

Programming and Statistics with R

Patrick Bucher

Contents

1 Basics	5
1.1 Environment	5
1.1.1 Scoping	5
1.1.2 Sessions	6
1.1.3 Packages	7
1.1.4 Help	7
1.2 Language Basics	7
1.2.1 Calculations	8
1.2.2 Argument Matching	8
1.3 Vectors	9
1.3.1 Sequences	9
1.3.2 Repetitions	9
1.3.3 Sorting	10
1.3.4 Accessing Elements	10
1.3.5 Arithmetic on Vectors	11
1.4 Matrices	11
1.4.1 Operations and Algebra	14
1.5 Multidimensional Arrays	16
1.6 Logical Values	16
1.6.1 Logical Operations	17
1.6.2 Element Selection	17
1.7 Strings	19
1.7.1 Concatenation	19
1.7.2 Substrings and Replacements	20
1.8 Factors	20
1.8.1 Cutting	21
1.8.2 Splitting	21
1.9 Lists	22
1.10 Data Frames	23
1.11 Special Values	24
1.11.1 Infinity	24
1.11.2 Not a Number	25
1.11.3 NULL	25
1.12 Objects	26
1.12.1 Attributes	26
1.12.2 Classes	26

Contents

1.13	Plotting	28
1.13.1	Additional Elements	29
1.13.2	TODO: Barplots	29
1.13.3	TODO: Pie Charts	29
1.13.4	TODO: Histograms	29
1.13.5	TODO: Box-and-Whisker Plots	29
1.13.6	TODO: Scatterplots	29
1.13.7	Saving Plots to Files	29
1.13.8	Plotting Examples	30
1.14	Pre-Installed Data Sets	33
1.15	Working with Files	35
1.15.1	Reading Text Files	35
1.15.2	Reading CSV Files	36
1.15.3	Writing Files	36
1.16	Timing	37
2	Programming	39
2.1	Conditions	39
2.1.1	If, Else If, Else	39
2.1.2	Switch-Case	39
2.2	Loops	40
2.2.1	For Loops	40
2.2.2	While Loops	40
2.2.3	Implicit Looping	41
2.2.4	Leave a Loop	41
2.2.5	Skip a Loop Item	42
2.2.6	Repeat	42
2.3	Functions	43
2.4	Warnings and Exceptions	44
3	Statistics	47
3.1	Basic Terms	47
3.2	Summary Statistics	47
3.2.1	Mean, Median, Mode	47
3.2.2	Quantiles, Percentiles	48
3.2.3	Spread	49
3.2.4	Covariance and Correlation	49
3.3	TODO: Probability	51
3.4	Probability Distributions	51
3.4.1	Mass Functions (for Discrete Variables)	51
3.4.2	Density Functions (for Continuous Variables)	54

1 Basics

1.1 Environment

Set a custom prompt (R>):

```
options(prompt = "R> ")
```

List variables, objects and user-defined functions of the current session:

```
ls()
```

Remove an item from the current session:

```
ls() # nothing  
foo <- "bar"  
ls() # "foo"
```

```
rm(foo)  
ls() # nothing
```

Empty the current session:

```
rm(list = ls())
```

Leave R:

```
q()
```

1.1.1 Scoping

List environments:

```
search() # ".GlobalEnv" "tools:rstudio" ... "package:base"
```

When referring to a symbol (a variable, object, function etc.), R searches through the environments listed by `search()` from left to right.

Determine the environment in which an object lives:

```
environment(seq) # <environment: namespace:base>  
environment(plot) # <environment: namespace:graphics>
```

1 Basics

List the content of an environment:

```
ls("package:graphics")
```

When a package is no longer used, it can be detached in order to clean up the namespace:

```
search() # ".GlobalEnv" "package:stats" "package:graphics" etc.
library("MASS")
search() # ".GlobalEnv" "package:MASS" "package:stats" etc.
detach("package:MASS", detach = TRUE)
search() # ".GlobalEnv" "package:stats" "package:graphics" etc.
```

For easier lookup, objects can be attached:

```
person <- data.frame(name = "Patrick", age = 30)
person.name # "Patrick"
person.age # 30
```

```
name # Error: object 'name' not found
age # Error: object 'age' not found
```

```
attach(person)
name # "Patrick"
age # 30
```

```
detach(person)
name # Error: object 'name' not found
age # Error: object 'age' not found
```

In order to keep the namespace clean, attaching objects to the namespace (“mounting”) should be avoided.

1.1.2 Sessions

Find out and set the current working directory:

```
getwd()
setwd("~/my-workspace")
```

Save the current session:

```
save.image("my-session.RData")
```

Load a stored session:

```
load("my-session.RData")
```

1.1.3 Packages

Install a new package (MASS, for example):

```
install.packages("MASS")
```

List installed packages:

```
installed.packages()
```

Load the installed MASS library:

```
library("MASS")
```

Update installed packages:

```
update.packages()
```

Uninstall a package (using default library paths):

```
remove.packages("MASS", .libPaths())
```

1.1.4 Help

Get help for a specific keyword (the mean function, for example):

```
help("mean")
?mean # shortcut
```

Search for a help topic (random, for example):

```
help.search("random")
??"random" # shortcut
```

1.2 Language Basics

Comments, starting with # to the end of the line:

```
1 + 1 # calculates one plus one, which is two
```

Exponential notation:

```
1e3 # 1 * 10^3 = 1000
1e-3 # 1 * 10^(-3) = 0.001
```

Assignments:

```
a <- 17
b = 42
```

1 Basics

1.2.1 Calculations

Basic arithmetic:

```
3 + 5 # 8
5 - 2 # 3
2 * 3 # 6
9 / 3 # 3
2 ^ 3 # 8 (2 to the power of 3)
sqrt(16) # 4
13 %% 5 # 3 (modulus, the remainder of 13 divided by 5)
```

Logarithms:

```
log(x = 8, base = 2) # 3, because 2^3 = 8
log(8, 2) # same but shorter
```

Euler's number as an exponential function:

```
exp(1) # 2.718282
```

The `log()` function uses Euler's number as the default base (natural logarithm):

```
log(100) # 4.60517
log(x = 100, base = exp(1)) # same with an explicit base
```

The `exp()` function is the reverse function of `log()`:

```
log(exp(23)) # 23
```

1.2.2 Argument Matching

Exact matching (fully spell out all the parameters):

```
matrix(data=1:16, ncol=4, nrow=4, byrow=TRUE, dimnames=list(1:4, 1:4))
```

Partial matching (abbreviate the parameter names):

```
matrix(dat=1:16, nc=4, nr=4, byr=TRUE, dim=list(1:4, 1:4))
```

Positional matching (rely solely on the argument order, which can be found out using the `args()` function – `args(matrix)`):

```
matrix(1:16, 4, 4, TRUE, list(1:4, 1:4))
```

Mixed matching (don't name most common arguments but special ones):

```
matrix(1:16, ncol=4, byrow=TRUE, dimnames=list(1:4, 1:4))
matrix(1:16, nc=4, byr=TRUE, dim=list(1:4, 1:4))
```


A lot of R functions have the parameter `x`, which is usually not explicitly named upon invocation.

Some function accept variadic arguments, represented by an ellipsis (`...`). Any argument not matching a named parameter will be matched to the variadic parameter:

```
args(cat)
# function (... , file="", sep=" ", fill=FALSE, labels=NULL, append=FALSE)

cat("foo", "bar", sep="-", "qux") # foo-bar-qux
```

1.3 Vectors

Make a vector from individual elements:

```
c(1, 2, 3) # 1 2 3
```

1.3.1 Sequences

Make a sequence from one to ten:

```
seq(from = 1, to = 10, by = 1)
1:10 # same but shorter (default step = 1)
```

Make a sequence with a specific length instead of step size, which will be calculated automatically:

```
seq(from = 1, to = 10, length.out = 4) # 1 4 7 10
```

Make a sequence with a specific length and step size, but omit the upper boundary:

```
seq(from = 1, by = 2, length.out = 5) # 1 3 5 7 9
```

1.3.2 Repetitions

Repeat a number:

```
rep(x = 1, times = 3) # 1 1 1
rep(1, 3) # same but shorter
```

Repeat a sequence:

```
rep(c(1, 2, 3), 2) # 1 2 3 1 2 3
rep(1:3, 2), # same but shorter
```

Repeat items instead of the whole sequence:

```
rep(1:3, each = 2) # 1 1 2 2 3 3
```

Repeat using `each` and `times` combined:

1 Basics

```
rep(1:2, each = 2, times = 2) # 1 1 2 2 1 1 2 2
```

1.3.3 Sorting

Sort (in ascending order):

```
sort(3:-3) # -3 -2 -1 0 1 2 3
```

Sort (in descending order):

```
sort(1:5, decreasing = TRUE) # 5 4 3 2 1
```

Reverse the order of a vector's elements:

```
rev(1:5) # 5 4 3 2 1
```

1.3.4 Accessing Elements

For the following examples, the vector `v` is used:

```
v <- seq(from = 10, to = 50, by = 10) # 10 20 30 40 50
```

Access the first element (the first index is 1):

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

Access the last element (the last index is the vector's length):

```
v[length(v)] # 50
```

Access multiple elements:

```
v[c(1, 3, 5)] # 10 30 50
```

```
v[1:3] # 10 20 30
```

Omit the element at a certain index:

```
v[-1] # 20 30 40 50
```

```
v[-length(v)] # 10 20 30 40
```

Omit multiple elements:

```
v[-c(1, 2, 3)] # 40 50
```

Overwrite vector elements:

```
v[1] = 11 # v = 11 20 30 40 50
```

```
v[c(2, 3)] = c(22, 33) # v = 11 22 33 40 50
```

```
v[c(4, 5)] = 44 # v = 11 22 33 44 44, 44 was used twice!
```

```
v[1:4] = c(1, 2) # v = 1 2 1 2 44
```

The vector on the left hand side must either have:

1. the same size as the vector on the right hand side, or
2. a size multiple times as big as the vector on the right hand side.

In the second case, the shorter vector is *recycled*, i.e. used repeatedly to fill up to the length of the longer vector.

1.3.5 Arithmetic on Vectors

Multiply every item of the vector by 2:

```
1:6 * 2 # 2 4 6 8 10 12
```

Multiply the items of the vector by 1 and -1, respectively:

```
1:6 * c(1, -1) # 1 -2 3 -4 5 -6
```

For the vector's sizes, the same rule applies as stated above.

