sd(x, na.rm=TRUE)==> considering there are NA in data for a list/vector

want to create a sequence of numbers.

```
> 1:5
[1] 1 2 3 4 5
```

also

```
seq(from=0, to=20)
> seq(from=0, to=20, by=2)
[1] 0 2 4 6 8 10 12 14 16 18 20
> seq(from=0, to=20, by=5)
[1] 0 5 10 15 20
```

```
# Exponentiation
2^5

# Modulo
28 %% 6

# Exponentiation
2^5

# Modulo
28 %% 6
```

```
v <- c(3, pi, 4)
> v == pi # Compare a 3-element vector against one number
[1] FALSE TRUE FALSE
```

```
> v != pi
[1] TRUE FALSE TRUE
```

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

Addition of vectors:->

```
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
```

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)
```

```
poker_vector <- c(140, -50, 20, -120, 240)
total_poker <- sum(poker_vector)
```

Access elements of Vector:-

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], 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)]
```

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"]
```

•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.

```
selection_vector <- poker_vector > 0

# 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]
```

In the matrix() function:

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

- 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.

Create matrix for Boxoffice :-

```
# 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)

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

# 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

rownames(star_wars_matrix) <- titles

colnames(star_wars_matrix)

<- region

star_wars_matrix

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)

```
# 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 Jedi"),

                        c("US", "non-US")))


#
```

Calculate worldwide box office figures

worldwide_vector <- rowSums(star_wars_matrix)

Adding a column for the Worldwide box office

100xp

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 ...)
```

```
# 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 Jedi"),

                        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)
```

Rowbind:-

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

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

The total box office revenue for the entire saga

colSums is similar to rowSums

```
# all_wars_matrix is available in your workspace
all_wars_matrix

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

# Print out

total_revenue_vector
total_revenue_vector
```

Selection of matrix elements

100xp

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

```
# all_wars_matrix is available in your workspace

all_wars_matrix


# Select the non-US revenue for all movies

non_us_all <- all_wars_matrix[,2]


# Average non-US revenue

mean(non_us_all)


# 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)
```

A little arithmetic with matrices

100xp

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.

```
# all_wars_matrix is available in your workspace
all_wars_matrix

# Estimate the visitors
visitors <- all_wars_matrix / 5

# Print the estimate to the console
visitors
```

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

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

all_wars_matrix

ticket_prices_matrix
```

```
# 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)
```

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

100xp

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".

Factor

What's a factor and why would you use it? (2)

100xp

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)
```

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

100xp

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".

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

# 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
```

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
```

FACTOR and level(need to check)

Summarizing a factor

100xp

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


# Generate summary for survey_vector

summary(survey_vector)
```

# Generate summary for factor_survey_vector

summary

(factor_survey_vector)

# 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

Ordered factors

70xp

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.

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.

```
# Create speed_vector

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

Ordered factors (2)

100xp

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.

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
summary(factor_speed_vector)
```

Comparing ordered factors

100xp

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.

- 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

What's a data frame?

100xp

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.

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

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.

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.

str(mtcars)

Creating a data frame

100xp

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)

Selection of data frame elements

100xp

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]


# Print out data for Mars (entire


fourth row)

planets_df[4,]
```

Selection of data frame elements (2)

100xp

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"]
```

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

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

Only planets with rings

100xp

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
```

•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
```

Only planets with rings (2)

100xp

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, ]
```

Only planets with rings but shorter

100xp

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 = planets_df$diameter < 1)
```

Sorting

100xp

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
a<- c(10,15,5)
order(a)
a[order(a)]
```

Sorting your data frame

0xp

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, ]
```

Lists, why would you need them? (2)

100xp

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!

Creating a list

100xp

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

100xp

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


# 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(my_vector, my_matrix, my_df)

names(my_list)  <-c("vec","mat","df")


#


Print out my_list

my_list
```

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).

Selecting elements from a list

100xp

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

# Print the second element of the vector

representing the actors

shining_list$actors[2]

Adding more movie information to the list

100xp

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)

Equality

100xp

The most basic form of comparison is equality. Let's briefly recap its syntax. The following statements all evaluate to TRUE (feel free to try them out in the

console).

