R Course Lecture Notes

Marco Del Vecchio[1]

[1]marco.del.vecchio@outlook.com, www.marcodelvecchioblog.wordpress.com

Acknowledgements

I would like to thank all the execs of the Warwick MORSE society who have helped with the organisation of the course in the academic year 2016/2017.

Preface

This is the compel set of lecture notes that I have used to teach "The R Course" for the Warwick MORSE society in the academic year 2016/2017.

Contents

1 Operator Syntax

1.1 Assignment operators

Operator	Description	Associativity
<-, <<-, =	Leftwards assignment	Right to Left
->, ->>	Rightwards assignment	Left to Right

Note: The operators <- and = can be used, almost interchangeably, to assign to variable in the same environment. <<- is used for assigning to variables in the parent environments.

```
x <- 10
10 -> x
x = 10
```

1.2 Arithmetic Operators

Operator	Description	Associativity
+	Addition	Left to Right
-	Subtraction	Left to Right
*	Multiplication	Left to Right
/	Division	Left to Right
^	Exponent	Right to Left
%%	Modulus (Remainder from division)	Left to Right

```
x <- 10
y <- 2

x + y
## [1] 12
x - y
## [1] 8
x * y
## [1] 20
x / y
## [1] 5
x ^ y
## [1] 100
x %% y
## [1] 0
x %/% y
## [1] 5
```

Note: all operations on vectors are carried out in an element-wise fashion.

```
x <- c(1,2,3)
y <- c(10,20,30)

x + y
```

```
## [1] 11 22 33
x * y
## [1] 10 40 90
```

1.3 Relational operators

Operator	Description	Associativity
<	Less than	Left to Right
>	Greater than	Left to Right
<=	Less than or equal to	Left to Right
>=	Greater than or equal to	Left to Right
==	Equal to	Left to Right
!=	Not equal to	Left to Right

```
x <- 10
y <- 2

x < y
## [1] FALSE
x > y
## [1] TRUE
x <= y
## [1] FALSE
x >= y
## [1] TRUE
x == y
## [1] FALSE
x != y
## [1] TRUE
```

Note: all operations on vectors are carried out in element-wise fashion.

```
x <- c(1,2,3)
y <- c(10,20,30)

x > y
## [1] FALSE FALSE FALSE
x == y
## [1] FALSE FALSE FALSE
```

1.4 Logical operators

Operator	Description	Associativity
!	Logical NOT	Left to Right
&	Element-wise logical AND	Left to Right
&&	Logical AND	Left to Right
|	Element-wise logical OR	Left to Right
||	Logical OR	Left to Right

Note: Operators `&` and `|` perform element-wise operation producing result having length of the longer operand. `&&` and `||` examines only the first element of the operands resulting into a unit length logical vector.

```
x <- FALSE
y <- TRUE

!x
## [1] TRUE
x & y
## [1] FALSE
x | y
## [1] TRUE

x <- c(FALSE, TRUE, TRUE)
y <- c(TRUE, TRUE, FALSE)

!x
## [1]  TRUE FALSE FALSE
x & y
## [1] FALSE  TRUE FALSE
x && y
## [1] FALSE
x | y
## [1] TRUE TRUE TRUE
x || y
## [1] TRUE
```

1.5 Operator precedence

The order of precedence (highest first) of the operators is:

- `::`
- `$ @`
- `^`
- `- +` (unary)
- `:`
- `%xyz%`
- `* /`
- `+ -` (binary)
- `> >= < <= == !=`
- `!`
- `& &&`
- `| ||`
- `~` (unary and binary)
- `-> ->>`
- `=` (as assignment)
- `<- <<-`

```
# : precedes precedes binary +/-, but not ^, so
1:5-1 # does not yield 1,2,3,4 but,
## [1] 0 1 2 3 4
# and
1:2^2  # does not yield 1,4 but,
## [1] 1 2 3 4
```

```
# howver, : is precededed by unary +/- so,
-1:5 # does not yeild -1,-2,-3,-4,-5 but,
## [1] -1  0  1  2  3  4  5

# && precedes || so,
TRUE || FALSE && FALSE # is not equal to FALSE but,
## [1] TRUE

# = is preceded by <-. Consequently, this gives a non-catchable error:
x <- y = 5
## Error in x <- y = 5: could not find function "<-<-"
```

1.6 Precedence and associativity with (

If you would like to change the order in which operation are executed or the direction of associativity, you can do so by grouping expressions with (.

```
# We said that : precedes binary +/- however, we can change this.
# Compare
1:(5-1)
## [1] 1 2 3 4
# with
1:5-1
## [1] 0 1 2 3 4
# in the last expression, R is sicretly doing the following:
(1:5) - 1
## [1] 0 1 2 3 4
# We also said that # && precedes || but, again, we can change this.
(TRUE || FALSE) && FALSE
## [1] FALSE

# The same is true for the direction of associativity
10/10/2
## [1] 0.5
# In the above example, 10/10/2 is executed as (10/10)/2 due to left
# to right associativity of the / operator.
#However, this order too can be changed using parentheses.
10/(10/2)
## [1] 2
```

2 Data structures

R's base data structures can be organised along two dimensions: dimensionality (whether it is one-dimensional, two-dimensional or n-conditional) and homogeneity (whether all its components must be of the same type). The following table summaries R's base data structures according to this two features.

	Homogeneous	Heterogeneous
one-dimensional	Atomic vector	List
two-dimensional	Matrix	Data frame
n-dimensional	Array	

Given an object, we can learn more about its structure and properties by using the function `str()`.

2.1 Vectors

The simplest of these data structures is the vector object. Vectors come in two different forms: atomic vectors and lists. They differ in the types of their elements: all elements of an atomic vector must be of the same type (homogeneous), whereas the elements of a list can have different types (heterogeneous). They have three common properties:

- Type, which is accessible by using `typeof()` and outputs what the object is.
- Length, which is accessible by using `length()` and outputs how many elements it contains.
- Attributes, which is accessible by using `attributes()` and outputs any additional arbitrary metadata.

2.1.1 Atomic vectors

There are six data types of these atomic vectors, also termed as six classes of vectors. We usually create atomic vectors with `c()`, short for combine. Those are: logical (or boolean), numeric (or double), integer, character, complex and raw.

```
# Use TRUE or FALSE or T and F to create logical or so called boolean vectors.
logicalVector <- c(TRUE, FALSE, FALSE, T, F)
typeof(logicalVector)
## [1] "logical"
```

```
# Use real numbers to create numeric vectors.
numericVector <- c(1, .99, 12)
typeof(numericVector)
## [1] "double"
```

```
# We can use the suffix L to get an integer instead of a numeric vector.
IntegerVector <- c(1L, 2L, 3L)
typeof(IntegerVector)
## [1] "integer"
# Note the difference when excluding the L
typeof(c(1,2,3))
## [1] "double"
```

```
# A character object is used to represent string values.
characterVector <- c("R", "helloWorld", "3.14")
typeof(characterVector)
## [1] "character"
```

```
# Use the usual formatting for complex numbers to create a complex vector.
complexVector <- c(1 + 2i, 1.8 + 1.9i, 1i)
# Note how we need to put a 1 in front of i. This is beacuse if we do not,
# R will look for a variale named i.
typeof(complexVector)
## [1] "complex"
# The following produces an error as we have not specified that -1 is a
# complex number.
sqrt(-1)
## Warning in sqrt(-1): NaNs produced
## [1] NaN
# Here, NaN statds for Not a Number.
# Instead, we need to use the complex number -1 + 0i.
sqrt(-1 + 0i)
## [1] 0+1i
```

```
# The raw type is intended to hold raw bytes.
rawVector <- c(charToRaw("foo"))
# Which looks like
rawVector
## [1] 66 6f 6f
typeof(rawVector)
## [1] "raw"
```

Atomic vectors are always flat, even if you nest `c()`'s:

```
c(1,c(2,c(3)))
## [1] 1 2 3
# Which is the same as
c(1,2,3)
## [1] 1 2 3
```

2.1.1.1 Types and tests

We have already seen how we can determine the type of an object with `typeof()` now, we will look at how we can check if an object it's of a specific type. The function used to do so are: `is.logical()`, `is.numeric()`, `is.double()`, `is.integer()`, `is.character`, `is.complex()`, `is.raw`, or, more generally, `is.atomic()`.

```
is.logical(c(T, F))
## [1] TRUE
is.double(c(1.1, 2.9))
## [1] TRUE
is.integer(c(1L, 2L))
## [1] TRUE
# Note how
is.integer(c(1,2,3))
## [1] FALSE
# does not yield the expected output since we did not use the suffix L.
```

NB: `is.numeric()` is a general test for the "numberliness," if we may say so, of a vector and returns TRUE for both integer and double vectors. It is not a specific test for double vectors, which are often called numeric.

```
is.numeric(c(1.2,1.3,1.4))
## [1] TRUE
is.numeric(c(1L,2L,3L))
## [1] TRUE
```

2.1.1.2 Missing values

Missing values are specified with `NA`, which is a logical vector of length 1. `NA` will always be coerced to the correct type if used inside `c()`, alternatively, you can create `NAs` of a specific type with `NA_real_` (a double vector[2]), `NA_integer_` and `NA_character_`.

```
is.logical(NA)
## [1] TRUE
is.numeric(NA)
## [1] FALSE
is.double(NA_real_)
## [1] TRUE
is.integer(NA_integer_)
## [1] TRUE
# Here, since the NA is made of integers, the NA value will be coerced to an NA_integer_
str(c(1L, 2L, NA))
##  int [1:3] 1 2 NA
```

Further, we can use `is.na()` to test objects if they are NA, and`is.nan()` to test for NaN.

NB: an `NaN` is also a `NA` but the converse is not true.

```
is.na(NaN)
## [1] TRUE
is.nan(NA)
## [1] FALSE
```

2.1.1.3 Coercion

As it was stated at the beginning of this section, all elements of an atomic vector must be the same type, so when we attempt to combine different types they will be coerced to the most flexible type.

For example, when we try to create a vector which contains both character and numeric values, the numeric values will be coerced to characters as characters cannot always be cohered to numeric.

```
str(c(1, 2, "hello", "3"))
##  chr [1:4] "1" "2" "hello" "3"
str(c(1, 2, "3"))
##  chr [1:3] "1" "2" "3"
```

When a logical value is mixed with numeric ones, `TRUE` becomes 1 and `FALSE` becomes 0.

```
str(c(1, T, F, 3))
##  num [1:4] 1 1 0 3
```

NB: Coercion often happens automatically. You will usually get a warning message if the coercion might lead to an information loss.

We can also explicitly coerce variables with functions such as: `as.numeric`, `as.logical`, `as.character`, `as.double`, `as.integer`.

```
str(as.double("3.12"))
##  num 3.12
str(as.numeric(TRUE))
##  num 1
str(as.character(1))
##  chr "1"
```

[2]If you feel frustrated by the fact that we use real instead of double which sometimes is referred to as numeric... know that you are not alone.

Sometimes, R will not be able to perform coercion and this will result in `NAs` being produced.

```
str(as.numeric(c("a", "b", "c")))
## Warning in str(as.numeric(c("a", "b", "c"))): NAs introduced by coercion
##  num [1:3] NA NA NA
```

2.1.2 Lists

Lists are different from atomic vectors because their elements can be of any type, including lists. You construct lists by using `list()` instead of `c()`:

```
x <- list(c(1,2,3), 100, c(TRUE, FALSE, TRUE), list("hello", "world"))
str(x)
## List of 4
##  $ : num [1:3] 1 2 3
##  $ : num 100
##  $ : logi [1:3] TRUE FALSE TRUE
##  $ :List of 2
##   ..$ : chr "hello"
##   ..$ : chr "world"
# Note how the nested list is not being flattened as opposite to c(1,c(2)). For this
# reason, lists are sometimes called recursive vectors, because a list can contain
# other lists. So,
x <- list(list(list(list())))
# is a list containing a list conatining a list which itself conatins a list.
str(x)
## List of 1
##  $ :List of 1
##   ..$ :List of 1
##   .. ..$ : list()
# Further, we can check if a list is recursive with
is.recursive(x)
## [1] TRUE
```

Lists are a key data type in R and an in-depth knowledge of them will enable you to get the most out of R. You can check whether an object is list with `is.list()`.

It is possible to combine several lists into one with `c()`. However, given a combination of atomic vectors and lists, c() will coerce the vectors to list before combining them.

```
x <- c(list(1, c(T, F)), c("how", "are", "you"))
str(x)
## List of 5
##  $ : num 1
##  $ : logi [1:2] TRUE FALSE
##  $ : chr "how"
##  $ : chr "are"
##  $ : chr "you"
y <- list(list(1, c(T, F)), c("how", "are", "you"))
str(y)
## List of 2
##  $ :List of 2
##   ..$ : num 1
##   ..$ : logi [1:2] TRUE FALSE
##  $ : chr [1:3] "how" "are" "you"
```

```
# As you can see, when we created the variable x with the function c(), the vector
# c("how", "are", "you") was coerced into a list. In fact, we would have gotten the
# result if we typed
x <- c(list(1, c(T, F)), list("how", "are", "you"))
str(x)
## List of 5
##  $ : num 1
##  $ : logi [1:2] TRUE FALSE
##  $ : chr "how"
##  $ : chr "are"
##  $ : chr "you"
```

2.1.3 Initialising vectors

It is possible to initialise an atomic vector or a list of a prespecified length with `vector()`.

```
x <- vector( mode = "logical", length = 3)
x
## [1] FALSE FALSE FALSE
y <- vector( mode = "list", length = 3)
y
## [[1]]
## NULL
##
## [[2]]
## NULL
##
## [[3]]
## NULL
```

2.2 Attributes

All objects can have arbitrary additional attributes, used to store metadata about the object. Attributes can be accessed individually with `attr()` or all at once (as a list) with `attributes()`.

```
x <- c(1,2,3)
attributes(x)
## NULL
attr(x, "attribute1") <- "I am an atomic vector"
attr(x, "attribute2") <- "I am a numeric vector to be specific"
attributes(x)
## $attribute1
## [1] "I am an atomic vector"
##
## $attribute2
## [1] "I am a numeric vector to be specific"
```

As you can see, attributes can be very useful for writing readable code and self-describing objects. So, for instance, if we were to draw 10 samples from a normal distribution with mean 10 and sd 2 to then store them in a vector, and we wanted to "save" those information inside the vector itself, we would create attributes in the following way:

```
samples <- rnorm(n = 5, mean = 10, sd = 2)
samples
## [1]  8.583764 11.699110 10.514065 11.093554 12.586373
attr(samples, "distribution") <- "normal"
attr(samples, "mean") <- 10
attr(samples, "sd") <- 2
attributes(samples)
## $distribution
## [1] "normal"
##
## $mean
## [1] 10
##
## $sd
## [1] 2
```

By default, most attributes are lost when modifying a vector:

```
attributes(sum(samples))
## NULL
# Here, all the attributes created above were lost.
```

The only attributes not lost are the three most important:

- Names, a character vector giving each element a name.
- Dimensions, used to turn vectors into matrices and arrays.
- Class, used to implement the S3 object system.

Each of these attributes has a specific accessor function to get and set values. When working with these attributes, use `names(x)`, `class(x)`, and `dim(x)`, not `attr(x, "names")`, `attr(x, "class")`, and `attr(x, "dim")` as the latter will not work.

```
x <- c(T,F,F)
attr(x, "class")
## NULL
class(x)
## [1] "logical"
```

Most importantly, do not create a new attribute using the names "dim", "names" and "class" as this will overwrite the `names`, `dim` and `class` attributes which can be accessed with `names(x)`, `class(x)`, and `dim(x)`.

```
x <- c(T,F,F)
class(x)
## [1] "logical"
attr(x, "class") <- "character"
class(x)
## [1] "character"
# Yet, the type of the vector x is still logical
typeof(x)
## [1] "logical"
# which leads to all sorts of confusions.
```

2.2.1 Names

You can name a vector in three ways:

- When creating it: `x <- c("low" = 1, "medium" = 2, "high" = 3)`.
- By modifying an existing vector: `x <- 1:3; names(x) <- c("low", "medium", "high")`.
- By creating a modified copy of a vector: `x <- setNames(1:3, c("low", "medium", "high"))`.

In all three cases, the resulting vector will look like this:

```
##    low medium   high
##      1      2      3
```

Names need not be unique. However, names uniqueness is particularly important when sub-setting comes into play. Moreover, not all elements of a vector need to have a name. If some names are missing, `names()` will return an empty string for those elements. If all names are missing, `names()` will return `NULL`.

2.2.2 Factors

One major use of attributes is to define factors. A factor is a vector that can contain only predefined values, and is used to store categorical data. You can think of a factor has an integer vector where each integer has a label. In fact, they are built on top of integer vectors using two attributes: the `class()`, "factor", which makes them behave differently from regular integer vectors, and the `levels()`, which defines the set of allowed values (labels).

