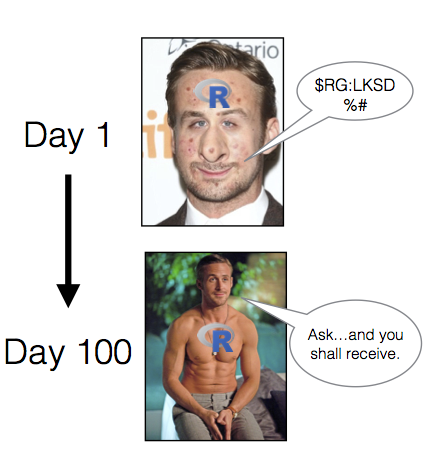
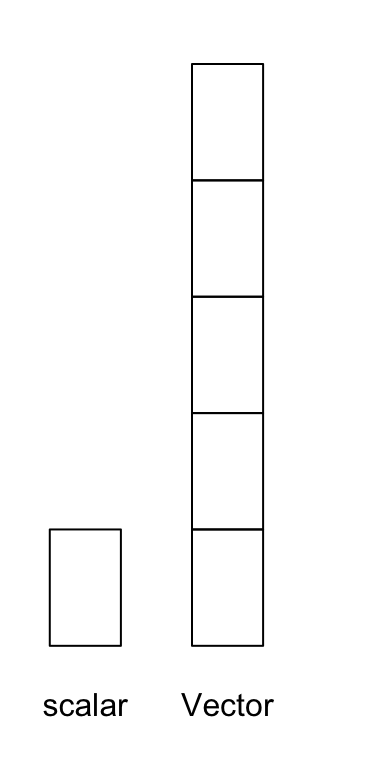
YaRrr – The Pirate’s Guide to R : NOTES



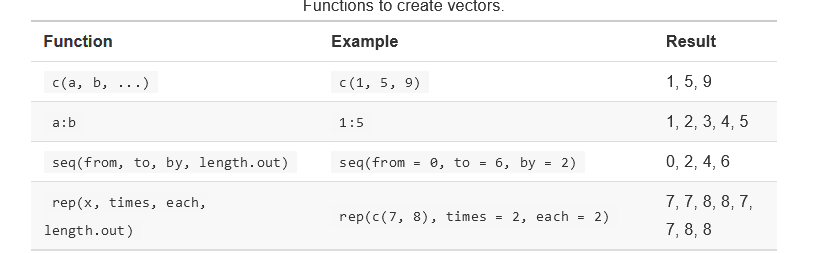
# Use the pirateplot() function without loading the yarrr package first

yarrr::pirateplot(formula = weight ~ Diet,

data = ChickWeight)



A vector object is just a combination of several scalars stored as a single object. For example, the numbers from one to ten could be a vector of length 10, and the characters in the English alphabet could be a vector of length 26. Like scalars, vectors can be either numeric or character (but not both!).



The simplest way to create a vector is with the c() function. The c here stands for concatenate, which means “bring them together”.

seq(from = 0, to = 100, length.out = 3)

rep(x = 3, times = 10)

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

rep(x = c(1, 2), each = 3)

## [1] 1 1 1 2 2 2

rep(x = 1:3, length.out = 10)

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

rep(x = 1:3, each = 2, times = 2)

chest <- c(rep("gold", 20),

rep("silver", 30),

rep("bronze", 50))

**Generating random data**

# From the integers 1:10, draw 5 numbers

sample(x = 1:10, size = 5)

sample(x = 1:5, size = 10, replace = TRUE)

Weighted samples

sample(x = c("a", "b"),

prob = c(.9, .1),

size = 10,

replace = TRUE)

### Normal (Gaussian)

# 5 samples from a Normal dist with mean = 0, sd = 1

rnorm(n = 5, mean = 0, sd = 1)

## [1] -0.0046 -0.0016 1.2226 1.2509 1.8195

# 3 samples from a Normal dist with mean = -10, sd = 15

rnorm(n = 3, mean = -10, sd = 15)