```
3 == (2 + 1)

"intermediate" != "r"

TRUE != FALSE

"Rchitect" != "rchitect"
```

Notice from the last expression that R is case sensitive: "R" is not equal to "r". Keep this in mind when solving the exercises in this chapter!

Instructions

•In the editor on the right, write R code to see if TRUE equals FALSE.

•Likewise, check if -6 * 14 is not equal to 17 - 101.

•Next up: comparison of character strings. Ask R whether the strings "useR" and "user" are equal.

•Finally, find out what happens if you compare logicals to numerics: are TRUE and 1 equal?

```
# Comparison of logicals

TRUE == FALSE


# Comparison of numerics

-6 * 14 != 17 - 101
```

# Comparison of character strings

"useR" == "user"

# Compare a logical with a numeric

TRUE

== 1

Greater and less than

100xp

Apart from equality operators, Filip also introduced the less than and greater than operators: < and >. You can also add an equal sign to express less than or equal to

or greater than or equal to, respectively. Have a look at the following R expressions, that all evaluate to FALSE:

(1 + 2) > 4

"dog" < "Cats"

TRUE <= FALSE

Remember that for string comparison, R determines the greater than relationship based on alphabetical order. Also, keep in mind that TRUE corresponds to 1 in R, and

FALSE coerces to 0 behind the scenes. Therefore, FALSE < TRUE is TRUE.

Instructions

Write R expressions to check whether:

- -6 * 5 + 2 is greater than or equal to -10 + 1.
- "raining" is less than or equal to "raining dogs".
- TRUE is greater than FALSE.

```
# Comparison of numerics
-6 * 5 + 2 >=  -10+1


# Comparison of character strings
"raining" <= "raining dogs"


# Comparison of logicals
TRUE > FALSE
```

Compare vectors

100xp

You are already aware that R is very good with vectors. Without having to change anything about the syntax, R's relational operators also work on vectors.

Let's go back to the example that was started in the video. You want to figure out whether your activity on social media platforms have paid off and decide to look at

your results for LinkedIn and Facebook. The sample code in the editor initializes the vectors linkedin and facebook. Each of the vectors contains the number of profile

views your LinkedIn and Facebook profiles had over the last seven days.

Instructions

Using relational operators, find a logical answer, i.e. TRUE or FALSE, for the following questions:

•On which days did the number of LinkedIn profile views exceed 15?

•When was your LinkedIn profile viewed only 5 times or fewer?

•When was your LinkedIn profile visited more often than your Facebook profile?

```
# The linkedin and facebook vectors have already been created for you

linkedin <- c(16, 9, 13, 5, 2, 17, 14)

facebook <- c(17, 7, 5, 16, 8, 13, 14)


# Popular days

linkedin

> 15


# Quiet days
```

linkedin <=5

# LinkedIn more popular than Facebook

linkedin > facebook

Compare matrices

100xp

R's ability to deal with different data structures for comparisons does not stop at vectors. Matrices and relational operators also work together seamlessly!

Instead of in vectors (as in the previous exercise), the LinkedIn and Facebook data is now stored in a matrix called views. The first row contains the LinkedIn

information; the second row the Facebook information. The original vectors facebook and linkedin are still available as well.

Instructions

Using the relational operators you've learned so far, try to discover the following:

•When were the views exactly equal to 13? Use the views matrix to return a logical matrix.

•For which days were the number of views less than or equal to 14? Again, have R return a logical matrix.

```
# The social data has been created for you

linkedin <- c(16, 9, 13, 5, 2, 17, 14)

facebook <- c(17, 7, 5, 16, 8, 13, 14)

views <- matrix(c(linkedin, facebook), nrow = 2,

byrow = TRUE)


# When does views equal 13?

views == 13


# When is views less than or equal to 14?

views <= 14
```

& and |

100xp

Before you work your way through the next exercises, have a look at the following R expressions. All of them will evaluate to TRUE:

```
TRUE & TRUE

FALSE | TRUE

5 <= 5 & 2 < 3

3 < 4 | 7 < 6
```

Watch out: 3 < x < 7 to check if x is between 3 and 7 will not work; you'll need 3 < x & x < 7 for that.

In this exercise, you'll be working with the last variable. This variable equals the last value of the linkedin vector that you've worked with previously. The linkedin

vector represents the number of LinkedIn views your profile had in the last seven days, remember? Both the variables linkedin and last have already been defined in the

editor.

Instructions

& ==> AND

| ==>OR

&& only first element is checked

|| only first element is checked

Write R expressions to solve the following questions concerning the variable last:

•Is last under 5 or above 10?

•Is last between 15 and 20, excluding 15 but including 20?

