Introduction to R

Lessons from DataCamp

Contents

Introduction

The following documentoutlines the written portion of the lessons from DataCamp's "Introduction to R" course. This is a beginner course that requires little knowledge in R programming.

As a note: All text is completely copied and pasted from the course. There are insances where the document refers to the "editor on the right", please note, that in this notebook document all of the instances are noted in the "r-chunks" (areas containing working r-code), which occurs below the text, rather than to the right.

If you have this document open on "R-Notebook", simply click "run" -> "Run all" (Or just press 'ctrl + alt + r'), let the "r-chunks" run (This might take a bit of time) then click "Preview". All necssary data is embedded within the code, no need to set a working directory or open an R-project.

This document was created by Neil Yetz on 09/24/2017. Please send any questions or concerns in this document to Neil at ndyetz@gmail.com

Chapter 1: Intro to Basics

In this chapter, you will take your first steps with R. You will learn how to use the console as a calculator and how to assign variables. You will also get to know the basic data types in R. Let's get started!

How It works

In the editor on the right you should type R code to solve the exercises. When you hit the 'Submit Answer' button, every line of code is interpreted and executed by R and you get a message whether or not your code was correct. The output of your R code is shown in the console in the lower right corner.

R makes use of the # sign to add comments, so that you and others can understand what the R code is about. Just like Twitter! Comments are not run as R code, so they will not influence your result. For example, Calculate `3 + 4` in the editor on the right is a comment.

You can also execute R commands straight in the console. This is a good way to experiment with R code, as your submission is not checked for correctness.

Instructions

In the editor on the right there is already some sample code. Can you see which lines are actual R code and which are comments?

Add a line of code that calculates the sum of 6 and 12, and hit the 'Submit Answer' button.

```
# Calculate 3 + 4
3 + 4
```

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

```
# Calculate 6 + 12
6+12
```

```
## [1] 18
```

Arithmetic with R

In its most basic form, R can be used as a simple calculator. Consider the following arithmetic operators:

```
Addition: +

Subtraction: -

Multiplication: *

Division: /

Exponentiation: ^

Modulo: %%
```

The last two might need some explaining:

The `^` operator raises the number to its left to the power of the number to its right: for example `3^2` is 9.

The modulo returns the remainder of the division of the number to the left by the number on its right, for example 5 modulo 3 or `5 %% 3` is 2.

With this knowledge, follow the instructions below to complete the exercise.

Instructions

Type `2^5` in the editor to calculate 2 to the power 5.

Type `28 %% 6` to calculate 28 modulo 6.

Click 'Submit Answer' and have a look at the R output in the console.

Note how the # symbol is used to add comments on the R code.

```
# An addition
5 + 5
```

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

```
# A subtraction
5 - 5
```

```
## [1] 0
```

```
# A multiplication
3 * 5
```

```
## [1] 15
```

```
 # A division
(5 + 5) / 2
```

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

```
# Exponentiation
2^5
```

```
## [1] 32
```

```
# Modulo
28 %% 6
```

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

Variable Assignment

A basic concept in (statistical) programming is called a variable.

A variable allows you to store a value (e.g. 4) or an object (e.g. a function description) in R. You can then later use this variable's name to easily access the value or the object that is stored within this variable.

You can assign a value 4 to a variable my_var with the command

my_var <- 4

Instructions

Over to you: complete the code in the editor such that it assigns the value `42` to the variable `x` in the editor.

Click 'Submit Answer'. Notice that when you ask R to print x, the value 42 appears.

```
# Assign the value 42 to x
x <- 42

# Print out the value of the variable x
x
```

```
## [1] 42
```

Variable Assignment(2)

Suppose you have a fruit basket with five apples. As a data analyst in training, you want to store the number of apples in a variable with the name my_apples.

Instructions

Type the following code in the editor: `my_apples <- 5`. This will assign the value `5` to `my_apples`.

Type: `my_apples` below the second comment. This will print out the value of `my_apples`.

Click 'Submit Answer', and look at the console: you see that the number 5 is printed. So R now links the variable my_apples to the value 5.

```
# Assign the value 5 to the variable my_apples
my_apples <- 5

# Print out the value of the variable my_apples
my_apples
```

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

Variable Assignment(3)

Every tasty fruit basket needs oranges, so you decide to add six oranges. As a data analyst, your reflex is to immediately create the variable `my_oranges` and assign the value `6` to it. Next, you want to calculate how many pieces of fruit you have in total. Since you have given meaningful names to these values, you can now code this in a clear way:

```
my_apples + my_oranges
```

Instructions

Assign to my_oranges the value 6.

Add the variables `my_apples` and `my_oranges` and have R simply print the result.

Assign the result of adding `my_apples` and `my_oranges` to a new variable `my_fruit`.

```
# Assign a value to the variables my_apples and my_oranges
my_apples <- 5
my_oranges <- 6

# Add these two variables together
my_apples + my_oranges
```

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

```
# Create the variable my_fruit
my_fruit <- my_apples + my_oranges
```

Apples and oranges

Common knowledge tells you not to add apples and oranges. But hey, that is what you just did, no :-)? The my_apples and my_oranges variables both contained a number in the previous exercise. The + operator works with numeric variables in R. If you really tried to add "apples" and "oranges", and assigned a text value to the variable my_oranges (see the editor), you would be trying to assign the addition of a numeric and a character variable to the variable `my_fruit`. This is not possible.

Instructions

Click 'Submit Answer' and read the error message. Make sure to understand why this did not work.

Adjust the code so that R knows you have 6 oranges and thus a fruit basket with 11 pieces of fruit.

```
# Assign a value to the variable my_apples
my_apples <- 5

# Fix the assignment of my_oranges
my_oranges <- 6

# Create the variable my_fruit and print it out
my_fruit <- my_apples + my_oranges
my_fruit
```

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

Basic data types in R

R works with numerous data types. Some of the most basic types to get started are:

Decimals values like `4.5` are called **numerics**.

Natural numbers like `4` are called **integers**. Integers are also numerics.

Boolean values (`TRUE` or `FALSE`) are called **logical**.

Text (or string) values are called **characters**.

Note how the quotation marks on the right indicate that `"some text"` is a character.

Instructions

Change the value of the:

`my_numeric` variable to `42`.

`my_character` variable to `"universe"`. Note that the quotation marks indicate that `"universe"` is a character.

`my_logical` variable to `FALSE`.

Note that R is case sensitive!

```
# Change my_numeric to be 42
my_numeric <- 42

# Change my_character to be "universe"
my_character <- "universe"

# Change my_logical to be FALSE
my_logical <- FALSE
```

What's that data type?

Do you remember that when you added `5 + "six"`, you got an error due to a mismatch in data types? You can avoid such embarrassing situations by checking the data type of a variable beforehand. You can do this with the `class()` function, as the code on the right shows.

Instructions

Complete the code in the editor and also print out the classes of `my_character` and `my_logical`

```
#Declare variables of different types
my_numeric <- 42
my_character <- "universe"
my_logical <- FALSE

# Check class of my_numeric
class(my_numeric)
```

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

```
# Check class of my_character
class(my_character)
```

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

```
# Check class of my_logical
class(my_logical)
```

```
## [1] "logical"
```

Chapter 2: Vectors

In this free R course, we'll take you on a trip to Vegas, where you will learn how to analyze your gambling results using vectors in R! After completing this chapter, you will be able to create vectors in R, name them, select elements from them and compare different vectors.

Create a vector

Feeling lucky? You better, because this chapter takes you on a trip to the City of Sins, also known as Statisticians Paradise!

Thanks to R and your new data-analytical skills, you will learn how to uplift your performance at the tables and fire off your career as a professional gambler. This chapter will show how you can easily keep track of your betting progress and how you can do some simple analyses on past actions. Next stop, Vegas Baby... VEGAS!!

Instructions

Do you still remember what you have learned in the first chapter? Assign the value `"Go!"` to the variable `vegas`. Remember: R is case sensitive!

```
# Define the variable vegas
vegas <- "Go!"
```

Create a vector (2)

Let us focus first!

On your way from rags to riches, you will make extensive use of vectors. Vectors are one-dimension arrays that can hold numeric data, character data, or logical data. In other words, a vector is a simple tool to store data. For example, you can store your daily gains and losses in the casinos.

In R, you create a vector with the combine function `c()`. You place the vector elements separated by a comma between the parentheses. For example:

```
numeric_vector <- c(1, 2, 3)
```

```
character_vector <- c("a", "b", "c")
```

Once you have created these vectors in R, you can use them to do calculations.

