size.
- It might be difficult to perform this task in a single chain, so create as many variables you need to finish the task.
- When your plot becomes too dense with long axis labels, you can use `coord_flip()` to flip vertical coordinates to horizontal ones.

6.6 Exercise 6: When were the trees planted?

Difficulty level - Difficult

Trees fall in the Spermatophytes group, meaning that they are plants that produce seeds. There are five groups of seed-producing plants, the cycads, ginkgo, conifers, gnetophytes, and the largest groups - flowering plants.

For this task we want to see how many trees from each group planted per each decade. In species list we only have coniferous trees, flowering trees, and the ginkgo tree - which has only one species in the group. For simplicity, we can refer to the trees from flowering plants group as deciduous trees. Below is a list of all coniferious genera included in the species list which you can use to create a vector containing coniferous species.

Conifer genera: "Abies", "Cedrus", "Chamaecyparis", "Larix", "Metasequoia", "Picea", "Pinus", "Pseudotsuga", "Sequoiadendron", "Taxodium", "Taxus", "Thuja", "Tsuga"

Task:
- Calculate how many trees were planted in each decade from group of conifers, deciduous, or ginkgo. Present your results visually.

To get you started, these are some of the important things you need to do:
- Get genus name.
- Get decade from the year.
- Reclassify genera so they belong to either conifers, ginkgo, or deciduous trees.
- Count number of trees planted in each decade from each group of trees.
- Visualize the results.

Hints:
- You can use nested if/else statements to rename multiple variables, or you can use a vectorised if version which is `case_when()`.
- If you use function `case_when()`, read function help carefully since the examples are particularly useful.
- Decade can be calculated from the year relatively easy. If you are stuck, don't be afraid to search for solutions online.
- Remember that you can use more than one variable when grouping.
- Use ggplot faceting to show trends for different groups in separate plot panels. - Line graphs are useful when showing trends over time. - Function `facet_wrap()` will use a common scale for all panels. If it is difficult to see trends in panels where there are fewer trees, so set the argument `scales` to `scales = "free"`.

This task may tricky to solve during the first day, but check the solution later to see one of the ways to solve it.

Learning unit 10

Efficient R coding

Selwyn Hoeks

Contents

1 Learning goals

After this learning unit is completed, you should be able to:
Start thinking about how and when to implement the concept of writing efficient code

- In terms of computational resources
- In terms of execution time
- In terms of coding effort (?)

Please note that do to the nature of this topic I chose to make the live coding part a bit longer compared to the assignment. In order to support my main goal to show a selection of general concepts of writing fast and more effecient code.

2 What is efficient code?

If you are familiar with other programming languages you might have noticed that R is a very dynamic and versatile language. Almost anything can be modified after it is created. Which is not the case for other languages like Java, JavaScript, C, C++ or Python. It is obvious that this comes with many advantages, making R a very popular tool in science and general data analysis. This unfortunately this comes also at a cost, compared to other languages (base) R usually does not earn many awards when it comes to efficiency.

Generally the effeciency ( ) is expressed by the ratio between the amount of work done (W) per unit effort (Q) (see equation below).

$$\eta = \frac{W}{Q}$$

In programming or data analysis W can be quantified by the operations required, e.g.:

- Reading a specific number of lines in a file
- Running some operations on the dataset
- Calculating necessary statistics
- Plotting results

All of which require a certain effort (Q). In order to simply increase our efficiency, we can reduce the number of operations required (reducing W), by carefully checking whether all actions in our code are required. However, in cases where we cannot reduce the amount of operations needed, we will need to improve the efficiency by reducing the amount of effort (Q) needed for each of our operations.

Q can, amongst other things, be defined as a combination of:

- Computation time
  - Amount of time needed to run code
- Resources used
  - Required hardware, possible to run on small laptop or expensive powerful pc

- Coding time/effort
  - Investment vs gains, is it in our interest to spend time on optimizing code?