Factor objects can be created with the `factor()` function.

```
cities <- factor(c("Rome", "Milan", "Naples", "Florence"))
cities
## [1] Rome     Milan    Naples   Florence
## Levels: Florence Milan Naples Rome
class(cities)
## [1] "factor"
levels(cities)
## [1] "Florence" "Milan"    "Naples"   "Rome"

# One important concept to keep in mind is that a factor can only contain
# a predefined set of values. In fact, if we try to use a new value
# which is not in the levels, we get an error
cities[1] <- "London"
## Warning in `[<-.factor`(`*tmp*`, 1, value = "London"): invalid factor
## level, NA generated
cities
## [1] <NA>     Milan    Naples   Florence
## Levels: Florence Milan Naples Rome

# NB: you can't combine factors
class(c(factor("Rome"), factor("Milan")))
## [1] "integer"
```

Factors are useful when you know the possible values a variable may take, even if you do not see all values in a given dataset.

```
gender <- c("male", "male", "male")
genderFactor <- factor(gender, levels = c("male", "female"))
# Notice how we have specified that there are two levels (male and female) even though the
# vector of observations only contains males
table(genderFactor)
## genderFactor
```

```
##   male female
##      3      0
```

One should be careful when coercing a factor into a numeric vector.

```
x <- factor(c(0, 0.5, 1, 2.5, 1.5, 2), levels = c(0, 0.5, 1, 1.5, 2, 2.5, 3))
as.numeric(x)
## [1] 1 2 3 6 4 5
# As you can see, if we coerce a factor into a numeric vector straight away,
# we do not get the expected result. This is because, as it was stated earlier,
# factors are build on top of integer vectors and this means that every value in a
# factor is stored as an integer. In fact, if we "unclass" a factor, we can see
# its underlying representation.
unclass(x)
## [1] 1 2 3 6 4 5
## attr(,"levels")
## [1] "0"   "0.5" "1"   "1.5" "2"   "2.5" "3"
# Hence, the "correct" way to coerce a factor into a numeric, is to first coerce the
# factor into a character vector and then covert the character vector into a numeric one.
as.numeric(as.character(x))
## [1] 0.0 0.5 1.0 2.5 1.5 2.0
```

Furthermore, you should be careful when treating factors as characters. This is because, while factors look (and often behave) like character vectors, they are actually integers (as we have just seen). Hence, it is often recommended to explicitly convert factors to character vectors if you need string-like behavior.

2.3 Matrices and arrays

Adding a `dim()` (short for dimension) attribute to an atomic vector allows it to behave like a multi-dimensional array. The dimension attribute is itself an integer vector whose length determines the number of dimensions of the array. A special case of the array is the matrix which has two dimensions.

Matrices and arrays are created with `matrix()` and `array()` respectively:

```
# We can create a matrix with both the matrix() and array() functions.
m <- matrix(nrow = 2, ncol = 2)
m
##      [,1] [,2]
## [1,]   NA   NA
## [2,]   NA   NA
a <- array(dim = c(2,2))
a
##      [,1] [,2]
## [1,]   NA   NA
## [2,]   NA   NA
# As you can see, both matrices have been filled with NAs. This is because
# we did not specify what values should be used to fill them.
# To do so, we need to provide a vector of values to the matrix function.

# NB: Matrices are constructed column-wise, so entries can be thought of
# starting in the "upper left" corner and running down the columns.
m <- matrix(data = c(1:4), nrow = 2, ncol = 2)
m
##      [,1] [,2]
## [1,]    1    3
```

```
## [2,]    2    4

# However, we normally use the array function when we want to create a
# a data structure with more than two dimensions
a <- array(dim = c(2,2,2))
a
## , , 1
##
##      [,1] [,2]
## [1,]   NA   NA
## [2,]   NA   NA
##
## , , 2
##
##      [,1] [,2]
## [1,]   NA   NA
## [2,]   NA   NA
```

Matrices and arrays can also be created directly from vectors by adding the dimension attribute.

```
x <- c(1:6)
dim(x) <- c(3,2)
x
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
dim(x) <- c(2,3)
x
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

`length()` and `names()`have high-dimensional generalizations:

- `length()` generalises to `nrow()` and `ncol()` for matrices, and `dim()` for arrays.
- `names()` generalises to `rownames()` and `colnames()` for matrices, and `dimnames()` for arrays.

As we have previously seen, `c()` can be used to concatenate vectors. In the context of matrices and arrays, `c()` generalises to `cbind()` and `rbind()` for matrices, and to `abind()` (provided by the abind package) for arrays.

```
# Matrices can be created by column-binding with cbind().
x <- cbind(c(1,2,3),c(4,5,6), c(7,8,9))
x
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
# Or by row-binding with rbind().
y <- rbind(c(1,2,3),c(4,5,6), c(7,8,9))
y
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
```

```
# We can also append a new row or column with the same functions.
m <- matrix(data = c(1:4), nrow = 2, ncol = 2)
m
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
# Let's add a new column,
m <- cbind(m, c(5,6))
m
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
# and a new row
m <- rbind(m, c(1,1,1))
m
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
## [3,]    1    1    1
```

You can transpose a matrix with `t()`; the generalised equivalent for arrays is `aperm()`.

While atomic vectors are most commonly turned into matrices, the dimension attribute can also be set on lists to make list-matrices or list-arrays. In so doing, we can generate relatively esoteric data structures where elements do not need to be homogeneous.

```
l <- list(c("U.K", "U.S.A"), 3.1 + 2i, 10L, FALSE, matrix(1:4, ncol = 2, nrow =2 ), 0.99)
dim(l) <- c(2,3)
l
##      [,1]        [,2]  [,3]
## [1,] Character,2 10    Integer,4
## [2,] 3.1+2i      FALSE 0.99
# In this example, the list-matrix we have just created contains: a character vector,
# a complex number, an integer, a logical value, a double and a matrix!
```

2.4 Data frames

The data frame is a key data structure in statistics and in R. Data frames are build on top of a list of equal-length vectors. Each element of the list can be thought of as a column and its length as the number of rows in the data frame (hence the constraint that the vectors much be of equal length). Given these properties, a data frame is a 2-dimensional structure which shares some similarities with matrices and lists. It has `names()`, `colnames()`, and `rownames()`, even though `names()` yields the same result as `colnames()`. The `length()` of a data frame is the length of the underlying list and so is the same as `ncol()`; `nrow()` gives the number of rows. However, one important difference is that unlike matrices, data frames, takes to their underlying lists, can store different classes of objects in each column (they are heterogeneous).

2.4.1 Creation

We can explicitly create a data frame with `data.frame()` which takes named vectors as input:

```
df <- data.frame(name = c("John", "Jane", "Mary"),
                 age = c(52, 29, 25),
                 civilStatus = c("married", "single", "married"))
```

```
df
##   name age civilStatus
## 1 John  52     married
## 2 Jane  29      single
## 3 Mary  25     married
str(df)
## 'data.frame':    3 obs. of  3 variables:
##  $ name       : Factor w/ 3 levels "Jane","John",..: 2 1 3
##  $ age        : num  52 29 25
##  $ civilStatus: Factor w/ 2 levels "married","single": 1 2 1
```

As you can see from the `str()` output above, data.frame()'s default behavior is to turn strings into factors. To suppress this behavior we can use stringAsFactors = FALSE:

```
df <- data.frame(name = c("John", "Jane", "Mary"),
                 age = c(52, 29, 25),
                 civilStatus = c("married", "single", "married"),
                 stringsAsFactors = FALSE)
str(df)
## 'data.frame':    3 obs. of  3 variables:
##  $ name       : chr  "John" "Jane" "Mary"
##  $ age        : num  52 29 25
##  $ civilStatus: chr  "married" "single" "married"
```

2.4.2 Testing and coercion

To check whether an object is a data frame, you can use `class()` or test explicitly with `is.data.frame()`. You can coerce an object to a data frame with `as.data.frame()`:

- A vector will create a one-column data frame.
- A list will create one column for each element (this will result in an error if the elements are not of the same length)
- A matrix will create a data frame with the same number of columns and rows.

```
dfFromVector <- as.data.frame(x=c(1,2,3))
names(dfFromVector) <- c("var1")
dfFromVector
##   var1
## 1    1
## 2    2
## 3    3

dfFromList <- as.data.frame(list(c(1,2,3), c("a", "b", "c"), c(T,T,F)))
names(dfFromList) <- c("var1", "var2", "var3")
dfFromList
##   var1 var2  var3
## 1    1    a  TRUE
## 2    2    b  TRUE
## 3    3    c FALSE

dfFromMAtrix <- as.data.frame(matrix(1:9, ncol = 3, nrow = 3))
names(dfFromMAtrix) <- c("var1", "var2", "var3")
dfFromMAtrix
##   var1 var2 var3
```

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

2.4.3 Combining data frames

You can combine data frames using `cbind()` and `rbind()`. However, keep in mind that when combining data frames column-wise, only the number of rows must match. Whereas, when combining data frames row-wise, both the number of rows and the names of the columns must match.

```
# Combining column-wise:
df <- cbind(data.frame(var1 = c(1L,2L,3L), var2 = c("a","b","c")),
            data.frame(var3 = c(T,T,T)))
df
##   var1 var2 var3
## 1    1    a TRUE
## 2    2    b TRUE
## 3    3    c TRUE

# Combining row-wise
df <- rbind(data.frame(var1 = c(1L,2L,3L), var2 = c("a","b","c"), var3 = c(T,T,T)),
            data.frame(var1 = c(4L,5L), var2 = c("d","e"), var3 = c(F,T)))
df
##   var1 var2  var3
## 1    1    a  TRUE
## 2    2    b  TRUE
## 3    3    c  TRUE
## 4    4    d FALSE
## 5    5    e  TRUE
```

2.4.4 Exotic columns

Since the underlying structure of a data frame is a list, it is possible for a data frame to have column which is a list, matrix, or array. To do so when explicitly creating a new data frame, we have to use `I()`, which forces `data.frame()` to treat the list, matrix or array as one unit:

```
# We can make a list into a column in the following way
df <- data.frame(var1 = c("a", "b", "c"), var2 = I(list(1:1, 1:2, 1:3)))
str(df)
## 'data.frame':    3 obs. of  2 variables:
##  $ var1: Factor w/ 3 levels "a","b","c": 1 2 3
##  $ var2:List of 3
##   ..$ : int 1
##   ..$ : int  1 2
##   ..$ : int  1 2 3
##   ..- attr(*, "class")= chr "AsIs"
df
##   var1    var2
## 1    a       1
## 2    b    1, 2
## 3    c 1, 2, 3

# We can make a matrix into a column in the following way
```

```
df <- data.frame(var1 = c("a", "b", "c"), var2 = I(matrix(1:6, nrow = 3, ncol = 2)))
# In this case, we need to check that the number of rows of the matrix matches the
# number of rows of the data frame. If this is not the case, an error will occur.
str(df)
## 'data.frame':    3 obs. of  2 variables:
##  $ var1: Factor w/ 3 levels "a","b","c": 1 2 3
##  $ var2: 'AsIs' int [1:3, 1:2] 1 2 3 4 5 6
df
##   var1 var2.1 var2.2
## 1    a      1      4
## 2    b      2      5
## 3    c      3      6
# NB: the columns "var2.1" and "var2.2" are not part of the data frame's columns' names,
# they represent the the columns of the matrix. In fact,
names(df)
## [1] "var1" "var2"
# does not contain them.
```

As you have probably noticed from the `str()` output, `I()` adds the `AsIs` class to its input, however, this can usually be ignored without consequences.

3 Subsetting

3.1 Data types

3.1.1 Atomic vectors

Let us explore atomic-vector-subsetting with the following vector:

```
x <- c("a.1", "b.2", "c.3", "d.4")
# Note that the number after the full stop gives the original position in the vector.
```

There are five things that you can use to subset a vector.

3.1.1.1 Positive integers

Positive integers return elements at the specified positions:

```
x[c(1,2)]
## [1] "a.1" "b.2"

# Duplicate indices yield duplicate values
x[c(1,1,2,2)]
## [1] "a.1" "a.1" "b.2" "b.2"
```

3.1.1.2 Negative integers

Negative omit elements at the specified positions:

```
x[-c(1,2)]
## [1] "c.3" "d.4"

# NB: There are some restrictions on this. We cannot mix positive indices and
# negative indices in a single subsetting call
x[c(1,-2)]
## Error in x[c(1, -2)]: only 0's may be mixed with negative subscripts
```

3.1.1.3 Zero

Zero[3] returns a zero-length vector.

```
x[0]
## character(0)
```

3.1.1.4 Logical vectors

Logical vectors select elements where the corresponding logical value is `TRUE`.

```
x[c(TRUE, TRUE, FALSE, FALSE)]
## [1] "a.1" "b.2"
```

If the logical vector is shorter than the vector being subsetted, it will be recycled to be the same length.

[3] As you might have already noticed, the starting position of a vector/matrix/array/data frame is 1 and not 0

```
x[c(TRUE, FALSE)]
## [1] "a.1" "c.3"
# Yields the same result as:
x[c(TRUE, FALSE, TRUE, FALSE)]
## [1] "a.1" "c.3"
```

We can also use logical operators to generate the logical vector that will be used to perform the subsetting:

```
x[x == "a.1"]
## [1] "a.1"
# This is possible because R is a vectorised language. So, the command x == "a.1"
# returns a logical vector with as many values as there are in x and the result
# contains TRUEs and FALSEs according to the condition:
x == "a.1"
## [1]  TRUE FALSE FALSE FALSE

# We can also use multiple logical operators at once by concatenating them
# with &, |, or !
z <- c(10, 20, 30, 35, 40)
z[z > 25]
## [1] 30 35 40
z[z > 25 & z <= 30]
## [1] 30
z[z > 25 & z <= 30 & z!= 35]
## [1] 30
```

A missing value in the index always yields a missing value in the output:

```
x[c(TRUE, TRUE, NA, FALSE)]
## [1] "a.1" "b.2" NA
```

3.1.1.5 Nothing

Nothing returns the original vector.

```
x[]
## [1] "a.1" "b.2" "c.3" "d.4"
```

3.1.1.6 Character Vectors

Character vectors return elements with matching names. It goes without saying that this kind of subsetting only works if we have set attribute names in advance.

```
y <- setNames(x, c("element1", "element2", "element3", "element4"))
y
## element1 element2 element3 element4
##    "a.1"    "b.2"    "c.3"    "d.4"
y[c("element1", "element4")]
## element1 element4
##    "a.1"    "d.4"

# As with integer indices, repeated names yield repeated results.
y[c("element1", "element1")]
## element1 element1
##    "a.1"    "a.1"
```

```
# When subsetting with [, names must be matched exactly
y[c("elem")]
## <NA>
##   NA
```

3.1.2 Lists

Subsetting a list works in the same way as subsetting an atomic vector. However, using `[` will always return a list.

```
l <- list(1:5, c("a", "b", "c"), 10L)
l[1]
## [[1]]
## [1] 1 2 3 4 5
class(l[1])
## [1] "list"
```

In order to pull out its components, we will have to use `[[` or `$` which will be covered later.

3.1.3 Matrices and arrays

We can subset higher-dimensional structures like matrices and arrays in two ways: with multiple vectors or with a matrix.

Subsections of an array or matrix may be specified by giving a sequence of 1d index vectors (one for each dimension) separated by a comma.

```
m <- matrix(1:9, ncol = 3, nrow = 3)
rownames(m) <- c("row1", "row2", "row3")
colnames(m) <- c("col1", "col2", "col3")
m
##      col1 col2 col3
## row1    1    4    7
## row2    2    5    8
## row3    3    6    9
m[1:2, 1:3]
##      col1 col2 col3
## row1    1    4    7
## row2    2    5    8
m[-3, 1:2]
##      col1 col2
## row1    1    4
## row2    2    5

# We can also mix different kinds of indices. In this case, we are using
# logical subsetting to select the rows and character subsetting to select
# the columns.
m[c(T,F,F), c("col1", "col2")]
## col1 col2
##    1    4

# NB: By default, `[` will simplify the result to the lowest possible dimensionality:
class(m[1:3,1])
```

```
## [1] "integer"
# In this case, we extracted a vector so the resulting object is no longer a matrix
# compare this with:
class(m[1:3,1:2])
## [1] "matrix"
```

Blank subsetting now carries a meaning; missing indices enables us to access entire rows or columns of a matrix.

```
# We can select all the rows by leaving the first index blank
m[ ,1:2]
##      col1 col2
## row1    1    4
## row2    2    5
## row3    3    6
# and all the columns by leaving the second index blank
m[2:3, ]
##      col1 col2 col3
## row2    2    5    8
## row3    3    6    9
```

Let's see how we can subset a 3d array with a matrix.

```
# Let us create an array whose elements represent their coordinates.
a <- array(data = c("1,1,1", "2,1,1",
                    "1,2,1", "2,2,1",
                    "1,1,2", "2,1,2",
                    "1,2,2", "2,2,2"), dim = c(2, 2, 2))
a
## , , 1
##
##      [,1]    [,2]
## [1,] "1,1,1" "1,2,1"
## [2,] "2,1,1" "2,2,1"
##
## , , 2
##
##      [,1]    [,2]
## [1,] "1,1,2" "1,2,2"
## [2,] "2,1,2" "2,2,2"

# We now create a matrix where each row is the set of coordinates of
# the element we want to pull out.
select <- matrix (data = c(1,1,1,
                           2,2,2), nrow = 2, ncol = 3, byrow = TRUE)
select
##      [,1] [,2] [,3]
## [1,]    1    1    1
## [2,]    2    2    2
a[select]
## [1] "1,1,1" "2,2,2"
```

3.1.4 Data frames

Data frames posses the characteristics of both lists and matrices: if you subset them with a single vector, they behave like lists, if you subset them with two vectors they behave like matrices.