Instructions

Complete the code such that `boolean_vector` contains the three elements: `TRUE`, `FALSE` and `TRUE` (in that order).

```
numeric_vector <- c(1, 10, 49)
character_vector <- c("a", "b", "c")

# Complete the code for boolean_vector
boolean_vector <- c(TRUE, FALSE, TRUE)
```

Create a vector (3)

After one week in Las Vegas and still zero Ferraris in your garage, you decide that it is time to start using your data analytical superpowers.

Before doing a first analysis, you decide to first collect all the winnings and losses for the last week:

For `poker_vector`:

On Monday you won $140
Tuesday you lost $50
Wednesday you won $20
Thursday you lost $120
Friday you won $240

For `roulette_vector`:

On Monday you lost $24
Tuesday you lost $50
Wednesday you won $100
Thursday you lost $350
Friday you won $10

You only played poker and roulette, since there was a delegation of mediums that occupied the craps tables. To be able to use this data in R, you decide to create the variables `poker_vector` and `roulette_vector`.

Instructions

Assign the winnings/losses for roulette to the variable `roulette_vector`.

```
# Poker winnings from Monday to Friday
poker_vector <- c(140, -50, 20, -120, 240)

# Roulette winnings from Monday to Friday
roulette_vector <-  c(-24, -50, 100, -350, 10)
```

Naming a vector

As a data analyst, it is important to have a clear view on the data that you are using. Understanding what each element refers to is therefore essential.

In the previous exercise, we created a vector with your winnings over the week. Each vector element refers to a day of the week but it is hard to tell which element belongs to which day. It would be nice if you could show that in the vector itself.

You can give a name to the elements of a vector with the `names()` function. Have a look at this example:

```
some_vector <- c("John Doe", "poker player")
names(some_vector) <- c("Name", "Profession")
```

This code first creates a vector `some_vector` and then gives the two elements a name. The first element is assigned the name `Name`, while the second element is labeled Profession. Printing the contents to the console yields following output:

```
Name     Profession      "John Doe" "poker player"
```

Instructions

The code on the right names the elements in `poker_vector` with the days of the week.

Add code to do the same thing for `roulette_vector`

```
# Poker winnings from Monday to Friday
poker_vector <- c(140, -50, 20, -120, 240)

# Roulette winnings from Monday to Friday
roulette_vector <- c(-24, -50, 100, -350, 10)

# Assign days as names of poker_vector
names(poker_vector) <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")

# Assign days as names of roulette_vectors
names(roulette_vector) <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
```

Naming a vector(2)

If you want to become a good statistician, you have to become lazy. (If you are already lazy, chances are high you are one of those exceptional, natural-born statistical talents.)

In the previous exercises you probably experienced that it is boring and frustrating to type and retype information such as the days of the week. However, when you look at it from a higher perspective, there is a more efficient way to do this, namely, to assign the days of the week vector to a **variable**!

Just like you did with your poker and roulette returns, you can also create a variable that contains the days of the week. This way you can use and re-use it.

Instructions A variable `days_vector` that contains the days of the week has already been created for you.

Use `days_vector` to set the names of `poker_vector` and `roulette_vector`.

```
# Poker winnings from Monday to Friday
poker_vector <- c(140, -50, 20, -120, 240)

# Roulette winnings from Monday to Friday
roulette_vector <- c(-24, -50, 100, -350, 10)
```

```
# The variable days_vector
days_vector <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")

# Assign the names of the day to roulette_vector and poker_vector
names(poker_vector) <- days_vector
names(roulette_vector) <- days_vector
```

Calculating total winnings

Now that you have the poker and roulette winnings nicely as named vectors, you can start doing some data analytical magic.

You want to find out the following type of information:

How much has been your overall profit or loss per day of the week? Have you lost money over the week in total?

Are you winning/losing money on poker or on roulette?

To get the answers, you have to do arithmetic calculations on vectors.

It is important to know that if you sum two vectors in R, it takes the element-wise sum. For example, the following three statements are completely equivalent:

```
c(1, 2, 3) + c(4, 5, 6)
c(1 + 4, 2 + 5, 3 + 6)
c(5, 7, 9)
```

You can also do the calculations with variables that represent vectors:

```
a <- c(1, 2, 3)
b <- c(4, 5, 6)
c <- a + b
```

Instructions

Take the sum of the variables `A_vector` and `B_vector` and it assign to `total_vector`.

Inspect the result by printing out `total_vector`.

```
A_vector <- c(1, 2, 3)
B_vector <- c(4, 5, 6)

# Take the sum of A_vector and B_vector
total_vector <- A_vector + B_vector

# Print out total_vector
total_vector
```

```
## [1] 5 7 9
```

Calculating total winnings (2)

Now you understand how R does arithmetic with vectors, it is time to get those Ferraris in your garage! First, you need to understand what the overall profit or loss per day of the week was. The total daily profit is the sum of the profit/loss you realized on poker per day, and the profit/loss you realized on roulette per day.

In R, this is just the sum of `roulette_vector` and `poker_vector`.

Instructions Assign to the variable `total_daily` how much you won or lost on each day in total (poker and roulette combined).

```
# Poker and roulette winnings from Monday to Friday:
poker_vector <- c(140, -50, 20, -120, 240)
roulette_vector <- c(-24, -50, 100, -350, 10)
days_vector <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")

names(poker_vector) <- days_vector
names(roulette_vector) <- days_vector

# Assign to total_daily how much you won/lost on each day
total_daily <- roulette_vector + poker_vector
```

Calculating total winnings (3)

Based on the previous analysis, it looks like you had a mix of good and bad days. This is not what your ego expected, and you wonder if there may be a very tiny chance you have lost money over the week in total?

A function that helps you to answer this question is `sum()`. It calculates the sum of all elements of a vector. For example, to calculate the total amount of money you have lost/won with poker you do:

```
total_poker <- sum(poker_vector)
```

Instructions

Calculate the total amount of money that you have won/lost with roulette and assign to the variable `total_roulette`.

Now that you have the totals for roulette and poker, you can easily calculate `total_week` (which is the sum of all gains and losses of the week).

Print out `total_week`.

```
# Poker and roulette winnings from Monday to Friday:

poker_vector <- c(140, -50, 20, -120, 240)
roulette_vector <- c(-24, -50, 100, -350, 10)
days_vector <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")

names(poker_vector) <- days_vector
names(roulette_vector) <- days_vector

# Total winnings with poker
total_poker <- sum(poker_vector)

# Total winnings with roulette
total_roulette <-  sum(roulette_vector)

# Total winnings overall
total_week <- total_poker + total_roulette

# Print out total_week
  total_week
```

```
## [1] -84
```

Comparing total winnings

Oops, it seems like you are losing money. Time to rethink and adapt your strategy! This will require some deeper analysis...

After a short brainstorm in your hotel's jacuzzi, you realize that a possible explanation might be that your skills in roulette are not as well developed as your skills in poker. So maybe your total gains in poker are higher (or `>`) than in roulette.

Instructions

Calculate `total_poker` and `total_roulette` as in the previous exercise. Use the `sum()` function twice.

Check if your total gains in poker are higher than for roulette by using a comparison. Simply print out the result of this comparison. What do you conclude, should you focus on roulette or on poker?

```
# Poker and roulette winnings from Monday to Friday:
poker_vector <- c(140, -50, 20, -120, 240)
roulette_vector <- c(-24, -50, 100, -350, 10)
days_vector <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
names(poker_vector) <- days_vector
names(roulette_vector) <- days_vector

# Calculate total gains for poker and roulette
total_poker <- sum(poker_vector)
total_roulette <- sum(roulette_vector)

# Check if you realized higher total gains in poker than in roulette
total_poker > total_roulette
```

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

Vector selection: the good times

Your hunch seemed to be right. It appears that the poker game is more your cup of tea than roulette.

Another possible route for investigation is your performance at the beginning of the working week compared to the end of it. You did have a couple of Margarita cocktails at the end of the week...

To answer that question, you only want to focus on a selection of the `total_vector`. In other words, our goal is to select specific elements of the vector. To select elements of a vector (and later matrices, data frames, ...), you can use square brackets. Between the square brackets, you indicate what elements to select. For example, to select the first element of the vector, you type `poker_vector[1]`. To select the second element of the vector, you type `poker_vector[2]`, etc. Notice that the first element in a vector has index 1, not 0 as in many other programming languages.

Instructions Assign the poker results of Wednesday to the variable `poker_wednesday`.

```
# Poker and roulette winnings from Monday to Friday:
poker_vector <- c(140, -50, 20, -120, 240)
roulette_vector <- c(-24, -50, 100, -350, 10)
days_vector <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
names(poker_vector) <- days_vector
names(roulette_vector) <- days_vector

# Define a new variable based on a selection
poker_wednesday <- poker_vector[3]
```

Vector selection: the good times (2)

How about analyzing your midweek results?

