T = TRUE

F = FALSE

X <- ….

Strings “”

Sum(optellen)

Rep(repeat)

Sqrt(kwadraat)

Help(functienaam)

Example(functienaam)

List.files()

To run a script, pass a string with its name tot source function: “bottle1.R”

Lwd = lijndikte

Col = collor

X/Ylim = min tot max op een as van een boxplot

?datasets

Summary(database) 🡪 samenvatting van een database

Library(help=”datasets”)

Head(ToothGrowth) namen boven de kolommen en de eerst 6.

Tail(ToothGrowth) laatste 6 in de database

Dim(ToothGrowth) hoe groot is de database

?(ToothGrowth) waar gaat de database over

? Extract

Boxplot( databse ~ database2)

Attach(cars)

Hist(dist)

boxplot(count ~ spray, data = InsectSprays, main =("Effectiveness of insect sprays"), ylab = ("aantal getelde insecten"), xlab = ("type spray"), col = ("lightblue"), sub = ("bron"))

?par 🡪 opmaak van grafieken of boxplot

boxplot(weight~Diet, data = ChickWeight, subset=Time==21, col="Yellow", main="Gewicht van kuikens in gram op dag 21 bij verschillende diëten", xlab="dieet", ylab="gewicht in gram", sub="bron")

read.table(file.choose(), header=TRUE)

bloemblaadjes<- read.csv(file.choose(), header=TRUE)

summary(bloemblaadjes)

als waardes niet gescheiden zijn door komma’s gebruik je de read.table funtie.

Met het seq argument geef je aan wat het ‘separator character’ is. voor een ‘tab character’ kun je ‘\t’gebruiken.

Flusub1<- erad.table(file.choose(), sep=’\t’, header=TRUE)

Summary(flusub1)

griepvirus<-merge(x=flusub1,y=flusub2)

Les 2

A vector's values can be numbers, strings, logical values, or any other type, as long as they're all the same type

The c function (c is short for Combine) creates a new vector by combining a list of values.

C(‘a’, 6)

If you need a vector with a sequence of numbers you can create it with start:end notation.

5:9 of seq(5:9) of seq(1. 9, 0.5)

We're going to create a vector with some strings in it for you, and store it in the sentence variable.

You can retrieve an individual value within a vector by providing its numeric index in square brackets.

> sentence <- c('walk', 'the', 'plank')

> sentence[3]

For this challenge, we'll make a 3-item vector for you, and store it in the ranks variable.

You can assign names to a vector's elements by passing a second vector filled with names to the names assignment function, like this:

[Redo](http://tryr.codeschool.com/levels/2/challenges/21) *Complete*

> ranks <- 1:3

> names(ranks) <- c("first", "second", "third")

Assigning names for a vector can act as useful labels for the data. Below, you can see what our vector looks like now.

You can also use the names to access the vector's values. Try getting the value for the "first" rank:

[Redo](http://tryr.codeschool.com/levels/2/challenges/21) *Complete*

> ranks

first second third

1 2 3

> ranks["first"]

first

1

The barplot function draws a bar chart with a vector's values. We'll make a new vector for you, and store it in the vesselsSunk variable.

Barplot(c(4, 5, 1), main = (“title”), y/xlab = (“asnaam”))

Most arithmetic operations work just as well on vectors as they do on single values. We'll make another sample vector for you to work with, and store it in the a variable.

If you add a scalar (a single value) to a vector, the scalar will be added to each value in the vector, returning a new vector with the results. Try adding 1 to each element in our vector:

> a <- c(1, 2, 3)

> a + 1

[1] 2 3 4

You can also take two vectors and compare each item. See which values in the a vector are equal to those in a second vector:

[Redo](http://tryr.codeschool.com/levels/2/challenges/30) *Complete*

> a == c(1, 99, 3)

[1] TRUE FALSE TRUE

Functions that normally work with scalars can operate on each element of a vector, too. Try getting the sine of each value in our vector:

[Redo](http://tryr.codeschool.com/levels/2/challenges/32) *Complete*

> sin(a)

[1] 0.8414710 0.9092974 0.1411200

The plot function takes two vectors, one for X values and one for Y values, and draws a graph of them.

Let's draw a graph showing the relationship of numbers and their sines.

First, we'll need some sample data. We'll create a vector for you with some fractional values between 0 and 20, and store it in the x variable.

Now, try creating a second vector with the sines of those values:

[Redo](http://tryr.codeschool.com/levels/2/challenges/34) *Complete*

> x <- seq(1, 20, 0.1)

> y <- sin(x)

Then simply call plot with your two vectors:

[Redo](http://tryr.codeschool.com/levels/2/challenges/35) *Complete*

> plot(x,y)

sum can take an optional named argument, na.rm. It's set to FALSE by default, but if you set it to TRUE, all NA arguments will be removed from the vector before the calculation is performed.