```
# Let us create an array whose elements represent their coordinates.
df <- data.frame(name = c("Lucy", "Alex", "john"),
                 age = c(16, 22, 20),
                 gender = c("female", "male", "male"),
                 stringsAsFactors = FALSE)

# There are two ways to select a column from a data frame:

# like a list
df[c("name", "age")]
##   name age
## 1 Lucy  16
## 2 Alex  22
## 3 john  20

# like a matrix
df[, c("name", "age")]
##   name age
## 1 Lucy  16
## 2 Alex  22
## 3 john  20

# The difference between them arises when we select only one column:
# subsetting like a matrix simplifies the output by default, subsetting
# like a list does not.
str(df["name"])
## 'data.frame':    3 obs. of  1 variable:
##  $ name: chr  "Lucy" "Alex" "john"
str(df[,"name"])
##  chr [1:3] "Lucy" "Alex" "john"
```

3.2 Other subsetting operators

Beside the subseting operator that we have seen so far -`[` - there are other two: `[[` and `$`. `[[` is similar to `[` but when applied, it returns only one value. `$` is a useful shorthand for `[[` when combined with character subsetting.

3.2.1 The `[[` operator

```
l <- list("a", "b", "c", "d", "e")
# We can use [[ to pull a single element out of a list:
l[[1]]
## [1] "a"
str(l[[1]])
##  chr "a"
# Notice the difference when we use [:
str(l[1])
```

```
## List of 1
##  $ : chr "a"

# If we name the elements, we can also use character subsetting:
l <- list("element1" = "a", "element2" = "b")
l[["element1"]]
## [1] "a"
```

3.2.2 The $ operator

`$` is a shorthand, where `data$col1` is equivalent to `data[["col1", exact = FALSE]]`. `$` is commonly used to access variables in a data frame.

```
df <- data.frame(name = c("Lucy", "Alex", "john"),
                 age = c(16, 22, 20),
                 gender = c("female", "male", "male"),
                 stringsAsFactors = FALSE)

df$name
## [1] "Lucy" "Alex" "john"
df[["name", exact = FALSE]]
## [1] "Lucy" "Alex" "john"
```

The difference between `[[` and `$` is that `$` does partial matching: a variable name need not to be specified in its whole:

```
# Here we can still acess the column gender even though we have not spelt
# its entire name.
df$gen
## [1] "female" "male"   "male"

# A thing which could not have been done if we had used [[:
df[["gen"]]
## NULL
```

3.2.3 Simplifying vs. preserving subsetting

Generally speaking, there are two kinds of subsetting: simplifying and preserving subsetting. Simplifying subsetting returns the simplest possible data structure that can be used to represent the output. Preserving subsetting, on the other hand, returns the output with the same data structure as the input.

The table below summaries the ways in which we can switch between the two types of subsetting.

	Simplifying	Preserving
Vector	`x[[1]]`	`x[1]`
List	`x[[1]]`	`x[1]`
Factor	`x[1:3, drop = T]`	`x[1:3]`
Array	`x[1, ] or x[, 1]`	`x[1, , drop = F] or x[, 1, drop = F]`
Data frame	`x[, 1] or x[[1]]`	`x[, 1, drop = F] or x[1]`

Now, whilst the preserving behavior is the same across all data types, simplifying behavior is not homogeneous. In fact, it has the following characteristics.

Atomic vector: removes names.

```
x <- c(Italy = "Rome", UnitedKingdom = "London" )

# Simplifying behaviour:
x[[1]]
## [1] "Rome"
str(x[[1]])
##  chr "Rome"

# Preserving behaviour:
x[1]
##  Italy
## "Rome"
str(x[1])
##  Named chr "Rome"
##  - attr(*, "names")= chr "Italy"
```

List: returns the object inside the list.

```
x <- list(Italy = "Rome", UnitedKingdom = "London" )

# Simplifying behaviour:
x[[1]]
## [1] "Rome"
str(x[[1]])
##  chr "Rome"

# Preserving behaviour:
x[1]
## $Italy
## [1] "Rome"
str(x[1])
## List of 1
##  $ Italy: chr "Rome"
```

Factor: drops any unused levels.

```
x <- factor(c("Rome", "London"))

# Simplifying behaviour:
x[1, drop = T]
## [1] Rome
## Levels: Rome
str(x[1, drop = T])
##  Factor w/ 1 level "Rome": 1

# Preserving behaviour:
x[1]
## [1] Rome
## Levels: London Rome
str(x[1])
##  Factor w/ 2 levels "London","Rome": 2
```

Array: if any dimension has length one, drops that dimension.

```
a <- matrix(1:2, nrow = 2, ncol = 1) # you can think of it as a 2d row vector

# Simplifying behaviour:
a[, 1]
## [1] 1 2
dim(a[, 1])
## NULL

# Preserving behaviour:
a[, 1, drop = F ]
##      [,1]
## [1,]    1
## [2,]    2
dim(a[, 1, drop = F ])
## [1] 2 1
```

Data frame: if the output is a single column returns a vector instead of a data frame.

```
df <- data.frame(Italy = c("Rome", "Milan"),
                 UnitedKingdom = c("London", "Edinburgh"),
                 stringsAsFactors = F)

# Simplifying behaviour:
df[,1]
## [1] "Rome"  "Milan"
str(df[,1])
##  chr [1:2] "Rome" "Milan"

df[[1]]
## [1] "Rome"  "Milan"
str(df[[1]])
##  chr [1:2] "Rome" "Milan"

# Preserving behaviour:
df[2]
##   UnitedKingdom
## 1        London
## 2     Edinburgh
str(df[2])
## 'data.frame':    2 obs. of  1 variable:
##  $ UnitedKingdom: chr  "London" "Edinburgh"

df[, 2, drop = F]
##   UnitedKingdom
## 1        London
## 2     Edinburgh
str(df[, 2, drop = F])
## 'data.frame':    2 obs. of  1 variable:
##  $ UnitedKingdom: chr  "London" "Edinburgh"
```

3.3 Subsetting nested data strucures

The [[operator can take an integer sequence if you want to extract a nested element of a list or data frame.

Consider a list:

```
l <- list(list(1,2,3), c("a", "b", "c"), matrix(1:4, nrow = 2, ncol = 2))

# Let us get the second element of the first element of the list.
l[[c(1,2)]]
## [1] 2

# Or, equivalently  (in a more human readable way)
l[[1]][[2]]
## [1] 2

# The same can be done with slightly  more complex nested elements such as a matrix:
# let us get the first row of the third element of the list (which is a matrix)
l[[3]][1, , drop = F]
##      [,1] [,2]
## [1,]    1    3
```

Consider a data frame:

```
df <- data.frame(a = c(1, 2, 3),
                 b = c("foo1", "foo2", "foo3"),
                 stringsAsFactors = F)

# Let us take the first element of the second column.
df[[2]][1]
## [1] "foo1"

# Or, equivaletly
df$b[1]
## [1] "foo1"
```

4 Control Structures

Control structures in R allow you to control the flow of execution of a series of R expressions. The most common and useful ones include:

- if-then-else
- for
- while

Control structures which might not be found in other programming languages but are featured in R are:

- repeat
- break
- next

4.1 If-then-else

This structure allows you to conditionally execute a group of statements, depending on the value of a logic expression and It is coded into the R language via the functions `if` and `else`:

```
if(<condition>){
  # Do something if the condition is true
} else{
  # Do something if the condition is false
}
```

It is possible to have a series of tests by following the initial if with any number of `else if`s.

```
if(<condition1>){
  # Do something if the condition1 is true
} else if(<condition2>){
  # Do something if the condition1 is false and condition2 is true
} else if(<condition3>){
  # Do something if the condition1 is false and condition3 is true
} else{
  # Do something if none of the conditions are true
}
```

Note: an `else if` statement is checked if and only if the `if` is false. As a case in point, see the following example:

```
number <- 30

if(number > 0){
  print("The number is greater than 0")
} else if(number > 10){
  print("The number is greater than 10")
} else{
  print("The number is less or equal to 0")
}
## [1] "The number is greater than 0"

# Here, the condition number > 10 has not been evaluated since the condition
# number > 1 is already true.
```

Note: the `else` clause is not necessary. It is possible to have a series of `if` clauses that always get executed if their respective conditions are true. Conversely, it is not possible to have an `else` clause without an `if` one.

4.2 For loop

The for loop is a control flow statement for specifying iteration, which allows code to be executed repeatedly, in R, it has the following syntax:

```
for(<variable> in <sequence>){
  # Do something
}
```

Consider the following trivial example:

```
for(i in 1:5){
  print(i^2)
}
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25

# Simply speaking, what the for function is doing is taking the i variable
# and in each iteration of the loop assigning it values 1, 2, 3, ..., 5, and
# executing the code within the curly braces (which in this case is squaring
# the variable i), and then exiting the loop.
```

A function which is often used together with `for()` is `seq_along()`. This function generates an integer sequence based on the length of an object. Suppose that you want to iterate through the elements of an atomic vector. We can do so in the following way:

```
x <- list(c(1:10), matrix(1:4, ncol = 2, nrow = 2), "hello")

for(i in seq_along(x)){
  print(x[i])
}
## [[1]]
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## [[1]]
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## [[1]]
## [1] "hello"
```

We can also iterate through a data structure in the following way:

```
for(element in x){
  print(element)
}
##  [1]  1  2  3  4  5  6  7  8  9 10
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
## [1] "hello"

# In this case, the variable element is no longer an iterator variable
```

```
# but it is a copy of the element inside the vector. Because of this however,
# we will end up consuming way more memory:
for(i in seq_along(x)){
  print(object.size(i))
}
## 48 bytes
## 48 bytes
## 48 bytes

for(element in x){
  print(object.size(element))
}
## 88 bytes
## 216 bytes
## 96 bytes
```

Nested loops are allowed:

```
# Suppose that I want to square each element of a matrix.
m <- matrix(1:8, ncol = 2, nrow = 4)
m
##      [,1] [,2]
## [1,]    1    5
## [2,]    2    6
## [3,]    3    7
## [4,]    4    8
for(i in 1:dim(m)[1]){
  for(j in 1:dim(m)[2]){
    m[i,j] <- m[i,j]^2
  }
}
m
##      [,1] [,2]
## [1,]    1   25
## [2,]    4   36
## [3,]    9   49
## [4,]   16   64

# Of course, the example above is only illustrative. I could
# have done the exact same thing with a lot less code and effort:
matrix(1:8, ncol = 2, nrow = 4)^2
##      [,1] [,2]
## [1,]    1   25
## [2,]    4   36
## [3,]    9   49
## [4,]   16   64

# Note: even though we are using the variables i and j inside the for
# function, they will not be removed once the loop terminates:
i; j;
## [1] 4
## [1] 2
```

4.3 While loop

While loops begin by testing a condition. If it is true, then they execute the loop body. Once the loop body is executed, the condition is tested again, and so forth, until the condition is false, after which the loop exits. In R, it has the following syntax:

```
while(<condition>){
  # Do something
}
```

Note: While loops can potentially result in infinite loops if not written properly.

```
counter <- 1

while(counter < 5){
  counter <- counter + 2
  print(counter)
}
## [1] 3
## [1] 5
```

4.4 Next and break

These two commands are used in conjunction with loop function. `next` allows you to skip an iteration of a loop, and `break` is used to exit a loop.

```
for(i in 1:20){
  if(i %% 2 == 0){
    next
  }else{
    print(i)
  }
}
## [1] 1
## [1] 3
## [1] 5
## [1] 7
## [1] 9
## [1] 11
## [1] 13
## [1] 15
## [1] 17
## [1] 19
# Here, we skip all the iteration where i is an even number

for(i in 1:20){
  if(i %% 10 == 0){
    break
  }
  print(i)
}
## [1] 1
## [1] 2
## [1] 3
## [1] 4
```

```
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
# Here instead, we exit the loop as soon as i is divisible by 10
```

4.5 Case study: convergence of sequences

Given a real-valued sequence $\{a_n\}_{n\in 1,2,...}$ consider the following definition of convergence to a limit l: Given $\epsilon > 0$ and $N \in \mathbb{N}$, we say that

$$\lim_{n\to\infty} a_n = l$$

if

$$|a_i - l| < \epsilon, |a_{i+1} - l| < \epsilon, \ldots, |a_{i+N} - l| < \epsilon,$$

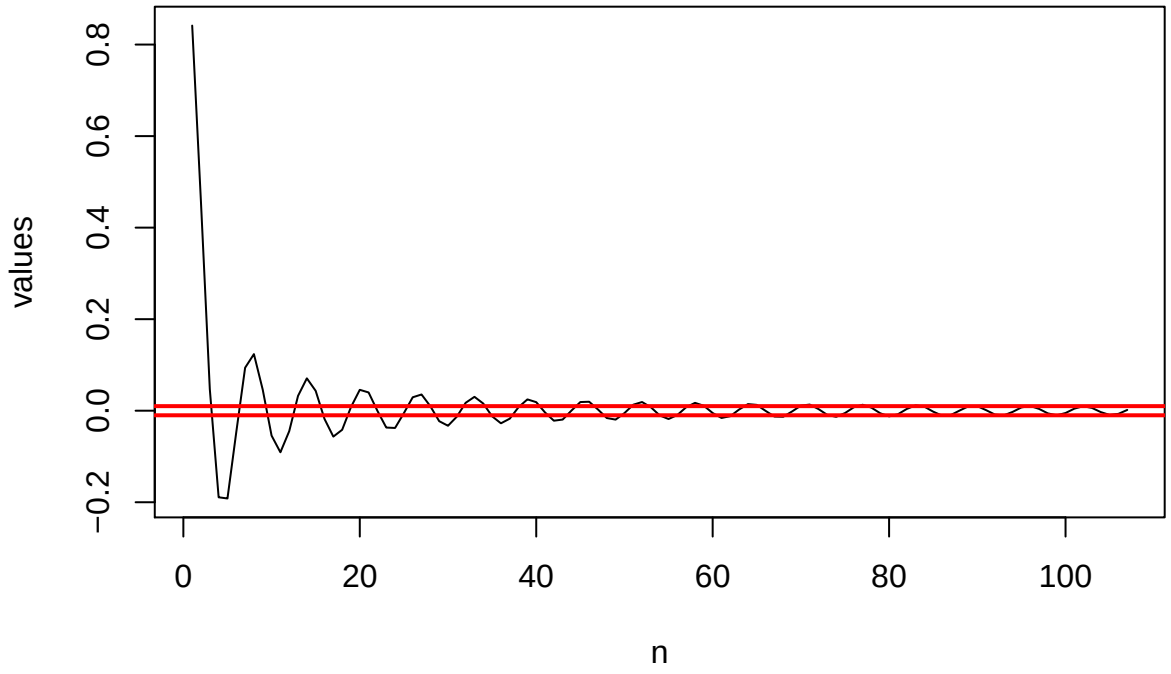
for $i \in \mathbb{N}$ s.t. $i >= N$

```
convergence_to_value_test <- function(f, epsilon, limit, tail){
  converged <- FALSE
  n <- 1
  values <- c()
  tail <- 10
  max_iteration <- 1000
  while(converged == FALSE && n <= max_iteration){
    values <- c(values, f(n))
    if(n >= tail){
      tmp <- sum(abs(values[(length(values) - tail):length(values)] - limit) < epsilon)
     if(tmp >= tail){
      converged <- TRUE
     }
    }
    n <- n + 1
  }
  plot(x = 1:(n-1), y = values, type = "l", xlab = "n")
  abline(h = (limit + epsilon), lwd=2, col="red")
  abline(h = (limit - epsilon), lwd=2, col="red")
  if(converged) {
    print(paste0("the series converges to ", limit))
  } else{
    print(paste0("the series does not converge to ", limit))
  }
}
```

Let us test this function on a couple of sequences:

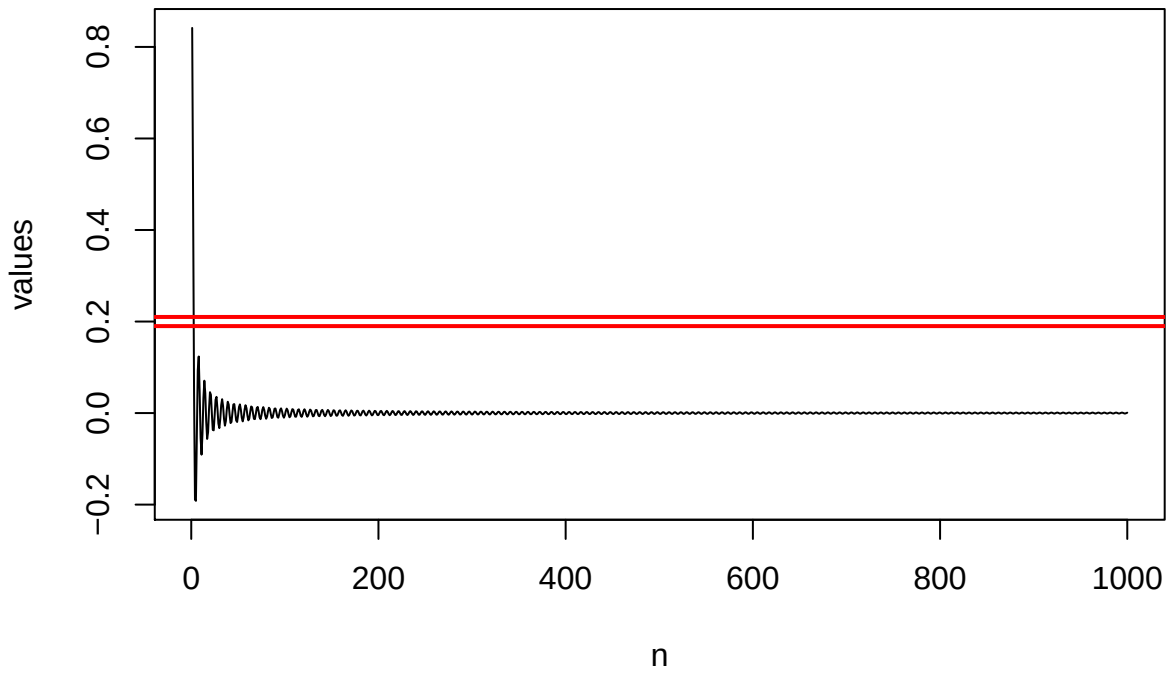
Consider $a_n = \frac{1}{n} sin(n)$