For the average R programmer, the first two points are in most cases the bottleneck in increasing efficiency. Especially compared to other languages, which generally require more coding time. Besides the performance limitations due to design and implementation of R, most issues have to do with the fact that most 'slow R code' is slow simply because it's poorly written. As the majority of people use R as a tool to understand data, they consider code efficiency a secondary objective. "It's more important to get an answer quickly than to develop a system that will work in a wide variety of situations". Although this might be seen as a issue, it also means that is most cases it is relatively easy to make R code much faster, as we'll see in the following exercises.

For R there are some easy rules to follow to help us make code run as efficient as possible:

- Don't grow objects
  - Do not attach new elements iteratively to an object (e.g. rbind)
- Avoid loops
  - Use vectorized functions, functions that can operate over an entire vector or matrix
  - The `apply` family can also considered loops and are equally bad in performance
- Consider the use of packages with more efficient implementations of functions
  - Brief introduction at the end of this learning unit (fst package)
- Code parallelization
  - Brief introduction at the end of this learning unit (mclapply package)
- Use of different languages combined with R
  - Brief introduction at the end of this learning unit (Rcpp package) just to show it can be faster and many packages you download and install use it without you knowing it.

Besides making code run as fast and as efficient as possible, we also need to consider the time we spend on writing the actual code. For example, if a basic task can be written in R relatively fast without any effort and the resulting code is very inefficient it might still provide the fastest method to reach your goal. However, when the code needs to be used many times and involves heavy computations, a little more effort will help us increase our overall efficiency!

Another way to increase to speed of writing the actual code might be to get to learn our IDE (integrated development environment) of choice, in our case this is RStudio. For example, see: https://support.rstudio.com/hc/en-us/articles/200711853-Keyboard-Shortcuts to learn some handy RStudio shortcuts (some of which will also be demonstrated in class).

3 Live example: implementing efficient code

Here we will run through some example code and try to make it run more efficient by reducing the amount of computation time required to execute. This example code involves calculating the Menhinick's index for a larger data set. The formula to calculate the Menhinick's index is shown below:

$$MI = \frac{n}{\sqrt{N}}$$

where:

- n = Number of species
- N = Total number of individuals

Next we will discuss the technical implementation of a method computing the Menhinick's index on a large data set which we will generate ourselves, similar to learning unit ...

3.1 Packages required

As we have seen in the previous learning unit, R packages in some cases provide a large advantage of base R functions. In this learning unit we will be using the following packages:

```r
library(microbenchmark) # evaluating the computing time of R code
library(fst) # fast, easy and small data reading and writing
library(parallel) # for using the parallel version of lapply
```

3.2 Generating a large data set

As we can see from equation 2, the Menhinick's index requires a data set containing species abundances. In reality these are of course recorded in the field by several observation methods. Think of bird counts, fish population inventories, species monitoring or vegetation mappings.

Here we will create our own data set which contains a set number of species (`n_species`). These species were recorded at several time steps (`n_timesteps`). Multiplying `n_species` by `n_timesteps` gives us the total number of records in the data set.

```r
n_species <- 1e4 # set number of species to consider
n_timesteps <- 1e3 # set number of timesteps to consider
total_length <- n_species * n_timesteps # sample length

# generating a vector containing all of our species abundances
random_species_abundances <- sample(1:350,
                                    size = total_length,
                                    replace = T)
```

Next we will place our long vector (`random_species_abundances`) containing all records in a matrix, in this matrix the columns will be the time steps. Our matrix row will be the species.

```r
# matrix: columns are timesteps, rows are species-specific abundances
m <- matrix(random_species_abundances, nrow = n_species, ncol = n_timesteps)
head(m[1:10,1:10])
```