Try calling sum again, with na.rm set to TRUE:

[Redo](http://tryr.codeschool.com/levels/2/challenges/39) *Complete*

> sum(a, na.rm = TRUE)

[1] 20

## Matrices 3.1

+75 points

Let's make a matrix 3 rows high by 4 columns wide, with all its fields set to 0.

> matrix(0, 3, 4)

[,1] [,2] [,3] [,4]  
[1,]    0    0    0    0  
[2,]    0    0    0    0  
[3,]    0    0    0    0

1. +75 points

You can also use a vector to initialize a matrix's value. To fill a 3x4 matrix, you'll need a 12-item vector. We'll make that for you now:

[a <- 1:12](http://tryr.codeschool.com/levels/3/challenges/25)

[Redo](http://tryr.codeschool.com/levels/3/challenges/25)*Complete*

1. +75 points

If we print the value of a, we'll see the vector's values, all in a single row:

[print(a)](http://tryr.codeschool.com/levels/3/challenges/25)

[Redo](http://tryr.codeschool.com/levels/3/challenges/25)*Complete*

> print(a)

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

1. +75 points

Now call matrix with the vector, the number of rows, and the number of columns:

[Redo](http://tryr.codeschool.com/levels/3/challenges/25)*Complete*

> matrix(a, 3, 4)

[,1] [,2] [,3] [,4]  
[1,]    1    4    7   10  
[2,]    2    5    8   11  
[3,]    3    6    9   12

1. +75 points

The vector's values are copied into the new matrix, one by one. You can also re-shape the vector itself into a matrix. Create an 8-item vector:

[plank <- 1:8](http://tryr.codeschool.com/levels/3/challenges/25)

1. +75 points

The dim assignment function sets dimensions for a matrix. It accepts a vector with the number of rows and the number of columns to assign.

Assign new dimensions to plank by passing a vector specifying 2 rows and 4 columns (c(2, 4)):

[dim(plank) <- c(2, 4)](http://tryr.codeschool.com/levels/3/challenges/25)

1. +75 points

If you print plank now, you'll see that the values have shifted to form 2 rows by 4 columns:

[print(plank)](http://tryr.codeschool.com/levels/3/challenges/25)

> print(plank)

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

1. +75 points

The vector is no longer one-dimensional. It has been converted, in-place, to a matrix.Now, use the matrix function to make a 5x5 matrix, with its fields initialized to any values you like.

> matrix(1, 5, 5)

[,1] [,2] [,3] [,4] [,5]  
[1,]    1    1    1    1    1  
[2,]    1    1    1    1    1  
[3,]    1    1    1    1    1  
[4,]    1    1    1    1    1  
[5,]    1    1    1    1    1

## Matrix Access 3.2

+75 points

Getting values from matrices isn't that different from vectors; you just have to provide two indices instead of one.

Let's take another look at our plank matrix:

> print(plan­k)

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

>

1. +75 points

Try getting the value from the second row in the third column of plank:

> plank[2, 3]

[1] 6

1. +75 points

Now, try getting the value from first row of the fourth column:

> plank[1,4]

[1] 7

1. +75 points

As with vectors, to set a single value, just assign to it. Set the previous value to 0:

> plank[1, 4] <- 0

1. +75 points

You can get an entire row of the matrix by omitting the column index (but keep the comma). Try retrieving the second row:

> plank[2,]

[1] 2 4 6 8

1. +75 points

To get an entire column, omit the row index. Retrieve the fourth column:

> plank[, 4]

[1] 7 8

15. +75 points

You can read multiple rows or columns by providing a vector or sequence with their indices. Try retrieving columns 2 through 4:

> plank[, 2:4]

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

## Matrix Plotting 3.3

+75 points

Text output is only useful when matrices are small. When working with more complex data, you'll need something better. Fortunately, R includes powerful visualizations for matrix data.

We'll start simple, with an elevation map of a sandy beach.

It's pretty flat - everything is 1 meter above sea level. We'll create a 10 by 10 matrix with all its values initialized to 1 for you:

> elevation <- matri­x(1, 10, 10)

1. +75 points

Oh, wait, we forgot the spot where we dug down to sea level to retrieve a treasure chest. At the fourth row, sixth column, set the elevation to 0

> elevation[­4, 6] <- 0

1. +75 points

You can now do a contour map of the values simply by passing the matrix to the contour function:

> contour(el­evation)

R SVG Plot! 0.0 0.2 0.4 0.6 0.8 1.0 0.0 0.2 0.4 0.6 0.8 1.0

1. +75 points

> persp(elev­ation)

R SVG Plot! elevation Y Z

1. +75 points

The perspective plot looks a little odd, though. This is because persp automatically expands the view so that your highest value (the beach surface) is at the very top.