## [1] -10.67 0.61 -25.94

### Uniform

# 5 samples from Uniform dist with bounds at 0 and 1

runif(n = 5, min = 0, max = 1)

#### Use set.seed() to control random samples

# Fix sampling seed to 100, so the next sampling functions

# always produce the same values

set.seed(100)

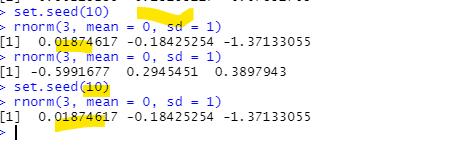
# The result will always be -0.5022, 0.1315, -0.0789

rnorm(3, mean = 0, sd = 1)

## [1] -0.502 0.132 -0.079

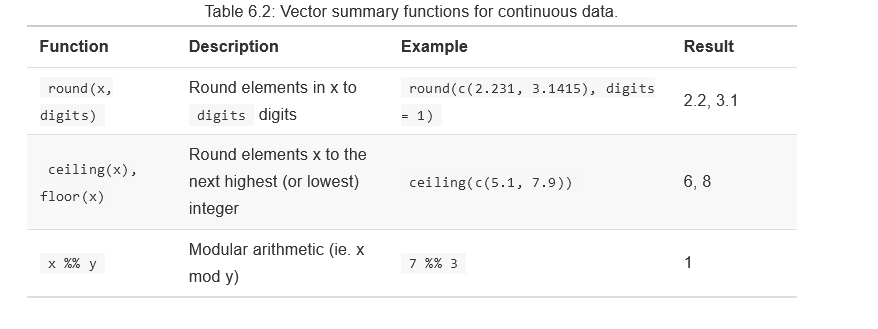
You can set seed value to any, not just 100. Its just a hook. So for 100

This will be the value.

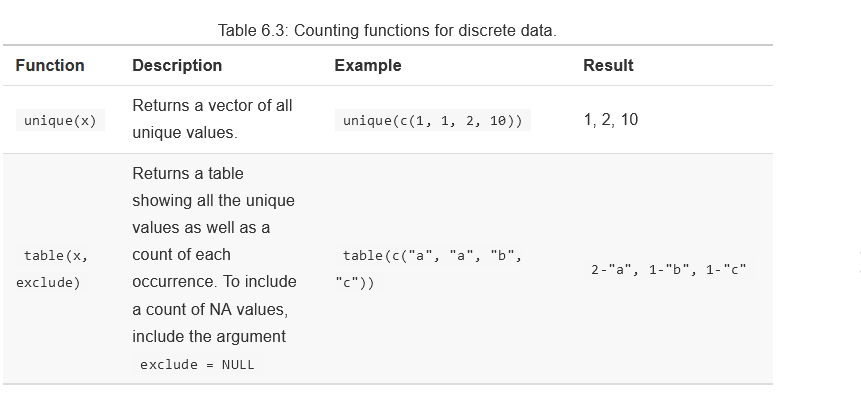


## Summary Stats

#### Rounding, ceiling, flooring



Unique:



If you want to get a table of percentages instead of counts, you can just divide the result of the table() function by the sum of the result:

table(vec) / sum(table(vec))

table(vec)/length(vec)

### Missing (NA) values

na.rm = TRUE

a <- c(1, 5, NA, 2, 10)

mean(a, na.rm = TRUE)

## Standardization (z-score)

grogg <- c(12, 8, 1, 6, 2)

climbing <- c(100, 520, 430, 200, 700)