Calculate the sum of a vector:

```
sum(1, 2, 3, 4) # 1+2+3+4=10
sum(1:100) # 5050
```

Calculate the product of a vector:

```
prod(1, 2, 3, 4) # 1*2*3*4=4!=24
prod(1:4) # same but shorter
```

1.4 Matrices

Create a 2x2 matrix:

```
matrix(data = c(1, 2, 3, 4), nrow = 2, ncol = 2)
```

Either `nrow` or `ncol` can be omitted:

```
matrix(1:16, nrow = 4) # 4x4 matrix (16 items)
matrix(1:25, ncol = 5) # 5x5 matrix (25 items)
```

If both `nrow` and `ncol` are omitted, a one-row matrix will be created:

```
matrix(1:10) # 1x10 matrix (10 items)
```

By default, the matrix is filled up by column:

```
matrix(1:6, ncol = 2)
```

```
1  4
2  5
3  6
```

1 Basics

This behaviour can be changed using the `byrow` parameter:

```
matrix(1:6, ncol = 2, byrow = TRUE)
```

```
1  2
3  4
5  6
```

Matrices can be built up from vectors of same lengths:

```
rbind(1:3, 4:6) # by row
```

```
1  2  3
4  5  6
```

```
cbind(1:3, 4:6) # by column
```

```
1  4
2  4
3  6
```

Find out the dimensions of a matrix:

```
m <- matrix(1:12, nrow = 3, ncol = 4)
```

```
1  4  7 10
2  5  8 11
3  6  9 12
```

```
dim(m) # 3 4 (a vector)
dim(m)[1] # 3, number of rows
nrow(m) # same but shorter
dim(m)[2] # 4, number of cols
ncol(m) # same but shorter
```

Access a matrix element:

```
m <- matrix(1:6, ncol = 3)
```

```
1  3  5
2  4  6
```

```
m[1,2] # 3 [row, col]
```

Access a whole row or column (returns a vector):

```
m[1,] # 1 3 5, first row
m[,2] # 3 4, second column
```

Rows and columns can be accessed using vectors:

```
m[1:2,] # rows 1 and 2
m[,c(1,3)] # cols 1 and 3
```

This makes it possible to select parts of a matrix:

```
m <- matrix(1:9, ncol = 3)
```

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

```
m[1:2, c(1,3)] # rows 1 and 2, cols 1 and 3
```

```
1  7
2  8
```

Access the diagonal values as a vector:

```
diag(m) # 1 5 9
```

Omit parts of a matrix:

```
m[-1,] # omit the first row
m[,-2] # omit the second column
m[-(2:3), -c(1,4)] # omit rows 2 to 3, columns 1 and 4
```

Matrix rows and columns can be overwritten like any vector (the same length rules apply):

```
m[1,] = 6 # set every value in the first row to 6
m[1:2, 2:3] = 7 # the values in the sub-matrix [1,2] to [2,3] are set to 7
m[,2] = c(1,2) # the values in the second column are set to 1, 2, 1, 2 etc.
m[1,] = m[2,] # overwrite the first row using the values of the second row
m[c(1, nrow(m)), c(1, ncol(m))] = -1 # set the values in the "corners" to -1
```

Name the dimensions of a matrix:

```
m <- (1:4, ncol = 2, dimnames = list(c("R1", "R2"), c("C1", "C2")))
```

```
   C1  C2
R1  1   3
R2  2   4
```

Dimension names can also be provided after the creation:

```
m <- (1:4, ncol = 2)
dimension(m) <- list(c("R1", "R2"), c("C1", "C2"))
```

1 Basics

1.4.1 Operations and Algebra

Transpose a matrix (A^T is the transposed matrix of A):

```
A <- matrix(1:9, ncol = 3)
```

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

```
t(A)
```

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

Create an identity matrix of size n (I_n):

```
I <- diag(x = 3)
```

```
1  0  0
0  1  0
0  0  1
```

Scalar multiplication of a matrix:

```
A <- rbind(1:3, 4:6)
```

```
1  2  3
4  5  6
```

```
A * 2
```

```
2  4  6
8 10 12
```

Addition and subtraction of matrices:

```
A <- matrix(1:4, ncol = 2)
```

```
B <- matrix(5:8, ncol = 2)
```

```
A + B
```

```
1  3      5  7      6 10
2  4      6  8      8 12
```

+ =

B - A

$$\begin{array}{rcc} 5 & 7 & \\ 6 & 8 & \end{array} + \begin{array}{rcc} 1 & 3 & \\ 2 & 4 & \end{array} = \begin{array}{rcc} 4 & 4 & \\ 4 & 4 & \end{array}$$

Two matrices, $A(m, n)$ and $B(p, q)$, can be multiplied if $n = p$ holds true (first matrix' cols = second matrix' rows), resulting in a matrix with m rows and q cols:

```
A <- matrix(c(2,6,5,1,2,4), ncol = 3) # n = 3
B <- matrix(c(3,-1,1,-3,1,5), nrow = 3) # p = 3
```

A %*% B

$$\begin{array}{rcc} \text{x} & & \\ & \text{B} = & \begin{array}{cc} 3 & -3 \\ -1 & 1 \\ 1 & 5 \end{array} \end{array}$$

A =

$$\begin{array}{rcc} 2 & 5 & 2 \\ 6 & 1 & 4 \end{array} \begin{array}{c} | \\ | \\ | \end{array} \begin{array}{cc} 3 & 9 \\ 21 & 3 \end{array} = \text{AxB}$$

A^{-1} is the inverse of a matrix A . A multiplied by A^{-1} results in the identity matrix:

```
A <- matrix(3,4,1,2), ncol = 2)
```

$$\begin{array}{rcc} 3 & 1 & \\ 4 & 2 & \end{array}$$

solve(A)

$$\begin{array}{rcc} 1 & -0.5 & \\ -2 & 1.5 & \end{array}$$

A %*% solve(A) # check the result: is it the identity matrix?

$$\begin{array}{rcc} 1 & 0 & \\ 0 & 1 & \end{array}$$

Summary:

- inverse matrix: A^{-1} , solve(A)
- transposed matrix: A^T , t(A)
- identity matrix: I_n , diag(x = n)

1.5 Multidimensional Arrays

Define arrays of different dimension:

```
array(data = 1:24) # vector 1 2 3 ... 24, 1 dimension
array(data = 1:24, dim = c(24)) # same with explicit dimension
array(data = 1:24, dim = c(4, 6)) # a 4x6 matrix, 2 dimensions
array(data = 1:24, dim = c(2, 3, 4)) # a 2x3x4 "cube", 3 dimensions
```

The dimension (2, 3, 4) stands for 2 rows, 3 cols and 4 layers. The product of the elements in the dimension vector must be equal to the length of the data vector.

Accessing parts of a multidimensional array:

```
AR <- array(1:24, c(2, 3, 4))
AR[1,,] # access the first row
AR[,2,] # access the second column
AR[, ,3] # access the third layer
```

```
AR[1,,c(1, 2)] # first row of first and second layer
```

For arrays, the same assignment rules of vectors and matrices also apply.

1.6 Logical Values

Boolean values:

```
TRUE
T # shorter for TRUE
FALSE
F # shorter for FALSE
```

Logical operations:

```
6 == 3 * 2 # equal to, TRUE
10 != 5 * 2 # not equal to, FALSE
7 > 5 # greater than, TRUE
8 < 3 # less than, FALSE
8 >= 4 * 2 # greater than or equal to, TRUE
7 <= 3 * 3 # less than or equal to, FALSE
```

TRUE and FALSE represent 1 and 0, respectively:

```
1 == TRUE # TRUE
0 == FALSE # TRUE
2 == TRUE # FALSE
```



```
T + T + T # 3
F - 4*T + 3*T # 0 - 4 + 3 = -1
```

Logical operations can be applied to vectors, matrices and arrays, applying the operator on every element and returning a vector consisting of TRUE and FALSE:

```
1:3 == seq(from = 1, to = 3) # TRUE TRUE TRUE
1:3 == c(1, 2, 4) # TRUE TRUE FALSE
```

Like assignments, comparisons can be performed on vectors of different lengths (according to the same rules, a shorter right hand side vector will be recycled):

```
1:4 > 2:3 # 1>2, 2>3, 3>2, 4>3; evaluates to FALSE FALSE FALSE TRUE
```

Check if at least one element evaluates to TRUE:

```
any(1:3 > 2) # TRUE, 3 is bigger than 2
```

Check if all elements evaluate to TRUE:

```
all(10:20 >= 11) # FALSE, 10 is smaller than 11
```

1.6.1 Logical Operations

Compare boolean values using double operators:

```
TRUE && TRUE # logical AND, returns TRUE
FALSE || TRUE # logical OR, returns TRUE
!TRUE # logical NOT, returns FALSE
```

Compare elements of a vector (or a matrix, or an array of higher dimensions) using single operators:

```
c(T, F, F) & c(T, T, F) # TRUE FALSE FALSE
c(T, T, T) | c(T, T, F) # TRUE TRUE FALSE
```

Single operators have the same behaviour as double operators when applied to scalar values rather than vectors. Double operators applied to vectors will only apply to the first elements of the vectors involved:

```
TRUE & FALSE # FALSE
FALSE | TRUE # TRUE
```

```
c(T, F, F) && c(T, T, T) # TRUE
c(F, T, T) || c(F, T, T) # FALSE
```

1.6.2 Element Selection

Select elements of a vector (or a matrix, or an array) using logical flags:

1 Basics

```
v <- 1:5 # 1 2 3 4 5
v[c(T, T, F, T, F)] # using a "flag" vector, returns 1 2 4
v[v >= 3] # using a condition, returns 3 4 5
```

Select every other element using vector recycling:

```
v <- 1:10
v[c(1,0)] # 1 3 5 7 9
```

Select all leap years of a range of years:

```
y <- 1987:2017
y[y %% 4 == 0 & (y %% 100 != 0 | y %% 400 == 0)]
# 1988 1992 1996 2000 2004 2008 2012 2016
```

Set all negative values to zero:

```
v <- -3:3 # -3 -2 -1 0 1 2 3
v[v < 0] = 0 # 0 0 0 0 1 2 3
```

Find out the indices of items matching a condition using the `which()` function:

```
v <- 3:8 # 3 4 5 6 7 8
which(x = (v %% 2 == 0)) # indices of even numbers: 2 4 6
```

The resulting vector can be used to invert the selection:

```
v[-which(x = (v %% 2 == 0))] # indices of odd numbers: 1 3 5
```

By default, `which()` treats matrices just like vectors:

```
m <- matrix(2:10, ncol = 3)
```

```
2  5  8
3  6  9
4  7 10
```

```
which(x = (m %% 2 == 1)) ## odd element's indices: 2 4 6 8
```

To get row/col coordinates, use the `arr.ind` flag:

```
which(x = (m %% 2 == 1), arr.ind = TRUE)
```

```
row col
2    1
1    2
3    2
2    3
```

1.7 Strings

Store a simple string:

```
s <- "This is a simple string!"
```

Find out the length of a string:

```
nchar("foobar") # 6
length("foobar") # 1, a string is considered a vector of length 1
```

Compare strings:

```
"foo" == "foo" # TRUE
"foo" == "bar" # FALSE
"bar" == c("foo", "bar", "qux") # FALSE TRUE FALSE
```

Compare strings using alphabetic order:

```
"Anna" > "Berti" # TRUE
```

Uppercase strings are considered bigger than lowercase string:

```
"A" > "a" # TRUE
"B" <= "b" # FALSE
```

This distinction only applies to alphabetically equivalent strings:

```
"A" > "z" # FALSE
```

Almost all characters can be used within a string. Double quotes and backslashes have to be escaped using a backslash:

```
"He said: \"a backslash: \\...\"" # He said: "a backslash: \..."
```

Other escape sequences are:

```
\n line break
\t tab
\b backspace
```

For a complete list of escape sequences, type `?Quotes`.