We can fix that by specifying our own value for the expand parameter.

> persp(elev­ation, expan­d=0.2)

R SVG Plot! elevation Y Z

1. +75 points

Okay, those examples are a little simplistic. Thankfully, R includes some sample data sets to play around with. One of these is volcano, a 3D map of a dormant New Zealand volcano.

It's simply an 87x61 matrix with elevation values, but it shows the power of R's matrix visualizations.

Try creating a contour map of the volcano matrix:

> contour(vo­lcano)

R SVG Plot! 0.0 0.2 0.4 0.6 0.8 1.0 0.0 0.2 0.4 0.6 0.8 1.0

1. +75 points

Try a perspective plot (limit the vertical expansion to one-fifth again):

> persp(volc­ano, expan­d=0.2)

> R SVG Plot! volcano Y Z

1. +75 points

The image function will create a heat map:

> image(volcano)

Les 4

A quick way to assess our battle-readiness would be to get the average of the crew's appendage counts. Statisticians call this the "mean". Call the mean function with the limbs vector.

> mean(limbs)

[1] 3.5

Here's a barplot of that vector:

[Redo](http://tryr.codeschool.com/levels/4/challenges/5) *Complete*

> barplot(limbs)

Draw a horizontal line across the plot at the mean:

[Redo](http://tryr.codeschool.com/levels/4/challenges/5) *Complete*

> abline(h=mean(limbs))

Call the median function on the vector:

[Redo](http://tryr.codeschool.com/levels/4/challenges/7) *Complete*

> median(limbs)

[1] 4

That's more like it. Let's show the median on the plot. Draw a horizontal line across the plot at the median.

[Redo](http://tryr.codeschool.com/levels/4/challenges/8) *Complete*

> abline(h=median(limbs))

Let's see a plot showing the mean value:

[Redo](http://tryr.codeschool.com/levels/4/challenges/9) *Complete*

> abline(h=meanValue)

If that sounds like a lot of work, don't worry. You're using R, and all you have to do is pass a vector to the sd function. Try calling sd on the pounds vector now, and assign the result to the deviation variable:

[Redo](http://tryr.codeschool.com/levels/4/challenges/10) *Complete*

> deviation <- sd(pounds)

We'll add a line on the plot to show one standard deviation above the mean (the top of the normal range)...

[Redo](http://tryr.codeschool.com/levels/4/challenges/11) *Complete*

> abline(h = meanValue + deviation)

Now try adding a line on the plot to show one standard devation below the mean (the bottom of the normal range):

[Redo](http://tryr.codeschool.com/levels/4/challenges/12) *Complete*

> abline(h = meanValue - deviation)

Les 5

## Creating Factors 5.1

+75 points

It's time to take inventory of the ship's hold. We'll make a vector for you with the type of booty in each chest.

To categorize the values, simply pass the vector to the factor function:

> chests <- c('gold', 'silver', 'gems', 'gold', 'gems')

> types <- factor(chests)

1. +75 points

There are a couple differences between the original vector and the new factor that are worth noting. Print the chests vector:

> print(chests)

[1] "gold" "silver" "gems" "gold" "gems"

1. +75 points

You see the raw list of strings, repeated values and all. Now print the types factor:

> print(types)

[1] gold silver gems gold gems

Levels: gems gold silver

Printed at the bottom, you'll see the factor's "levels" - groups of unique values. Notice also that there are no quotes around the values. That's because they're not strings; they're actually integer references to one of the factor's levels.

1. +75 points

Let's take a look at the underlying integers. Pass the factor to the as.integer function:

> as.integer(types)

[1] 2 3 1 2 1

1. +75 points

You can get only the factor levels with the levels function:

> levels(types)

[1] "gems" "gold" "silver"

## Plots With Factors 5.2

+75 points

You can use a factor to separate plots into categories. Let's graph our five chests by weight and value, and show their type as well. We'll create two vectors for you; weights will contain the weight of each chest, and prices will track how much the chests are worth.

Now, try calling plot to graph the chests by weight and value.

> weights <- c(300, 200, 100, 250, 150)

> prices <- c(9000, 5000, 12000, 7500, 18000)

> plot(weights, prices)

1. +75 points

We can't tell which chest is which, though. Fortunately, we can use different plot characters for each type by converting the factor to integers, and passing it to the pch argument of plot.