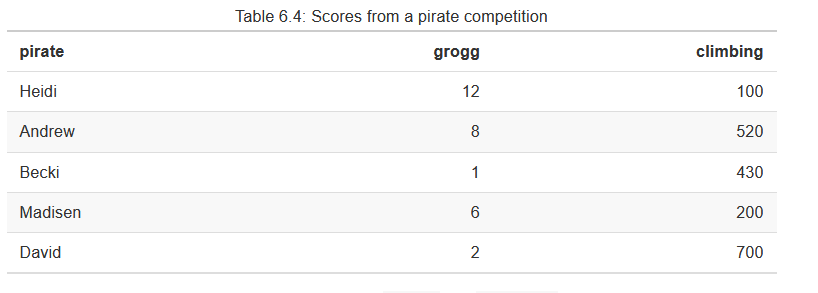
grogg.z <- (grogg - mean(grogg)) / sd(grogg)

climbing.z <- (climbing - mean(climbing)) / sd(climbing)

### Evaluating a competition

Your gluten-intolerant first mate just perished in a tragic soy sauce incident and it’s time to promote another member of your crew to the newly vacated position. Of course, only two qualities really matter for a pirate: rope-climbing, and grogg drinking. Therefore, to see which of your crew deserves the promotion, you decide to hold a climbing and drinking competition. In the climbing competition, you measure how many feet of rope a pirate can climb in an hour. In the drinking competition, you measure how many mugs of grogg they can drink in a minute. Five pirates volunteer for the competition – here are their results:

We can represent the main results with two vectors grogg and climbing:



grogg <- c(12, 8, 1, 6, 2)

climbing <- c(100, 520, 430, 200, 700)

But this are not on same scale, so its not easy to compare, so standardize it .

grogg.z <- (grogg - mean(grogg)) / sd(grogg)

climbing.z <- (climbing - mean(climbing)) / sd(climbing)

grogg.z

## [1] 1.379 0.489 -1.068 0.044 -0.845

climbing.z

## [1] -1.20 0.54 0.17 -0.78 1.28

It looks like there were two outstanding performances in particular. In the grogg drinking competition, the first pirate (Heidi) had a z-score of 1.4. We can interpret this by saying that Heidi drank 1.4 more standard deviations of mugs of grogg than the average pirate. In the climbing competition, the fifth pirate (David) had a z-score of 1.3. Here, we would conclude that David climbed 1.3 standard deviations more than the average pirate.

average.z <- (grogg.z + (climbing.z)) / 2

## Indexing Vectors with [ ]

# Boat sale. Creating the data vectors

boat.names <- c("a", "b", "c", "d", "e", "f", "g", "h", "i", "j")

boat.colors <- c("black", "green", "pink", "blue", "blue",

"green", "green", "yellow", "black", "black")

boat.ages <- c(143, 53, 356, 23, 647, 24, 532, 43, 66, 86)

boat.prices <- c(53, 87, 54, 66, 264, 32, 532, 58, 99, 132)

boat.costs <- c(52, 80, 20, 100, 189, 12, 520, 68, 80, 100)

# What was the price of the first boat?

boat.prices[1]

## [1] 53

# What were the ages of the first 5 boats?

boat.ages[1:5]

## [1] 143 53 356 23 647

# What were the names of the black boats?

boat.names[boat.colors == "black"]

## [1] "a" "i" "j"

# What were the prices of either green or yellow boats?

boat.prices[boat.colors == "green" | boat.colors == "yellow"]

## [1] 87 32 532 58

# Change the price of boat "s" to 100

boat.prices[boat.names == "s"] <- 100

# What was the median price of black boats less than 100 years old?

median(boat.prices[boat.colors == "black" & boat.ages < 100])

## [1] 116

# How many pink boats were there?

sum(boat.colors == "pink")

## [1] 1

# What percent of boats were older than 100 years old?

mean(boat.ages < 100)

## [1] 0.6

my.index <- 3:5

boat.names[my.index]

## [1] "c" "d" "e"

# Which boats had a higher price than cost?

boat.prices > boat.costs

Derivation using logical indexing:

# Which boats are older than 100 years?

boat.ages > 100

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

# Writing the logical index by hand (you'd never do this!)

# Show me all of the boat prices where the logical vector is TRUE:

boat.prices[c(TRUE, FALSE, TRUE, FALSE, TRUE, FALSE, TRUE, FALSE, FALSE, FALSE)]