1.7.1 Concatenation

Strings can be concatenated:

```
cat("hello", "world") # prints "hello world"
paste("hello", "world") # returns "hello world"
```

The separator (a space character, by default) can be defined:

1 Basics

```
cat("foo", "bar", "qux", sep="---") # "foo---bar---qux"
cat("foo", "bar", "qux", sep="") # "foobarqux"
```

Numbers are automatically converted to strings (*coercion*):

```
numbers <- 5:1
cat("Countdown:", numbers) # Countdown: 5 4 3 2 1
cat(2, "times", 3, "is", 2 * 3) # 2 times 3 is 6
cat("is", 5, "bigger than", 7, 5 > 7) # is 5 bigger than 7 FALSE
```

1.7.2 Substrings and Replacements

Extract a substring (using 1-based inclusive indices):

```
substr(x = "this is", start = 1, stop = 4) # "this"
```

Substrings can be replaced by other strings of the same length:

```
s <- "this is cool"
substr(x = s, start = 1, stop = 4) <- "that" # "that is cool"
```

Replacements are done more effectively using `sub()` (replaces the first occurrence) and `gsub()` (replaces all occurrences):

```
s <- "foo too"
sub(pattern = "oo", x = s, replacement = "u") # fu too
gsub(pattern = "oo", x = s, replacement = "u") # fu tu
```

1.8 Factors

Factors are a special kind of vectors for storing categorical data, similar to enumerations in Java or C. Next to the value, factors also store a level:

```
colors <- factor(c("red", "green", "blue"))
```

When a factor is subsetted, *some* of the values but *all* of the levels stay:

```
colors[1:2]
```

```
red green
```

```
Levels: blue red green
```

Factors allow ordering:

```
weekdays <- c("Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun")
workdays <- c("Mon", "Tue", "Wed", "Thu", "Fri")
factor(x = workdays, levels = weekdays, ordered = TRUE)
```

```
Mon Tue Wed Thu Fri
Levels: Mon < Tue < Wed < Thu < Fri < Sat < Sun
```

1.8.1 Cutting

The `cut()` function can be used to break up data points on a continuum into discrete intervals:

```
weights <- c(72, 83, 61, 119, 88, 155)
w.breaks <- c(0, 70, 90, 120, 200)
cut(x = weights, breaks = w.breaks)
```

Output:

```
(70,90] (70,90] (0,70] (90,120] (70,90] (120,200]
Levels: (0,70] (70,90] (90,120] (120,200]
```

(70,90] means: from 70 exclusive to 90 inclusive. Use the parameter `right = FALSE` for inclusive/exclusive intervals ([70,90)).

The intervals can be named using labels:

```
w.labels <- c("low", "normal", "high", "obese")
cut(x = weights, breaks = w.breaks, labels = w.labels)
```

Output:

```
normal normal low high normal obese
Levels: low normal high obese
```

1.8.2 Splitting

Split the data up into a list grouped by the factor:

```
segments <- cut(x = weights, breaks = w.breaks, labels = w.labels)
groups <- split(x = weights, f = segments)
```

Output:

```
$low
[1] 61

$normal
[1] 72 83 88

$high
[1] 119

$obese
```

1.9 Lists

Lists can contain elements of different data types, including other lists, matrices etc.

Create a list containing three different sized vectors of different types:

```
l <- list(c("a", "b", "c"), c(1:5), c(TRUE, FALSE))
```

To access list elements, use double square brackets:

```
l[[1]] # the vector "a" "b" "c"  
l[[2]] # the vector 1 2 3 4 5  
l[[3]] # the vector TRUE FALSE
```

```
l[[1]] <- c("X", "Y") # overwrite the first element
```

```
l[[3]][2] # the second vector element of the third list item (FALSE)
```

To access multiple elements at once, use list slicing rather than double square brackets:

```
l[c(2,3)] # the vectors 1 2 3 4 5 and TRUE FALSE
```

List elements can be named:

```
names(l) <- c("chars", "numbers", "logicals")
```

List elements can also be named upon initialization:

```
l <- list(chars = c("a", "b", "c"), numbers = 1:5, logicals = c(T, F))  
names(l) # "chars" "numbers" "logicals"
```

To access list elements by name (rather than index), use dollar notation:

```
l$chars # "a" "b" "c"  
l$numbers # 1 2 3 4 5  
l$logicals # TRUE FALSE
```

```
l$chars[1] # "a"  
l$numbers[5] # 5  
l$logicals[2] # FALSE
```

An element can be added to the list by assignment:

```
l$newElement <- c("new", "character", "vector")
```

Lists can also be nested:

```
foo <- list(char = "A", num = 1, logical = TRUE)
bar <- list(char = "z", num = 9, logical = FALSE)

l <- list(first = foo, second = bar)

l$first$char # "A"
l$second$logical # FALSE
```

1.10 Data Frames

A data frame is a special kind of list with the restriction that the members must be all vectors of equal length. (Shorter vectors will be recycled, if possible).

Create a data frame:

```
s <- c("cow", "spider", "whale")
l <- c(4, 8, 0)
m <- c(T, F, T)
animals <- data.frame(species = s, legs = l, mammal = m)
```

Output (a table with named columns and numbered rows):

	species	legs	mammal
1	cow	4	TRUE
2	spider	8	FALSE
3	whale	0	TRUE

Elements can be accessed like matrices using row and column indices:

```
animals[3][1] # whale
animals[,2] # 4 8 0
animals[c(1,3),1] # cow whale
```

Since the element vectors of a data frame are named, they can be accessed using that name:

```
animals$species # cow spider whale
animals$species[2] # spider
```

The dimensions of a data frame can be explored using the same functions as for matrices:

```
nrow(animals) # 3
ncol(animals) # 3
dim(animals) # 3 3
```

String values are treated as factors by default. This can be prevented upon creation:

```
a <- data.frame(species = s, legs = l, mammal = m, stringsAsFactors = F)
```

1 Basics

If certain (but not all) non-numeric columns should be factors, they have to be created as factors in the first place:

```
s <- c("cow", "spider", "whale")
l <- c(4, 8, 0)
m <- factor(c(T, F, T))
a <- data.frame(species = s, legs = l, mammal = m, stringsAsFactors = F)
```

Rows can be added to a data frame by creating a new data frame of similar structure and adding it to the existing data frame:

```
bird <- data.frame(species = "bird", legs = 2, mammal = FALSE)
a <- rbind(a, bird)
```

Columns can be added to a data frame by creating a new vector and adding it:

```
area <- c("land", "land", "sea", "air")
a <- cbind(a, area)
```

New columns can also be made up using values of existing columns:

```
a$toesPerFoot = c(0, 0, 0, 3)
a$toes = a$legs * a$toesPerFoot
```

Rows can be selected using logical expressions:

```
a[a$mammal == TRUE, 1] # selects the first mammal of the data frame
```

To select multiple columns, a vector of names can be used:

```
a[1:2, c("species", "mammal")] # columns species and mammal of row 1 and 2
```

1.11 Special Values

1.11.1 Infinity

Infinity (Inf) is not a number, but a concept describing a number higher than the highest representable number, which is platform dependent::

```
12800 ^ 75 # 1.098368e+308
12900 ^ 75 # Inf
```

There is positive and negative infinity (-Inf):

```
Inf > 10e24 # TRUE
-Inf < -10e24 # TRUE
```

Arithmetic operations involving infinity always result in (positive or negative) infinity:


```
10e24 - Inf # -Inf
2 * Inf == Inf # TRUE
Inf + Inf - 2 * -Inf == 0 # FALSE
```

Expressions can be tested for finity/infinity:

```
is.finite(12800^75) # TRUE (on my machine)
is.infinite(12900^75) # TRUE (ditto)
is.finite(5 / 0) # FALSE
```

1.11.2 Not a Number

Some expressions cannot be represented as a number. They are represented as NaN:

```
0 / 0 # NaN
-Inf + Inf # NaN
Inf / Inf # NaN
```

NaN is not considered finite:

```
is.finite(NaN) # FALSE
```

Expressions can be tested if they are “not a number”:

```
is.nan(NaN) # TRUE
is.nan(134) # FALSE
is.nan(5 / 0) # FALSE, it's a number considered infinite
is.nan(0 / 0) # TRUE
is.nan(sqrt(-1)) # TRUE

!is.nan(13.7) # TRUE
!is.nan(13000 ^ 75) # TRUE, it's a infinite number (on my machine)
```

1.11.3 NULL

NULL stands for emptiness – in contrast to NA, which stands for a missing entry. As opposed to NA, NULL cannot be part of a vector:

```
v <- c(1, 2, NULL, 4) # 1 2 4
length(v) # 3, not 4
c(NULL, NULL, NULL) # NULL
```

NULL values can be detected:

```
foo <- NULL
is.null(foo) # TRUE
```

1 Basics

```
bar <- "hello"  
is.null(bar) # FALSE
```

NULL can be used in arithmetic expressions with the effect of returning the resulting type.

```
17 + NULL # numeric(0)  
NULL >= 5 # logical(0)
```

NULL dominates in combination with Inf, NaN and NA:

```
NULL + Inf - NaN + 3 * NA # numeric(0)
```

1.12 Objects

1.12.1 Attributes

Every object can store additional attributes:

```
o <- 42  
o.description = "The answer to everything"
```

Show the attributes of an object:

```
m <- matrix(1:4, ncol = 2)  
attributes(m)
```

```
$dim  
2 2
```

Access an attribute:

```
attributes(m)$dim # 2 2  
attr(x = m, which = "dim") # same using a string
```

Some attributes have their own function:

```
dim(m) # 2 2
```

1.12.2 Classes

Find out the class of an object:

```
class(c(1, 2, 3)) # "numeric"  
class("foo") # "character"  
class(matrix(1:4)) # "matrix"  
class(array(1:100)) # "array"  
class(factor(c("R", "G", "B")))) # "factor"
```

```
class(5 > 3) # "logical"
class(length(c(1, 2, 3))) # "integer"
```

Some objects have multiple classes due to inheritance:

```
bits <- factor(x = c(1, 0, 0, 1, 0), levels = c(0, 1), ordered = TRUE)
class(bits) # "ordered" "factor"
```

Objects can be checked whether or not they are of a certain class:

```
is.numeric(3) # TRUE
is.character("abc") # TRUE
is.matrix(matrix(1:4)) # TRUE
is.array(array(1:100)) # TRUE
is.factor(factor(c(1, 0, 1, 1, 0))) # TRUE
is.logical(5 > 3) # TRUE
is.integer(length(1:3)) # TRUE
is.vector(1:3) # TRUE
```

Convert explicitly from one type to another (coercion):

```
as.numeric("12") # 12
as.numeric("1.2e5") # 12000
as.character(13) # "13"
as.numeric("howdy!") # NA
as.logical(0) # FALSE
as.logical(as.numeric(c("1", "0", "0", "1"))) # TRUE FALSE FALSE TRUE
```

```
m <- matrix(1:4, ncol = 2)
as.vector(m) # 1 2 3 4
```

```
a <- array(1:8, dim = c(2, 2, 2))
as.matrix(a)
```

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

```
as.vector(a) # 1 2 3 4 5 6 7 8
```

1.13 Plotting

Simple plots can be drawn using two vectors of x and y coordinates of the same length:

```
x <- c(3, 2, 7, 8)
y <- c(2, 5, 6, 1)
plot(x, y)
```

When using matrices for plotting, the first column holds the x values, and the second column holds the y values:

```
m <- matrix(1:4, ncol = 2)

1  3  # point (1;3)
2  4  # point (2;4)

plot(m)
```

When using a single vector for plotting, each value serves as the x and the y coordinate at the same time:

```
plot(1:3) # plots (1;1), (2;2), (3;3)
```

Plots can be (optically) enhanced using various options:

- `type`: the plotting style, a single character
 - `"p"`: points
 - `"l"`: lines
 - `"b"`: both (points and lines)
 - `"c"`: empty points joined by lines
 - `"o"`: overplotted points and lines
 - `"s"`: stair steps (lower Riemann areas)
 - `"S"`: stair steps (upper Riemann areas)
 - `"n"`: none
- `main`: the main title of the plot, characters
- `xlab`, `ylab`: labels for the x and y axis, characters
- `pch`: the character to be used to draw the dots, either a number from 1 to 25 (pre-defined styles) or any single character
- `cex`: character expansion, stretches the point by the given factor, a number
- `lty`: the line type (`"solid"`, `"dotted"` or `"dashed"`)
- `lwd`: the line width, a number
- `xlim`, `ylim`: horizontal and vertical ranges as vectors (`c(lower, upper)`)
- `col`: the color for the dots/lines, either a color name, a number (from 1 to

8) or a hex code (like `#ffffff` for white)

1.13.1 Additional Elements

Calling the `plot()` function always creates a new output and removes the old one. Elements can be added to an existing plot using these functions:

```
lines(x = c(1, 3, 2), y = c(3, 1, 2))
# plots a line connecting P(1;3), P(3;1) and P(2;2)

points(x = c(1, 3, 2), y = c(3, 1, 2))
# plots the points P(1;3), P(3;1) and P(2;2)

text(x = 5, y = 3, "Nothing") # the text "Nothing" centered around P(5;3)

arrows(x0 = 1, y0 = 2, x1 = 4, y1 = 7)
# draws an arrow pointing from P(1;2) to P(4;7)

abline(a = 2, b = 3) # line with the slope 2 and y intercept 3 (P(0;3), that is)

abline(h = 2) # a horizontal line on y = 2
abline(v = 1) # a vertical line on x = 1

segments(x0 = c(1, 4), y0 = c(1, 1), x1 = c(1, 4), y1 = c(5, 5))
# two vertical lines from P(1;1) to P(1;5) and from P(4;1) to P(4;5)

legend(x = "bottomleft", legend = c("Male", "Female"), pch = c("+", "x"))
# a legend on the bottom left position (+: Male, x: Female)
```

1.13.2 TODO: Barplots**1.13.3 TODO: Pie Charts****1.13.4 TODO: Histograms****1.13.5 TODO: Box-and-Whisker Plots****1.13.6 TODO: Scatterplots****1.13.7 Saving Plots to Files**

Plots can be saved to files. Various formats are supported, for example PNG, JPEG, TIFF, BMP, PDF and EPS (postscript). Plotting to a screen is a special case of plotting: it draws directly on the screen device. Files are handled as devices, too..

In order to save a plot to a file, just open the file device, do the plots and close the device:

1 Basics

```
png(filename = "plot.png")
plot(1:10, 2:11)
dev.off()
```

The dimensions are 480x480 pixels by default, but can be overwritten:

```
png(filename = "plot.png", width = 800, height = 600)
```

Other units than pixels can be used when providing the DPI resolution:

```
png(filename = "plot.png", width = 9, height = 8, units = "cm", res = 300)
```

For postscript and PDF output, the default unit is inches – and the filename parameter is called file:

```
pdf(file = "plot.pdf", width = 8, height = 6)
```

For SVG output, parameters such as the font and the background can be defined:

```
svg(filename = "plot.svg", family = "serif", bg = "grey")
```

1.13.8 Plotting Examples

1.13.8.1 Example 1: Drawing Features

```
plot(x = c(), xlim = c(-3, 3), ylim = c(7, 13), xlab="", ylab="")
abline(v = c(-3,3), h = c(7,13), lty = "dashed", col = "grey", lwd = 3)
x0 = c(-2.5, -2.5, -2.5, 2.5, 2.5, 2.5)
y0 = c(7.5, 10, 12.5, 12.5, 10, 7.5)
x1 = c(-1, -1, -1, 1, 1, 1)
y1 = c(9.5, 10, 10.5, 10.5, 10, 9.5)
arrows(x0, y0, x1, y1)
text(x = 0, y = 10, "SOMETHING\nPROFOUND")
```

1.13.8.2 Example 2: Colored Graph and Legend

```
fw <- c(55, 42, 58, 67)
fh <- c(161, 154, 170, 178)
mw <- c(85, 75, 93, 63, 75, 89)
mh <- c(185, 174, 188, 178, 167, 181)
plot(x = c(), main = "Female/Male: Height by Weight",
     xlab = "Weight (kg)", ylab = "Height (cm)",
     xlim = c(40, 100), ylim = c(150, 200))
points(fw, fh, pch = "♀", col = "red")
points(mw, mh, pch = "♂", col = "blue")
legend(x = "topleft", legend = c("Female", "Male"),
      pch = c("♀", "♂"), col = c("red", "blue"))
```

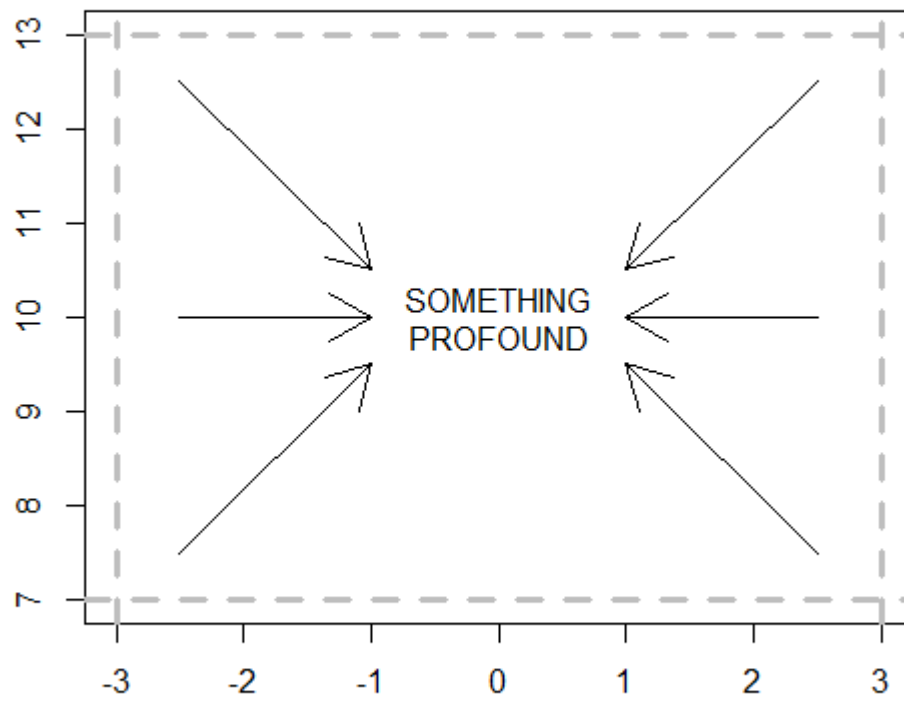


Figure 1.1: Example 1: Various Drawing Features

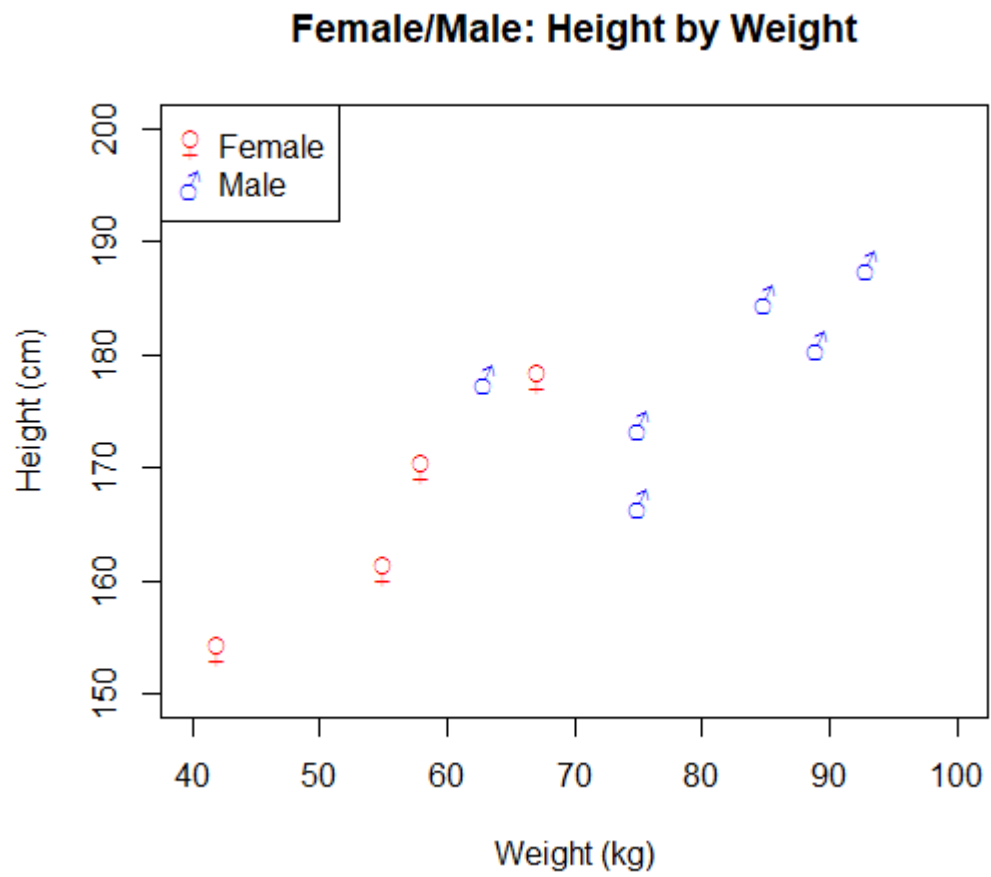


Figure 1.2: Example 2: Colored Graph with Legend

1.13.8.3 Example 3: Law of Large Numbers

```

nTosses <- 500

## toss the coin, get head ("H") or tails ("T")
tosses <- sample(c('H', 'T'), size = nTosses, replace = TRUE)

v <- rep(0, length(tosses))
heads <- data.frame(tossNumber = v, headCount = v, headRatio = v)

for (n in 1:length(v)) {
  h <- 0
  for (i in 1:n) {
    if (tosses[i] == 'H') {
      h <- h + 1
    }
  }
  heads$tossNumber[n] = n
  heads$headCount[n] = h
  heads$headRatio[n] = heads$headCount[n] / n
}

plot(x = c(), type = "l", xlim = c(0, nTosses), ylim = c(0,1),
     main = "Coin Tosses", xlab = "Toss Number", ylab = "Heads Ratio")
abline(h = 0.5, col = "red")
lines(x = heads$tossNumber, y = heads$headRatio)
axis(side = 2, at = c(0.5))

```

1.14 Pre-Installed Data Sets

R comes with of pre-installed data sets, which can be listed:

```
library(help = "datasets")
```

To get more information about one of the data sets listed, just use the help function:

```
?Titanic # Survival of passengers on the Titanic
```

Other data sets can be installed just like packages, for example the tseries package:

```
install.packages("tseries")
```