> plot(weights, prices, pch=as.integer(types))

"Circle", "Triangle", and "Plus Sign" still aren't great descriptions for treasure, though. Let's add a legend to show what the symbols mean.

1. +75 points

The legend function takes a location to draw in, a vector with label names, and a vector with numeric plot character IDs.

> legend("topright", c("gems", "gold", "silver"), pch=1:3)

Next time the boat's taking on water, it would be wise to dump the silver and keep the gems!

1. +75 points

If you hard-code the labels and plot characters, you'll have to update them every time you change the plot factor. Instead, it's better to derive them by using the levels function on your factor:

> legend('topright', levels(types), pch=1:length(levels(types)))

Les 6

## Data Frames 6.1

+75 points

Our vectors with treasure chest data are perfect candidates for conversion to a data frame. And it's easy to do. Call the data.frame function, and pass weights, prices, and types as the arguments. Assign the result to the treasure variable:

> treasure <- data.frame(weights, prices, types)

1. +75 points

Now, try printing treasure to see its contents:

> print(treasure)

weights prices types

1 300 9000 gold

2 200 5000 silver

3 100 12000 gems

4 250 7500 gold

5 150 18000 gems

There's your new data frame, neatly organized into rows, with column names (derived from the variable names) across the top.

## Data Frame Access 6.2

+75 points

Just like matrices, it's easy to access individual portions of a data frame.

You can get individual columns by providing their index number in double-brackets. Try getting the second column (prices) of treasure:

> treasure[[2]]

[1] 9000 5000 12000 7500 18000

1. +75 points

You could instead provide a column name as a string in double-brackets. (This is often more readable.) Retrieve the "weights" column:

> treasure[["weights"]]

[1] 300 200 100 250 150

1. 75 points

Typing all those brackets can get tedious, so there's also a shorthand notation: the data frame name, a dollar sign, and the column name (without quotes). Try using it to get the "prices" column:

> treasure$prices

[1] 9000 5000 12000 7500 18000

1. +75 points

Now try getting the "types" column:

> treasure[["types"]]

[1] gold silver gems gold gems

Levels: gems gold silver

## Loading Data Frames 6.3

+75 points

Typing in all your data by hand only works up to a point, obviously, which is why R was given the capability to easily load data in from external files.

We've created a couple data files for you to experiment with:

> list.files()

[1] "targets.csv" "infantry.txt"

Our "targets.csv" file is in the CSV (Comma Separated Values) format exported by many popular spreadsheet programs. Here's what its content looks like:

"Port","Population","Worth"

"Cartagena",35000,10000

"Porto Bello",49000,15000

"Havana",140000,50000

"Panama City",105000,35000

You can load a CSV file's content into a data frame by passing the file name to the read.csv function. Try it with the "targets.csv" file:

> read.csv("targets.csv")

Port Population Worth

1 Cartagena 35000 10000

2 Porto Bello 49000 15000

3 Havana 140000 50000

4 Panama City 105000 35000

1. +75 points

The "infantry.txt" file has a similar format, but its fields are separated by tab characters rather than commas. Its content looks like this:

Port Infantry

Porto Bello 700

Cartagena 500

Panama City 1500

Havana 2000

For files that use separator strings other than commas, you can use the read.table function. The sep argument defines the separator character, and you can specify a tab character with "\t".

Call read.table on "infantry.txt", using tab separators:

> read.table("infantry.txt", sep="\t")

V1 V2

1 Port Infantry

2 Porto Bello 700

3 Cartagena 500

4 Panama City 1500

5 Havana 2000

1. +75 points

Notice the "V1" and "V2" column headers? The first line is not automatically treated as column headers with read.table. This behavior is controlled by the header argument. Call read.table again, setting header to TRUE:

> read.table("infantry.txt", sep="\t", header=TRUE)

Port Infantry

1 Porto Bello 700

2 Cartagena 500

3 Panama City 1500

4 Havana 2000

## Merging Data Frames 6.4

+75 points

We want to loot the city with the most treasure and the fewest guards. Right now, though, we have to look at both files and match up the rows. It would be nice if all the data for a port were in one place...

R's merge function can accomplish precisely that. It joins two data frames together, using the contents of one or more columns. First, we're going to store those file contents in two data frames for you, targets and infantry.