```
convergence_to_value_test(f = function(n) (1/n)*sin(n),
                          epsilon = 1e-2,
                          limit = 0,
                          tail = 10)
```




```
## [1] "the series converges to 0"
```

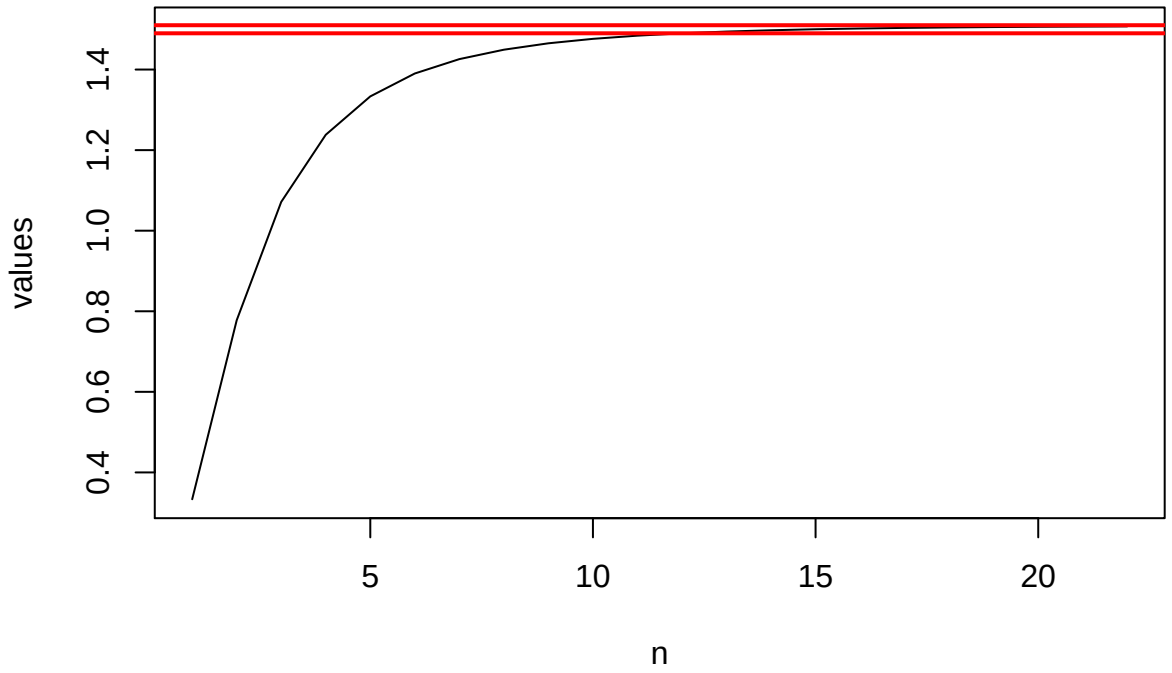
```
convergence_to_value_test(f = function(n) (1/n)*sin(n),
                          epsilon = 1e-2,
                          limit = 0.2,
                          tail = 10)
```



```
## [1] "the series does not converge to 0.2"
```

Consider $a_n = \frac{3n^2+n}{2n^2+10}$

```
convergence_to_value_test(f = function(n) (3 * n^2 + n)/(2 * n^2 + 10),
                          epsilon = 1e-2,
                          limit = 3/2,
                          tail = 10)
```




```
## [1] "the series converges to 1.5"
```

5 Functions

Functions in a R are object in their own right. In fact, a function is an object whose class is the class `function`. Importantly, the reader will find it helpful to remember that in R, "everything that exists is an object and everything that happens is a function call." With this being said, we remark two properties that R functions have:

- Functions can be passed to other functions as arguments (this should not be surprising since we have already said that functions are indeed objects).
- Functions can be nested.

5.1 Function definition

In R, we can define a function using `function()` :

```
foo <- function(<arguments>){
  # function body
}
```

5.2 Function components

Every R functions - except primitives, more on that later - have three parts:

- The `body()`, which is the code inside the function.
- The `formals()`, which is the list of argument of the function.
- The `environment()`, which is the map of the location of the function's variables.

5.3 Primitive functions

As it has been stated earlier, there is one exception to the rule that all functions have three components: Primitive functions - like `length()` or `sum()`. Those functions do not contain any R code but, call C code directly with `.Primitive()`. Therefore, their `formals()`, `body()`, and `environment()` are `NULL`.

```
length
## function (x)  .Primitive("length")
body(length)
## NULL
formals(length)
## NULL
environment(length)
## NULL
```

5.4 Return Values

In R, the last expression evaluated in the function body becomes the return value. As a consequence, we are not bound to use `return()` to specify what the output of the function needs to be[4].

Note: functions can return only a single object. However, this is not a limitation because, we could return a list which contains as many object as we please.

[4]Even though we do not need to do so, in my opinion is good practice to use `return()` to make very explicit to whoever is reading our code what that the function is returning

```
foo <- function(x){
  if(x > 0){
    "The number is greater than 0"
  } else{
    "The number is not greater than 0"
  }
}

foo(-1)
## [1] "The number is not greater than 0"
foo(1)
## [1] "The number is greater than 0"

# We could have used the function return() to obtain the same result

foo <- function(x){
  if(x > 0){
    return("The number is greater than 0")
  } else{
    return("The number is not greater than 0")
  }
}

foo(-1)
## [1] "The number is not greater than 0"
foo(1)
## [1] "The number is greater than 0"
```

5.5 Lexical Scoping

Generally speaking, scoping is the set of rules which allow R to bind values to symbols. In the example below, scoping is the set of rules that R applies to go from the symbol `v` to its value 30.

```
v <- 30
v
## [1] 30
```

There are four basic principles behind R's implementation of lexical scoping:

- name masking
- functions vs. variables
- a fresh start
- dynamic lockup

5.5.1 A diversion on R's environments

We have described local scoping as the set of rules that binds values to symbols, however, where does R actually look for this "mapping" ? The answer is environments: the data structure that powers scoping.

We can think of an environment as a bag of names where each name points to an object stored elsewhere in memory.

The objects do not live in the environment so, multiple names can point to the same object, which may or may not have the same value.

Every environment has a parent environment - apart from the empty environment. The parent environment. is used to implement lexical scoping: if a name is not found in an environment, then R will look one level up - in its parent - and so on.

Note: we do not speak of children environment for a reason: given an environment there is no way to find its children.

More formally, each environment is made up of two components:

- the frame, which contains the name-object mapping
- the parent environment.

There are four special environments:

- The `globalenv()`, or global environment, is the interactive work space. Its parent is the last package that you attached with `library()` or `require()`.
- The `baseenv`, or the base environment, is the environment of the base package. Its parent is the empty environment.
- The `emptyen()`, or empty environment, is the ultimate ancestor of all environments, and therefore, does not have a parent.
- The `environment` is the current environment.

`search()` lists all parents of the global environment.

5.5.2 Name masking

```
# Here, both variables a and b were defined inside the function. Therefore, R did not
# have to look in any other environment but in the one where the function was defined
# in order to return the vector (a,b)
f <- function(){
  a <- 1
  b <- 2
  return(c(a,b))
}
f()
## [1] 1 2
rm(f)

# In this case the function has to return the triplet (a,b,c), however, the variable c
# is not defined inside the function which is returning it. As a consequence, R will
# look one level up; all the way to the globalenviroment and further - like loaded
# packages - if necessary.
c <- 3
f <- function(){
  a <- 1
  b <- 2
  return(c(a,b,c))
}
f()
## [1] 1 2 3
rm(f)

# The same principle applies to nested functions: functions created inside
# other functions

f <- function(){
```

```
  a <- 1
  g <- function(){
    b <- 2
    h <- function(){
      c <- 3
      return(c(a,b,c))
    }
    return(h())
  }
  return(g())
}

f()
## [1] 1 2 3
rm(f)
```

5.5.3 Functions vs. variables

The same principle described above applies to functions with only one tweak to the rule: if you are using a name in a context where it is obvious that you are referring to a function - e.g. `f()` - then R will ignore objects that are not functions when searching.

```
# In this example, the symbol n points to two distinct objects: n() points to
# the function whose name is n and n points to the variable called n.
n <- function(x){
  return(log10(x))
}
f <- function(){
  n <- 100
  return(n(n))
}

f()
## [1] 2
rm(f,n)
```

5.5.4 A fresh start

In R, every time a function is called, a new environment is created to host execution. As a consequence, each invocation is completely independent from the previous ones.

```
# As you can see here, even though the first time we call the function, a new variable,
# x, is created, as soon as the execution of the function is terminated, the environment
# where x is stored is destroyed.

f <- function(){
  if(!exists("x")){
    x <- 1
  } else{
    x <- x + 1
  }
  return(x)
}
```

```
f()
## [1] 1
f()
## [1] 1
rm(f)
```

5.5.5 Dynamic lookup

In R, the value of an object inside a function is looked-up when the function is run, not when it is created.

```
# As you can see here, even though the first time we call the function, a new
# variable, x, is created, as soon as the execution of the function is terminated,
# the environment where x is stored is destroyed.
x <- 10
f <- function(){
  return(x)
}

x <- 20
f()
## [1] 20
rm(f)
# In fact, although, when f was created the value of x was 10, by the time
# it was run, it changed to 20.
```

5.6 R and function calls

In R, every operation is a function call being this infix operators like `+` and `*`, control flow operators like `for` and `while`, subsetting operators like `[` and `$`, and `{`.

Thus, each pair of function calls in the examples below are absolutely equivalent.

```
# Note: the `, the backtick, let's you refer to functions or variables whose
# names are reserved or illegal.
3 * 2
## [1] 6
`*`(3,2)
## [1] 6

for(i in 1:3) print(i)
## [1] 1
## [1] 2
## [1] 3
`for`(i, 1:3, print(i))
## [1] 1
## [1] 2
## [1] 3

if(i == 1) print("yes") else print("no")
## [1] "no"
`if`(i==1, print("yes"), print("no"))
## [1] "no"
```

```
x <- c(1,2,3)
x[2]
## [1] 2
`[`(x,2)
## [1] 2
```

5.7 Function arguments

It is worth distinguishing between the formals or formal arguments, which are a property of the function itself and the actual arguments with which the function is invoked. In this section, we will discuss how actual arguments are matched to formals.

5.7.1 Calling functions

When calling a function, one can specify arguments by:

- exact name
- partial name
- position

Note: R will try to match the arguments in the order given above.

```
f <- function(x, yx, xyz){
  return(c(x,yx,xyz))
}

# In this case, R will use positional matching as neither exact nor partial names
# were provided.
f(1, 2, 3)
## [1] 1 2 3

# Here, names are matched  by exact name.
f(x = 1, yx = 2, xyz = 3)
## [1] 1 2 3

# Here, the arguments xz and xy are matched  by exact name, and x by position.
f( 1, xyz = 3, yx = 2)
## [1] 1 2 3

# Here, partial name matching does not work because there are two arguments which start
# by the letter x. As a result, the argument xx has not been matched whilst the other
# two were matched by position.
f(x = 1, yx = 2, x = 3 )
## Error in f(x = 1, yx = 2, x = 3): formal argument "x" matched by multiple actual arguments

rm(f)
```

5.7.2 Calling a function given a list of arguments

Suppose that we had a list of function arguments.

```
args <- list(n = 5, min = 0, max = 1)
```

In order to call a function given a list of arguments, we need `do.call()`

```
do.call(runif, args)
## [1] 0.89552472 0.52612402 0.02118261 0.54147344 0.83113382
```

5.7.3 Lazy evaluation

By default, R function arguments are only evaluated if they are used inside the function.

```
# In this case, the function will not stop and return an error because the argument
# y is not actually used within the function.
f <- function(x,y) {return(x)}
f(10)
## [1] 10

# Compare this above code with the one below:
f <- function(x,y) {return(c(x,y))}
f(10)
## Error in f(10): argument "y" is missing, with no default

rm(f)

# This is particularly useful in if statements:

x <- NULL
if(!is.null(x) && x > 100) print("Ok")
# Here, in order for the if statement to be true, both conditions have to be satisfied.
# In this case, x is NULL which means that no matter what the second condition (x > 100)
# evaluates to, the if statement will be false. Yet, if R were to evaluate the
# second condition, we would get an error in return (since saying NULL > 100 does not
# make any sense). But, since arguments are evaluated lazily, the second statement  is not
# actually used, so we do not get an error.

# compare the above code with:
if(!is.null(x) || x > 100) print("Ok")
## Error in if (!is.null(x) || x > 100) print("Ok"): missing value where TRUE/FALSE needed
```

5.7.4 Default and missing arguments

In R, functions can have default values

```
f <- function(x = 1, y){
  return(c(x,y))
}
f(y = 2)
## [1] 1 2
# We can also overwrite the default argument by specifying the new value when
# calling a function:
f(x = 2, y = 2)
## [1] 2 2
```

```
rm(f)
```

Further, given that R uses lazy evaluation, the default value of an argument can be defined in terms of other arguments, or in terms of variables defined inside the function:

```
f <- function(x, y, z = sum(x,y)){return(c(x,y,z))}

f(2,3)
## [1] 2 3 5
rm(f)

f <- function(x, y, z = w){
  w <- x * y
  return(c(x,y,z))
}

f(2,3)
## [1] 2 3 6
rm(f)
```

We can determine if an argument was supplied or not via `missing()`.

```
f <- function(x, y, z){
  return(c(missing(x),
           missing(y),
           missing(z)))
}
f(1,2,3)
## [1] FALSE FALSE FALSE
f(1,2)
## [1] FALSE FALSE  TRUE
f(1)
## [1] FALSE  TRUE  TRUE

rm(f)
```

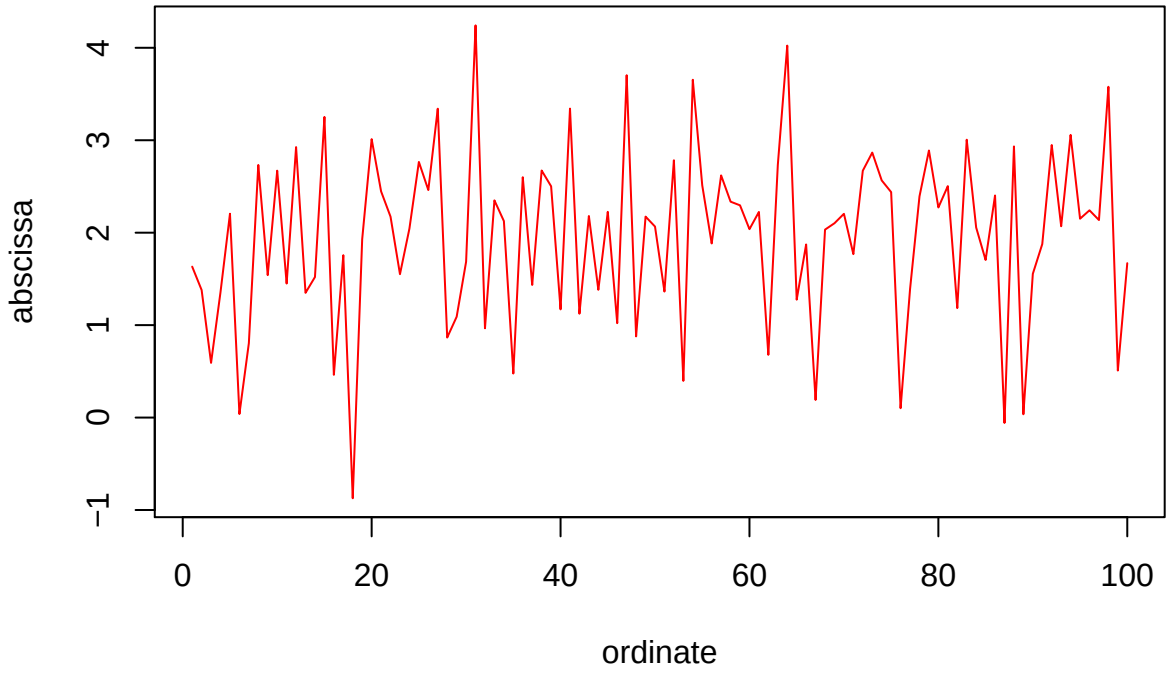
5.7.5 ...

There is a special argument in R known as the ... argument, which indicate a variable number of arguments that are usually passed on to other functions.

One of the classic examples of the ... argument in use id the `plot`:

```
plot
## function (x, y, ...)
## UseMethod("plot")
## <bytecode: 0x7fcf4a6a8380>
## <environment: namespace:graphics>
# As you can see, the plot function needs to be invoked with two arguments x and y, which
# are the x and y coordinates.