To select multiple elements from a vector, you can add square brackets at the end of it. You can indicate between the brackets what elements should be selected. For example: suppose you want to select the first and the fifth day of the week: use the vector `c(1, 5)` between the square brackets. For example, the code below selects the first and fifth element of `poker_vector`:

```
poker_vector[c(1, 5)]
```

Instructions

Assign the poker results of Tuesday, Wednesday and Thursday to the variable `poker_midweek`.

```
# Poker and roulette winnings from Monday to Friday:
poker_vector <- c(140, -50, 20, -120, 240)
roulette_vector <- c(-24, -50, 100, -350, 10)
days_vector <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
names(poker_vector) <- days_vector
names(roulette_vector) <- days_vector

# Define a new variable based on a selection
poker_midweek <- poker_vector[c(2, 3, 4)]
```

Vector selection: the good times (3)

Selecting multiple elements of `poker_vector` with `c(2, 3, 4)` is not very convenient. Many statisticians are lazy people by nature, so they created an easier way to do this: `c(2, 3, 4)` can be abbreviated to `2:4`, which generates a vector with all natural numbers from 2 up to 4.

So, another way to find the mid-week results is `poker_vector[2:4]`. Notice how the vector `2:4` is placed between the square brackets to select element 2 up to 4.

Instructions Assign to `roulette_selection_vector` the roulette results from Tuesday up to Friday; make use of `:` if it makes things easier for you.

```
# Poker and roulette winnings from Monday to Friday:
poker_vector <- c(140, -50, 20, -120, 240)
roulette_vector <- c(-24, -50, 100, -350, 10)
days_vector <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
names(poker_vector) <- days_vector
names(roulette_vector) <- days_vector

# Define a new variable based on a selection
roulette_selection_vector <- roulette_vector[c(2:5)]
```

Vector selection: the good times (4)

Another way to tackle the previous exercise is by using the names of the vector elements (Monday, Tuesday, ...) instead of their numeric positions. For example,

```
poker_vector["Monday"]
```

will select the first element of `poker_vector` since `"Monday"` is the name of that first element.

Just like you did in the previous exercise with numerics, you can also use the element names to select multiple elements, for example:

```
poker_vector[c("Monday","Tuesday")]
```

Instructions Select the first three elements in `poker_vector` by using their names: `"Monday"`, `"Tuesday"` and `"Wednesday"`. Assign the result of the selection to `poker_start`.

Calculate the average of the values in `poker_start` with the `mean()` function. Simply print out the result so you can inspect it.

```
# Poker and roulette winnings from Monday to Friday:
poker_vector <- c(140, -50, 20, -120, 240)
roulette_vector <- c(-24, -50, 100, -350, 10)
days_vector <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
names(poker_vector) <- days_vector
names(roulette_vector) <- days_vector

# Select poker results for Monday, Tuesday and Wednesday
poker_start <- poker_vector[c("Monday", "Tuesday", "Wednesday")]

# Calculate the average of the elements in poker_start
mean(poker_start)
```

```
## [1] 36.66667
```

Selection by comparison - Step 1

By making use of comparison operators, we can approach the previous question in a more proactive way.

The (logical) comparison operators known to R are:

`<` for less than
`>` for greater than
`<=` for less than or equal to
`>=` for greater than or equal to
`==` for equal to each other
`!=` not equal to each other

As seen in the previous chapter, stating `6 > 5` returns `TRUE`. The nice thing about R is that you can use these comparison operators also on vectors. For example:

```
> c(4, 5, 6) > 5
[1] FALSE FALSE TRUE
```

This command tests for every element of the vector if the condition stated by the comparison operator is `TRUE` or `FALSE`.

Instructions

Check which elements in `poker_vector` are positive (i.e. > 0) and assign this to `selection_vector`.

Print out selection_vector so you can inspect it. The printout tells you whether you won (`TRUE`) or lost (`FALSE`) any money for each day.

```
# Poker and roulette winnings from Monday to Friday:
poker_vector <- c(140, -50, 20, -120, 240)
roulette_vector <- c(-24, -50, 100, -350, 10)
days_vector <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
names(poker_vector) <- days_vector
```

```
names(roulette_vector) <- days_vector

# Which days did you make money on poker?
selection_vector <- poker_vector > 0

# Print out selection_vector
selection_vector
```

```
##    Monday   Tuesday Wednesday  Thursday    Friday
##      TRUE     FALSE      TRUE     FALSE      TRUE
```

Selection by comparison - Step 2

Working with comparisons will make your data analytical life easier. Instead of selecting a subset of days to investigate yourself (like before), you can simply ask R to return only those days where you realized a positive return for poker.

In the previous exercises you used `selection_vector <- poker_vector > 0` to find the days on which you had a positive poker return. Now, you would like to know not only the days on which you won, but also how much you won on those days.

You can select the desired elements, by putting `selection_vector` between the square brackets that follow `poker_vector`:

`poker_vector[selection_vector]`

R knows what to do when you pass a logical vector in square brackets: it will only select the elements that correspond to `TRUE` in `selection_vector`.

Instructions Use `selection_vector` in square brackets to assign the amounts that you won on the profitable days to the variable `poker_winning_days`.

```
# Poker and roulette winnings from Monday to Friday:
poker_vector <- c(140, -50, 20, -120, 240)
roulette_vector <- c(-24, -50, 100, -350, 10)
days_vector <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
names(poker_vector) <- days_vector
names(roulette_vector) <- days_vector

# Which days did you make money on poker?
selection_vector <- poker_vector > 0

# Select from poker_vector these days
poker_winning_days <- poker_vector[selection_vector]
```

Advanced selection

Just like you did for poker, you also want to know those days where you realized a positive return for roulette.

Instructions

Create the variable `selection_vector`, this time to see if you made profit with roulette for different days.

Assign the amounts that you made on the days that you ended positively for roulette to the variable `roulette_winning_days`. This vector thus contains the positive winnings of `roulette_vector`.

```r
# Poker and roulette winnings from Monday to Friday:
poker_vector <- c(140, -50, 20, -120, 240)
roulette_vector <- c(-24, -50, 100, -350, 10)
days_vector <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
names(poker_vector) <- days_vector
names(roulette_vector) <- days_vector

# Which days did you make money on roulette?
selection_vector <- roulette_vector > 0

# Select from roulette_vector these days
roulette_winning_days <- roulette_vector[selection_vector]
```

Chapter 3: Matrices

In this chapter you will learn how to work with matrices in R. By the end of the chapter, you will be able to create matrices and to understand how you can do basic computations with them. You will analyze the box office numbers of Star Wars to illustrate the use of matrices in R. May the force be with you!

What's a matrix?

In R, a matrix is a collection of elements of the same data type (numeric, character, or logical) arranged into a fixed number of rows and columns. Since you are only working with rows and columns, a matrix is called two-dimensional.

You can construct a matrix in R with the `matrix()` function. Consider the following example:

```r
matrix(1:9, byrow = TRUE, nrow = 3)
```

In the `matrix()` function:

The first argument is the collection of elements that R will arrange into the rows and columns of the matrix. Here, we use `1:9` which is a shortcut for `c(1, 2, 3, 4, 5, 6, 7, 8, 9)`.

The argument byrow indicates that the matrix is filled by the rows. If we want the matrix to be filled by the columns, we just place `byrow = FALSE`.

The third argument `nrow` indicates that the matrix should have three rows.

Instructions

Construct a matrix with 3 rows containing the numbers 1 up to 9, filled row-wise.

```r
# Construct a matrix with 3 rows that contain the numbers 1 up to 9
matrix(1:9, byrow = TRUE, nrow =3)
```

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

Analyzing matrices, you shall

It is now time to get your hands dirty. In the following exercises you will analyze the box office numbers of the Star Wars franchise. May the force be with you!

In the editor, three vectors are defined. Each one represents the box office numbers from the first three Star Wars movies. The first element of each vector indicates the US box office revenue, the second element refers to the Non-US box office (source: Wikipedia).

In this exercise, you'll combine all these figures into a single vector. Next, you'll build a matrix from this vector.

Instructions Use `c(new_hope, empire_strikes, return_jedi)` to combine the three vectors into one vector. Call this vector `box_office`.

Construct a matrix with 3 rows, where each row represents a movie. Use the `matrix()` function to this. The first argument is the vector `box_office`, containing all box office figures.