```
# The linkedin and last variable are already defined for you

linkedin <- c(16, 9, 13, 5, 2, 17, 14)

last <- tail(linkedin, 1)
```

# Is last under 5 or above 10?

last <5 | last

> 10

# Is last between 15 (exclusive) and 20 (inclusive)?

last >15 & < =20

Like relational operators, logical operators work perfectly fine with vectors and matrices.

Both the vectors linkedin and facebook are available again. Also a matrix - views - has been defined; its first and second row correspond to the linkedin and facebook

vectors, respectively. Ready for some advanced queries to gain more insights into your social outreach?

Instructions

•When did LinkedIn views exceed 10 and did Facebook views fail to reach 10 for a particular day? Use the linkedin and facebook vectors.

•When were one or both of your LinkedIn and Facebook profiles visited at least 12 times?

•When is the views matrix equal to a number between 11 and 14, excluding 11 and including 14?

```
# The social data (linkedin, facebook, views) has been created for you

# linkedin exceeds 10 but facebook below 10
linkedin > 10  & facebook < 10

# When were one or both

visited at least 12 times?
linkedin >=12 | facebook >= 12

# When is views between 11 (exclusive) and 14 (inclusive)?
views > 11 & views <= 14
```

Blend it all together

200xp

With the things you've learned by now, you're able to solve pretty cool problems.

Instead of recording the number of views for your own LinkedIn profile, suppose you conducted a survey inside the company you're working for. You've asked every

employee with a LinkedIn profile how many visits their profile has had over the past seven days. You stored the results in a data frame called li_df. This data frame

is available in the workspace; type li_df in the console to check it out.

Instructions

•Select the entire second column, named day2, from the li_df data frame as a vector and assign it to second.

•Use second to create a logical vector, that contains TRUE if the corresponding number of views is strictly greater than 25 or strictly lower than 5 and FALSE

otherwise. Store this logical vector as extremes.

•Use sum() on the extremes vector to calculate the number of TRUEs in extremes (i.e. to calculate the number of employees that are either very popular or very low-

profile). Simply print this number to the console.

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


# Select the second column, named day2, from li_df: second


second <- li_df$day2

# Build a logical vector, TRUE if value in second


is extreme: extremes

extremes <- second <5 | second > 25


# Count the number of TRUEs in extremes

sum(extremes == TRUE)


# Solve it with a one-liner
```

```
sum((second < 5 & second

> 25 ) == TRUE)
```

The if statement

Before diving into some exercises on the if statement, have another look at its syntax:

```
if (condition) {
  expr
}
```

Remember your vectors with social profile views? Let's look at it from another angle. The medium variable gives information about the social website; the num_views

variable denotes the actual number of views that particular medium had on the last day of your recordings. Both these variables have already been defined in the

editor.

Instructions

•Examine the if statement that prints out "Showing LinkedIn information" if the medium variable equals "LinkedIn".

•Code an if statement that prints "You're popular!" to the console if the num_views variable exceeds 15.

The if statement

100xp

Before diving into some exercises on the if statement, have another look at its syntax:

```
if (condition) {
  expr
}
```

Remember your vectors with social profile views? Let's look at it from another angle. The medium variable gives information about the social website; the num_views

variable denotes the actual number of views that particular medium had on the last day of your recordings. Both these variables have already been defined in the

editor.

Instructions

•Examine the if statement that prints out "Showing LinkedIn information" if the medium variable equals "LinkedIn".

•Code an if statement that prints "You're popular!" to the console if the num_views variable exceeds 15.

```
# Variables related to your last day of recordings

medium <- "LinkedIn"

num_views <- 14


# Examine the if statement for medium

if (medium == "LinkedIn") {

  print("Showing

LinkedIn information")

}


# Write the if statement for num_views

 if (num_views > 15) {

   print ("You're popular!")

 }
```

Add an else

100xp

You can only use an else statement in combination with an if statement. The else statement does not require a condition; its corresponding code is simply run if all of

the preceding conditions in the control structure are FALSE. Here's a recipe for its usage:

```
if (condition) {
  expr1
} else {
  expr2
}
```

It's important that the else keyword comes on the same line as the closing bracket of the if part!

Both if statements that you coded in the previous exercises are already available in the editor. It's now up to you to extend them with the appropriate else

statements!