# If we wanted to create a wrapper function which plots only lines we can do so in
# the following way.

plotLine <- function(x, y, type = "l", ...){
```

```
  return(plot(x, y, type = type, ...))
}

plotLine(1:100,
         rnorm(100, mean = 2, sd = 1),
         col = "red",
         xlab = "ordinate",
         ylab = "abscissa" )
```



Two catches with ... is that one, any arguments that appear after ... on the argument list must be named explicitly and cannot be partially matched or matched positionally, and two, any misspelled argument will not raise an error.

5.8 Special calls

R supports two additional syntaxes for calling two special types of functions: infix and replacement functions.

5.8.1 Infix functions

Most R functions are prefix operators, meaning that the function's name comes before the list of arguments. It is also possible to create infix functions where the function name comes in between its arguments. Examples of such functions are: `+`, `-`, `*`, `<`, `>`, etc.. All users-created infix function must start and end with `%`. Let us see an example:

```
# In Python, for instance, when we want to concatenate strings together, we can do
# so by simply using the + operators. The same can be achieved in R if we define a
# new + operator in the following way:

`%+%` <- function(x,y) paste0(x, y)
```

```
"Hello" %+% " " %+% "World"
## [1] "Hello World"
```

5.8.1.1 Infix operators application: piping with `magrittr`

Some programming languages, i.e, F#, have what is called a forward pipe operator. This operator, allows you to pipe a value forward into an expression or function call; something along the lines of `x -> f()`, rather than `f(x)`. Unfortunately, R does not come with a built-in pipe operator, however, thanks to a gentleman called Stefan Milton Bache, there now is a package called `magrittr`, which according to the author must be pronounced with a sophisticated french accent, that provides this exact feature.

The operator introduced by the `magrittr` package has the following syntax: `%>%`. Let us see it in action.

```
library(magrittr)
mean_mpg <- mtcars %>%                      # get the mtcars dataset about cars.
  subset(cyl > 6 & hp > 100 & wt < 4) %>% # subset it so that we only have cars
                                            # with 6 cylinders and 100 HP
  `$`(mpg) %>%                              # extract the miles per gallon column
  mean() %>%                                # calcualte its mean value
  round(0)                                  # round it to the nearest integer

mean_mpg
## [1] 16
```

The same result could have been achieved in the following, less human readable way :

```
mean_mpg <- round(mean(subset(mtcars,
                              cyl > 6 & hp > 100 & wt < 4)
                       $mpg),
                  0)

mean_mpg
## [1] 16

# Or even worst
car_df_subset <- subset(mtcars, cyl > 6 & hp > 100 & wt < 4)
mean_mpg <- round(mean(car_df_subset$cyl), 0)

mean_mpg
## [1] 8
```

5.8.2 Replacement functions

Replacement functions allow you to modify their argument in place. They can have as many arguments as you wish but, the most used are `x` and `value`, and any additional one has to be placed between them.

```
`modify<-` <- function(x, position, value){
  x[position] <- value
  return(x)
}

v <- c(1,2,3)
modify(v,2) <- 10
v
## [1]  1 10  3
```

```
# Note: behind the scene, R turns modify(v,2) <- 10 into
# v <- `modify<-` (v,2,10)
```

5.9 Copy-on-modify behaviour

In R, modifying a function argument does not change the original value

```
x <- 1
foo <- function(x){
  x <- 2
  return(x)
}

foo(x)
## [1] 2
x
## [1] 1
```

6 Import Export

6.1 Spreadsheet-like data

6.1.1 Import

The function `read.table` is the most convenient way to read in a rectangular grid of data. Some of the issues to consider when using this functions are:

- **Encoding**: this can be specified by the `fileEncoding` argument, for example
  `fileEncoding = "UCS-2LE"  # Windows 'Unicode' files`
  `fileEncoding = "UTF-8"`
- **Header line**: if the first record in the file is a header record containing column (field) names then, we can import those names by setting `header = TRUE`
- **Separator**: normally looking at the file will determine the field separator to be used. With white-space separated files there may be a choice between the default `sep = ""` which uses any white space (spaces, tabs or newlines) as a separator, `sep = " "` and `sep = "\t"`.
- **Quoting**: by default character strings can be quoted by either `"` or `'`, and in each case all the characters up to a matching quote are taken as part of the character string. The set of valid quoting characters (which might be none) is controlled by the quote argument. For `sep = "\n"` the default is changed to `quote = ""`.
- **Missing values**: by default the file is assumed to contain the character string `NA` to represent missing values, but this can be changed by the argument `na.strings`, which is a vector of one or more character representations of missing values. Empty fields in numeric columns are also regarded as missing values. Note: in numeric columns, the values `NaN`, `Inf` and `-Inf` are accepted.
- **Unfilled lines**: sometimes, a file exported from a spreadsheet will have all trailing empty fields (and their separators) omitted. To read such files set `fill = TRUE`.
- **White space in character fields**: if a separator is specified, leading and trailing white space in character fields is regarded as part of the field. To strip the space, use argument `strip.white = TRUE`.
- **Blank lines**: By default, read.table ignores empty lines. This can be changed by setting `blank.lines.skip = FALSE`, which will only be useful in conjunction with `fill = TRUE`, for instance to use blank rows to indicate missing cases.
- **Classes for the variables**: unless you take any special action, `read.table` reads all the columns as character vectors and then tries to select a suitable class for each variable. It tries in turn logical, integer, numeric and complex, moving on if any entry is not missing and cannot be converted. If all of these fail, the variable is converted to a factor. If you want to control directly which class is assigned to which column then you can use `colClasses`. This argument allows the user to specify a character vector indicating the class of each column in the file. The classes you can specify are: factor, character, integer, numeric, logical, complex, and Date. Dates that are in the form `%Y-%m-%d` or `Y/%m/%d` will import correctly. There is also another argument `as.is` that when is set to `TRUE` will suppress conversion of character vectors to factors (only).

A general read.table call would look like this:

```
df <- read.table("<FileName>.<FileFormat>",
                 header = FALSE,
                 sep=",",
                 strip.white = TRUE,
                 na.strings = "NA",
                 colClasses = c("factor",
                                "character",
```

```
                                "integer",
                                "numeric",
                                "character"))
```

Because of the many possibilities, there are several other functions that call `read.table` but change a group of default arguments. Those are:

- `read.csv`: this function is primarily used to read `.csv` files (Comma Separated Values), and therefore, has the following default arguments: `sep = ","`, `dec = "."`, `header = TRUE`, `fill = TRUE`, `quote = "\""`, `comment.char = ""`.
- `read.csv2`: this function is like the one above but `sep = ";"`, `dec = ","`. That is the field separator character is a semicolon and the character used in the file for decimal points is a comma.
- `read.delim`: this function is another variant of `read.table` and has the following default arguments: `header = TRUE`, `sep = "\t"`, `quote = "\""`, `dec = "."`, `fill = TRUE`, `comment.char = ""`.
- `read.delim2`: this function is like the one above but the character used in the file for decimal points is a comma instead of a dot.

Note: `read.table` and its brother and sisters will convert any character value into a factor variable. This is because `stringsAsFactors` is set to `TRUE` as a default. To suppress this behavior simply set `stringsAsFactors = FALSE`.

6.1.1.1 The `readr` package

The `readr` package provides replacements for functions like `read.table()` and `read.csv()`. The analogous functions in `readr` are `read_table()` and `read_csv()`. This packaged was developed by Hadley Wickham to deal with large and flat files quickly and efficiently. These functions are not only much faster than their base R analogues, but they also provide a few other features such as progress bars and warnings. Further, for the most part, you can read use `read_table()` and `read_csv()` pretty much anywhere you might use `read.table()` and `read.csv()`.

6.1.1.2 Example

Suppose we want to load the following file:

```
street,              city,       beds, baths, sq__ft, type,        price
3526 HIGH ST,                  , 2,    1,     836,    Residential, 59,222.2
51 OMAHA CT,         SACRAMENTO, 3,    1,     1,167,             , 68,212.1
2796 BRANCH ST,
2805 JANETTE WAY,    SACRAMENTO, 2,    1,     852,    Residential, 69,307.12
6001 MCMAHON DR,     SACRAMENTO, 2,    1,     797,    Residential, 81,900.1
...
```

Note: white spaces have been introduced to increase readability and are not present in the actual file.

We can identify the following problems:

1. the comma is being used as both a field separator character and a thousands separator (look at the variables `sq_ft` and `price`) while the dot is being used as a decimal point.
2. missing values seems to have been encoded as empty strings.
3. some of the rows have unequal length.

Solution:

1. Firstly, replace all commas intended to be used as field separators with semicolons (might not be as trivial as it sounds). Secondly, set `sep = ";"` and `dec = "."`. Finally, after having imported the file replace the commas used as thousands separator with an empty string and then convert the values into

numeric.

2. set `na.strings = ""`.
3. set `fill = TRUE`.

```
df <- read.table(file = "ExampleData/Import_Export_data.txt",
                 header = TRUE,
                 sep = ";",
                 dec = ".",
                 na.strings = "",
                 fill = TRUE)

df
##                 street       city beds baths sq__ft        type     price
## 1        3526 HIGH ST       <NA>    2     1    836 Residential  59,222.2
## 2         51 OMAHA CT SACRAMENTO    3     1  1,167        <NA>  68,212.1
## 3      2796 BRANCH ST       <NA>   NA    NA   <NA>        <NA>      <NA>
## 4    2805 JANETTE WAY SACRAMENTO    2     1    852 Residential 69,307.12
## 5     6001 MCMAHON DR SACRAMENTO    2     1    797 Residential  81,900.1
## 6  5828 PEPPERMILL CT SACRAMENTO    3     1  1,122       Condo  89,921.2
## 7 6048 OGDEN NASH WAY SACRAMENTO    3     2  1,104 Residential  90,895.9
## 8       2561 19TH AVE SACRAMENTO    3     1  1,177 Residential  91,002.0

df$sq__ft <- as.numeric(gsub(",", "", as.character(df$sq__ft)))
df$price <- as.numeric(gsub(",", "", as.character(df$price)))

df
##                 street       city beds baths sq__ft        type    price
## 1        3526 HIGH ST       <NA>    2     1    836 Residential 59222.20
## 2         51 OMAHA CT SACRAMENTO    3     1   1167        <NA> 68212.10
## 3      2796 BRANCH ST       <NA>   NA    NA     NA        <NA>       NA
## 4    2805 JANETTE WAY SACRAMENTO    2     1    852 Residential 69307.12
## 5     6001 MCMAHON DR SACRAMENTO    2     1    797 Residential 81900.10
## 6  5828 PEPPERMILL CT SACRAMENTO    3     1   1122       Condo 89921.20
## 7 6048 OGDEN NASH WAY SACRAMENTO    3     2   1104 Residential 90895.90
## 8       2561 19TH AVE SACRAMENTO    3     1   1177 Residential 91002.00
```

6.1.2 Export

The function `write.table` is the most convenient way to write a rectangular grid of data to a text file. Beside the arguments that are shared with `read.table`, `write.table` has some other ones that are required when writing files. Those are:

- `append`: when set to `TRUE` it will append to the existing file - or it will automatically create a new one if there is no existing file matching the given name. If `FALSE`, any existing file of the name is overwritten.
- `row.names` and `col.names`: as a default, `write.table` will write row and column names, which means that `row.names` and `col.names` are both set the `TRUE` as a default.

Again, because of the many possibilities, there are several other functions that call `write.table` but change a group of default arguments. Those are: * `write.csv`: it wraps `write.table` by setting `sep = ","`, `dec = "."`. * `write.csv2`: it wraps `write.table` by setting `sep = ";"`, `dec= ","`.

A general read.table call would look like this:

```
write.table(data, "<FileName>.<FileFormat>",
            row.names = FALSE,
            na="")
```

6.2 R objects

So far, we have seen how to read and write 2-dimensional datasets into a text file. However, as you might have noticed, we can neither import nor export special special attributes of the data structures, such as whether a column is a character type or factor, or the order of levels in factors. In order to do that, it should be written out in a special format for R.

6.2.1 .Rdmpd

`dump()` can be used to to output R source code which, when run, will re-create the objects that have been saved.

```
# Note: we do not pass the data frame to the function, rather we specify its name

# Save a single object text format
dump("data1", "data.Rdmpd")

# Can save multiple objects:
dump(c("data1", "data2"), "data.Rdmpd")

# To load the data again:
source("data.Rdmpd")

# Note: when loaded, the original data names will automatically be used.
```

This is how an object saved in `.Rdmpd` looks like.

How the object was created in R:

```
df <- data.frame(V1 = c(1L, 2L, 3L), V2 = c("a", "b", "c"))
```

How it is stored in a `.Rdmpd` file:

```
df <- structure(list(V1 = 1:3, V2 = c("a", "b", "c")), .Names = c("V1", "V2"), row.names
= c(NA, -3L), class = "data.frame")
```

6.2.2 .rds

We can write out individual data objects in RDS format via the function `saveRDS()`. This format can be binary or ASCII. Binary is more compact, while `ASCII` will be more efficient with version control systems like Git.

```
# Note: contrary to dump(), saveRDS() needs to be provided with the actual R object
# to be saved and not its name. Further, saveRDS() can only save one R object per file.

# Save a single object in binary RDS format
saveRDS(data1, "data.rds")

# Or, using ASCII format
saveRDS(data1, "data.rds", ascii=TRUE)
```

```r
# To load the data again:
data <- readRDS("data.rds")
```

6.2.3 .RData

If you wish to save multiple objects into an single file in either binary or ASCII,the `RData` format can be used. In order to write out objects in the `RData`, we use `save()`.

```r
# Saving multiple objects in binary RData format
save(data1, file="data.RData")

# Or, using ASCII format
save(data1, file="data.RData", ascii=TRUE)

# Can save multiple objects
save(data1, data2, file="data.RData")

# To load the data again:
load("data.RData")
```

An important difference between `saveRDS()` and `save()` is that, with the former, when you `readRDS()` the data, you specify the name of the object, and with the latter, when you `load()` the data, the original object names are automatically used.

6.3 Connections

Connections are powerful tools that let you interact with files or other external objects. Connections can be thought of as a translator that lets you talk to objects that are outside of R. Those outside objects could be anything from a data base, a text file, a webpage, or an API. Some of the most used functions to create a connection are:

- `file()`: creates a connection to a file.
- `gzfile()`: creates a connection to a file compressed with gzip.
- `bzfile()`: creates a connection to a file compressed with bzip2.
- `socketConnection` [5]: creates a connection to a socket.
- `url()`: creates a connection to a webpage.

One feature of connections is that you can incrementally read or write pieces of data from/to the connection using functions such as `readBin()`, `write()`, `readLines()`, and `writeLines()`. This, in turn, allows for asynchronous data processing.

The function listed above will create a connection, however, in order to open or close a connection we will need to use:

- `open()`: the argument `open` can be set to:
  - `"r"` or `"rt"`: Open for reading in text mode.
  - `"w"` or `"wt"`: Open for writing in text mode.
  - `"a"` or `"at"`: Open for appending in text mode.
  - `"rb"`: Open for reading in binary mode.
  - `"wb"`: Open for writing in binary mode.
  - `"ab"`: Open for appending in binary mode.
  - `"r+"`, `"r+b"`: Open for reading and writing.
  - `"w+"`, `"w+b"`: Open for reading and writing, truncating file initially.

[5] An (internet) socket is one endpoint of a two-way communication link between two programs running on a network.

  - `"a+"`, `"a+b"`: Open for reading and appending.
- `close()`: used to close a connection

Below, are some functions which can come in handy when handling files:

- `file.create()`
- `file.exists()`
- `file.remove()`
- `file.rename()`
- `file.append()`
- `file.copy()`

In this section we will look at `file()` and `url()`.

6.3.1 Connection to a file

```
file.create("ExampleData/sample.txt") # Create a new file
## [1] TRUE
con <- file("ExampleData/sample.txt") # Create a connection to the file
con                                   # Check the status of the connection
##              description                    class                     mode
## "ExampleData/sample.txt"                   "file"                      "r"
##                     text                   opened                 can read
##                   "text"                 "closed"                    "yes"
##                can write
##                    "yes"
open(con, "r+")                       # Open the connecion in reading and writing mode
writeLines("A line", con)             # Write some lines
writeLines("Another line", con)
readLines(con)                        # Read all the lines in the file
## [1] "A line"       "Another line"
close(con)                            # Close the connection
```

6.3.2 Connectio to a URL

```
# Create a connection to an URL
con <- url("http://www2.warwick.ac.uk/fac/sci/statistics/modules/st3")
# Check the status of the connection
con
##                                                  description
## "http://www2.warwick.ac.uk/fac/sci/statistics/modules/st3"
##                                                        class
##                                                        "url"
##                                                         mode
##                                                          "r"
##                                                         text
##                                                       "text"
##                                                       opened
##                                                     "closed"
##                                                     can read
##                                                        "yes"
##                                                    can write
##                                                         "no"
```

```
# Open the connecion in reading mode
open(con, "r")
# Read all the lines and display 8 through 12.
readLines(con)[8:12]
## [1] "<meta name=\"viewport\" content=\"width=device-width, initial-scale=1\">"
## [2] ""
## [3] "<title>Warwick Statistics: Third year modules</title>"
## [4] ""
## [5] "<meta name=\"description\" content=\"\">"
# Close the connection
close(con)
```

7 Graphics

7.1 Plotting with graphics

7.1.1 Histogram and density plot

We start by simulating some data:

```
data1 <- rnorm(n = 100,
               mean = 3,
               sd = 1)
```

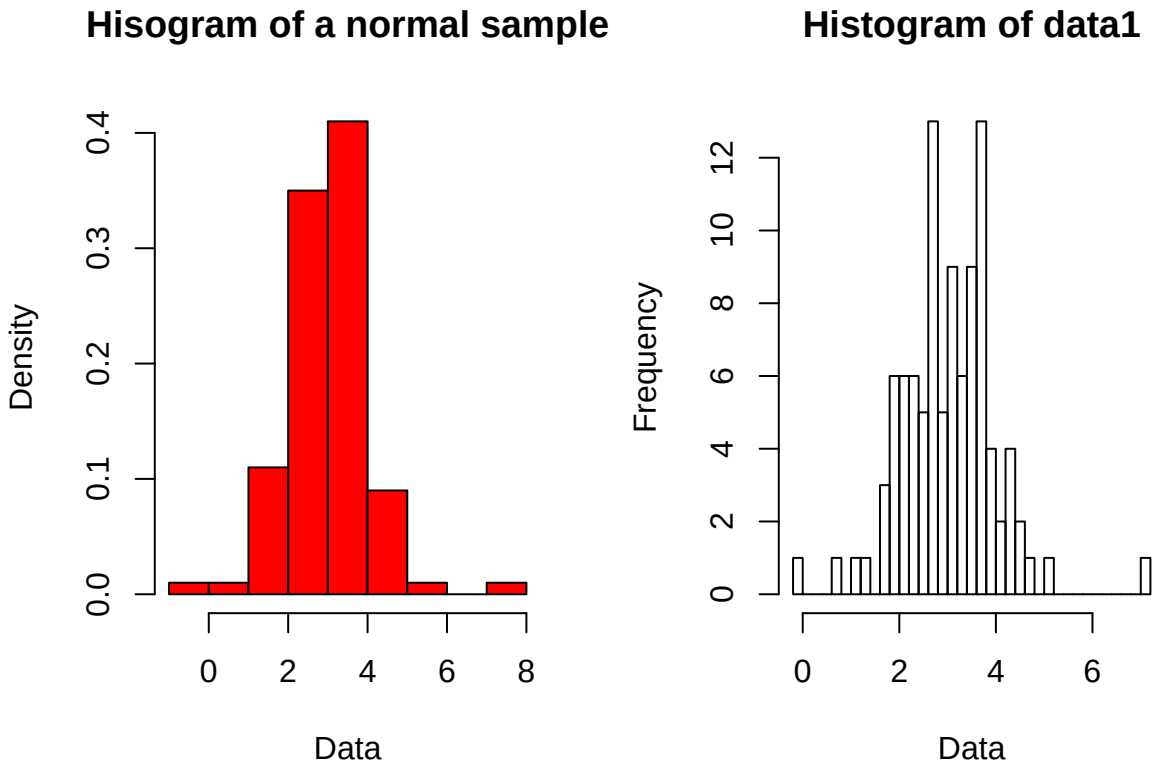
7.1.1.1 Histogram

We can plot an histogram of the data via the `hist` function. This function has many different arguments however, we will focus on:

- `x` : a vector of values for which the histogram is desired.
- `breaks`: one of:
  1. a vector giving the breakpoints between histogram cells,
  2. a function to compute the vector of breakpoints,
  3. a single number giving the number of cells for the histogram,
  4. a character string naming an algorithm to compute the number of cells (see `Details` in the help file),
  5. a function to compute the number of cells.
- `freq`: logical; if `TRUE`, the histogram graphic is a representation of frequencies, the counts component of the result; if `FALSE`, probability densities, component density, are plotted (so that the histogram has a total area of one).
- `col`: a colour to be used to fill the bars. The default of `NULL` yields unfilled bars.
- `main`, `xlab`, `ylab`: tile of the plot, label of the x and y axes respectively.
- `axes`: logical. If `TRUE` (default), axes are draw if the plot is drawn.

Let us this function in action.

```
par(mfrow = c(1, 2))
hist(x = data1,
     breaks = 10,
     col = "red",
     freq = FALSE,
     main = "Hisogram of a normal sample",
     xlab = "Data")
hist(x = data1,
     breaks = 50,
     xlab = "Data")
```



7.1.1.2 Density or kernel plot

In R, it is possible to compute kernel density estimates using the `density` function. Some of the arguments to be aware of are:

- `x`: the data from which the estimate is to be computed.
- `kernel`: a character string giving the smoothing kernel to be used. This must partially match one of "gaussian", "rectangular", "triangular", "epanechnikov", "biweight", "cosine" or "optcosine", with default "gaussian", and may be abbreviated to a unique prefix (single letter).
- `na.rm`: logical; if `TRUE`, missing values are removed from `x`. If `FALSE` any missing values cause an error.

```
kernelDensity <-  density(data1)
```

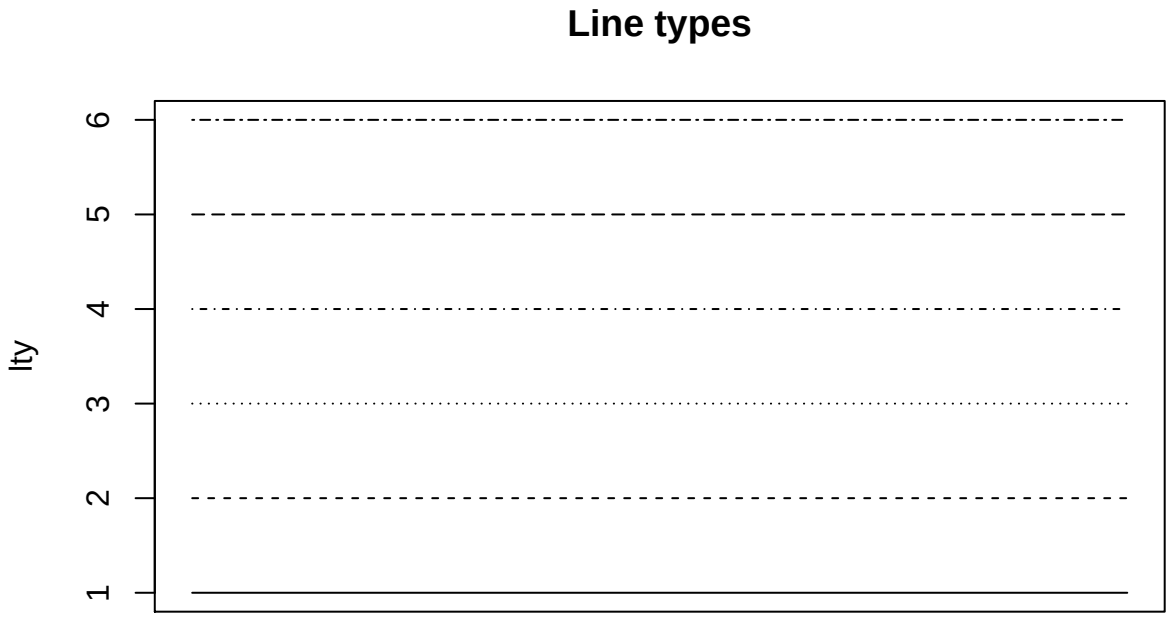
If we print the `kernelDensity` object we get some summary information about the estimated density

```
kernelDensity
##
## Call:
##  density.default(x = data1)
##
## Data: data1 (100 obs.);  Bandwidth 'bw' = 0.3317
##
##        x                y
##  Min.   :-1.147   Min.   :0.0001359
##  1st Qu.: 1.146   1st Qu.:0.0061565
##  Median : 3.440   Median :0.0190962
##  Mean   : 3.440   Mean   :0.1088922
##  3rd Qu.: 5.733   3rd Qu.:0.2025931
##  Max.   : 8.027   Max.   :0.4049654
# if you go back to how we generated the data, you will notice that we generated samples
# from a normal distribution centred around 3 which is what we can see from the summary
```

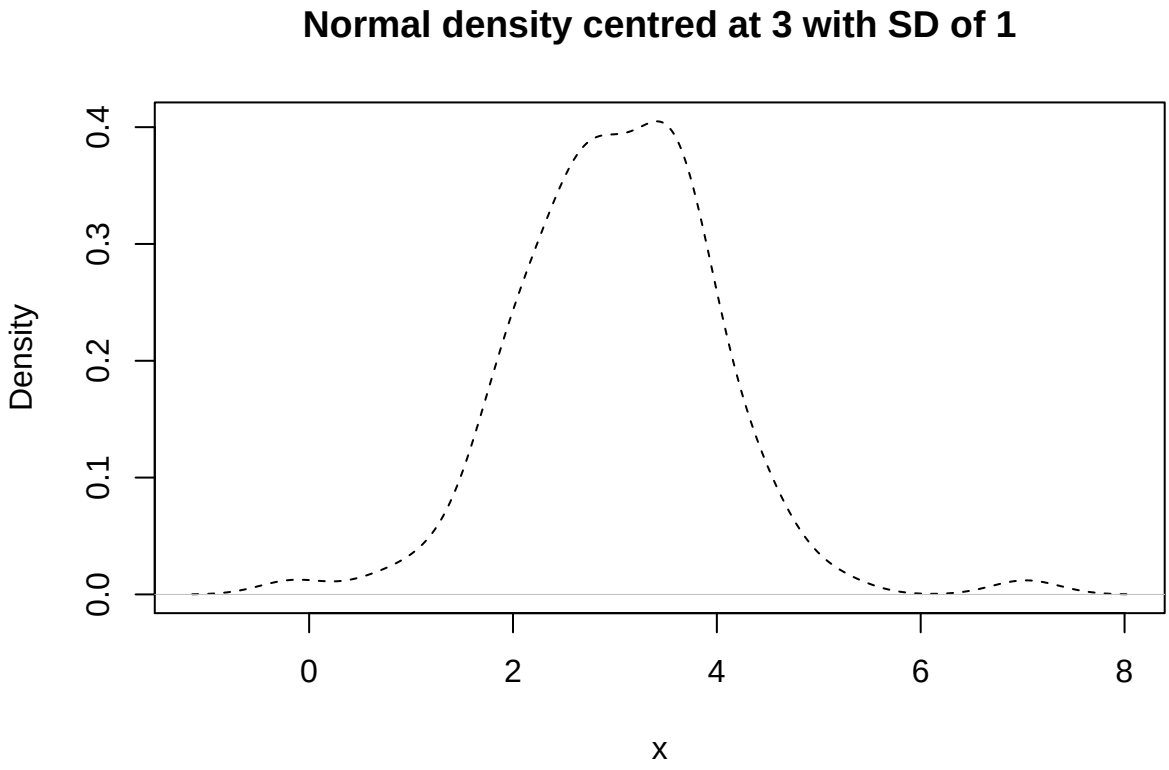
```
# above.
# It is also worth taking a look at what str returns when invked on this object
str(kernelDensity)
## List of 7
##  $ x         : num [1:512] -1.15 -1.13 -1.11 -1.09 -1.08 ...
##  $ y         : num [1:512] 0.000136 0.000159 0.000186 0.000218 0.000254 ...
##  $ bw        : num 0.332
##  $ n         : int 100
##  $ call      : language density.default(x = data1)
##  $ data.name: chr "data1"
##  $ has.na    : logi FALSE
##  - attr(*, "class")= chr "density"
```

We can plot the density via the `plot` function. Some of the most used arguments of this function are (we will see more in when looking at scatterplots):

- `x`: the coordinates of points in the plot. Alternatively, a single plotting structure, function or any R object with a `plot` method can be provided.
- `y`: the y coordinates of points in the plot, optional if x is an appropriate structure.
- `...`: Arguments to be passed to methods, such as graphical parameters. Many methods will accept the following arguments:
  1. `type`: a non-exhaustive list is "p" (points); "l" (lines); "b" (both points and lines), "h" (histogram-like), "s" (stair steps)
  2. `main`, `sub`, `xlab`, `ylab`: tile and subtitle of the plot and label of the x and y axes respectively.
  3. `lwd`: line width for drawing symbols.
  4. `col`: color code or name.
  5. `xaxt` and `yaxt`: a character which specifies the x/y axis type. Specifying "n" suppresses plotting.
  6. `lty`: line type (see plot below).



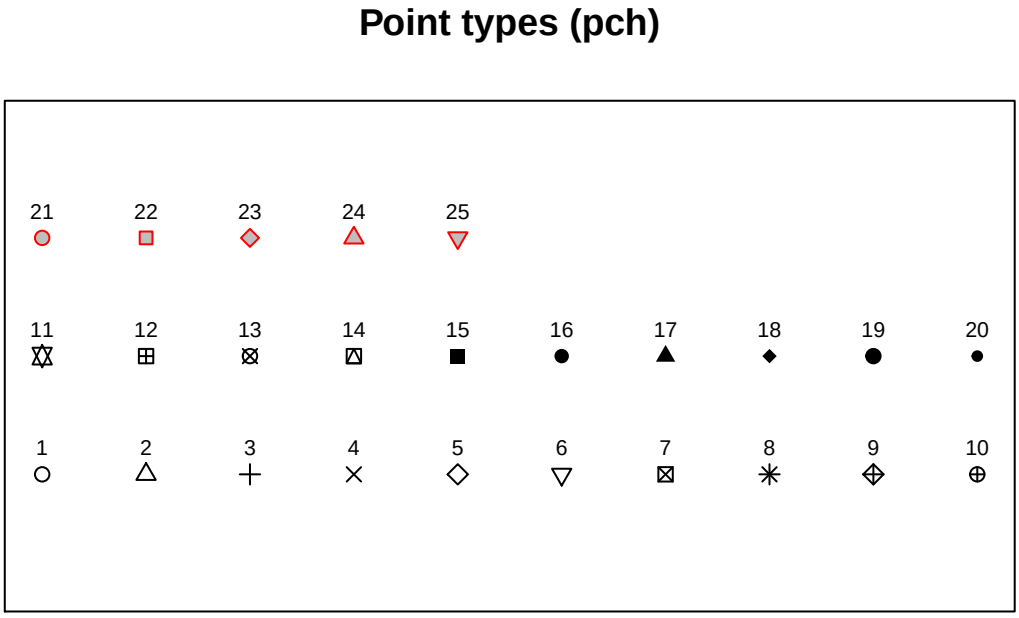

```
plot(kernelDensity,
     main = "Normal density centred at 3 with SD of 1",
     xlab = "x",
     lwd = 1,
     lty = 2)
```



7.1.2 Scatterplot

In R, we can make a 2-d scatter plot by specifying the x and y arguemnts of the `plot` function. Before we plot anything, we introduce a new agrument which can be specified when calling `plot`:

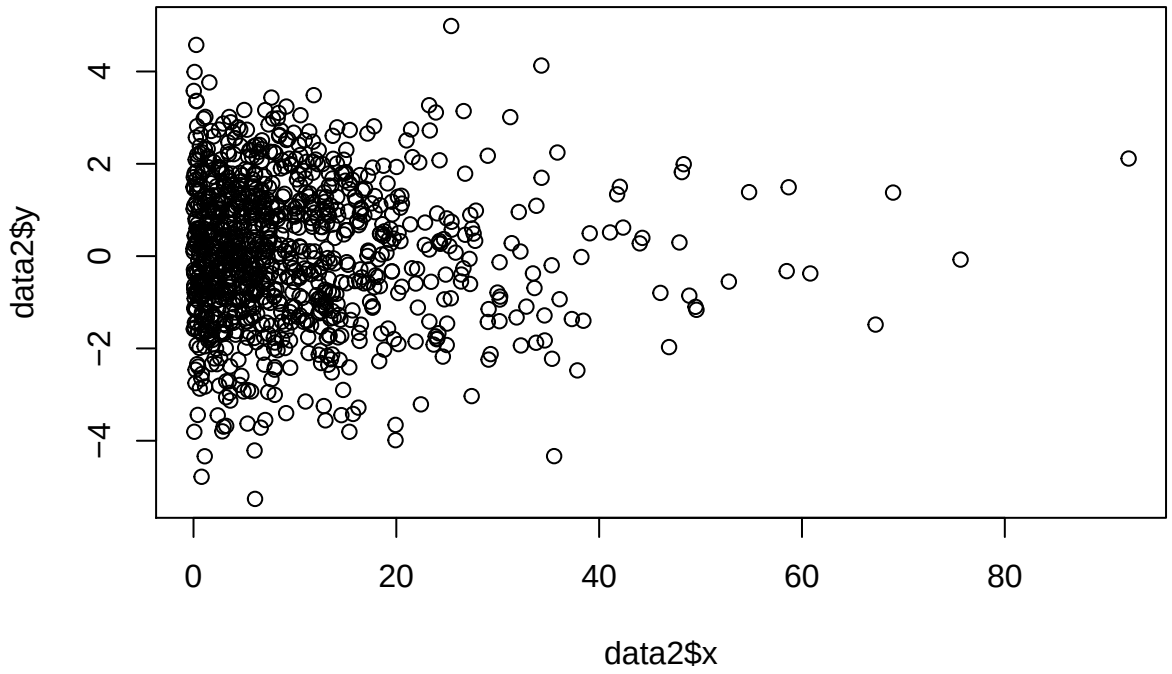
- `pch`: Either an integer specifying a symbol or a single character to be used as the default in plotting points. See the plot below for some examples.



Note: For symbols 21 through 25, specify border color `col` and fill color `bg`.

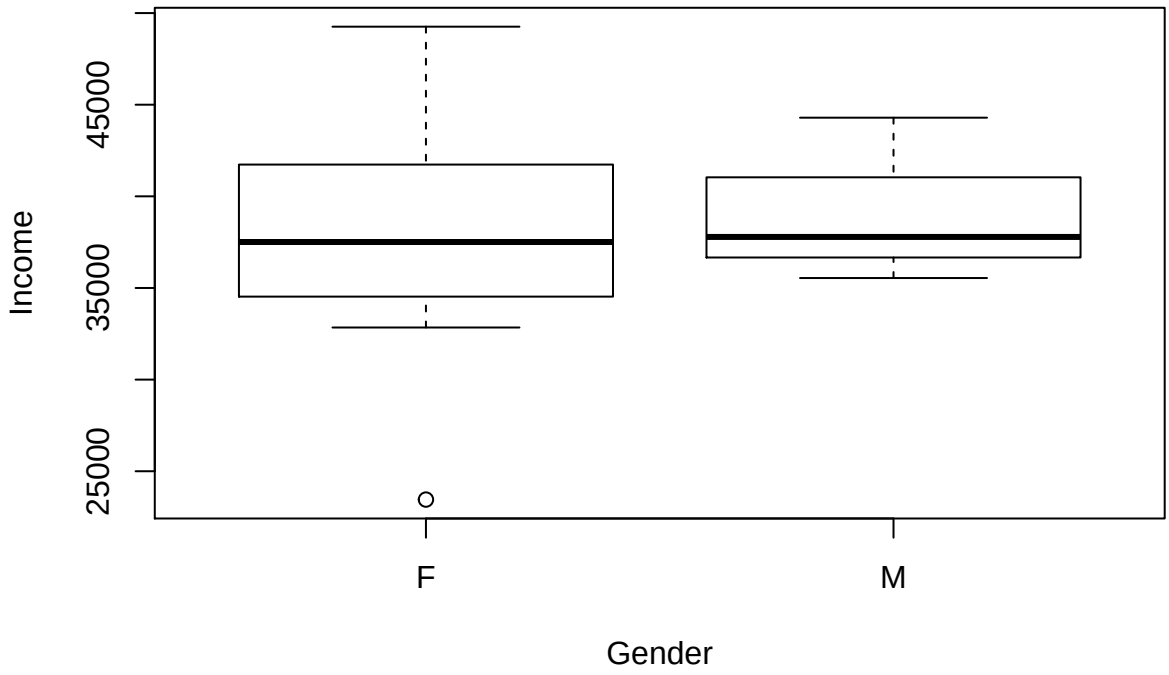
Let us now generate and plot some data:

```
data2 <- data.frame(x = rexp(n = 1000,
                             rate = 0.1),
                    y = rnorm(n = 1000,
                              mean = 0,
                              sd = 1.5))

plot(x = data2$x,
     y = data2$y,
     type = "p",
     pch = 1)
```

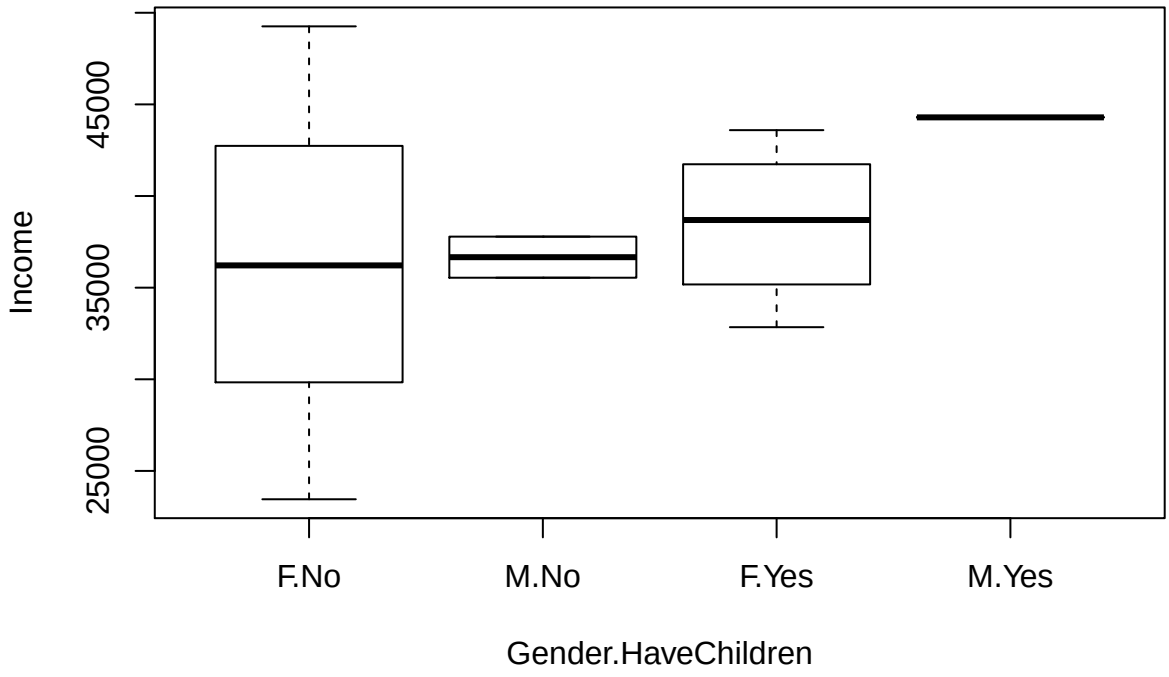


7.1.3 Box plot

```
set.seed(570)
data3 <- data.frame(age = round(rnorm(n = 10, mean = 35, sd = 9)),
                    gender = sample(x = c("M","F"), size = 10, replace = TRUE),
                    income = rnorm(n = 10, mean = 35000, sd = 10000),
                    haveChildren = sample(x = c("Yes","No"), size = 10, replace = TRUE))

boxplot(formula = income ~ gender,
        data = data3,
        xlab = "Gender",
        ylab = "Income")
```




```
set.seed(570)
boxplot(formula = income ~ interaction(gender,haveChildren),
        data = data3,
        xlab = "Gender.HaveChildren",
        ylab = "Income")
```

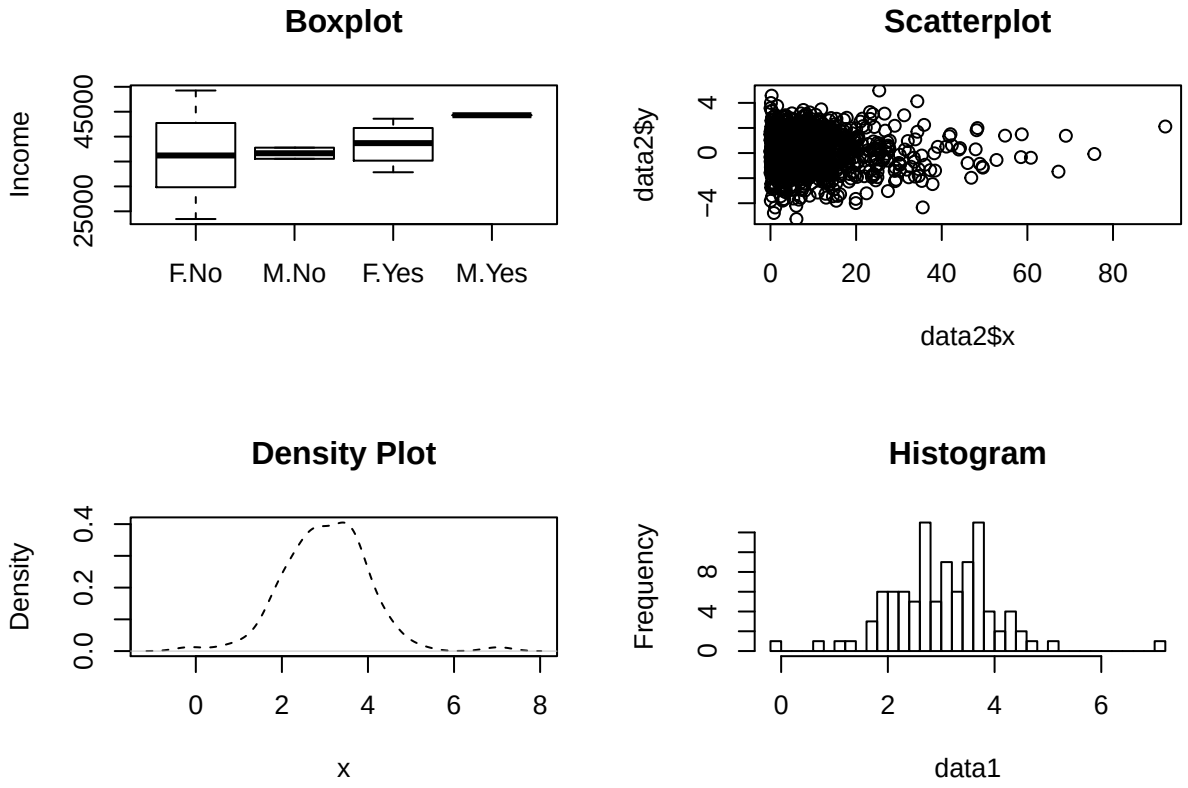


7.1.4 Combining plots

R makes it "easy" to combine multiple plots into one overall graph, using either the `par` or `layout` function. With the `par` function, you can include the option `mfrow = c(nrows, ncols)` to create a matrix of `nrows`

by ncols plots that are filled in by row. mfcol = c(nrows, ncols) fills in the matrix by columns.

```
par(mfrow = c(2, 2))
boxplot(formula = income ~ interaction(gender,haveChildren),
        ylab = "Income",
        data = data3,
        main = "Boxplot")
plot(x = data2$x,
     y = data2$y,
     type = "p",
     pch = 1,
     main = "Scatterplot")
plot(kernelDensity,
     xlab = "x",
     lwd = 1,
     lty = 2,
     main = "Density Plot")
hist(x = data1,
     breaks = 50,
     main = "Histogram")
```

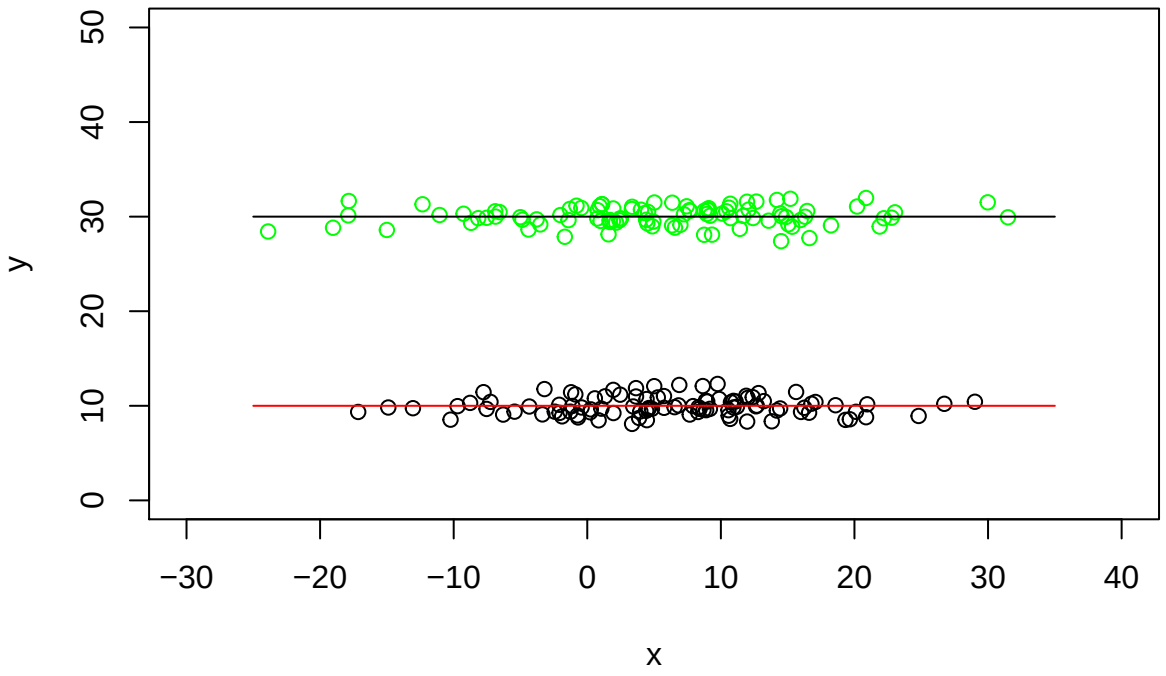


7.1.5 Overlaying plots

The functions lines and points will add to an existing plot. Let us see an exaple

```
set.seed(1)
# Generate a scatter plot
plot(x = rnorm(n = 100, mean = 5, sd = 10),
     y = rnorm(n = 100, mean = 10, sd = 1),
```

```
     ylim = c(0,50),
     xlim = c(-30,40),
     ylab = "y",
     xlab = "x")
# Add a line
lines(x = c(-25,35),
      y = c(10,10),
      col = "red",
      lwd = 1)
# Add some more points
points(x = rnorm(n = 100, mean = 5, sd = 10),
       y = rnorm(n = 100, mean = 30, sd = 1),
       col = "green")
# Add another line
lines(x = c(-25,35),
      y = c(30,30),
      col = "black",
      lwd = 1)
```



8 Functional Programming

8.1 A motivational case study: NAs handling

To start this section, we will look at how functional programming can enable us to create an missing values handling tool. Each step in the development of the tool is driven by the desire to reduce redundancy and increase generality.

Suppose that you have imported a csv file into R which uses the word "missing" to denote a missing value. Our job is to replace all the "missing"s with NAs.

```
set.seed(1341)
df <- data.frame(replicate(5,
                           sample(c(-10:10,"missing"),
                                  size = 10,
                                  replace = TRUE)
                           )
                 )
names(df) <- c(letters[1:5])
df
##          a  b       c       d       e
## 1        0  2      -7       9      -6
## 2  missing -4       4       6 missing
## 3        1 -9      -2 missing       5
## 4       -7  9       6      10      -4
## 5        8  2      -1       0       4
## 6        9 -1      -9       7       1
## 7        0  5       0       0       9
## 8       -5 -4 missing       3      -8
## 9        7  9      -9       4      -9
## 10     -10 -7      -2      -7       7
```

One way in which an inexperienced programmer might tackle the issue is with copy-and-paste.

```
df$a[df$a == "missing"] <- NA
df$b[df$b == "missing"] <- NA
df$c[df$c == "mising"] <- NA
df$d[df$g == "missing"] <- NA
df$e[df$e == "missing"] <- NA

df
##       a  b       c       d    e
## 1     0  2      -7       9   -6
## 2  <NA> -4       4       6 <NA>
## 3     1 -9      -2 missing    5
## 4    -7  9       6      10   -4
## 5     8  2      -1       0    4
## 6     9 -1      -9       7    1
## 7     0  5       0       0    9
## 8    -5 -4 missing       3   -8
## 9     7  9      -9       4   -9
## 10  -10 -7      -2      -7    7
```

One problem with this approach is that it is enormously error-prone (in fact in the code above there is a mistake).

We can start applying functional programming ideas by writing a function that replaces the missing value.

```
replace_missing_value <- function(x){
  x[x == "missing"] <- NA
  return(x)
}

df$a <- replace_missing_value(df$a)
df$b <- replace_missing_value(df$b)
df$c <- replace_missing_value(df$c)
df$d <- replace_missing_value(df$d)
df$e <- replace_missing_value(df$e)
```

This method, is definitely better than the previous one as it reduces the scope of possible mistakes. However, there is still a lot of repetition, and, even though we ca no longer misspell "missing", we might still make mistakes when writing the columns names. Hence, we need one more function which can potentially apply a function to all columns of a data frame. Something the functional `lapply()`.

```
df[] <- lapply(df, replace_missing_value)
# Note: lapply is called a functional because it takes a function as an argument.
#Further, we use df[] instead of df to tell R that we want a data frame and not a list.

# compare this two expression:
df[] <- lapply(df, replace_missing_value)
class(df)
## [1] "data.frame"

df <- lapply(df, replace_missing_value)
class(df)
## [1] "list"
```

The code above has several advantages over copy-and-paste:

- It is more compact.
- If the encoding for the missing value changes, we have to change the value of only one variable.
- We do not need to specify the name or number of the columns.

Now, consider this scenario: we are importing a data set which uses again the word "missing" to represent NAs. However, this time around, whoever created this file, did not pay to much attention to spelling and somehow the word "missing" is spelled either correctly or with an extra "s": "misssing".

In order to tackle this problem, we will create a closure: a function that creates and returns another function. The purpose of this function will be able to specify the encoding for the missing values when we are calling the function itself.

```
set.seed(111)
df <- data.frame(replicate(5,
                           sample(c(-10:10, "missing", "misssing"),
                                  size = 10,
                                  replace = TRUE)
                           )
                 )
names(df) <- c(letters[1:5])
df
##      a        b        c  d       e
## 1    3        2       -1 -9       4
## 2    6        3       -4  1      -3
```

```
## 3   -2       -9       -3  0       4
## 4    1       -9       -2  0 missing
## 5   -2       -7 misssing -2       3
## 6   -1        0       -3  6      -2
## 7  -10       -7        5 -8       0
## 8    2 misssing       -4  8       9
## 9   -1       -3        8  4       4
## 10  -8        4        3  8       8
```

```
general_replace_missing_value <- function(na_values){
  function(x){
    for(na_value in na_values){
      x[x == na_value] <- NA
    }
    return(x)
  }
}

df[] <- lapply(df, general_replace_missing_value(c("missing", "misssing")))
```

8.2 Anonymos functions

We have learned that when we want to define a new function, we have to give it a name by bonding it to a name. However, in R we are not required to give a name to a new function. As a consequence, if you decide to not give a name to a newly created function, you get an anonymous function.

Generally, you would use an anonymous function when it is not worth the effort to give it a name.

```
# Thus, this is a nomral function
foo <- function(x) length(unique(x))

# and this is its anonymous version
function(x) length(unique(x))
## function(x) length(unique(x))
```

Like all functions in R, anonymous functions have `formals()`, a `body()`, and a parent `environment()`.