## [1] 53 54 264 532

# Doing it all in one step! You get the same answer:

boat.prices[boat.ages > 100]

## [1] 53 54 264 532

#### %in%

The %in% function goes through every value in the vector x, and returns TRUE if it finds it in the vector of possible values – otherwise it returns FALSE

x <- c(1, 2, 3, -5, -5, -5, -5, -5)

We can create a logical vector to see which values are greater than 0:

x > 0

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

# Assign values greater than 10 to 10

x[x > 10] <- 10

Now, we’ll use sum() and mean() on that logical vector to see how many of the values in x are positive, and what percent are positive. We should find that there are 5 TRUE values, and that 50% of the values (5 / 10) are TRUE.

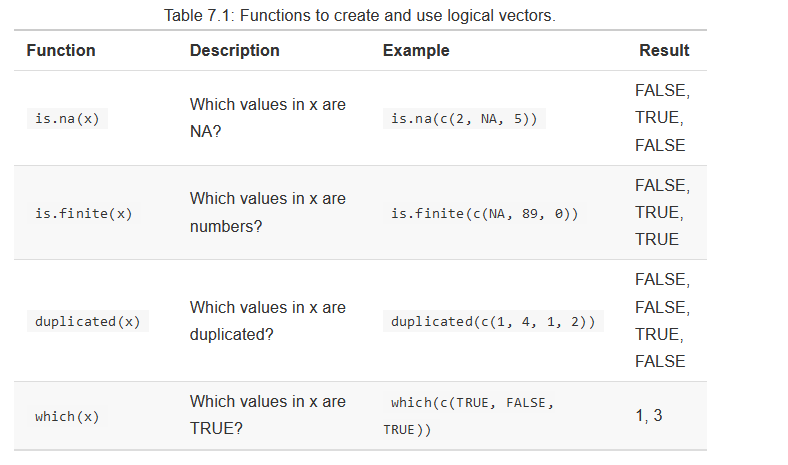
sum(x > 0)

## [1] 3

mean(x > 0)

## [1] 0.38

This is a really powerful tool. Pretty much any time you want to answer a question like “How many of X are Y” or “What percent of X are Y”, you use sum() or mean() function with a logical vector as an argument.



#### which

# A vector of sex information

sex <- c("m", "m", "f", "m", "f", "f")

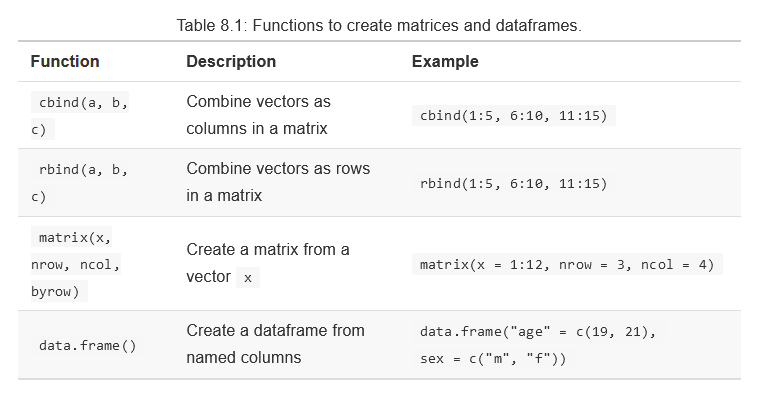
# Which values of sex are m?

which(sex == "m")

## [1] 1 2 4

## matrices and dataframes

While a matrix can contain either character or numeric columns, a dataframe can contain both numeric and character columns. matrices take up less computational space than dataframes.



#### With

Evaluate an R expression in an environment constructed from data, possibly modifying (a copy of) the original data.