Instructions

Add an else statement to both control structures, such that

- "Unknown medium" gets printed out to the console when the if-condition on medium does not hold.
- R prints out "Try to be more visible!" when the if-condition on num_views is not met.

```
# Variables related to your last day of recordings
medium <- "LinkedIn"
num_views <- 14

# Control structure for medium
if (medium == "LinkedIn") {
  print("Showing LinkedIn

information")
} else if (medium == "Facebook") {
  # Add code to print correct string when condition is TRUE
print("Showing Facebook information")
} else {
  print("Unknown

medium")
}

# Control structure for num_views
if (num_views > 15) {
  print("You're popular!")
} else if (num_views <= 15 & num_views > 10) {
  # Add code to print correct

string when condition is TRUE
print("Your number of views is average")
} else {
  print("Try to be more visible!")
}
```

Take control!

100xp

In this exercise, you will combine everything that you've learned so far: relational operators, logical operators and control constructs. You'll need it all!

In the editor, we've coded two values beforehand: li and fb, denoting the number of profile views your LinkedIn and Facebook profile had on the last day of recordings.

Go through the instructions to create R code that generates a 'social media score', sms, based on the values of li and fb.

Instructions

Finish the control-flow construct with the following behavior:

•If both li and fb are 15 or higher, set sms equal to double the sum of li and fb.

•If both li and fb are strictly below 10, set sms equal to half the sum of li and fb.

•In all other cases, set sms equal to li + fb.

•Finally, print the resulting sms variable to the console.

# Variables related to your last day of recordings

```
li <- 15

fb <- 9


# Code the control-flow construct

if (li > 15 & fb > 15) {

  sms <- 2 * (li + fb)

} else if (li < 10 & fb


<10) {

  sms <- 0.5 * (li + fb)

} else {

  sms <- li+fb

}


# Print the resulting sms to the console


sms
```

While loop

Write a while loop

100xp

Let's get you started with building a while loop from the ground up. Have another look at its recipe:

```
while (condition) {
  expr
}
```

Remember that the condition part of this recipe should become FALSE at some point during the execution. Otherwise, the while loop will go on indefinitely. In

DataCamp's learning interface, your session will be disconnected in this case.

Have a look at the code on the right; it initializes the speed variables and already provides a while loop template to get you started.

Instructions

Code a while loop with the following characteristics:

•The condition of the while loop should check if speed is higher than 30.

•Inside the body of the while loop, print out "Slow down!".

•Inside the body of the while loop, decrease the speed by 7 units. This step is crucial; otherwise your while loop will never stop.

```
# Initialize the speed variable
speed <- 64
```

```
# Code the while loop
while (speed > 30 ) {
print("Slow down!")
speed <- speed -7
  }

# Print out the speed variable
speed
```

Throw in more conditionals

100xp

In the previous exercise, you simulated the interaction between a driver and a driver's assistant: When the speed was too high, "Slow down!" got printed out to the

console, resulting in a decrease of your speed by 7 units.

There are several ways in which you could make your driver's assistant more advanced. For example, the assistant could give you different messages based on your speed

or provide you with a current speed at a given moment.

A while loop similar to the one you've coded in the previous exercise is already available in the editor. It prints out your current speed, but there's no code that

decreases the speed variable yet, which is pretty dangerous. Can you make the appropriate changes?

Instructions

•If the speed is greater than 48, have R print out "Slow down big time!", and decrease the speed by 11.

•Otherwise, have R simply print out "Slow down!", and decrease the speed by 6.

```
# Initialize the speed variable
speed <- 64

# Extend/adapt the while loop
while (speed > 30) {
  print(paste("Your speed is",speed))
  if ( speed > 48) {
  print("Slow down

big time!")
   speed <- speed -11

  } else {
  print("Slow down!")
  speed <- speed - 6



  }
```

Stop the while loop: break

100xp

There are some very rare situations in which severe speeding is necessary: what if a hurricane is approaching and you have to get away as quickly as possible? You

don't want the driver's assistant sending you speeding notifications in that scenario, right?

This seems like a great opportunity to include the break statement in the while loop you've been working on. Remember that the break statement is a control statement.

When R encounters it, the while loop is abandoned completely.

Instructions

Adapt the while loop such that it is abandoned when the speed of the vehicle is greater than 80. This time, the speed variable has been initialized to 88; keep it that

way.