Load the ice.river data set into the current workspace:

```
data(ice.river)
```

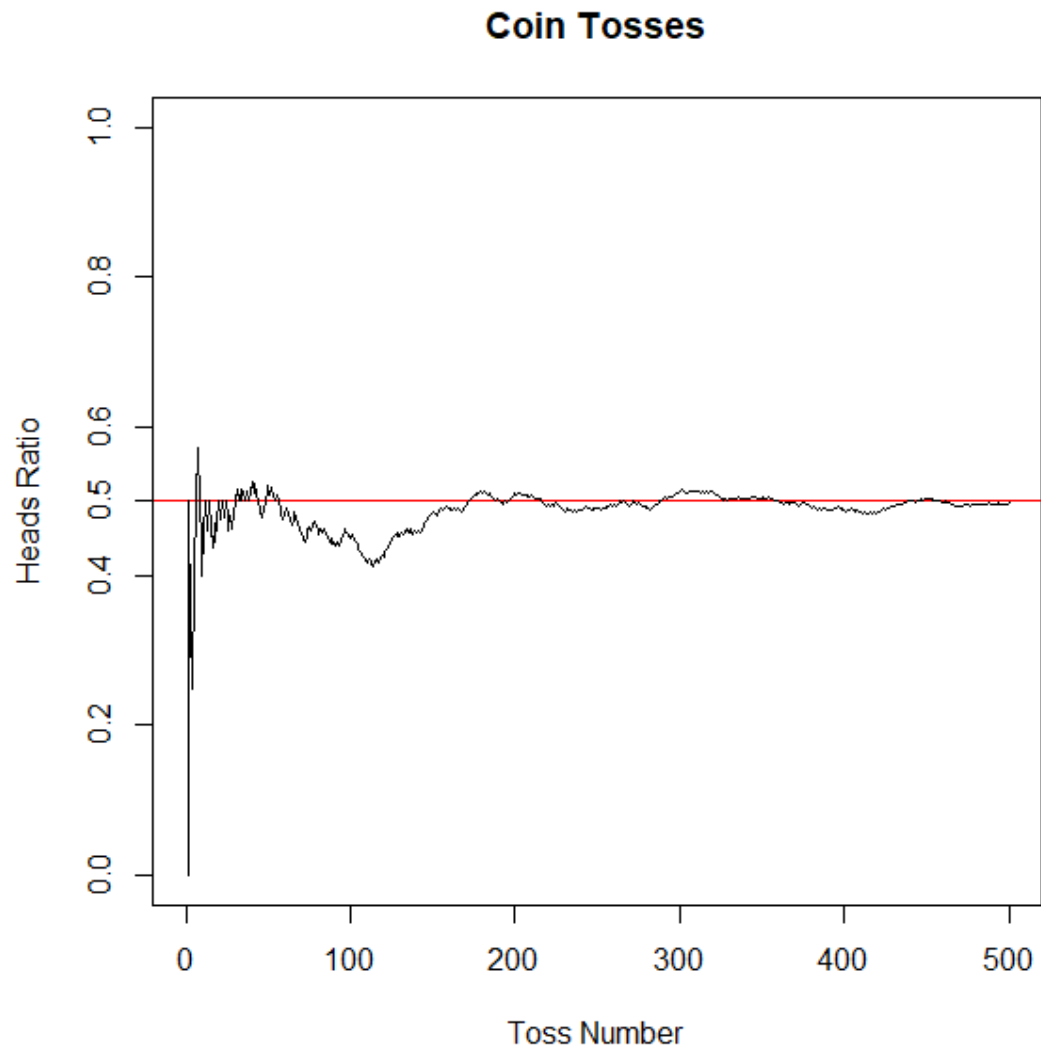


Figure 1.3: Example 3: Coin Tosses: Law of Large Numbers

Now the `ice.river` data set can be accessed and explored like any variable.

A good resource for free statistical example data is the Journal of Statistics Education (JSE).

1.15 Working with Files

1.15.1 Reading Text Files

Sample file (`table.txt`):

```
species mammal legs area
cow TRUE 4 land
spider FALSE 8 land
whale TRUE N/A sea
bird TRUE 2 air
```

Read tabular data from a file (`table.txt`):

```
animals <- read.table(file = "table.txt", header = TRUE, sep = " ",
  na.strings = "N/A", stringsAsFactors = FALSE)
```

Parameters:

- `file`: the (absolute or relative) file name
- `header`: whether or not the first line should be read as a header
- `sep`: the separator (here: one space, use `" "` for any amount of whitespace)
- `na.strings`: define which strings (either a single string or a vector of strings) should be recognized as NA values
- `stringsAsFactors`: whether or not string columns should be interpreted as factors (same parameter as for `data.frame`)

If *some* of the non-numeric columns should be interpreted as factors, simply overwrite them, providing optional levels (in case some possible values are missing in the data set):

```
animals$area = factor(animals$area)
animals$area = factor(animals$area, levels = c("air", "land", "sea"))
```

Either use an absolute file path or make sure to change your working directory:

```
getwd() # "C:/Users/patrick.bucher/Documents/R"
setwd("tables") # relative path (absolute paths are also possible)
getwd() # "C:/Users/patrick.bucher/Documents/R/tables"
```

List all files in the current working directory (for more information type `?list.files` and `?list.dirs`):

```
list.files()
```

Files can also be chosen interactively, returning the absolute path of the file selected:

1 Basics

```
myFile <- file.choose()
```

Files can also be read directly from the web:

```
forbes500 <- read.table(file = "http://forbes.com/assets/filthy-rich-people.txt")
```

1.15.2 Reading CSV Files

Sample file (table.csv):

```
species,mammal,legs,area
cow,TRUE,4,land
wolf spider,FALSE,8,land
whale,TRUE,N/A,sea
guinea pig,TRUE,4,land
```

Read the data from a CSV file (with the comma as the default separator):

```
animals <- read.csv(file = "table.csv", header = TRUE,
  stringsAsFactors = TRUE)
```

CSV files often use the semicolon (;) or the tab (\t) as the separator value, so make sure to define the sep parameter accordingly:

```
animals <- read.csv(file = "table.csv", header = TRUE,
  stringsAsFactors = TRUE, sep = ';')
```

1.15.3 Writing Files

Sample data frame:

```
names <- c("Sepp", "Max", "Uschi")
sex <- factor("M", "M", "F")
age <- c(42, 50, 61)
people <- data.frame(person = names, sex = sex, age = age)
```

Write the data frame to a tabular text file:

```
write.table(x = people, file = "people.txt")
```

Write the data frame to a CSV file:

```
write.csv(x = people, file = "people.csv", row.names = TRUE)
```

The argument row.names adds an unnamed column with a row counter to the output.

Store any single object in a file:

```
m <- matrix(1:64, ncol = 8) # 8x8 matrix
dput(x = m, file = "matrix.txt")
```

Retrieve a formerly stored object from a file:

```
m <- dget(file = "matrix.txt")
```

1.16 Timing

Pause the program execution for a given amount of time (in seconds):

```
for (i in 1:10) {
  print(i)
  Sys.sleep(0.5)
}
```

Display a textual progress bar:

```
from <- 1
to <- 10
prog <- txtProgressBar(min = from, max = to, char = "#", style = 3)
cat("counting from", from, "to", to, "\n")
for (i in from:to) {
  setTxtProgressBar(prog, value = i)
  Sys.sleep(0.5)
}
close(prog)
```

Get the current date and time:

```
Sys.time() # returns current date time
Sys.Date() # returns current date
```

Format date and time:

```
format(Sys.time(), "%d.%m.%Y %H:%M") # 30.07.2017 17:41
format(Sys.time(), "%s") # 1501429298, UNIX timestamp
```

Measure execution time (difference between two times):

```
start <- Sys.time()
Sys.sleep(1.23)
end <- Sys.time()
diff <- end - start # diff is of type "difftime"
as.numeric(diff, units = "secs") # time difference in seconds, around 1.23
```

Measure duration of a function call (similar to the UNIX command `time`):

```
system.time(sqrt(mean(1:1e9)))
```

Output:

1 Basics

user	system	elapsed
1.504	0.233	1.738

2 Programming

2.1 Conditions

2.1.1 If, Else If, Else

Simple if-else if-else conditions:

```
r <- sample(1:10, 1) # random number from 1 to 10

if (r > 6) {
  print("big")
} else if (r < 4) {
  print("small")
} else {
  print("medium")
}
```

Combined logical conditions:

```
year <- sample(c(1900:2100), 1) # a random year from 1900 to 2100

if (year %% 4 == 0 && (year %% 100 != 0 || year %% 400 == 0)) {
  cat(year, "is a leap year")
} else {
  cat(year, "is not a leap year")
}
```

Apply conditions to a series of numbers:

```
x <- sample(1:10, 5) # 5 numbers from 1 to 10
y <- sample(0:2, replace = TRUE, 5) # 5 numbers from 0 to 2

q <- ifelse(test = (y != 0), yes = (x / y), no = NA)
# divide x by y if y is not equal to y, otherwise return NA
```

2.1.2 Switch-Case

Switch-case in R is implemented as a function:

2 Programming

```
animal <- sample(c("Spider", "Cow", "Bird"), 1)
legs <- switch(EXPR = animal, Spider = 8, Cow = 4, Bird = 2)
cat(animal, legs) # prints either "Spider 8", "Cow 4" or "Bird 2"
```

2.2 Loops

2.2.1 For Loops

Loop over the elements of a vector (by value):

```
abc <- c("A", "B", "C")
for (i in abc) {
  print(i)
}
# prints "A" "B" "C"
```

Loop over the elements of a vector (by index):

```
abc <- c("A", "B", "C")
for (i in 1:length(abc)) {
  print(abc[i])
}
# prints "A" "B" "C", too
```

Nested loops (implementing the “Bubble Sort” algorithm):

```
x <- sample(1:10, 10)
print(x)
for (i in 1:length(x)) {
  for (j in 1:length(x)) {
    if (x[i] < x[j]) {
      tmp <- x[i]
      x[i] <- x[j]
      x[j] <- tmp
    }
  }
}
```

2.2.2 While Loops

Loop as long as a condition holds true:

```
x <- 3
while (x > 0) {
```



```

    print(x)
    x <- x - 1
}
# prints 3 2 1

```

Use a loop to get input from the user:

```

number = 0
while (number <= 0) {
  input = readline(prompt = "Enter a positive number: ")
  number = as.numeric(input)
}

```

2.2.3 Implicit Looping

Apply a function to the columns or rows of a matrix (define the dimension the function should be applied to using the MARGIN parameter):

```

m <- matrix(data = sample(1:16, 16), ncol = 2)
rowSums = apply(X = m, MARGIN = 1, FUN = sum)
colSums = apply(X = m, MARGIN = 2, FUN = sum)

```

Create a random matrix, sum up its rows and put the result in a new matrix:

```

numbers <- sample(1:8, 8)
m <- matrix(data = numbers, ncol = 2, dimnames = list(1:4, c("A", "B")))
rowSums <- apply(X = m, MARGIN = 1, FUN = sum)
columns <- cbind(m, rowSums)
m2 <- matrix(columns, ncol = 3, dimnames = list(1:4, c("A", "B", "Sum")))

```

Apply a function to every member of a list:

```

l <- list("a", 13, TRUE)
lapply(X = l, FUN = is.numeric) # returns a list (FALSE TRUE FALSE)
sapply(X = l, FUN = is.numeric) # returns a vector (FALSE TRUE FALSE)

```

2.2.4 Leave a Loop

A loop can be left prematurely using the break keyword (a binary search to guess a secret number):

```

min = 1
max = 100
secret = sample(min:max, 1)
print(paste("don't tell: the secret number is", secret))

guess = 0