Next, you'll have to specify `nrow = 3` and `byrow = TRUE`. Name the resulting matrix `star_wars_matrix`.

```
# Box office Star Wars (in millions!)
new_hope <- c(460.998, 314.4)
empire_strikes <- c(290.475, 247.900)
return_jedi <- c(309.306, 165.8)

# Create box_office
box_office <- c(new_hope, empire_strikes, return_jedi)

# Construct star_wars_matrix
star_wars_matrix <- matrix(box_office, byrow= TRUE, nrow = 3)
```

Naming a matrix

To help you remember what is stored in `star_wars_matrix`, you would like to add the names of the movies for the rows. Not only does this help you to read the data, but it is also useful to select certain elements from the matrix.

Similar to vectors, you can add names for the rows and the columns of a matrix

```
rownames(my_matrix) <- row_names_vector
colnames(my_matrix) <- col_names_vector
```

We went ahead and prepared two vectors for you: `region`, and `titles`. You will need these vectors to name the columns and rows of `star_wars_matrix`, respectively.

Instructions

Use `colnames()` to name the columns of `star_wars_matrix` with the `region` vector. Use `rownames()` to name the rows of `star_wars_matrix` with the `titles` vector. Print out `star_wars_matrix` to see the result of your work.

```
# Box office Star Wars (in millions!)
new_hope <- c(460.998, 314.4)
empire_strikes <- c(290.475, 247.900)
return_jedi <- c(309.306, 165.8)

# Construct matrix
star_wars_matrix <- matrix(c(new_hope, empire_strikes, return_jedi), nrow = 3, byrow = TRUE)

# Vectors region and titles, used for naming
region <- c("US", "non-US")
titles <- c("A New Hope", "The Empire Strikes Back", "Return of the Jedi")
```

```
# Name the columns with region
colnames(star_wars_matrix) <- region

# Name the rows with titles
rownames(star_wars_matrix) <- titles

# Print out star_wars_matrix
star_wars_matrix
```

```
##                             US non-US
## A New Hope              460.998  314.4
## The Empire Strikes Back 290.475  247.9
## Return of the Jedi      309.306  165.8
```

Calculating the worldwide box office

The single most important thing for a movie in order to become an instant legend in Tinseltown is its worldwide box office figures.

To calculate the total box office revenue for the three Star Wars movies, you have to take the sum of the US revenue column and the non-US revenue column.

In R, the function `rowSums()` conveniently calculates the totals for each row of a matrix. This function creates a new vector:

```
rowSums(my_matrix)
```

Instructionss

Calculate the worldwide box office figures for the three movies and put these in the vector named `worldwide_vector`.

```
# Construct star_wars_matrix
box_office <- c(460.998, 314.4, 290.475, 247.900, 309.306, 165.8)
star_wars_matrix <- matrix(box_office, nrow = 3, byrow = TRUE,
                           dimnames = list(c("A New Hope", "The Empire Strikes Back", "Return of the Je
                                           c("US", "non-US")))

# Calculate worldwide box office figures
worldwide_vector <- rowSums(star_wars_matrix)
```

Adding a column for the Worldwide box office

In the previous exercise you calculated the vector that contained the worldwide box office receipt for each of the three Star Wars movies. However, this vector is not yet part of `star_wars_matrix`.

You can add a column or multiple columns to a matrix with the `cbind()` function, which merges matrices and/or vectors together by column. For example:

```
big_matrix <- cbind(matrix1, matrix2, vector1 ...)
```

Instructions

Add `worldwide_vector` as a new column to the `star_wars_matrix` and assign the result to `all_wars_matrix`. Use the `cbind()` function.

```
# Construct star_wars_matrix
box_office <- c(460.998, 314.4, 290.475, 247.900, 309.306, 165.8)
star_wars_matrix <- matrix(box_office, nrow = 3, byrow = TRUE,
                           dimnames = list(c("A New Hope", "The Empire Strikes Back", "Return of the Je
                                           c("US", "non-US")))

# The worldwide box office figures
worldwide_vector <- rowSums(star_wars_matrix)

# Bind the new variable worldwide_vector as a column to star_wars_matrix
all_wars_matrix <- cbind(star_wars_matrix, worldwide_vector)
```

Adding a row

Just like every action has a reaction, every `cbind()` has an `rbind()`. (We admit, we are pretty bad with metaphors.)

Your R workspace, where all variables you defined 'live' (check out what a workspace is), has already been initialized and contains two matrices:

`star_wars_matrix` that we have used all along, with data on the first trilogy,

`star_wars_matrix2`, with similar data for the second trilogy.

Type the name of these matrices in the console and hit Enter if you want to have a closer look. If you want to check out the contents of the workspace, you can type `ls()` in the console.

Instructions

Use `rbind()` to paste together `star_wars_matrix` and `star_wars_matrix2`, in this order.

Assign the resulting matrix to `all_wars_matrix`.

```
#Neil adding in star_wars_matrix2 data (Only available in DataCamp)

box_office2 <- c(474.5, 552.5,310.7, 338.7, 380.3, 468.5)
star_wars_matrix2 <- star_wars_matrix <- matrix(box_office2, nrow = 3, byrow = TRUE,
                           dimnames = list(c("A New Hope", "The Empire Strikes Back", "Return of the Je
                                           c("US", "non-US")))

# star_wars_matrix and star_wars_matrix2 are available in your workspace
star_wars_matrix
```

```
##                           US non-US
## A New Hope              474.5  552.5
## The Empire Strikes Back 310.7  338.7
## Return of the Jedi      380.3  468.5
```

```
star_wars_matrix2
```

```
##                           US non-US
## A New Hope              474.5  552.5
## The Empire Strikes Back 310.7  338.7
## Return of the Jedi      380.3  468.5
```

```
# Combine both Star Wars trilogies in one matrix
all_wars_matrix <- rbind(star_wars_matrix, star_wars_matrix2)
```

The total box office revenue for the entire saga

Just like every `cbind()` has a `rbind()`, every `colSums()` has a `rowSums()`. Your R workspace already contains the `all_wars_matrix` that you constructed in the previous exercise; type `all_wars_matrix` to have another look. Let's now calculate the total box office revenue for the entire saga.

Instructions

Calculate the total revenue for the US and the non-US region and assign `total_revenue_vector`. You can use the `colSums()` function.

Print out `total_revenue_vector` to have a look at the results.

```
# all_wars_matrix is available in your workspace
all_wars_matrix
```

```
##                             US non-US
## A New Hope              474.5  552.5
## The Empire Strikes Back 310.7  338.7
## Return of the Jedi      380.3  468.5
## A New Hope              474.5  552.5
## The Empire Strikes Back 310.7  338.7
## Return of the Jedi      380.3  468.5
```

```
# Total revenue for US and non-US
total_revenue_vector <- colSums(all_wars_matrix)

# Print out total_revenue_vector
total_revenue_vector
```

```
##     US non-US
## 2331.0 2719.4
```

Selection of matrix elements

Similar to vectors, you can use the square brackets `[ ]` to select one or multiple elements from a matrix. Whereas vectors have one dimension, matrices have two dimensions. You should therefore use a comma to separate that what to select from the rows from that what you want to select from the columns. For example:

`my_matrix[1,2]` selects the element at the first row and second column.

`my_matrix[1:3,2:4]` results in a matrix with the data on the rows 1, 2, 3 and columns 2, 3, 4.

If you want to select all elements of a row or a column, no number is needed before or after the comma, respectively:

`my_matrix[,1]` selects all elements of the first column.

`my_matrix[1,]` selects all elements of the first row.

Back to Star Wars with this newly acquired knowledge! As in the previous exercise, `all_wars_matrix` is already available in your workspace.

Instructions

Select the non-US revenue for all movies (the entire second column of `all_wars_matrix`), store the result as `non_us_all`.

Use `mean()` on `non_us_all` to calculate the average non-US revenue for all movies. Simply print out the result.

This time, select the non-US revenue for the first two movies in `all_wars_matrix`. Store the result as `non_us_some`.

Use `mean()` again to print out the average of the values in `non_us_some`.

```
# all_wars_matrix is available in your workspace
all_wars_matrix
```

```
##                           US non-US
## A New Hope              474.5  552.5
## The Empire Strikes Back 310.7  338.7
## Return of the Jedi      380.3  468.5
## A New Hope              474.5  552.5
## The Empire Strikes Back 310.7  338.7
## Return of the Jedi      380.3  468.5
```

```
# Select the non-US revenue for all movies
non_us_all <- all_wars_matrix[,2]

# Average non-US revenue
mean(non_us_all)
```

```
## [1] 453.2333
```

```
# Select the non-US revenue for first two movies
non_us_some <- all_wars_matrix[1:2,2]

# Average non-US revenue for first two movies
mean(non_us_some)
```

```
## [1] 445.6
```

A little arithmetic with matrices

Similar to what you have learned with vectors, the standard operators like `+, -, /, *,` etc. work in an element-wise way on matrices in R.