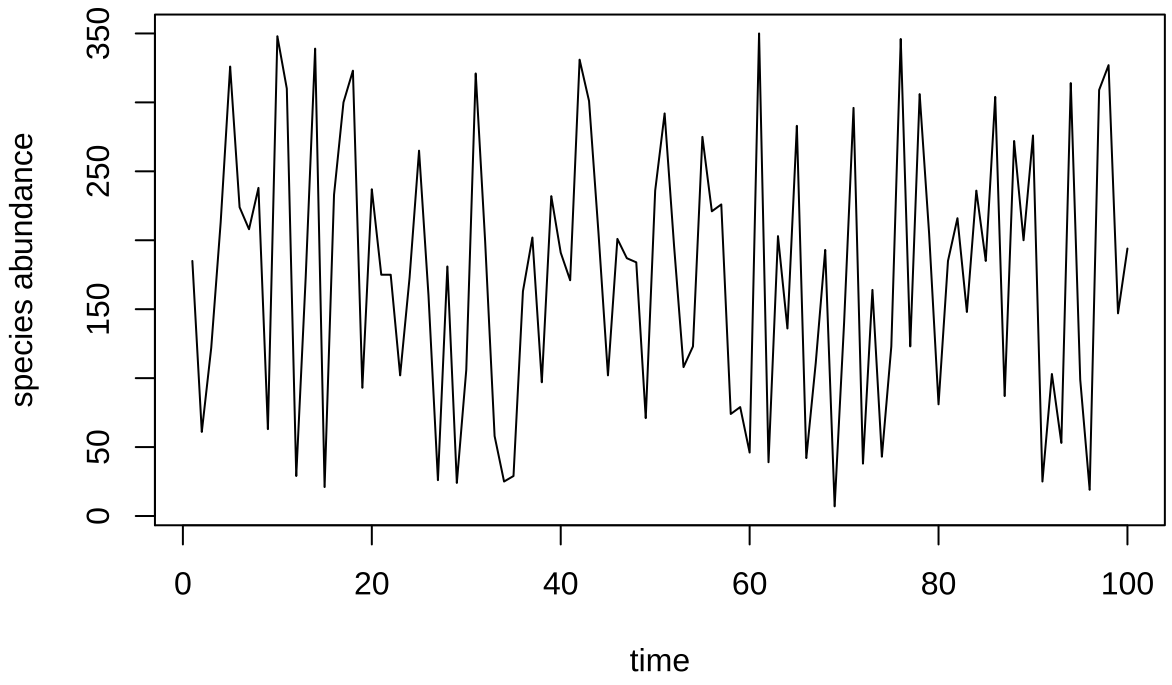
```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]  185   61  122  213  326  224  208  238   63   348
## [2,]   45  308   60    8   22  270   90  244   50   129
## [3,]  348  153  180  314  218  158   67  117  322   143
## [4,]  154  124   82  196  131  163  163  218  107    98
## [5,]  258  283  190   29   59   73  229  179   95    72
## [6,]  248   52    3  304  238   68  226  134  126   269
```

We can subset the matrix by for example getting all the abundances recorder for species 1 for first 100 timesteps. We do this by selecting the first row and then the 1:100 elements in that row.

```r
m[1,1:100]
```

```
##   [1] 185  61 122 213 326 224 208 238  63 348 310  29 173 339  21 233 300
##  [18] 323  93 237 175 175 102 173 265 160  26 181  24 106 321 199  58  25
##  [35]  29 163 202  97 232 191 171 331 301 205 102 201 187 184  71 236 292
##  [52] 196 108 123 275 221 226  74  79  46 350  39 203 136 283  42 111 193
##  [69]   7 140 296  38 164  43 123 346 123 306 205  81 185 216 148 236 185
##  [86] 304  87 272 200 276  25 103  53 314  99  19 309 327 147 194
```

```r
# plot abundance of species 1 for first 100 timesteps
plot(1:100,m[1,1:100],type="l",
     xlab="time",ylab="species abundance")
```