```

2 Programming

```
tries = 0
while (TRUE) {
  guess = as.integer((min + max) / 2)
  print(paste("guessed", guess))
  tries <- tries + 1
  if (guess == secret) {
    print("right")
    print(paste("found the secret after", tries, "attempts"))
    break
  } else {
    print("wrong")
    if (guess > secret) {
      max = guess
    } else {
      min = guess
    }
  }
}
```

2.2.5 Skip a Loop Item

A loop item can be skipped using the next keyword (a loop that performs divisions only on non-zero divisors):

```
n <- 10
dividends = sample(1:100, n)
divisors = sample(0:3, replace = TRUE, n)

for (i in 1:n) {
  if (divisors[i] == 0) {
    next
  }
  q <- dividends[i] / divisors[i]
  print(paste(dividends[i], "/", divisors[i], "=", q))
}
```

2.2.6 Repeat

Instead of writing while(TRUE), an endless loop can be defined using the repeat keyword (such a loop can only be ended using break):

```
print("CC: CrappyCalculator")
repeat {
```

```

i <- trimws(readline("Enter '+' for addition or 'q' to quit: "))
if (i == "q") {
  break
}
if (i != "+") {
  next
}
a <- as.numeric(readline("Enter a number: "))
b <- as.numeric(readline("Enter another number: "))
print(paste(a, "+", b, "is", a + b))
}

```

2.3 Functions

A recursive function to calculate Fibonacci numbers:

```

fib <- function(n) {
  if (n == 1 || n == 2) {
    return(1)
  } else {
    return(fib(n - 2) + fib(n - 1))
  }
}

```

If return is left away, the last object created in the lexical environment will be returned (this function works exactly like the one above):

```

fib <- function(n) {
  if (n == 1 || n == 2) {
    1
  } else {
    fib(n - 2) + fib(n - 1)
  }
}

```

The variadic arguments ... first have to be converted into a vector or list in order to work with them:

```

numberOfArguments <- function(...) {
  args <- c(...)
  return(length(args))
}
numberOfArguments(1, 2, 3)
numberOfArguments(1:10)

```

2 Programming

Function definitions can be nested within other functions:

```
average <- function(...) {  
  args = c(...)  
  sum <- function(x) {  
    s <- 0  
    for (i in x) {  
      s <- s + i  
    }  
    return(s)  
  }  
  return(sum(args) / length(args))  
}  
average(sample(1:10, replace = TRUE, 100))
```

Functions can also be defined *ad hoc*, so called disposable functions:

```
supply(1:10, FUN = function(x) { x ** 2 })  
# squares all the numbers from 1 to 10
```

When defining a function with a name already used (such as `sum`), the new user-defined function hides the original function:

```
sum(1:10) # calls the original function  
  
sum <- function(...) {  
  print("this sum function is user-defined")  
  total <- 0  
  for (i in c(...)) {  
    total <- total + i  
  }  
  return(total)  
}  
  
sum(1:10) # calls the user-defined function
```

The original function can be accessed using a package qualifier:

```
base::sum(1:10) # calls the original function
```

2.4 Warnings and Exceptions

Throw a warning or an exception:

```
safeDivide <- function(x, y) {  
  if (x == 0) {
```

```

    # throw a warning message
    warning("zero value can't be divided")
  }
  if (y == 0) {
    # throw an exception, halts the execution
    stop("can't divide by zero")
  }
  return(x / y)
}

```

```

saveDivide(0, 2) # causes warning
saveDivide(2, 0) # causes exception, halts execution
print("done") # this won't be executed

```

Catch an error (silent = TRUE suppresses the original error message):

```

x <- try(saveDivide(2, 0), silent = TRUE)
if ("try-error" == attr(x, "class")) {
  print("division failed")
} else {
  print(x)
}

```

Suppress a warning:

```

sqrt(-1): # returns NaN, arning: "NaNs produced"
suppressWarning(sqrt(-1)) # just returns NaN

```

Advanced error handling with tryCatch (using the saveDivide(x, y) function from above):

```

dividends = sample(0:3, 4)
divisors = sample(0:3, 4)

for (c in 1:4) {
  a = dividends[c]
  b = divisors[c]
  cat("try to divide", a, "by", b, "\n")

  result <- tryCatch({
    # try part
    saveDivide(a, b)
  }, warning = function(warning) {
    # catch part (for warnings)
    return(0)
  }, error = function(error) {
    # catch part (for errors)
    return(NA)
  })
}

```

2 Programming

```
    }, finally = {  
        # finally part (for cleanup)  
        cat("division", a, "by", b, "done\n")  
    })  
  
    cat("result:", result, "\n")  
}
```

Example output:

```
try to divide 2 by 2  
division 2 by 2 done  
result: 1  
try to divide 1 by 3  
division 1 by 3 done  
result: 0.3333333  
try to divide 3 by 1  
division 3 by 1 done  
result: 3  
try to divide 0 by 0  
division 0 by 0 done  
result: 0
```

3 Statistics

3.1 Basic Terms

- raw data: records or observations that make up a sample
- numeric variables
 - continuous: weight of a person, distance between two points
 - discrete: number of coin tosses, population of a village
- categorial variables
 - nominal (unranked categories): color, nationality
 - ordinal (ranked categories): level (low, medium, high)
- univariate data: only one dimension (weight, length, duration etc.)
- multivariate data: more than one dimension (latitude/longitude, x/y position, weight/height)
- parameter: characteristic of a population
 - true mean μ
- statistic: characteristic of a sample (drawn from the population)
 - sample mean \bar{x}

3.2 Summary Statistics

3.2.1 Mean, Median, Mode

Calculate the mean (arithmetic average) and median (the middle value for a set with odd length or the mean of the two middle values for a set with an even length, respectively):

```
data <- c(1,1,1,2,2,3,3,3,3,4,4,4,4,5,5,6,7,8,8,9)
```

```
mean(data) # 4.15
```

```
median(data) # 4
```

Calculate the mode (most common value(s) in a set):

```
mode <- function(v) {  
  # convert values to a table (containing the values and the number of  
  # occurrences for each value)  
  t <- table(v)
```

3 Statistics

```
# select the table entries with the biggest number of occurrences
m <- t[t == max(t)]

# return the value with the most occurrences as a number
return(as.numeric(names(m)))
}
mode(data) # 3 4 (the most common values, both occur four times)
```

Apply summarizing functions on a data frame, grouped by an index (see ?chickwts, for details):

```
tapply(X = chickwts$weight, INDEX = chickwts$feed, FUN = mean)
tapply(X = chickwts$weight, INDEX = chickwts$feed, FUN = median)
```

3.2.2 Quantiles, Percentiles

Calculate the quantile of a vector (0% minimum, 50% median, 100% maximum):

```
# 100 random numbers from 1 to 10
s <- sample(x = 1:10, replace = TRUE, size = 100)

quantile(s) # default quantiles: 0%, 25%, 50%, 75% and 100%
quantile(s, prob = c(0, 0.5, 1)) # just 0%, 50% and 100%
quantile(s, prob = seq(from = 0, to = 1, by = 0.1)) # 0%, 10% etcetera
```

Output:

```
0%   25%   50%   75%  100%
1.00  3.00  6.00  8.25 10.00

0%   50% 100%
1     6   10

0%  10%  20%  30%  40%  50%  60%  70%  80%  90% 100%
1.0  2.0  2.8  3.0  5.0  6.0  7.0  8.0  9.0 10.0 10.0
```

From this output, information can be obtained, such as:

- 25% of all the values in the sample are below 3
- 0% of all values in the sample are below or equal to 1.0 (not a single value)
- 100% of all values in the sample are below or equal to 10.0 (all values)

Obtaining the “five-number summary”:

```
summary(sample(x = 1:10, replace = TRUE, size = 100))
```

Output:

```
Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
```



```
1.00    3.00    6.00    5.66    8.00    10.00
```

3.2.3 Spread

Observations with different characteristics could have similar or equal means and medians:

```
a <- c(1,2,3,10,11,12,19,20,21)
b <- c(7,8,9,10,11,12,13,14,15)
```

```
mean(a) # 11
mean(b) # 11
median(a) # 11
median(b) # 11
```

The degree of spread of numeric observations can be measured with variance (average squared distance) and the standard deviation (the square root of the variance):

```
var(a) # 61.5
var(b) # 7.5

sd(a) # 7.842194, same as sqrt(var(a))
sd(b) # 2.738613, same as sqrt(var(b))
```

The interquartile range measures the width of the “middle 50%” of the data :

```
as.numeric(quantile(a, 0.75) - quantile(a, 0.25)) # 16
IQR(a) # same, but shorter

as.numeric(quantile(b, 0.75) - quantile(b, 0.25)) # 4
IQR(b) # 4
```

3.2.4 Covariance and Correlation

Two variables can be more or less related to each other:

```
height <- c(170, 168, 181, 188, 195, 182, 157, 175, 177, 183)
weight <- c(82, 67, 95, 112, 100, 82, 63, 90, 67, 75)
plot(height, weight)
```

Calculate the covariance (how much two variables “change together”) and the correlation (identify the direction and strength of the covariance):

```
cov(height, weight) # 128.3556, positive linear relationship
cor(height, weight) # 0.7424948, strong correlation, close to one
```

If the points in the plot would align to a straight line, the correlation would be exactly 1.

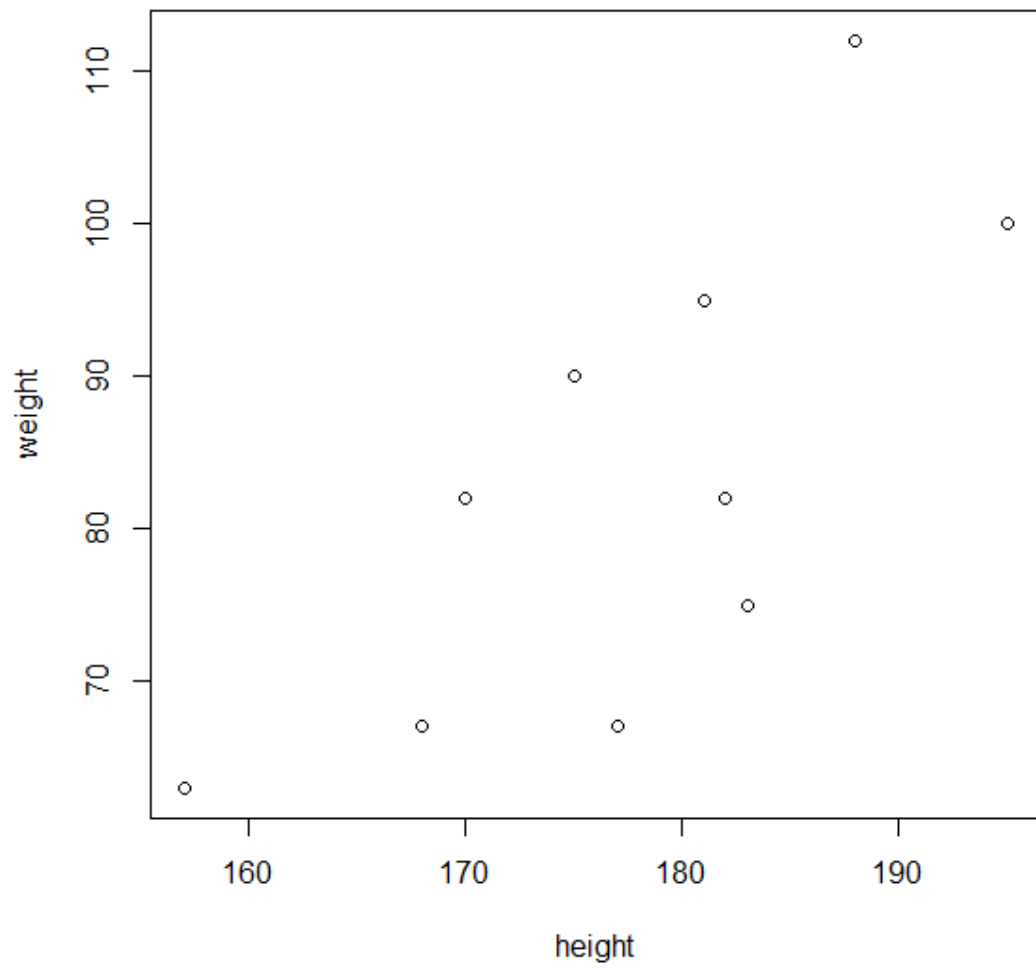


Figure 3.1: Weight by Height

3.3 TODO: Probability

3.4 Probability Distributions

R offers four types of functions for probability distributions, using a one-letter prefix:

- 'd' functions: providing specific probability values
- 'p' functions: providing cumulative probability values
- 'q' functions: providing quantiles (reverse function of the 'p' function)
- 'r' functions: providing random values

3.4.1 Mass Functions (for Discrete Variables)

3.4.1.1 Binomial distribution

The probability of getting n out of x when $p(n)$ is given.