Initialize the speed variable

speed <- 88

```
while (speed > 30) {
  print(paste("Your speed is", speed))


  # Break the while loop when speed exceeds 80
  if (speed > 80 ) {


 break


  }


  if (speed > 48) {
    print("Slow down big time!")
    speed <- speed - 11
  } else {
    print("Slow down!")
    speed <- speed - 6
  }
}
```

Build a while loop from scratch

100xp

The previous exercises guided you through developing a pretty advanced while loop, containing a break statement and different messages and updates as determined by

control flow constructs. If you manage to solve this comprehensive exercise using a while loop, you're totally ready for the next topic: the for loop.

Instructions

Finish the while loop so that it:

- prints out the triple of i, so 3 * i, at each run.
- is abandoned with a break if the triple of i is divisible by 8, but still prints out this triple before breaking.

```
# Initialize i as 1

i <- 1


# Code the while loop

while (i <= 10) {

  print(3*i)

  if (3*i %%8 == 0 ) {

  break

  }

  i <- i + 1

}
```

In the previous video, Filip told you about two different strategies for using the for loop. To refresh your memory, consider the following loops that are equivalent

in R:

```
primes <- c(2, 3, 5, 7, 11, 13)


# loop version 1
for (p in primes) {
  print(p)
}


# loop version 2
for (i in 1:length(primes)) {
  print(primes[i])
}
```

Remember our linkedin vector? It's a vector that contains the number of views your LinkedIn profile had in the last seven days. The linkedin vector has already been

defined in the editor on the right so that you can fully focus on the instructions!

Instructions

Write a for loop that iterates over all the elements of linkedin and prints out every element separately. Do this in two ways: using the loop version 1 and the loop

version 2 in the example code above.

```
# The linkedin vector has already been defined for you

linkedin <- c(16, 9, 13, 5, 2, 17, 14)


# Loop version 1


for (l in linkedin){

  print(l)


}



# Loop version 2

for (i in

1:length(linkedin)){

  print(linkedin[i])


}
```

Loop over a list

100xp

Looping over a list is just as easy and convenient as looping over a vector. There are again two different approaches here:

```
primes_list <- list(2, 3, 5, 7, 11, 13)


# loop version 1

for (p in primes_list) {

  print(p)

}


# loop version 2

for (i in 1:length(primes_list)) {

  print(primes_list[[i]])

}
```

Notice that you need double square brackets - [[ ]] - to select the list elements in loop version 2.

Suppose you have a list of all sorts of information on New York City: its population size, the names of the boroughs, and whether it is the capital of the United

States. We've already prepared a list nyc with all this information in the editor (source: Wikipedia).

Instructions

As in the previous exercise, loop over the nyc list in two different ways to print its elements:

•Loop directly over the nyc list (loop version 1).

•Define a looping index and do subsetting using double brackets (loop version 2).

```
# The nyc list is already specified
nyc <- list(pop = 8405837,
            boroughs = c("Manhattan", "Bronx", "Brooklyn", "Queens", "Staten Island"),


capital = FALSE)

# Loop version 1
for (info in nyc) {
  print(info)
}

# Loop version 2
for (i in 1:length(nyc)) {
  print(nyc[[i]])
}
```

Loop over a matrix

70xp

In your workspace, there's a matrix ttt, that represents the status of a tic-tac-toe game. It contains the values "X", "O" and "NA". Print out ttt in the console so

you can have a closer look. On row 1 and column 1, there's "O", while on row 3 and column 2 there's "NA".

To solve this exercise, you'll need a for loop inside a for loop, often called a nested loop. Doing this in R is a breeze! Simply use the following recipe:

```
for (var1 in seq1) {
  for (var2 in seq2) {
    expr
  }
}
```

Instructions

Finish the nested for loops to go over the elements in ttt:

- The outer loop should loop over the rows, with loop index i (use 1:nrow(ttt)).
- The inner loop should loop over the columns, with loop index j (use 1:ncol(ttt)).
- Inside the inner loop, make use of print() and paste() to print out information in the following format: "On row i and column j the board contains x", where x is the

value on that position.

```
# The tic-tac-toe matrix ttt has already been defined for you

# define the double for loop
for (i in 1:nrow(ttt)) {
  for (j in 1:ncol(ttt)) {
    print(paste("On row", i,

"and column", j ,"the board contains ", ttt[i,j]))
  }
}
```

Mix it up with control flow

100xp

Let's return to the LinkedIn profile views data, stored in a vector linkedin. In the first exercise on for loops you already did a simple printout of each element in

this vector. A little more in-depth interpretation of this data wouldn't hurt, right? Time to throw in some conditionals! As with the while loop, you can use the if

and else statements inside the for loop.