3.3 Calculating the Menhinick index

In this section we will create a function for calculating the Menhinick index. However, first we will start with calculating it for a single timestep the make sure it is working correctly. In order to do so, we will need to extract the first column, which contains all species abundances at t = 1. After getting all of the abundances for the first-time step `m_t1` we will need to calculate `N` and `n`. For the time being we will ignore most of the base R functions, this is done with the purpose of illustrating R performance considering multiple implementation methods. And after obtaining both `N` and `n`, we can proceed to compute `MI`.

```
# example of Menhinick's index at timestep = 1
m_t1 = m[,1] # extract abundaces at t = 1

# compute total number of individuals (N)
N <- 0
for(value in m_t1) {
  N <- N + value
}
print(N)
```

```
## [1] 1757192
```

```
# compute number of species (n) without using R functions
n <- 0
for(value in m_t1) {
  n <- n + 1
}
print(n)
```

```
## [1] 10000
```

```
# compute of Menhinick's index at t = 1
MI <- n / N ^ 0.5
print(MI)
```

```
## [1] 7.543804
```

3.4 Menhinick index: function 1

Next, we will create our function to easily calculate the Menhinick index for all timesteps. This function will implement the same method as we just created, it adds however a loop to loop over all time steps (columns) in our matrix.

```
MI_v1 <- function(m,n_timesteps) {

  MIs <- c() # create vector to store results
  # loop over ext_cols
  for(i in 1:n_timesteps) {

    ext_col <- m[,i]  # extract col from matrix

    # calculate N
    N <- 0
    for(value in ext_col) {
      N <- N + value
    }

    # calculate n
    n <- 0
    for(value in m_t1) {
      n <- n + 1
    }

    MI <- n / ( N ^ 0.5 ) # calculate MI
    MIs <- c(MIs,MI) # store MI in vector
  }

  return(MIs) # return results
}

# calculate just for time step 1
MI_v1(m,1)
```

```
## [1] 7.543804
```

```
# calculate for the first 10 time steps
MI_v1(m,10)
```

```
##  [1] 7.543804 7.544235 7.568745 7.546970 7.552347 7.538293 7.543679
##  [8] 7.543774 7.526534 7.551836
```

Now we are ready to test the performance of our function on the entire dataset:

```
print(n_timesteps)
```

```
## [1] 1000
```

```
microbenchmark(MI_v1(m, n_timesteps), times = 10 )
```

```
## Unit: milliseconds
##                   expr      min       lq     mean   median       uq
##  MI_v1(m, n_timesteps) 447.8825 451.2895 458.9715 458.9247 461.7376
##       max neval
##  480.0011    10
```

In the implementation above we violate some of the basic rules in writing efficient code.

1. We grow objects, in each loop the calculated MI is stored in MIs `MIs <- c(MIs,MI) # store MI in vector`, no vector is pre-allocated. This could be done by:

```
# allocating a vector with 0s
MIs <- c(0,0,0,0,0,0,0,0,0,0,0)
# replace the first 0 with result MI
MIs[1] <- 7.46
print(MIs)
```

```
##  [1] 7.46 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
```

This way R does not have to attach a new element to the vector at each iteration. The length of the vector will remain the same, the values within it will be replace by our results.

2. We do not consider any vectorized functions like calculating the `n` by using the function `sum()`, instead we are looping over our vector and adding up all values we encounter during the iteration.

3.5 Menhinick index: function 2

This implementation can be improved significantly by removing for example the inner loops and replacing them with regular base R functions. Besides, improving the performance it also increases the readability of our code. In addition to replacing most of the loops in our function we can also replace operators like `^ * +` by base R functions, in our case we replace `N ^ 0.5` by `sqrt(N)`, the performance difference is shown below.

```
# show impact of N^0.5 vs sqrt(N)
microbenchmark(sqrt(random_species_abundances),
               random_species_abundances^0.5,
               times = 10)
```

```
## Unit: milliseconds
##                              expr       min        lq      mean    median
##  sqrt(random_species_abundances)  69.16698  69.30056  70.10182  69.63933
##    random_species_abundances^0.5 702.27794 707.57103 719.63025 713.92481
##         uq       max neval
##   70.17823  74.44967    10
##  730.33128 753.72375    10
```