The **with function** evaluates an R expression in an environment constructed based on a data frame.

with(data, x1 + x2)

The **within function** evaluates an R expression in an environment constructed based on a data frame AND modifies the original data.

within(data, x3 <- x1 + x2)

fordataframe:

with(health, weight + height / age + 2 \* height)

# Workspace Management

| **Code** | **Description** |
| --- | --- |
| ls() | Display all objects in the current workspace |
| rm(a, b, ..) | Removes the objects a, b… from your workspace |
| rm(list = ls()) | Removes *all* objects in your workspace |
| getwd() | Returns the current working directory |
| setwd(file = "dir) | Changes the working directory to a specified file location |
| list.files() | Returns the names of all files in the working directory |
| write.table(x, file = "mydata.txt", sep) | writes the object x to a text file called mydata.txt. Define how the columns will be separated with sep (e.g.; sep = "," for a comma–separated file, and sep = \t" for a tab–separated file). |
| save(a, b, .., file = "myimage.RData) | Saves objects a, b, … to myimage.RData |
| save.image(file = "myimage.RData") | Saves *all* objects in your workspace to myimage.RData |
| load(file = "myimage.RData") | Loads objects in the file myimage.RData |
| read.table(file = "mydata.txt", sep, header) | Reads a text file called mydata.txt, define how columns are separated with sep (e.g. sep = "," for comma-delimited files, and sep = "\t" for tab-delimited files), and whether there is a header column with header = TRUE |

# Shortcut

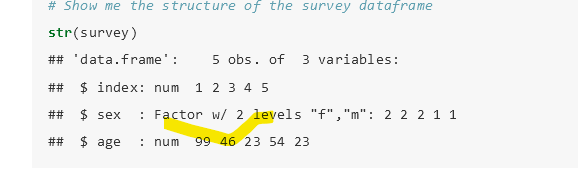
In **R**, press the “Ctrl” + “L” keys simultaneously. The screen will now be refreshed and the **console** should be cleared

f you try to create a matrix with both numbers and characters, it will turn all the numbers into characters:

#### stringsAsFactors = FALSE

By default, the data.frame() function will automatically convert any string columns to a specific type of object called a **factor** in R. A factor is a nominal variable that has a well-specified possible set of values that it can take on. For example, one can create a factor sex that can only take on the values "male" and "female".

For example: if you have a factor of sex data, but then you want to add a new value called other, R will yell at you and return an error.



Hence, in the R Core meetings in Toulouse in 2019, it was decided to move towards using stringsAsFactors = FALSE by default, ideally starting with the 4.0.0 release.

Dafeult datasets:

To see a complete list of all the datasets included in the datasets package, run the code: library(help = "datasets").

View(ChickWeight)

Now, there is one major potential problem with my method above – I had to manually enter the value of 1. But what if the column I want to change isn’t in the first column (either because I typed it wrong or because the order of the columns changed)? This could lead to serious problems later on.

To avoid these issues, it’s better to change column names using a logical vector using the format names(df)[names(df) == "old.name"] <- "new.name". Here’s how to read this: “Change the names of df, but only where the original name was "old.name", to "new.name"

#### Slicing with subset()

Let’s use the subset() function to create a new, subsetted dataset from the ToothGrowth dataframe containing data from guinea pigs who had a tooth length less than 20cm (len < 20), given the OJ supplement (supp == "OJ"), and with a dose greater than or equal to 1 (dose >= 1):

# Get rows of ToothGrowth where len < 20 AND supp == "OJ" AND dose >= 1

subset(x = ToothGrowth,

subset = len < 20 &

supp == "OJ" &

dose >= 1)

subset(x = ToothGrowth,

subset = len > 30 & supp == "VC",

select = c(len, dose))

# Debugging:

# [Launching R help: Error in file(out, “wt”) : cannot open the connection](https://stackoverflow.com/questions/38086356/launching-r-help-error-in-fileout-wt-cannot-open-the-connection)

Remedy is very simple. You don't actually need to restart R. Just re-create the temporary directory by:

tempdir()

# [1] "C:\Users\XYZ~1\AppData\Local\Temp\Rtmp86bEoJ\Rtxt32dcef24de2"

dir.create(tempdir())