```
formals(function(x, y) y * sqrt(sin(x) / cos(x)))
## $x
##
##
## $y

body(function(x, y) y * sqrt(sin(x) / cos(x)))
## y * sqrt(sin(x)/cos(x))

environment(function(x, y) y * sqrt(sin(x) / cos(x)))
## <environment: R_GlobalEnv>
```

If you want to call an anonymous, you have to use parentheses in two ways: first to call the function, and second to make it clear that you want to call the anonymous function itself, as opposed to calling a function inside it. Like this:

```
(function(x) sin(x) / cos(x))(10)
## [1] 0.6483608
# which is equivalent to
foo <- function(x) sin(x) / cos(x)
foo(10)
## [1] 0.6483608
```

8.3 Closures

One of the most common uses of anonymous functions is to create closures: functions written by functions. The name closure is due to the fact that this kind of functions enclose the environment of the aren't function and can access all its variables. This hierarchical structure allows us to have two levels of parameters: a parent level, and a child level.

The following example uses this idea to generate a family of L^p norms to measure the length of an n-dimensional vector, in which the parent function `Lp_norm()` creates two child functions `L1_norm()`, and `L2_norm()`.

We remember that for a real number $p \geq 1$, the $L^p - norm$ of x is defined by

$$||x||_p = \Big(\sum_{i=1}^{n} |x_i|^p \Big)^{1/p},$$

where n is the dimension of x

```
Lp_norm <- function(p){
  function(x){
    return((sum(abs(x)^p))^(1/p))
  }
}

L1_norm <- Lp_norm(1)
L1_norm(c(3,3,3))
## [1] 9
L2_norm <- Lp_norm(2)
L2_norm(c(3,3,3))
## [1] 5.196152
```

Another use of a closure would be to construct a family of power functions:

```
power <- function(exponent){
  function(base){
    return(base^exponent)
  }
}

square <- power(2)
cube <- power(3)

square(3)
## [1] 9
cube(3)
## [1] 27
```

When we print a closure, we do not anything useful in the function body, in fact, they are exactly the same although they perform two different operations (one has $p = 1$ and the other $p = 2$). This is because, what changes - that is to say what makes p have different values - is the enclosing environment.

```
L1_norm
## function(x){
##     return((sum(abs(x)^p))^(1/p))
##   }
## <environment: 0x7fcf4ba97540>
L2_norm
## function(x){
##     return((sum(abs(x)^p))^(1/p))
##   }
## <environment: 0x7fcf4aad6a38>
environment(L1_norm)
## <environment: 0x7fcf4ba97540>
environment(L2_norm)
## <environment: 0x7fcf4aad6a38>
# As you can see, the two closures do not have the same environment
```

One way to see the contents of an environment is to convert it to a list:

```
as.list(environment(L1_norm))
## $p
## [1] 1
as.list(environment(L2_norm))
## $p
## [1] 2

# As you can see, the two closures do not have the same environment
```

The parent environment of a closure is the execution environment of the function that created it, as shown below.

```
power <- function(exponent){
  print(environment())
  function(base){
    return(base^exponent)
  }
}

square <- power(2)
## <environment: 0x7fcf4bec53b8>
environment(square)
## <environment: 0x7fcf4bec53b8>
```

The execution environment normally disappears after a function returns a value. However, in this case the object to the returned is a function, and, in R functions capture their enclosing environments. In R almost every function is a closure - the only exception is primitive functions - and they remember the environment in which they were created:

```
foo <- function(x){
  return(mean(x)/var(x))
}
environment(foo)
## <environment: R_GlobalEnv>
```

8.3.1 The <<- operator

Having variables at two levels, allows us to maintain their state across function invocations. This is possible because while the executing environment is refreshed every time, the enclosing environment does not change. The way in which we can manage variables at different levels is via the `<<-` operator. Let us see it in action:

```
foo <- function(){
  counter <- 0
  function(){
    counter <<- counter + 1
    return(paste0("This function has been called ", counter, " time(s)"))
  }
}

foo_1 <- foo()
foo_1()
## [1] "This function has been called 1 time(s)"
foo_1()
## [1] "This function has been called 2 time(s)"
foo_2 <- foo()
foo_2()
## [1] "This function has been called 1 time(s)"
```

Unlike the single arrow operator (`<-`) that looks in the current environment, the double arrow operator (`<<-`) will keep looking up the chain of parent enthronements until it finds the matching name.

Thanks to this operator, in the example above we were able to access the variable `counter` within the closure even thought it was defined in another environment.

8.4 Lists of functions

In R functions can be stored in lists like any other object.

Suppose we want to summaries a sample in a number of different ways. We can start by creating a list that contains all the function we wish to include in the summary:

```
summary_functions <- list(mean = mean,
                          median = median,
                          sd = sd,
                          max = max,
                          min = min,
                          sum = sum)
```

To call a function from a list, we simply extract it and then call it:

```
summary_functions$median(rnorm(100,3.5,2))
## [1] 3.893971
```

To apply each function to the same sample, we can use `lapply()`:

```
# Create a sample from the normal distribution
x <- rnorm(100, 3.5, 2)

# We call each function with the argument na.rm (remove missing values) set to TRUE,
# and we apply on the vector x.
lapply(summary_functions, function(f) f(x, na.rm = TRUE))
## $mean
```

```
## [1] 3.433211
##
## $median
## [1] 3.443021
##
## $sd
## [1] 1.944665
##
## $max
## [1] 8.684455
##
## $min
## [1] -3.14667
##
## $sum
## [1] 343.3211
```

8.5 Case study: maximum likelihood

Suppose the following: having observed the outcomes n i.i.d random variables distributed normally, we would like to estimate the parameters of the distribution which generated them via the maximum likelihood method. We recall that the log-likelihood for n i.i.d Gaussian random variables is given by:

$$l(\theta|x) = \frac{n}{2}\log 2\pi - \frac{n}{2}\log\sigma^2 - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(x_i - \mu)^2$$

where,

$$\theta = (\mu, \sigma)$$

We begin by writing the log-likelihood in R syntax:

```
gaussian_log_likelihood <- function(x, theta){
  n <- length(x)
  return(((n / 2) * log(2 * pi) - (n/2) * log(theta[2]^2) -
            1 / (2 * theta[2]^2) * sum((x-theta[1])^2)))
}
```

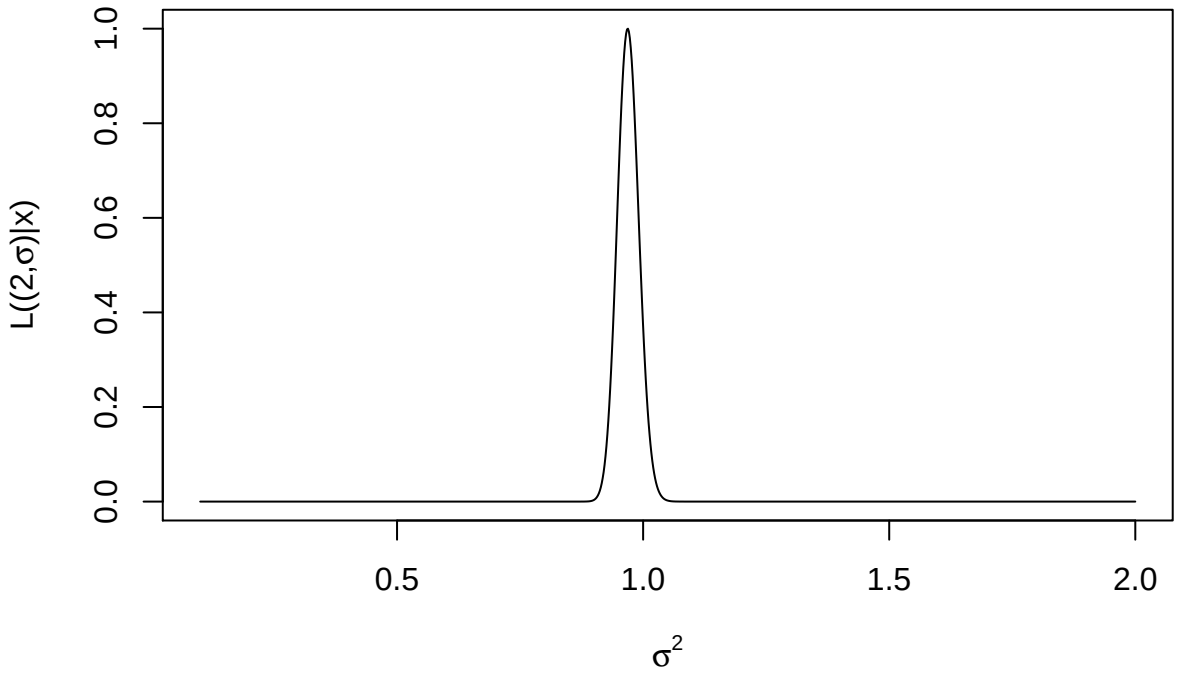
So far, we have a function which given the data and the parameters can tell us the value of the log-likelihood. However, what if what we want is to be able to have a log-likelihood for a sample without having to specify its parameters? We would need to create a closure:

```
gaussian_likelihood <- function(x, fixed_parameters = c(FALSE, FALSE)){
  theta <- fixed_parameters
  function(parameters){
    theta[!fixed_parameters] <- parameters
    n <- length(x)
    return(-((n / 2) * log(2 * pi) - (n/2) * log(theta[2]^2) -
            1 / (2 * theta[2]^2) * sum((x-theta[1])^2)))
  }
}

# If you are wondering why we are making the loglikelihood negative,
```

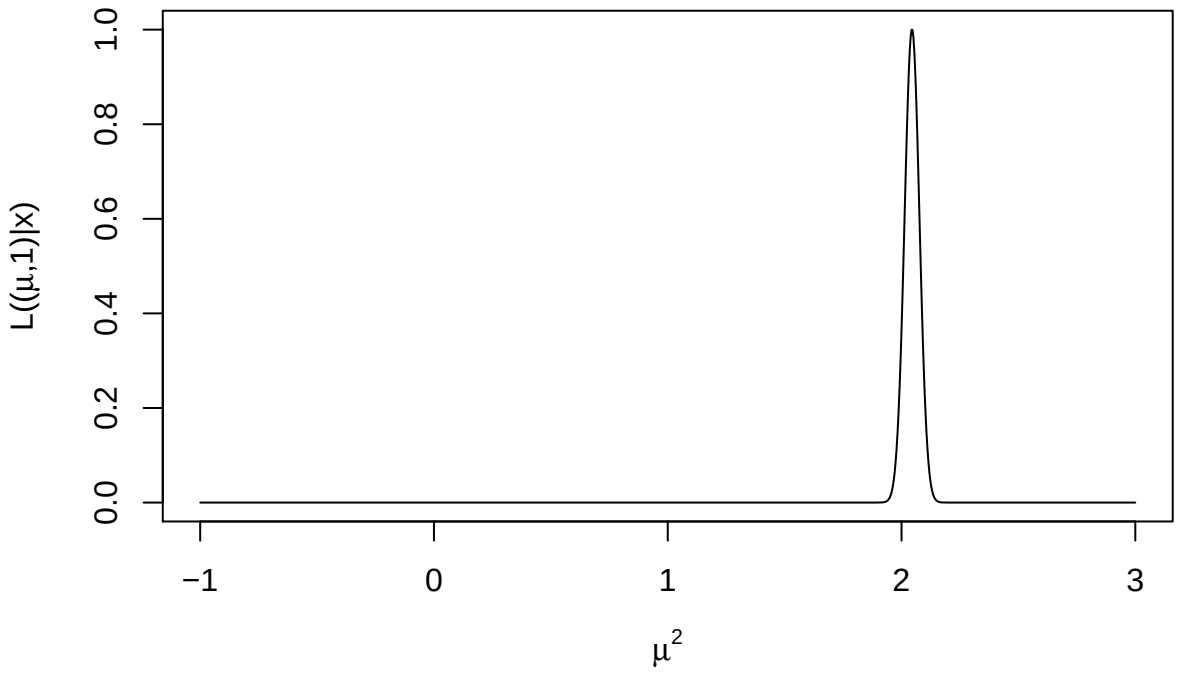
```
# it is because the function optim() minimizes functions, so, in order
# to get the maximum, we take the negative loglikelihood.
```

Now, what we have is a function that given the sample will return another function that accepts the parameters as its arguments.

```
# we call the function gaussian_likelihood and we fix the data.
gaussian_likelihood_fixed_data <- gaussian_likelihood(rnorm(n = 1000,
                                                            mean = 2,
                                                            sd = 1))
optim(par = c(0,1),
      fn = gaussian_likelihood_fixed_data,
      method = "L-BFGS-B",
      lower = c(-Inf,1e-10), upper =c(Inf,Inf))
## $par
## [1] 2.0123036 0.9932849
##
## $value
## [1] -425.6768
##
## $counts
## function gradient
##       13       13
##
## $convergence
## [1] 0
##
## $message
## [1] "CONVERGENCE: REL_REDUCTION_OF_F <= FACTR*EPSMCH"
x <- seq(from = 0.1, to = 2, len = 1000)
y <- sapply(x,
                  gaussian_likelihood(rnorm(1000,
                                            mean = 2,
                                            sd = 1),
                                      c(2, FALSE)))
plot(x = x,
     y = exp(-(y - min(y))),
     type = "l",
     xlab = expression(sigma^2),
     ylab = expression("L((" * 2 * "," * sigma * ")|" * "x)"))
```




```
x <- seq(from = -1, to = 3, len = 1000)
y <- sapply(x,
                  gaussian_likelihood(rnorm(1000,
                                            mean = 2,
                                            sd = 1),
                                      c(FALSE, 1)))
plot(x = x,
     y = exp(-(y - min(y))),
     type = "l",
     xlab = expression(mu^2),
     ylab = expression("L((" *mu * "," * 1 * ")|" * "x)"))
```



9 Functionals

9.1 Split-Apply-Combine Functionals

9.1.1 `lapply`

`lapply` takes a function, applies it to each element in a list and returns the results in the form of a list. Let us see this function in action.

```
# We will use the biult-in mtcars dataset as a toy dataset.

head(mtcars)
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1

# we will now calcualte the Coefficient of Variation for each column of the dataset

columnCV <- lapply(mtcars[,1:5], function(column){
  return(mean(column) / sd(column))
})

columnCV
## $mpg
## [1] 3.333466
##
## $cyl
## [1] 3.464598
##
## $disp
## [1] 1.861581
##
## $hp
## [1] 2.13946
##
## $drat
## [1] 6.726586

str(columnCV)
## List of 5
##  $ mpg : num 3.33
##  $ cyl : num 3.46
##  $ disp: num 1.86
##  $ hp  : num 2.14
##  $ drat: num 6.73
```

Since data frames are built on top of lists, `lapply` is also useful when you want to do something to each column of a data frame:

```
# Center each column by subtracting the mean
```

```
mtcars[] <- lapply(mtcars, function(column){
  return(column - mean(column))
})

class(mtcars)
## [1] "data.frame"
```

9.1.1.1 Looping patterns

Generally speaking, there are three basic ways to loop over a vector:

- loop over the elements: `for(x in y)`
- loop over the numeric indices: `for(i in seq_along(y))`
- loop over the names: `for(name in names(y))`

Note: The first choice is usually not a good one as it leads to inefficient ways of saving output.

Just as there are three basic ways to use a for loop, there are three ways to use `lapply`

- `lapply(y, function(x){})`
- `lapply(seq_along(y), function(i){})`
- `lapply(names(y), function(name){})`

9.1.2 `vapply` and `sapply`

`vapply` and `sapply` are very similar to `lapply` except they simplify the output to produce an atomic vector. The difference between `vapply` and `sapply` is that while the latter guesses the output type, the former takes an additional argument which specifies exactly that.

```
# Center each column by subtracting the mean

sapply(mtcars[,1:5], sd)
##         mpg         cyl        disp          hp        drat
##   6.0269481   1.7859216 123.9386938  68.5628685   0.5346787

vapply(mtcars[,1:5], sd, numeric(1))
##         mpg         cyl        disp          hp        drat
##   6.0269481   1.7859216 123.9386938  68.5628685   0.5346787
```

9.1.3 `apply`

`apply` is a variant of `sapply` that works with matrices and arrays. This function has 4 arguments:

- `X`: an array, including a matrix.
- `MARGIN`: a vector giving the subscripts which the function will be applied over. E.g., for a matrix 1 indicates rows, 2 indicates columns, c(1, 2) indicates rows and columns.
- `FUN`: the function to be applied
- ...: optional arguments passed on to `FUN`.

```
# applied to a 2-d matrix

m <- matrix(rbeta(100, 0.2,0.9) * 10, ncol = 5)

# compute the mean of each row
```

```
apply(m, 1, mean)
##  [1] 0.0726771 0.7532630 0.2950209 3.9440199 0.7344725 2.2503118 0.7136749
##  [8] 1.1358668 5.2001834 3.4208288 0.7658929 1.3952486 3.1104192 2.6274865
## [15] 2.3158826 0.9840738 0.7238562 2.2285094 0.4210871 2.4268507

# compute the mean of each column

apply(m, 2, mean)
## [1] 1.630367 1.099271 2.045602 1.862391 2.242276

# applied to a 3-d array

a <- array(rbeta(20, 0.2,0.9) * 10, c(2,2,5))
apply(a, 1, mean)
## [1] 3.081504 2.236752
apply(a, 2, mean)
## [1] 2.610572 2.707685
apply(a, 3, mean)
## [1] 2.030530 2.811400 2.530415 2.086321 3.836974
apply(a, c(1,2), mean)
##          [,1]     [,2]
## [1,] 2.791875 3.371133
## [2,] 2.429268 2.044236
```