R functions perform better generally than manual use of arithmetic operators. Below the implementation of our improved function:

```
MI_v2 <- function(m,n_timesteps) {
```

```r
  # create vector to store results
  MIs <- c()

  # loop over ext_cols
  for(i in 1:n_timesteps) {

    ext_col <- m[,i]  # extract ext_col from matrix
    N <- sum(ext_col) # calculate N, loop replaced
    n <- length(ext_col) # calculate n, loop replaced
    MI <- n / sqrt(N) # calculate MI
    MIs <- c(MIs,MI) # store row sum in vector
  }

  # return results, vector with row sums
  return(MIs)
}

# MI_v1 and MI_v2 same results?
all.equal(MI_v1(m,n_timesteps),MI_v2(m,n_timesteps))
```

```
## [1] TRUE
```

```r
# show performance between functions: v1 and v2
microbenchmark(MI_v1(m,n_timesteps),
               MI_v2(m,n_timesteps),
               times = 10)
```

```
## Unit: milliseconds
##                   expr       min       lq      mean    median        uq
##  MI_v1(m, n_timesteps) 447.89524 448.6557 456.00642 453.19050 459.87643
##  MI_v2(m, n_timesteps)  29.40031  29.4662  35.69261  33.24082  37.72503
##        max neval
##  479.91794    10
##   53.10194    10
```

```r
# large performance increase
```

3.6 Menhinick index: function 3

This will be our final function. Because R performance is generally not that good when using loops, it is best to think as much as possible about methods that will stay away from loops. Performance of the majority of standard base R protocols is the highest when they involve a direct computation on a vector. Which makes for example dividing all elements in two vectors directly much faster than looping through all elements and dividing them one by one. The implementation below fully removes all loops from our function.

```
MI_v3 <- function(m,n_timesteps) {

  # calculate n for all cols
  n_all_ext_cols = nrow(m)
  # does assume all species every time present

  # calculate N for all cols
  N_all_ext_cols <- colSums(m)

  # calculate MI and return results
  return(n_all_ext_cols / sqrt(N_all_ext_cols))
}

# MI_v2 and MI_v3 same results?
all.equal(MI_v2(m,n_timesteps),MI_v3(m,n_timesteps))
```

```
## [1] TRUE
```

```
# yes!
```

Check the performance of our final function (compared to the other implementations)

```
# show performance between functions: v1, v2 and v3
microbenchmark(MI_v1(m,n_timesteps),
               MI_v2(m,n_timesteps),
               MI_v3(m,n_timesteps),
               times = 10)
```

```
## Unit: milliseconds
##                   expr        min         lq       mean     median
##  MI_v1(m, n_timesteps) 449.802093 451.696369 463.637858 459.930994
##  MI_v2(m, n_timesteps)  29.719764  32.248369  35.607471  32.284848
##  MI_v3(m, n_timesteps)   9.491938   9.496096   9.791414   9.513956
##          uq       max neval
##  469.620864 502.31369    10
##   39.920962  52.90319    10
##    9.534547  12.29682    10
```

conclusion: the way R functions are written combined with vectorization enables better performance