Instructions

Add code to the for loop that loops over the elements of the linkedin vector:

•If the vector element's value exceeds 10, print out "You're popular!".

•If the vector element's value does not exceed 10, print out "Be more visible!"

```
# The linkedin vector has already been defined for you
linkedin <- c(16, 9, 13, 5, 2, 17, 14)

# Code the for loop with conditionals
for (li in linkedin) {
  if ( li > 10)

{
    print("You're popular!")
  } else {
    print("Be more visible!")
  }
  print(li)
}
```

Next, you break it

0xp

In the editor on the right you'll find a possible solution to the previous exercise. The code loops over the linkedin vector and prints out different messages

depending on the values of li.

In this exercise, you will use the break and next statements:

•The break statement abandons the active loop: the remaining code in the loop is skipped and the loop is not iterated over anymore.

•The next statement skips the remainder of the code in the loop, but continues the iteration.

Instructions

Extend the for loop with two new, separate if tests in the editor as follows:

•If the vector element's value exceeds 16, print out "This is ridiculous, I'm outta here!" and have R abandon the for loop (break).

•If the value is lower than 5, print out "This is too embarrassing!" and fast-forward to the next iteration (next).

```
# The linkedin vector has already been defined for you

linkedin <- c(16, 9, 13, 5, 2, 17, 14)
```

```
# Adapt/extend the for loop
for (li in linkedin) {
  if (li > 10) {
    print

("You're popular!")
  } else {
    print("Be more visible!")
  }

  # Add if statement with break
  if (li > 16) {
    print("This is ridiculous, I'm outta here!")
    break


 }

  # Add if statement with next
  if (li < 5) {
    print("This is too embarrassing!")
    next
  }

  print(li)
}
```

Build a for loop from scratch

100xp

This exercise will not introduce any new concepts on for loops.

In the editor on the right, we already went ahead and defined a variable rquote. This variable has been split up into a vector that contains separate letters and has

been stored in a vector chars with the strsplit() function.

Can you write code that counts the number of r's that come before the first u in rquote?

Instructions

•Initialize the variable rcount, as 0.

•Finish the for loop: ◦if char equals "r", increase the value of rcount by 1.

◦if char equals "u", leave the for loop entirely with a break.

•Finally, print out the variable rcount to the console to see if your code is correct.

```
# Pre-defined variables

rquote <- "r's internals are irrefutably intriguing"

chars <- strsplit(rquote, split = "")[[1]]
```

```
# Initialize rcount

rcount <- 0


# Finish the for


loop

for (char in chars) {

  if (char == "r")

  { rcount = rcount + 1}

  else if

  ( char == "u") {

  break }

}

 rcount

# Print out rcount
```

Function documentation

100xp

Before even thinking of using an R function, you should clarify which arguments it expects. All the relevant details such as a description, usage, and arguments can be found in the documentation. To consult the documentation on the sample() function, for example, you can use one of following R commands:

help(sample)

?sample

If you execute these commands in the console of the DataCamp interface, you'll be redirected to www.rdocumentation.org.

A quick hack to see the arguments of the sample() function is the args() function. Try it out in the console:

args(sample)

In the next exercises, you'll be learning how to use the mean() function with increasing complexity. The first thing you'll have to do is get acquainted with the mean

() function.

Instructions

•Consult the documentation on the mean() function: ?mean or help(mean).

•Inspect the arguments of the mean() function using the args() function.

Function documentation

100xp

Before even thinking of using an R function, you should clarify which arguments it expects. All the relevant details such as a description, usage, and arguments can be

found in the documentation. To consult the documentation on the sample() function, for example, you can use one of following R commands:

```
help(sample)
```

```
?sample
```

If you execute these commands in the console of the DataCamp interface, you'll be redirected to www.rdocumentation.org.

A quick hack to see the arguments of the sample() function is the args() function. Try it out in the console:

```
args(sample)
```

In the next exercises, you'll be learning how to use the mean() function with increasing complexity. The first thing you'll have to do is get acquainted with the mean

() function.