For example, `2 * my_matrix` multiplies each element of `my_matrix` by two.

As a newly-hired data analyst for Lucasfilm, it is your job is to find out how many visitors went to each movie for each geographical area. You already have the total revenue figures in `all_wars_matrix`. Assume that the price of a ticket was 5 dollars. Simply dividing the box office numbers by this ticket price gives you the number of visitors.

Instructions

Divide `all_wars_matrix` by 5, giving you the number of `visitors` in millions. Assign the resulting matrix to visitors.

Print out `visitors` so you can have a look.

```
# all_wars_matrix is available in your workspace
all_wars_matrix
```

```
##                           US non-US
## A New Hope              474.5  552.5
## The Empire Strikes Back 310.7  338.7
## Return of the Jedi      380.3  468.5
## A New Hope              474.5  552.5
```

```
## The Empire Strikes Back 310.7  338.7
## Return of the Jedi      380.3  468.5
```

```
# Estimate the visitors
visitors <- all_wars_matrix / 5

# Print the estimate to the console
visitors
```

```
##                            US non-US
## A New Hope              94.90 110.50
## The Empire Strikes Back 62.14  67.74
## Return of the Jedi      76.06  93.70
## A New Hope              94.90 110.50
## The Empire Strikes Back 62.14  67.74
## Return of the Jedi      76.06  93.70
```

A little arithmetic with matrices (2)

Just like `2 * my_matrix` multiplied every element of `my_matrix` by two, `my_matrix1 * my_matrix2` creates a matrix where each element is the product of the corresponding elements in `my_matrix1` and `my_matrix2`.

After looking at the result of the previous exercise, big boss Lucas points out that the ticket prices went up over time. He asks to redo the analysis based on the prices you can find in ticket_prices_matrix (source: imagination).

Those who are familiar with matrices should note that this is not the standard matrix multiplication for which you should use `%*%` *in R.*

Instructions

Divide `all_wars_matrix` by `ticket_prices_matrix` to get the estimated number of US and non-US visitors for the six movies. Assign the result to `visitors`.

From the `visitors` matrix, select the entire first column, representing the number of visitors in the US. Store this selection as `us_visitors`.

Calculate the average number of US visitors; print out the result.

```
#Neil adding ticket_prices_matrix (Only avaliable in DataCamp)

tix_price <- c(5.0, 5.0, 6.0, 6.0, 7.0, 7.0, 4.0, 4.0, 4.5, 4.5, 4.9, 4.9)
ticket_prices_matrix <- matrix(tix_price, nrow = 6, byrow = TRUE,
                               dimnames = list(c("A New Hope", "The Empire Strikes Back", "Return of the Je
                                               c("US", "non-US")))



# all_wars_matrix and ticket_prices_matrix are available in your workspace
all_wars_matrix
```

```
##                            US non-US
## A New Hope              474.5  552.5
## The Empire Strikes Back 310.7  338.7
## Return of the Jedi      380.3  468.5
## A New Hope              474.5  552.5
## The Empire Strikes Back 310.7  338.7
## Return of the Jedi      380.3  468.5
```

```
ticket_prices_matrix
```

```
##                           US non-US
## A New Hope              5.0    5.0
## The Empire Strikes Back 6.0    6.0
## Return of the Jedi      7.0    7.0
## The Phantom Menace      4.0    4.0
## Attack of the Clones    4.5    4.5
## Revenge of the Sith     4.9    4.9
```

```
# Estimated number of visitors
visitors <- all_wars_matrix / ticket_prices_matrix

# US visitors
us_visitors <- visitors[,1]

# Average number of US visitors
mean(us_visitors)
```

```
## [1] 77.7156
```

Chapter 4: Factors

Very often, data falls into a limited number of categories. For example, humans are either male or female. In R, categorical data is stored in factors. Given the importance of these factors in data analysis, you should start learning how to create, subset and compare them now!

What's a factor and why would you use it?

In this chapter you dive into the wonderful world of **factors**.

The term factor refers to a statistical data type used to store categorical variables. The difference between a categorical variable and a continuous variable is that a categorical variable can belong to a **limited number of categories**. A continuous variable, on the other hand, can correspond to an infinite number of values.

It is important that R knows whether it is dealing with a continuous or a categorical variable, as the statistical models you will develop in the future treat both types differently. (You will see later why this is the case.)

A good example of a categorical variable is the variable 'Gender'. A human individual can either be "Male" or "Female", making abstraction of inter-sexes. So here "Male" and "Female" are, in a simplified sense, the two values of the categorical variable "Gender", and every observation can be assigned to either the value "Male" of "Female".

Instructions

Assign to variable `theory` the value `"factors for categorical variables"`.

```
# Assign to the variable theory what this chapter is about!
theory <- "factors for categorical variables"
```

What's a factor and why would you use it?

To create factors in R, you make use of the function `factor()`. First thing that you have to do is create a vector that contains all the observations that belong to a limited number of categories. For example,

`gender_vector` contains the sex of 5 different individuals:

`gender_vector <- c("Male","Female","Female","Male","Male")`

It is clear that there are two categories, or in R-terms **'factor levels'**, at work here: "Male" and "Female".

The function `factor()` will encode the vector as a factor:

`factor_gender_vector <- factor(gender_vector)`

Instructions

Convert the character vector `gender_vector` to a factor with `factor()` and assign the result to factor_gender_vector

Print out `factor_gender_vector` and assert that R prints out the factor levels below the actual values.

```
# Gender vector
gender_vector <- c("Male", "Female", "Female", "Male", "Male")

# Convert gender_vector to a factor
factor_gender_vector <- factor(gender_vector)

# Print out factor_gender_vector
factor_gender_vector
```

```
## [1] Male   Female Female Male   Male
## Levels: Female Male
```

What's a factor and why would you use it? (3)

There are two types of categorical variables: a **nominal categorical variable** and an **ordinal categorical variable**.

A nominal variable is a categorical variable without an implied order. This means that it is impossible to say that 'one is worth more than the other'. For example, think of the categorical variable `animals_vector` with the categories `"Elephant", "Giraffe", "Donkey"` and `"Horse"`. Here, it is impossible to say that one stands above or below the other. (Note that some of you might disagree ;-)).

In contrast, ordinal variables do have a natural ordering. Consider for example the categorical variable `temperature_vector` with the categories: `"Low", "Medium"` and `"High"`. Here it is obvious that `"Medium"` stands above `"Low"`, and `"High"` stands above `"Medium"`.

Instructions Click 'Submit Answer' to check how R constructs and prints nominal and ordinal variables. Do not worry if you do not understand all the code just yet, we will get to that.

```
# Animals
animals_vector <- c("Elephant", "Giraffe", "Donkey", "Horse")
factor_animals_vector <- factor(animals_vector)
factor_animals_vector
```

```
## [1] Elephant Giraffe  Donkey   Horse
## Levels: Donkey Elephant Giraffe Horse
```

```
# Temperature
temperature_vector <- c("High", "Low", "High","Low", "Medium")
factor_temperature_vector <- factor(temperature_vector, order = TRUE, levels = c("Low", "Medium", "High
factor_temperature_vector
```

```
## [1] High   Low    High   Low    Medium
## Levels: Low < Medium < High
```

Factor levels

When you first get a data set, you will often notice that it contains factors with specific factor levels. However, sometimes you will want to change the names of these levels for clarity or other reasons. R allows you to do this with the function `levels()`:

```
levels(factor_vector) <- c("name1", "name2",...)
```

A good illustration is the raw data that is provided to you by a survey. A standard question for every questionnaire is the gender of the respondent. You remember from the previous question that this is a factor and when performing the questionnaire on the streets its levels are often coded as `"M"` and `"F"`.

```
survey_vector <- c("M", "F", "F", "M", "M")
```

Next, when you want to start your data analysis, your main concern is to keep a nice overview of all the variables and what they mean. At that point, you will often want to change the factor levels to `"Male"` and `"Female"` instead of `"M"` and `"F"` to make your life easier.

Watch out: the order with which you assign the levels is important. If you type `levels(factor_survey_vector)`, you'll see that it outputs `[1] "F" "M"`. If you don't specify the levels of the factor when creating the vector, R will automatically assign them alphabetically. To correctly map `"F"` to `"Female"` and `"M"` to `"Male"`, the levels should be set to `c("Female", "Male")`, in this order order.

Instructions

Check out the code that builds a factor vector from `survey_vector`. You should use `factor_survey_vector` in the next instruction.