Example: Roll a die (probability of $1/6$ of getting a certain value):

```
# getting the desired value 5 times (x) when rolling a die 10 times (size)
dbinom(x=5, size=10, prob=1/6) # 0.01302381

# getting the desired value up to five times
sum(dbinom(x=0:5, size=10, prob=1/6) # 0.9975618
pbinom(x=5, size=10, prob=1/6) # same, using the cumulative function

# how many desired values to get with a given probability of 50%
qbinom(p=0.5, size=10, prob=1/6) # 2

# amount of desired values after 10 die rolls (run n=5 random experiments)
rbinom(n=5, size=10, prob=1/6) # 3 1 0 1 3 (for example)
```

3.4.1.2 Poisson distribution

The probability of making x observations when, on average, there are λ observations.

Example: Probability of getting a certain amount of defects in a production series:

```
# the probability of getting 10 defects (for an average of 7 defects)
dpois(x=10, lambda=7) # 0.07098327

# the probability of getting up to 10 defects (for an average of 7 defects)
ppois(q=10, lambda=7)
```

3 Statistics

```
# how much defects to expect in the best 10% ( $p=0.1$ ) of the production line
qpois( $p=0.1$ ,  $\lambda=7$ ) # 4
```

```
# how many defects every product will have (run  $n=10$  random experiments)
rpois( $n=10$ ,  $\lambda=7$ ) # 9 6 9 7 4 4 5 8 9 9 (for example)
```

3.4.1.3 Geometric distribution

The numbers of failures to expect before the first success.

Example: Toss a coin (probability = $1/2$), expecting head, calculating the probability of first getting x times tails:

```
# getting three tails before getting one head
dgeom( $x=3$ ,  $\text{prob}=1/2$ ) # 0.0625
```

```
# getting up to three tails before getting one head
pgeom( $q=3$ ,  $\text{prob}=1/2$ ) # 0.9375
```

```
# in ninety percent of the cases, 3 or less tails occur before the first head
qgeom( $p=0.9$ ,  $\text{prob}=1/2$ ) # 3
```

```
# how many tails to expect before the first head (run  $n=10$  experiments)
rgeom( $n=10$ ,  $\text{prob}=1/2$ ) # 1 0 0 2 1 0 0 0 0 4 (for example)
```

3.4.1.4 Negative binomial distribution

Generalized version of the geometric distribution with an additional size parameter.

Example: Toss a coin (probability = $1/2$), calculate the chance of getting a specific number of tails (x) before a specific number of heads:

```
# one failure ( $x=1$ ) before the first ( $\text{size}=1$ ) success
# first throw tails and then head
dnbinom( $x=1$ ,  $\text{size}=1$ ,  $\text{prob}=1/2$ ) # 0.25
# four possible outcomes: HT, TH, HH and TT.
# TH is one out of those four, hence  $1/4=0.25$ 
```

```
# up to one failure ( $q=1$ ) before the first ( $\text{size}=1$ ) success
# first throw head or tails and then head
pnbinom( $q=1$ ,  $\text{size}=1$ ,  $\text{prob}=1/2$ )
# four possible outcomes: HT, TH, HH and TT.
# HT, HH (no failure) and TH (one failure) are 3 out of 4, hence  $3/4=0.75$ 
```

```
# in ninety percent of the cases, 3 or less tails occur before the first head
qnbinom(p=0.9, size=1, prob=1/2)
```

```
# how many tails to expect before the first head (run n=10 experiments)
rnbinom(n=10, size=1, prob=1/2) # 0 0 0 1 1 1 0 0 0 0 (for example)
```

3.4.1.5 Hypergeometric distribution

Sampling without replacement, when events have an impact on the probabilities of upcoming events.

Example: A urn is filled $m = 10$ white and $n = 90$ black balls ($m + n = 100$ balls):

```
# x: number of white balls drawn (without replacement)
# m: total number of white balls in the urn
# n: total number of black balls in the urn
# k: number of balls to be drawn from the urn
```

```
# getting one white ball when drawing ten times
dhyper(x=1, m=10, n=90, k=10) # 0.4079953
```

```
# getting zero or one white ball when drawing ten times
phyper(q=1, m=10, n=90, k=10) # 0.7384715
```

```
# in 95% of the cases, 3 or less white balls will be drawn
qhyper(p=0.95, m=10, n=90, k=10) # 3
```

```
# how many white balls expected to be drawn (run nn=10 experiments)
rhyper(nn=10, m=10, n=90, k=10) # 2 1 1 1 2 2 1 0 1 0
```

3.4.1.6 Multinomial distribution

Generalized version of the binomial distribution. A success can happen in more than one category at each trial.

Example: Throw a rigged coin with a heavier heads side, see how probable a certain outcome is:

```
# rigged coin, head twice as probable as tails
probs <- c(2/3, 1/3)
```

```
# expecting six heads and three tails
outcome <- c(6, 3) # 3 heads, 2 tails
```

```
# calculate the odds of that exact outcome
dmultinom(x=outcome, size=sum(outcome), prob=probs) # 0.2731291
```

3 Statistics

```
# what head/tails ratio to expect when running n=5 experiments with 5 tosses
rmultinom(n=5, size=sum(outcome), prob=probs) # returns a matrix
```

Output of the matrix:

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	7	8	6	8	8
[2,]	2	1	3	1	1

3.4.2 Density Functions (for Continuous Variables)

3.4.2.1 Uniform distribution

The uniform distribution has a constant value between a defined interval and the value zero outside that interval. The width (the interval) and the height (the value within the interval) multiply to 1.

Example: A machine cuts off pieces between 2.9 and 3.1 centimeters.

```
# the piece is cut off at exactly 3 cm length
dunif(x=3, min=2.9, max=3.1) # 5
```

The result 5 is not a probability, of course, but the function value. In order to calculate the probability, the area of an interval must be calculated:

```
# the interval of possible outcomes
a <- 2.9
b <- 3.1
```

```
# the interval of desired outcomes
l <- 2.99
r <- 3.01
```

```
# calculate the height: any value within the interval will do
h <- dunif(x=3, min=a, max=b)
```

```
# plot the interval function
plot(x = c(a, b), y = c(h, h), type="o",
      xlim = c(a, b), ylim=c(0, h),
      xlab="x", ylab="f(x)")
```

```
# draw the borders
abline(h=0, lty=2)
abline(v=a, lty=2)
abline(v=b, lty=2)
```

```
# draw a polygon denoting the desired interval
polygon(x=c(r, r, l, l), y=c(h, 0, 0, h),
        col="gray", border=NA)

# calculate the area of the polygon and display it
area <- round(h * (r - l), digits=3)
text(x = 3, y = h/2, area)
```

So the probability of the piece cut off between 2.99 and 3.01 is 10%.

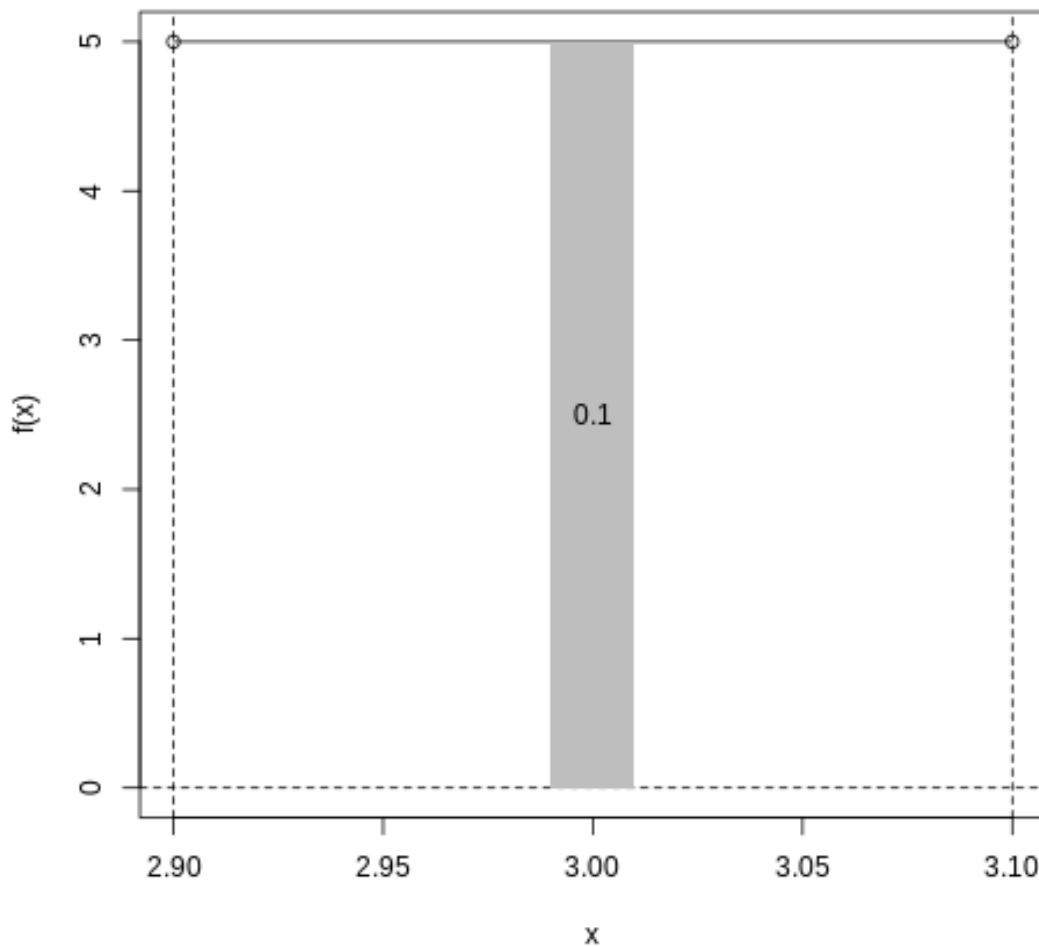


Figure 3.2: Plot of the uniform distribution (probability = 10%)

The same calculation can be achieved easier with the `punif` function, which operates on intervals

3 Statistics

(and calculates the resulting area):

```
# the area from the lower interval (a) to the upper desired interval
area_a_to_r <- punif(q=3.01, min=2.9, max=3.1)

# the area from the lower interval (a) to the lower desired interval
area_a_to_l <- punif(q=2.99, min=2.9, max=3.1)

# the desired interval is the difference between those two areas
area = area_a_to_r - area_a_to_l # 0.1
```

Calculating the range that will result with a certain probability:

```
a <- 2.9
b <- 3.1

# in 95% of the cases, the piece will be cut off at 3.09 cm maximum
qunif(p=0.95, min=a, max=b) # 3.09

# in 95% of the cases, the piece will be cut off at 2.91 cm minimum
qunif(p=1-0.95, min=a, max=b) # 2.91
```

Calculate n random values in the given range:

```
l <- runif(n=10, min=2.9, max=3.1)
round(l, digits=3)
# 2.930 2.945 2.979 2.939 3.086 2.990 3.025 2.993 2.920 3.021 (for example)
```

3.4.2.2 Normal distribution

Describes a bell-shaped distribution curve, which is also known as “Gaussian”, and is often to describe distributions in nature. The curve is symmetric, unimodal, and the area below it adds up to 1, and the tails are asymptotic to the x axis.