The merge function takes arguments with an x frame (targets) and a y frame (infantry). By default, it joins the frames on columns with the same name (the two Port columns). See if you can merge the two frames:

[merge(x = targets, y = infantry)](http://tryr.codeschool.com/levels/6/challenges/12)

[Redo](http://tryr.codeschool.com/levels/6/challenges/12)*Complete*

> targets <- read.csv("targets.csv")

> infantry <- read.table("infantry.txt", sep="\t", header=TRUE)

> merge(x = targets, y = infantry)

Port Population Worth Infantry

1 Cartagena 35000 10000 500

2 Havana 140000 50000 2000

3 Panama City 105000 35000 1500

4 Porto Bello 49000 15000 700

Les 7

## Some Real World Data 7.1

+75 points

Modern pirates plunder software, not silver. We have a file with the software piracy rate, sorted by country. Here's a sample of its format:

Country,Piracy

Australia,23

Bangladesh,90

Brunei,67

China,77

...

We'll load that into the piracy data frame for you:

> piracy <- read.csv("piracy.csv")

We also have another file with GDP per capita for each country (wealth produced, divided by population):

Rank Country GDP

1 Liechtenstein 141100

2 Qatar 104300

3 Luxembourg 81100

4 Bermuda 69900

...

That will go into the gdp frame:

> gdp <- read.table("gdp.txt", sep=" ", header=TRUE)

We'll merge the frames on the country names:

> countries <- merge(x = gdp, y = piracy)

Let's do a plot of GDP versus piracy. Call the plot function, using the "GDP" column of countries for the horizontal axis, and the "Piracy" column for the vertical axis:

> plot(countries$GDP, countries$Piracy)

1. +75 points

It looks like there's a negative correlation between wealth and piracy - generally, the higher a nation's GDP, the lower the percentage of software installed that's pirated. But do we have enough data to support this connection? Is there really a connection at all?

R can test for correlation between two vectors with the cor.test function. Try calling it on the GDP and Piracy columns of the countries data frame:

> cor.test(countries$GDP, countries$Piracy)

Pearson's product-moment correlation

data: countries$GDP and countries$Piracy

t = -14.8371, df = 107, p-value < 2.2e-16

alternative hypothesis: true correlation is not equal to 0

95 percent confidence interval:

-0.8736179 -0.7475690

sample estimates:

cor

-0.8203183

The key result we're interested in is the "p-value". Conventionally, any correlation with a p-value less than 0.05 is considered statistically significant, and this sample data's p-value is definitely below that threshold. In other words, yes, these data do show a statistically significant negative correlation between GDP and software piracy.

1. +75 points

We have more countries represented in our GDP data than we do our piracy rate data. If we know a country's GDP, can we use that to estimate its piracy rate?

We can, if we calculate the linear model that best represents all our data points (with a certain degree of error). The lm function takes a model formula, which is represented by a response variable (piracy rate), a tilde character (~), and a predictor variable (GDP). (Note that the response variable comes first.)

Try calculating the linear model for piracy rate by GDP, and assign it to the line variable:

> line <- lm(countries$Piracy ~ countries$GDP)

1. +75 points

You can draw the line on the plot by passing it to the abline function. Try it now:

> abline(line)

Now, if we know a country's GDP, we should be able to make a reasonable prediction of how common piracy is there!

## ggplot2 7.2

+75 points

The functionality we've shown you so far is all included with R by default. (And it's pretty powerful, isn't it?) But in case the default installation doesn't include that function you need, there are still more libraries available on the servers of the Comprehensive R Archive Network, or CRAN. They can add anything from new statistical functions to better graphics capabilities. Better yet, installing any of them is just a command away.

Let's install the popular ggplot2 graphics package. Call the install.packages function with the package name in a string:

> install.packages("ggplot2")

1. +75 points

You can get help for a package by calling the help function and passing the package name in the package argument. Try displaying help for the "ggplot2" package:

> help(package = "ggplot2")

Information on package 'ggplot2'

Description:

Package: ggplot2

Type: Package

Title: An implementation of the Grammar of Graphics

Version: 0.9.1

...

1. +75 points

Here's a quick demo of the power you've just added to R. To use it, let's revisit some data from a previous chapter.

> weights <- c(300, 200, 100, 250, 150)

> prices <- c(9000, 5000, 12000, 7500, 18000)

> chests <- c('gold', 'silver', 'gems', 'gold', 'gems')

> types <- factor(chests)

The qplot function is a commonly-used part of ggplot2. We'll pass the weights and values of our cargo to it, using the chest types vector for the color argument:

> qplot(weights, prices, color = types)

Not bad! An attractive grid background and colorful legend, without any of the configuration hassle from before!

ggplot2 is just the first of many powerful packages awaiting discovery on CRAN. And of course, there's much, much more functionality in the standard R libraries. This course has only scratched the surface!