Change the factor levels of `factor_survey_vector` to `c("Female", "Male")`. Mind the order of the vector elements here.

```
# Code to build factor_survey_vector
survey_vector <- c("M", "F", "F", "M", "M")
factor_survey_vector <- factor(survey_vector)

# Specify the levels of factor_survey_vector
levels(factor_survey_vector) <- c("Female", "Male")

factor_survey_vector
```

```
## [1] Male   Female Female Male   Male
## Levels: Female Male
```

Summarizing a factor

After finishing this course, one of your favorite functions in R will be `summary()`. This will give you a quick overview of the contents of a variable:

```
summary(my_var)
```

Going back to our survey, you would like to know how many `"Male"` responses you have in your study, and how many `"Female"` responses. The `summary()` function gives you the answer to this question.

Instructions

Ask a `summary()` of the `survey_vector` and `factor_survey_vector`. Interpret the results of both vectors. Are they both equally useful in this case?

```
# Build factor_survey_vector with clean levels
survey_vector <- c("M", "F", "F", "M", "M")
```

```
factor_survey_vector <- factor(survey_vector)
levels(factor_survey_vector) <- c("Female", "Male")
factor_survey_vector
```

```
## [1] Male   Female Female Male   Male
## Levels: Female Male
```

```
# Generate summary for survey_vector
summary(survey_vector)
```

```
##    Length     Class      Mode
##         5 character character
```

```
# Generate summary for factor_survey_vector
summary(factor_survey_vector)
```

```
## Female   Male
##      2      3
```

Battle of the sexes

In `factor_survey_vector` we have a factor with two levels: Male and Female. But how does R value these relatively to each other? In other words, who does R think is better, males or females?

Instructions

Read the code in the editor and click 'Submit Answer' to see whether males are worth more than females.

```
# Build factor_survey_vector with clean levels
survey_vector <- c("M", "F", "F", "M", "M")
factor_survey_vector <- factor(survey_vector)
levels(factor_survey_vector) <- c("Female", "Male")

# Male
male <- factor_survey_vector[1]

# Female
female <- factor_survey_vector[2]

# Battle of the sexes: Male 'larger' than female?
male > female
```

```
## Warning in Ops.factor(male, female): '>' not meaningful for factors
```

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

```
#Note from Neil: That is an intended error
```

Ordered factors

Since `"Male"` and `"Female"` are unordered (or nominal) factor levels, R returns a warning message, telling you that the greater than operator is not meaningful. As seen before, R attaches an equal value to the levels for such factors.

But this is not always the case! Sometimes you will also deal with factors that do have a natural ordering between its categories. If this is the case, we have to make sure that we pass this information to R...

Let us say that you are leading a research team of five data analysts and that you want to evaluate their performance. To do this, you track their speed, evaluate each analyst as `"slow"`, `"fast"` or `"insane"`, and save the results in `speed_vector`.

Instructions

As a first step, assign speed_vector a vector with 5 entries, one for each analyst. Each entry should be either "slow", "fast", or "insane". Use the list below:

Analyst 1 is fast,
Analyst 2 is slow,
Analyst 3 is slow,
Analyst 4 is fast and
Analyst 5 is insane
. No need to specify these are factors yet.

```
# Create speed_vector
speed_vector <- c("fast", "slow", "slow", "fast", "insane")
```

Ordered factors (2)

`speed_vector` should be converted to an ordinal factor since its categories have a natural ordering. By default, the function `factor()` transforms `speed_vector` into an unordered factor. To create an ordered factor, you have to add two additional arguments: "'ordered and levels"".

```
factor(some_vector,
       ordered = TRUE,
       levels = c("lev1", "lev2" ...))
```

By setting the argument `ordered` to `TRUE` in the function `factor()`, you indicate that the factor is ordered. With the argument `levels` you give the values of the factor in the correct order.

Instructions

From `speed_vector`, create an ordered factor vector: `factor_speed_vector`. Set ordered to `TRUE`, and set `levels` to `c("slow", "fast", "insane")`.

```
# Create speed_vector
speed_vector <- c("fast", "slow", "slow", "fast", "insane")

# Convert speed_vector to ordered factor vector
factor_speed_vector <- factor(speed_vector,
ordered = TRUE,
levels = c("slow", "fast", "insane"))

# Print factor_speed_vector
factor_speed_vector
```

```
## [1] fast   slow   slow   fast   insane
## Levels: slow < fast < insane
```

```
summary(factor_speed_vector)
```

```
##   slow   fast insane
##      2      2      1
```

Comparing ordered factors

Having a bad day at work, 'data analyst number two' enters your office and starts complaining that 'data analyst number five' is slowing down the entire project. Since you know that 'data analyst number two' has the reputation of being a smarty-pants, you first decide to check if his statement is true.

The fact that `factor_speed_vector` is now ordered enables us to compare different elements (the data analysts in this case). You can simply do this by using the well-known operators.

Instructions Use `[2]` to select from `factor_speed_vector` the factor value for the second data analyst. Store it as `da2`.

Use `[5]` to select the `factor_speed_vector` factor value for the fifth data analyst. Store it as `da5`.

Check if `da2` is greater than `da5`; simply print out the result. Remember that you can use the `>` operator to check whether one element is larger than the other.

```
# Create factor_speed_vector
speed_vector <- c("fast", "slow", "slow", "fast", "insane")
factor_speed_vector <- factor(speed_vector, ordered = TRUE, levels = c("slow", "fast", "insane"))

# Factor value for second data analyst
da2 <- factor_speed_vector[2]

# Factor value for fifth data analyst
da5 <- factor_speed_vector[5]

# Is data analyst 2 faster than data analyst 5?
da2 > da5
```

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

Chapter 5: Data Frames

Most data sets you will be working with will be stored as data frames. By the end of this chapter focused on R basics, you will be able to create a data frame, select interesting parts of a data frame and order a data frame according to certain variables.

What's a dataframe?

You may remember from the chapter about matrices that all the elements that you put in a matrix should be of the same type. Back then, your data set on Star Wars only contained numeric elements.

When doing a market research survey, however, you often have questions such as:

'Are your married?' or 'yes/no' questions (`logical`) 'How old are you?' (`numeric`) 'What is your opinion on this product?' or other 'open-ended' questions (`character`) ... The output, namely the respondents' answers to the questions formulated above, is a data set of different data types. You will often find yourself working with data sets that contain different data types instead of only one.

A data frame has the variables of a data set as columns and the observations as rows. This will be a familiar concept for those coming from different statistical software packages such as SAS or SPSS.

Instructions Click 'Submit Answer'. The data from the built-in example data frame `mtcars` will be printed to the console.

```
# Print out built-in R data frame
mtcars
```

```
##                      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
## Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
## Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
## Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
## Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
## Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
## Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
## AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
## Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
## Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
## Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
## Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
## Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
## Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

Quick, have a look at your data set

Wow, that is a lot of cars!

Working with large data sets is not uncommon in data analysis. When you work with (extremely) large data sets and data frames, your first task as a data analyst is to develop a clear understanding of its structure and main elements. Therefore, it is often useful to show only a small part of the entire data set.

So how to do this in R? Well, the function `head()` enables you to show the first observations of a data frame. Similarly, the function `tail()` prints out the last observations in your data set.

Both `head()` and `tail()` print a top line called the 'header', which contains the names of the different variables in your data set.

Instructions

Call `head()` on the `mtcars` data set to have a look at the header and the first observations.

```
# Call head() on mtcars
head(mtcars)
```

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

Have a look at the structure

Another method that is often used to get a rapid overview of your data is the function ‘`str()`. The function `str()` shows you the structure of your data set. For a data frame it tells you:

The total number of observations (e.g. 32 car types) The total number of variables (e.g. 11 car features) A full list of the variables names (e.g. `mpg`, `cyl` ...) The data type of each variable (e.g. `num`) The first observations

Applying the `str()` function will often be the first thing that you do when receiving a new data set or data frame. It is a great way to get more insight in your data set before diving into the real analysis.

Instructions

Investigate the structure of `mtcars`. Make sure that you see the same numbers, variables and data types as mentioned above.

```
# Investigate the structure of mtcars
str(mtcars)
```

```
## 'data.frame':    32 obs. of  11 variables:
##  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num  160 160 108 258 360 ...
##  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
##  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
##  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
##  $ qsec: num  16.5 17 18.6 19.4 17 ...
##  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
##  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
##  $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
##  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

Creating a data frame

Since using built-in data sets is not even half the fun of creating your own data sets, the rest of this chapter is based on your personally developed data set. Put your jet pack on because it is time for some space exploration!