Instructions

•Consult the documentation on the mean() function: ?mean or help(mean).

•Inspect the arguments of the mean() function using the args() function.

Use a function

100xp

The documentation on the mean() function gives us quite some information:

•The mean() function computes the arithmetic mean.

•The most general method takes multiple arguments: x and ....

•The x argument should be a vector containing numeric, logical or time-related information.

Remember that R can match arguments both by position and by name. Can you still remember the difference? You'll find out in this exercise!

Once more, you'll be working with the view counts of your social network profiles for the past 7 days. These are stored in the linkedin and facebook vectors and have

already been defined in the editor on the right.

Instructions

•Calculate the average number of views for both linkedin and facebook and assign the result to avg_li and avg_fb, respectively. Experiment with different types of

argument matching!

•Print out both avg_li and avg_fb.

```
# The linkedin and facebook vectors have already been created for you

linkedin <- c(16, 9, 13, 5, 2, 17, 14)

facebook <- c(17, 7, 5, 16, 8, 13, 14)


# Calculate average number of views


avg_li <- mean(linkedin)

avg_fb <- mean(facebook)


# Inspect avg_li and avg_fb

avg_li

avg_fb
```

Use a function (2)

100xp

Check the documentation on the `mean()` function again:

```
?mean
```

The Usage section of the documentation includes two versions of the `mean()` function. The first usage,

```
mean(x, ...)
```

is the most general usage of the mean function. The 'Default S3 method', however, is:

```
mean(x, trim = 0, na.rm = FALSE, ...)
```

The `...` is called the ellipsis. It is a way for R to pass arguments along without the function having to name them explicitly. The ellipsis will be treated in more detail in future courses.
For the remainder of this exercise, just work with the second usage of the mean function. Notice that both `trim` and `na.rm` have default values. This makes them **optional arguments**.

Instructions

- Calculate the mean of the element-wise sum of `linkedin` and `facebook` and store the result in a variable `avg_sum`.
- Calculate the mean once more, but this time set the `trim` argument equal to 0.2 and assign the result to `avg_sum_trimmed`.
- Print out both `avg_sum` and `avg_sum_trimmed`; can you spot the difference?

```
# The linkedin and facebook vectors have already been created for you

linkedin <- c(16, 9, 13, 5, 2, 17, 14)

facebook <- c(17, 7, 5, 16, 8, 13, 14)



# Calculate the mean of the sum



avg_sum <-mean(linkedin+facebook)

# Calculate the trimmed mean of the sum

avg_sum_trimmed <- mean(linkedin+facebook,trim = 0.2 )



# Inspect both new variables

avg_sum

avg_sum_trimmed
```

Use a function (3)

100xp

In the video, Filip guided you through the example of specifying arguments of the `sd()` function. The `sd()` function has an optional argument, `na.rm` that specified whether or not to remove missing values from the input vector before calculating the standard deviation.
If you've had a good look at the documentation, you'll know by now that the `mean()` function also has this argument, `na.rm`, and it does the exact same thing. By default, it is set to `FALSE`, as the Usage of the default S3 method shows:

```
mean(x, trim = 0, na.rm = FALSE, ...)
```

Let's see what happens if your vectors `linkedin` and `facebook` contain missing values (`NA`).

Instructions

- Calculate the average number of LinkedIn profile views, without specifying any optional arguments. Simply print the result to the console.
- Calculate the average number of LinkedIn profile views, but this time tell R to strip missing values from the input vector.

# The linkedin and facebook vectors have already been created for you

linkedin <- c(16, 9, 13, 5, NA, 17, 14)

facebook <- c(17, NA, 5, 16, 8, 13, 14)

# Basic average of linkedin

mean(linkedin)

# Advanced average of linkedin

mean(linkedin,na.rm = TRUE)

Functions inside functions

100xp

You already know that R functions return objects that you can then use somewhere else. This makes it easy to use functions inside functions, as you've seen before:

```
speed <- 31
print(paste("Your speed is", speed))
```

Notice that both the `print()` and `paste()` functions use the ellipsis - `...` - as an argument. Can you figure out how they're used?

Instructions

Use `abs()` on `linkedin - facebook` to get the absolute differences between the daily Linkedin and Facebook profile views. Next, use this function call inside `mean()` to calculate the Mean Absolute Deviation. In the `mean()` call, make sure to specify `na.rm` to treat missing values correctly!