4 General do's and don'ts

When dealing with large datasets there are a few thinks to keep in mind. This starts already with the method we store our data e.g.:

```
# how is the performance considering different data types
head(m[,1:10]) # our matrix
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]  185   61  122  213  326  224  208  238   63   348
## [2,]   45  308   60    8   22  270   90  244   50   129
## [3,]  348  153  180  314  218  158   67  117  322   143
## [4,]  154  124   82  196  131  163  163  218  107    98
## [5,]  258  283  190   29   59   73  229  179   95    72
## [6,]  248   52    3  304  238   68  226  134  126   269
```

```
class(m)
```

```
## [1] "matrix"
```

```
df <- as.data.frame(m) # matrix to data.frame
class(df)
```

```
## [1] "data.frame"
```

```
head(df[,1:10]) # df has nice column names
```

```
##    V1  V2  V3  V4  V5  V6  V7  V8  V9 V10
## 1 185  61 122 213 326 224 208 238  63 348
## 2  45 308  60   8  22 270  90 244  50 129
## 3 348 153 180 314 218 158  67 117 322 143
## 4 154 124  82 196 131 163 163 218 107  98
## 5 258 283 190  29  59  73 229 179  95  72
## 6 248  52   3 304 238  68 226 134 126 269
```

The code below shows the difference in performance between for example a matrix and a similar data.frame

```
# show performance between v1, v2 and v3 for df and m
microbenchmark(MI_v1(m,n_timesteps),
               MI_v2(m,n_timesteps),
               MI_v3(m,n_timesteps),
               MI_v1(df,n_timesteps),
               MI_v2(df,n_timesteps),
               MI_v3(df,n_timesteps),
               times = 5)
```

```
## Unit: milliseconds
##                    expr        min         lq      mean     median
##   MI_v1(m, n_timesteps) 451.022577 451.813732 467.07602 462.744383
##   MI_v2(m, n_timesteps)  29.757642  31.729238  40.18199  33.832489
##   MI_v3(m, n_timesteps)   9.465679   9.490772   9.55359   9.518986
##  MI_v1(df, n_timesteps) 432.667759 432.780740 437.68453 438.513402
##  MI_v2(df, n_timesteps)  15.686000  15.752928  16.13643  15.779943
##  MI_v3(df, n_timesteps)  34.030798  34.482899  40.94444  36.445480
##          uq        max neval
##  465.219926 504.579479     5
##   34.312135  71.278452     5
##    9.537594   9.754921     5
##  441.546527 442.914237     5
##   15.818906  17.644396     5
##   49.704869  50.058174     5
```

Conclusion: if possible (all data has a similar format) store data in a matrix or vector. If not use data.frames. Although data.frames easier to use e.g. calling variables using names, combining data classes they are relatively slow when used in large computations.

Additionally, the method used to extract or subset data can greatly influence performance when performed regularly:

```
# different methods to extract the same data point:
df[32, 11]
```

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

```
df$V11[32]
```

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

```
df[[c(11, 32)]]
```

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

```
df[[11]][32]
```

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

```
.subset2(df, 11)[32]
```

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

```
# show performance between methods
microbenchmark(
  df[32, 11],
  df$V11[32],
  df[[c(11, 32)]],
  df[[11]][32],
  .subset2(df, 11)[32],
  times = 10
)
```

```
## Unit: nanoseconds
##                  expr   min    lq    mean  median    uq   max neval
##            df[32, 11] 16840 17831 21614.0 18074.5 19193 51873    10
##            df$V11[32]  1056  1156  2264.7  1335.5  1840  9742    10
##       df[[c(11, 32)]]  3689  3837  4788.7  4156.0  4556 10117    10
##          df[[11]][32]  3352  3739  8244.3  3978.0  5350 44774    10
##  .subset2(df, 11)[32]   232   266   339.3   296.0   316   775    10
```

Conclusion: for large datasets, think of the best approach to extract values or perform computations. Additionally, all simpler operations performed in a large function can add significantly to our computation time, even simple things like extracting a value.

```
# show performance for matrix compared to data.frame
microbenchmark(
  df[32, 11],
  m[32, 11],
  times = 10
)
```

```
## Unit: nanoseconds
##        expr   min    lq    mean  median    uq    max neval
##  df[32, 11] 16679 16778 32501.7 17687.5 19369 159275    10
##   m[32, 11]   152   196   814.2   255.0   366   5664    10
```