As a first goal, you want to construct a data frame that describes the main characteristics of eight planets in our solar system. According to your good friend Buzz, the main features of a planet are:

The type of planet (Terrestrial or Gas Giant). The planet’s diameter relative to the diameter of the Earth. The planet’s rotation across the sun relative to that of the Earth. If the planet has rings or not (TRUE or FALSE).

After doing some high-quality research on Wikipedia, you feel confident enough to create the necessary vectors: `name, type, diameter, rotation and rings`; these vectors have already been coded up on the right. The first element in each of these vectors correspond to the first observation.

You construct a data frame with the `data.frame()` function. As arguments, you pass the vectors from before: they will become the different columns of your data frame. Because every column has the same length, the vectors you pass should also have the same length. But don't forget that it is possible (and likely) that they contain different types of data.

Instructions

Use the function `data.frame()` to construct a data frame. Pass the vectors `name, type, diameter, rotation and rings` as arguments to `data.frame()`, in this order. Call the resulting data frame `planets_df`.

```
# Definition of vectors
name <- c("Mercury", "Venus", "Earth", "Mars", "Jupiter", "Saturn", "Uranus", "Neptune")
type <- c("Terrestrial planet", "Terrestrial planet", "Terrestrial planet",
          "Terrestrial planet", "Gas giant", "Gas giant", "Gas giant", "Gas giant")
diameter <- c(0.382, 0.949, 1, 0.532, 11.209, 9.449, 4.007, 3.883)
rotation <- c(58.64, -243.02, 1, 1.03, 0.41, 0.43, -0.72, 0.67)
rings <- c(FALSE, FALSE, FALSE, FALSE, TRUE, TRUE, TRUE, TRUE)

# Create a data frame from the vectors
planets_df <- data.frame(name, type, diameter, rotation, rings)
```

Creating a data frame (2)

The `planets_df` data frame should have 8 observations and 5 variables. It has been made available in the workspace, so you can directly use it.

Instructions Use `str()` to investigate the structure of the new `planets_df` variable.

```
# Check the structure of planets_df
str(planets_df)
```

```
## 'data.frame':    8 obs. of  5 variables:
##  $ name    : Factor w/ 8 levels "Earth","Jupiter",..: 4 8 1 3 2 6 7 5
##  $ type    : Factor w/ 2 levels "Gas giant","Terrestrial planet": 2 2 2 2 1 1 1 1
##  $ diameter: num  0.382 0.949 1 0.532 11.209 ...
##  $ rotation: num  58.64 -243.02 1 1.03 0.41 ...
##  $ rings   : logi  FALSE FALSE FALSE FALSE TRUE TRUE ...
```

Selection of data frame elements

Similar to vectors and matrices, you select elements from a data frame with the help of square brackets `[ ]`. By using a comma, you can indicate what to select from the rows and the columns respectively. For example:

`my_df[1,2]` selects the value at the first row and select element in `my_df`.

`my_df[1:3,2:4]` selects rows 1, 2, 3 and columns 2, 3, 4 in my_df. Sometimes you want to select all elements of a row or column. For example, `my_df[1, ]` selects all elements of the first row. Let us now apply this technique on `planets_df`!

Instructions

From `planets_df`, select the diameter of Mercury: this is the value at the first row and the third column. Simply print out the result.

From `planets_df`, select all data on Mars (the fourth row). Simply print out the result.

```
# The planets_df data frame from the previous exercise is pre-loaded

# Print out diameter of Mercury (row 1, column 3)
planets_df[1,3]
```

```
## [1] 0.382
```

```
# Print out data for Mars (entire fourth row)
planets_df[4,]
```

```
##   name               type diameter rotation rings
## 4 Mars Terrestrial planet    0.532     1.03 FALSE
```

Selection of data frame elements (2)

Instead of using numerics to select elements of a data frame, you can also use the variable names to select columns of a data frame.

Suppose you want to select the first three elements of the `type` column. One way to do this is

```
planets_df[1:3,1]
```

A possible disadvantage of this approach is that you have to know (or look up) the column number of `type`, which gets hard if you have a lot of variables. It is often easier to just make use of the variable name:

```
planets_df[1:3,"type"]
```

Instructions

Select and print out the first 5 values in the `"diameter"` column of `planets_df`

```
# The planets_df data frame from the previous exercise is pre-loaded

# Select first 5 values of diameter column
planets_df[1:5,"diameter"]
```

```
## [1]  0.382  0.949  1.000  0.532 11.209
```

Only planets with rings

You will often want to select an entire column, namely one specific variable from a data frame. If you want to select all elements of the variable `diameter`, for example, both of these will do the trick:

```
planets_df[,3]
planets_df[,"diameter"]
```

However, there is a short-cut. If your columns have names, you can use the `$` sign:

```
planets_df$diameter
```

Instructions

Use the `$` sign to select the `rings` variable from `planets_df`. Store the vector that results as `rings_vector`.

Print out `rings_vector` to see if you got it right.

```
# planets_df is pre-loaded in your workspace

# Select the rings variable from planets_df
```

```
rings_vector <- planets_df$rings

# Print out rings_vector
rings_vector
```

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

Only planets with rings (2)

You probably remember from high school that some planets in our solar system have rings and others do not. But due to other priorities at that time (read: puberty) you can not recall their names, let alone their rotation speed, etc.

Could R help you out?

If you type `rings_vector` in the console, you get:

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

This means that the first four observations (or planets) do not have a ring (`FALSE`), but the other four do (`TRUE`). However, you do not get a nice overview of the names of these planets, their diameter, etc. Let's try to use `rings_vector` to select the data for the four planets with rings.

Instructions

The code on the right selects the `name` column of all planets that have rings. Adapt the code so that instead of only the `name` column, all columns for planets that have rings are selected.

```
# planets_df and rings_vector are pre-loaded in your workspace

# Adapt the code to select all columns for planets with rings
planets_df[rings_vector,]
```

```
##      name      type diameter rotation rings
## 5 Jupiter Gas giant   11.209     0.41  TRUE
## 6  Saturn Gas giant    9.449     0.43  TRUE
## 7  Uranus Gas giant    4.007    -0.72  TRUE
## 8 Neptune Gas giant    3.883     0.67  TRUE
```

Only planets with rings but shorter

So what exactly did you learn in the previous exercises? You selected a subset from a data frame (`planets_df`) based on whether or not a certain condition was true (rings or no rings), and you managed to pull out all relevant data. Pretty awesome! By now, NASA is probably already flirting with your CV ;-).

Now, let us move up one level and use the function `subset()`. You should see the `subset()` function as a short-cut to do exactly the same as what you did in the previous exercises.

```
subset(my_df, subset = some_condition)
```

The first argument of `subset()` specifies the data set for which you want a subset. By adding the second argument, you give R the necessary information and conditions to select the correct subset.

The code below will give the exact same result as you got in the previous exercise, but this time, you didn't need the `rings_vector`!

```
subset(planets_df, subset = rings)
```

Instructions

Use `subset()` on `planets_df` to select planets that have a `diameter` smaller than Earth. Because the diameter variable is a relative measure of the planet's diameter w.r.t that of planet Earth, your condition is `diameter < 1`.

```
# planets_df is pre-loaded in your workspace

# Select planets with diameter < 1
subset(planets_df, subset = diameter < 1)
```

```
##      name               type diameter rotation rings
## 1 Mercury Terrestrial planet    0.382    58.64 FALSE
## 2   Venus Terrestrial planet    0.949  -243.02 FALSE
## 4    Mars Terrestrial planet    0.532     1.03 FALSE
```

Sorting

Making and creating rankings is one of mankind's favorite affairs. These rankings can be useful (best universities in the world), entertaining (most influential movie stars) or pointless (best 007 look-a-like).

In data analysis you can sort your data according to a certain variable in the data set. In R, this is done with the help of the function `order()`.

`order()` is a function that gives you the ranked position of each element when it is applied on a variable, such as a vector for example:

```
> a <- c(100, 10, 1000)
> order(a)
[1] 2 1 3
```

10, which is the second element in `a`, is the smallest element, so 2 comes first in the output of `order(a)`. 100, which is the first element in `a` is the second smallest element, so 1 comes second in the output of `order(a)`.

This means we can use the output of `order(a)` to reshuffle `a`:

```
> a[order(a)]
[1]   10  100 1000
```

Instructions

Experiment with the `order()` function in the console. Click 'Submit Answer' when you are ready to continue

```
# Play around with the order function in the console
```

Sorting your data frame

Alright, now that you understand the `order()` function, let us do something useful with it. You would like to rearrange your data frame such that it starts with the smallest planet and ends with the largest one. A sort on the `diameter` column.