Example: The height of a person (male) follows a normal distribution with a mean of 175.5 cm and a standard deviation of 7.37 cm:

```
height_mean <- 175.5
height_sd <- 7.37

delta <- 3 * height_sd
x <- seq(from=height_mean-delta, to=height_mean+delta, length=50)
y <- dnorm(x, mean=height_mean, sd=height_sd)
plot(x, y, main="Normal Distribution")
```

The points are evenly spread on the x-axis, but not so on the y-axis.

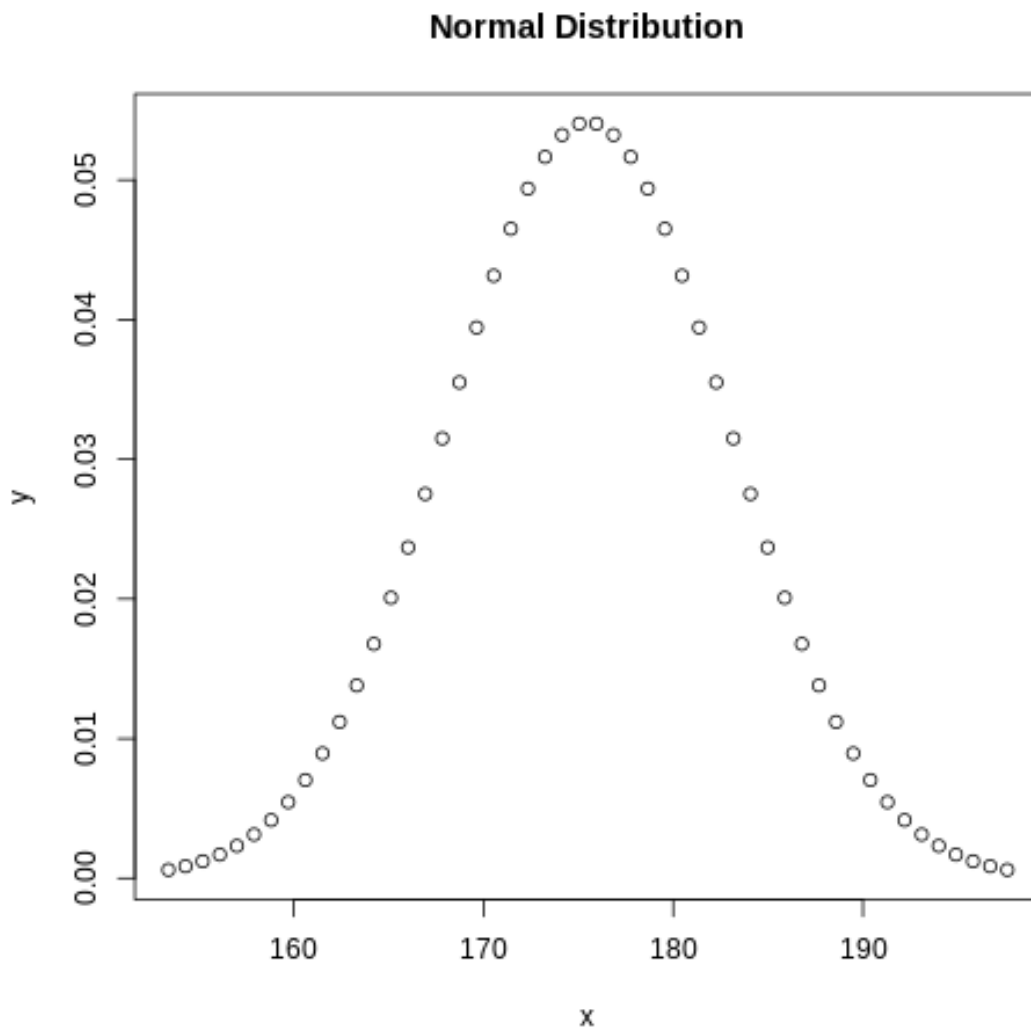


Figure 3.3: Normal Distribution: describing a bell-shaped curve

3 Statistics

To obtain a probability value, the area below the curve and inside an interval has to be calculated:

```
height_mean <- 175.5
height_sd <- 7.37

# calculate the probability of a value within one standard deviation
low <- height_mean - height_sd
high <- height_mean + height_sd
area_left <- pnorm(q=low, mean=height_mean, sd=height_sd)
area_right <- pnorm(q=high, mean=height_mean, sd=height_sd)
area <- area_right - area_left # 0.6826895

# calculate the probability of a value within two standard deviations
low <- height_mean - 2 * height_sd
high <- height_mean + 2 * height_sd
area_left <- pnorm(q=low, mean=height_mean, sd=height_sd)
area_right <- pnorm(q=high, mean=height_mean, sd=height_sd)
area <- area_right - area_left # 0.9544997

# calculate the probability of a value within three standard deviations
low <- height_mean - 3 * height_sd
high <- height_mean + 3 * height_sd
area_left <- pnorm(q=low, mean=height_mean, sd=height_sd)
area_right <- pnorm(q=high, mean=height_mean, sd=height_sd)
area <- area_right - area_left # 0.9973002
```

The calculated probabilities summarized:

- around 66% of all values are within *one* standard deviation
- around 95% of all values are within *two* standard deviations
- around 99% of all values are within *three* standard deviations

Obtaining quantiles, the cut-off points of a given probability can be calculated:

```
height_mean <- 175.5
height_sd <- 7.37

# calculating a height that marks the cut-off point of the lowest 10%
qnorm(p=0.1, mean=height_mean, sd=height_sd) # 166.055

# calculating a height that marks the cut-off point of the lowest 90%
qnorm(p=0.9, mean=height_mean, sd=height_sd) # 184.945
```

Create some random, normally distributed values:

```
height_mean <- 175.5
height_sd <- 7.37
```

```
heights <- rnorm(n=100, mean=height_mean, sd=height_sd)
plot(sort(heights), xlab="index", ylab="height")
```

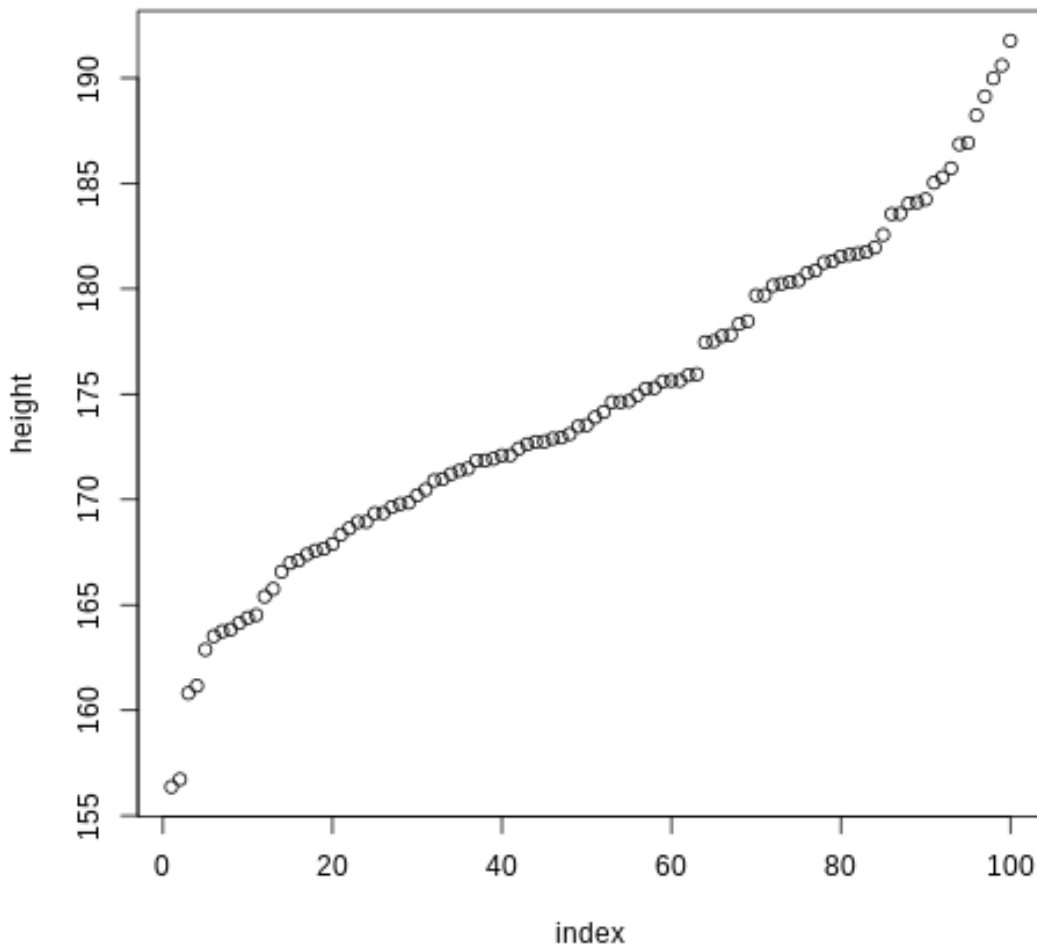


Figure 3.4: Random, normally distributed values

The values are roughly aligned on a straight line, with outliers at the left bottom and at the right top.

3.4.2.3 Other Density Distributions

Other density distributions and functions ([TODO: explain student's t and exponential in detail]):

3 Statistics

- Student's t distribution: bell-shaped distribution curve for samples (rather than populations)
 - Functions: `dt`, `pt`, `qt` and `rt`
- Exponential distribution: for random variables from 0 to ∞
 - Functions: `dexp`, `pexp`, `qexp` and `rexp`
- Chi-squared distribution: models sums of squared normal variates
 - Functions: `dchisq`, `pchisq`, `qchisq` and `rchisq`
- F distribution: models ratios of two chi-squared random variables
 - Functions: `df`, `pf`, `qf` and `rf`
- Gamma distribution: generalization of exponential and chi-squared distributions
 - Functions: `dgamma`, `pgamma`, `qgamma` and `rgamma`
- Beta distribution: for Bayesian modeling
 - Functions: `dbeta`, `pbeta`, `qbeta` and `rbeta`