5 Brief introduction of fastest data reading and writing

In the section 'What is efficient code?' we discussed that part of efficient code also has to do with the amount of computer resources used. Although we won't go in to much detail on this subject (we won't discuss memory usage and processor utilization), something quite easily presentable is the amount of hard disk space we need to store our data. The `fst` package (https://www.fstpackage.org/) is one of many packages that allows for fast and small data writing. An example is given below:

```
# this code writes our data.frame df
dim(df)
```

```
## [1] 10000  1000
```

```
# we didn't import or export any data yet
# and haven't our working directory
#setwd('~/Desktop')

# first we try the regular implementation
system.time(write.csv(df,'simulated_data_LU10.csv'))
```

```
##    user  system elapsed
##   3.419   0.083   3.587
```

```
# for this we use system.time() around the function to check
# the time required for writing our csv
```

On my laptop this took 4.960 seconds, the `.csv` file is 37 mb. (in this case we use `system.time()` around the function to check the time required for writing our `.csv`).

```
# now we use the package fst to write our data set
system.time(write.fst(df,'simulated_data_LU10.fst'))
```

```
##    user  system elapsed
##   0.085   0.024   0.114
```

On my laptop this took 0.124 second, the `.fst` file is only 15.8 mb! In this case we both saved on computation time and resources! However, do take in mind that the newly created `.fst` file can only be loaded by people also software similar to R with implementations similar to the `.fst` package (e.g. THE `.fst` CAN NOT BE USED IN EXCEL OR SIMILAR SOFTWARE). Nonetheless, image data sets of multiple gb's, it can sometimes take more than 10 minutes to load or save a file!

```
# load/read both files and check if they are the same
csv_read_data = read.csv('simulated_data_LU10.csv')
fst_read_data = read.fst('simulated_data_LU10.fst')
# as you might have noticed the read.fst is also faster

# show contents of first 20 items in fist column
csv_read_data$V1[1:20]
```

```
##  [1] 185  45 348 154 258 248  74 105  58  52 230 237 246 188 173 175 330
## [18]  23 202 291
```

```
fst_read_data$V1[1:20]
```

```
##  [1] 185  45 348 154 258 248  74 105  58  52 230 237 246 188 173 175 330
## [18]  23 202 291
```

6 Brief introduction of parallel R code

Since All modern computers contain processors with more than 1 core, we can do multiple calculations at once! To illustrate the benefits of parallel computation we could do the following:

```
# split our matrix (m) in 4 pieces or chucks

# show size of matrix
dim(m)
```

```
## [1] 10000  1000
```

```
# set the number of columns per chuck
n_cols_to_split = ncol(m)/4

# split matrix
m1 = m[,1:n_cols_to_split]
m2 = m[,(n_cols_to_split+1):(n_cols_to_split*2)]
m3 = m[,(n_cols_to_split*2+1):(n_cols_to_split*3)]
m4 = m[,(n_cols_to_split*3+1):ncol(m)]

# put all chucks in a list
m_list = list(m1,m2,m3,m4)

# Simple version of MI_v2
MI_v2_simple <- function(m) {
  return(MI_v2(m,ncol(m)))
}

# apply function to all matrices in our list using lapply
result1 <- lapply(m_list,MI_v2_simple)

# apply function to all matrices in our list using lapply
result2 <- mclapply(m_list,MI_v2_simple,mc.cores=4)

# same result?
all.equal(result1,result2)
```

```
## [1] TRUE
```

```
# yes!

# is the parallel version faster
microbenchmark(
  lapply(m_list,MI_v2_simple),
  mclapply(m_list,MI_v2_simple,mc.cores=4),
  times = 10
)
```

```
## Unit: milliseconds
##                                          expr      min        lq      mean
##                  lapply(m_list, MI_v2_simple) 29.03638  40.19885  68.55576
##  mclapply(m_list, MI_v2_simple, mc.cores = 4) 86.02531 109.58783 148.63802
##     median        uq      max neval
##   54.00502  92.08787 127.8602    10
##  143.17915 192.15941 204.3175    10
```

```r
# no!?

# we need a larger data set
n_species <- 1e2 # set number of species to consider
n_timesteps <- 1e2 # set number of timesteps to consider
total_length <- n_species * n_timesteps # sample length

# create random datasets in list
# this list contains 10 chunks of random data
m_list = list()
for(i in 1:20) {
  m_list[[i]] <- matrix(sample(1:350,size = total_length, replace = T),
                        nrow = n_species,
                        ncol = n_timesteps)
}

# is running in parallel faster?!!
microbenchmark(
  lapply(m_list,MI_v2_simple),
  mclapply(m_list,MI_v2_simple,mc.cores=4),
  times = 1
)
```

```
## Unit: milliseconds
##                                          expr       min        lq
##                  lapply(m_list, MI_v2_simple)  3.653926  3.653926
##  mclapply(m_list, MI_v2_simple, mc.cores = 4) 40.949136 40.949136
##       mean    median        uq       max neval
##   3.653926  3.653926  3.653926  3.653926     1
##  40.949136 40.949136 40.949136 40.949136     1
```

```r
# yeah!!
```