Instructions

Call `order()` on `planets_df$diameter` (the `diameter` column of `planets_df`). Store the result as `positions`.

Now reshuffle `planets_df` with the `positions` vector as row indexes inside square brackets. Keep all columns. Simply print out the result.

```
# planets_df is pre-loaded in your workspace

# Use order() to create positions
positions <-  order(planets_df$diameter)

# Use positions to sort planets_df
planets_df[positions, ]
```

```
##      name               type diameter rotation rings
## 1 Mercury Terrestrial planet    0.382    58.64 FALSE
## 4    Mars Terrestrial planet    0.532     1.03 FALSE
## 2   Venus Terrestrial planet    0.949  -243.02 FALSE
## 3   Earth Terrestrial planet    1.000     1.00 FALSE
## 8 Neptune          Gas giant    3.883     0.67  TRUE
## 7  Uranus          Gas giant    4.007    -0.72  TRUE
## 6  Saturn          Gas giant    9.449     0.43  TRUE
## 5 Jupiter          Gas giant   11.209     0.41  TRUE
```

Chapter 6: Lists

Lists, as opposed to vectors, can hold components of different types, just like your to-do list at home or at work. This intro to R chapter will teach you how to create, name and subset these lists.

Lists, why would you need them?

Congratulations! At this point in the course you are already familiar with:

Vectors (one dimensional array): can hold numeric, character or logical values. The elements in a vector all have the same data type.

Matrices (two dimensional array): can hold numeric, character or logical values. The elements in a matrix all have the same data type.

Data frames (two-dimensional objects): can hold numeric, character or logical values. Within a column all elements have the same data type, but different columns can be of different data type.

Pretty sweet for an R newbie, right? ;-)

Instructions

Click 'Submit Answer' to start learning everything about lists!

```
# Just click the 'Submit Answer' button.
```

Lists, why would you need them? (2)

A **list** in R is similar to your to-do list at work or school: the different items on that list most likely differ in length, characteristic, type of activity that has to do be done, ...

A list in R allows you to gather a variety of objects under one name (that is, the name of the list) in an ordered way. These objects can be matrices, vectors, data frames, even other lists, etc. It is not even required that these objects are related to each other in any way.

You could say that a list is some kind super data type: you can store practically any piece of information in it!

Instructions

Click 'Submit Answer' to start learning everything about lists!

```
# Just click the 'Submit Answer' button.
```

Creating a list

Let us create our first list! To construct a list you use the function `list()`:

```
my_list <- list(comp1, comp2 ...)
```

The arguments to the list function are the list components. Remember, these components can be matrices, vectors, other lists, ...

Instructions

Construct a list, named `my_list`, that contains the variables `my_vector`, `my_matrix` and `my_df` as list components.

```
# Vector with numerics from 1 up to 10
my_vector <- 1:10

# Matrix with numerics from 1 up to 9
my_matrix <- matrix(1:9, ncol = 3)

# First 10 elements of the built-in data frame mtcars
my_df <- mtcars[1:10,]

# Construct list with these different elements:
my_list <- list(my_vector, my_matrix, my_df)
```

Creating a named list

Well done, you're on a roll!

Just like on your to-do list, you want to avoid not knowing or remembering what the components of your list stand for. That is why you should give names to them:

```
my_list <- list(name1 = your_comp1,
                name2 = your_comp2)
```

This creates a list with components that are named `name1`, `name2`, and so on. If you want to name your lists after you've created them, you can use the `names()` function as you did with vectors. The following commands are fully equivalent to the assignment above:

```
my_list <- list(your_comp1, your_comp2)
names(my_list) <- c("name1", "name2")
```

Instructions

Change the code of the previous exercise (see editor) by adding names to the components. Use for `my_vector` the name `vec`, for `my_matrix` the name `mat` and for `my_df` the name `df`.

Print out `my_list` so you can inspect the output.

```
# Vector with numerics from 1 up to 10
my_vector <- 1:10
```

```r
# Matrix with numerics from 1 up to 9
my_matrix <- matrix(1:9, ncol = 3)

# First 10 elements of the built-in data frame mtcars
my_df <- mtcars[1:10,]

# Adapt list() call to give the components names
my_list <- list(vec = my_vector,
                mat = my_matrix,
                df = my_df)

# Print out my_list
my_list
```

```
## $vec
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $mat
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
##
## $df
##                    mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Duster 360        14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 240D         24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230          22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 280          19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
```

Creating a named list (2)

Being a huge movie fan (remember your job at LucasFilms), you decide to start storing information on good movies with the help of lists.

Start by creating a list for the movie "The Shining". We have already created the variables `mov`, `act` and `rev` in your R workspace. Feel free to check them out in the console.

Instructions

Complete the code on the right to create shining_list; it contains three elements:

moviename: a character string with the movie title (stored in `mov`)

actors: a vector with the main actors' names (stored in `act`)

reviews: a data frame that contains some reviews (stored in `rev`)

Do not forget to name the list components accordingly (names are moviename, actors and reviews).

```
# The variables mov, act and rev are available

#Neil Adding mov, act & rev (Only in DataCamp)
mov <- "The Shining"
act <- c("Jack Nicholson",   "Shelley Duvall" ,  "Danny Lloyd"  ,    "Scatman Crothers", "Barry Nelson")
scores <-  c(4.5, 4.0, 5.0)
sources <-  c("IMDb1", "IMDb1", "IMDb3")
comments <-  c("Best Horror Film I Have Ever Seen", "A truly brilliant and scary film from Stanley Kubr
rev <- data.frame(scores, sources, comments)

# Finish the code to build shining_list
shining_list <- list(moviename = mov,
                     actors = act,
                     reviews = rev)
```

Selecting elements from a list

Your list will often be built out of numerous elements and components. Therefore, getting a single element, multiple elements, or a component out of it is not always straightforward.

One way to select a component is using the numbered position of that component. For example, to "grab" the first component of `shining_list` you type

```
shining_list[[1]]
```

A quick way to check this out is typing it in the console. Important to remember: to select elements from vectors, you use single square brackets: `[ ]`. Don't mix them up!

You can also refer to the names of the components, with `[[ ]]` or with the `$` sign. Both will select the data frame representing the reviews:

```
shining_list[["reviews"]]
shining_list$reviews
```

Besides selecting components, you often need to select specific elements out of these components. For example, with `shining_list[[2]][1]` you select from the second component, `actors` (`shining_list[[2]]`), the first element (`[1]`). When you type this in the console, you will see the answer is Jack Nicholson.

Instructions

Select from `shining_list` the vector representing the actors. Simply print out this vector.

Select from `shining_list` the second element in the vector representing the actors. Do a printout like before.

```
# shining_list is already pre-loaded in the workspace

# Print out the vector representing the actors
shining_list$actors
```

```
## [1] "Jack Nicholson"   "Shelley Duvall"   "Danny Lloyd"
## [4] "Scatman Crothers" "Barry Nelson"
```

```
# Print the second element of the vector representing the actors
shining_list$actors[2]
```

```
## [1] "Shelley Duvall"
```

Adding more movie information to the list

Being proud of your first list, you shared it with the members of your movie hobby club. However, one of the senior members, a guy named M. McDowell, noted that you forgot to add the release year. Given your ambitions to become next year's president of the club, you decide to add this information to the list.

To conveniently add elements to lists you can use the `c()` function, that you also used to build vectors:

```
ext_list <- c(my_list , my_val)
```

This will simply extend the original list, `my_list`, with the component `my_val`. This component gets appended to the end of the list. If you want to give the new list item a name, you just add the name as you did before:

```
ext_list <- c(my_list, my_name = my_val)
```

Instructions

Complete the code below such that an item named `year` is added to the shining_list with the value 1980. Assign the result to `shining_list_full`. Finally, have a look at the structure of `shining_list_full` with the `str()` function.

```
# shining_list, the list containing movie name, actors and reviews, is pre-loaded in the workspace

# We forgot something; add the year to shining_list
shining_list_full <- c(shining_list, year = 1980)

# Have a look at shining_list_full
str(shining_list_full)
```

```
## List of 4
##  $ moviename: chr "The Shining"
##  $ actors   : chr [1:5] "Jack Nicholson" "Shelley Duvall" "Danny Lloyd" "Scatman Crothers" ...
##  $ reviews  :'data.frame':   3 obs. of  3 variables:
##   ..$ scores  : num [1:3] 4.5 4 5
##   ..$ sources : Factor w/ 2 levels "IMDb1","IMDb3": 1 1 2
##   ..$ comments: Factor w/ 2 levels "A truly brilliant and scary film from Stanley Kubrick",..: 2 1 1
##  $ year     : num 1980
```