7 Assignment

Will have more information once I know the assignment is okay. Code will be cleaned and included in the third document required once ready. The unfinished and not student friendly version of the code can be found in `LU10_Assignment1_2019-04-14.R`.

7.1 Create a hypothetical data set

Create a hypothetical data set in a `list()` format. In which:

- Each list entry contains a matrix and represents a abundance count at a specific timestep
- Columns in the matrix abundance counts for specific locations
- Rows are species-specific abundances for each of the locations (similar to the data set we created in the live coding example, however stored in a list format for different locations)

The following code can be used to create the testing dataset:

```r
n_locations <- 1e2 # set number of locations to consider
n_species <- 1e3 # set number of species to consider
n_timesteps <- 5  # set number of timesteps to consider

# create list in which
# - each list entry contains data for a single timestep
# - columns in each list entry contain abundance for a single location
# - rows in each list entry represent the species
abundance_list = list()
for(i in 1:n_timesteps){
  abundance_list[[i]] <- matrix(sample(1:350, size = n_species * n_locations, replace =
                                ncol=n_locations,
                                nrow=n_species)
}
```

7.2 Create a function to calculate the Shannon diversity index for a vector

Before moving to the entire data set we will create a function to calculate the Shannon diversity index for one specific location and one specific time step.

- Show how to extract data for a specific location and timestep. Store the extracted values in a vector.
- Calculate the Shannon diversity index for the extracted vector. The Shannon diversity index (H) is another index that is commonly used to characterize species diversity in a community. Shannon's index accounts for both abundance and evenness of the species present. The proportion of species is relative to the total number of species (pi) is calculated, and then multiplied by the natural logarithm of this proportion (lnpi). The resulting product is summed across species, and multiplied by -1. The equation for the Shannon diversity index can be found on: http://www.tiem.utk.edu/~gross/bioed/bealsmodules/shannonDI.html

7.3 Create a function to calculate the Shannon diversity index for an entire matrix

- Now modify the function created in the previous assignment to calculate the Shannon diversity index for all locations in the matrix of the first timestep (the entire first list entry).
- Is the function we just created for the entire matrix the fastest implementation you can think of?
- If so, consider the inefficient implementations we avoided
- If not, think about the various lines of code in our newly created function, which can be altered to increase performance?

7.4 Create a function to calculate the Shannon diversity index for all timesteps

- First try to make it as fast as possible without implementing the parallel function `mclapply()`
- How does performance differ when using either a for loop (`for(i in 1:length(list)){}`) or `lapply`?
- Once satisfied, try `mclapply()` on the list instead of the regular `lapply()`
- Is the parallel version faster?
- If not, increase the size of the simulated data, do we see a point at which our data set reaches a certain size and our parallel version becomes faster? _ If so, does this apply to smaller datasets as well?