# The linkedin and facebook vectors have already been created for you

linkedin <- c(16, 9, 13, 5, NA, 17, 14)

facebook <- c(17, NA, 5, 16, 8, 13, 14)

# Calculate the mean absolute deviation

abs(linkedin-facebook)

mean(abs(linkedin-facebook),na.rm =TRUE)

Write your own function

100xp

Wow, things are getting serious... you're about to write your own function! Before you have a go at it, have a look at the following function template:

```
my_fun <- function(arg1, arg2) {
  body
}
```

Notice that this recipe uses the assignment operator (`<-`) just as if you were assigning a vector to a variable for example. This is not a coincidence. Creating a function in R basically is the assignment of a function object to a variable! In the recipe above, you're creating a new R variable `my_fun`, that becomes available in the workspace as soon as you execute the definition. From then on, you can use the `my_fun` as a function.

Instructions

- Create a function `pow_two()`: it takes one argument and returns that number squared (that number times itself).
- Call this newly defined function with `12` as input.
- Next, create a function `sum_abs()`, that takes two arguments and returns the sum of the absolute values of both arguments.
- Finally, call the function `sum_abs()` with arguments `-2` and `3` afterwards.

# Create a function pow_two()

pow_two <- function(a){

  a^2

}

# Create a function pow_two()

```
pow_two <- function(a){
  a^2
}



# Use the function
pow_two(12)

# Create a function sum_abs()
sum_abs <- function(a,b){
  abs(a)+abs(b)
}



# Use the function
sum_abs(-2,3)
```

Write your own function (2)

100xp

There are situations in which your function does not require an input. Let's say you want to write a function that gives us the random outcome of throwing a fair die:

```
throw_die <- function() {
  number <- sample(1:6, size = 1)
  number
}

throw_die()
```

Up to you to code a function that doesn't take any arguments!

Instructions

- Define a function, `hello()`. It prints out "Hi there!" and returns `TRUE`. It has no arguments.

- Call the function `hello()`, without specifying arguments of course.

```
# Define the function hello()

hello <- function(){

print("Hi there!")

  return(TRUE)

}




# Call the function hello()

hello()
```

Write your own function (3)

100xp

Do you still remember the difference between an argument with and without default values? Have another look at the `sd()` function by typing `?sd` in the console. The usage section shows the following information:

```
sd(x, na.rm = FALSE)
```

This tells us that `x` has to be defined for the `sd()` function to be called correctly, however, `na.rm` already has a default value. Not specifying this argument won't cause an error.

You can define default argument values in your own R functions as well. You can use the following recipe to do so:

```
my_fun <- function(arg1, arg2 = val2) {
  body
}
```

The editor on the right already includes an extended version of the `pow_two()` function from before. Can you finish it?

Instructions

- Add an optional argument, named `print_info`, that is `TRUE` by default.
- Wrap an `if` construct around the `print()` function: this function should only be executed if `print_info` is `TRUE`.
- Feel free to experiment with the `pow_two()` function you've just coded.

Finish the pow_two() function

pow_two <- function(x,print_info = TRUE) {

 y <- x ^ 2

 if(print_info == TRUE){

 print(paste(x, "to the power two equals", y))}

 return(y)

}

R passes arguments by value

50xp

The title gives it away already: R passes arguments by value. What does this mean? Simply put, it means that an R function cannot change the variable that you input to that function. Let's look at a simple example (try it in the console):

```
triple <- function(x) {
  x <- 3*x
  x
}
a <- 5
triple(a)
a
```

Inside the `triple()` function, the argument `x` gets overwritten with its value times three. Afterwards this new `x` is returned. If you call this function with a variable `a` set equal to 5, you obtain 15. But did the value of `a` change? If R were to pass `a` to `triple()` *by reference*, the override of the `x` *inside* the function would ripple through to the variable `a`, outside the function. However, R passes *by value*, so the R objects you pass to a function can never change unless you do an explicit assignment. `a` remains equal to 5, even after calling `triple(a)`.

Can you tell which one of the following statements is false about the following piece of code?

```
increment <- function(x, inc = 1) {
  x <- x + inc
  x
}
count <- 5
a <- increment(count, 2)
b <- increment(count)
count <- increment(count, 2)
```

Possible Answers

Click or Press Ctrl+1 to focus

-

`a` and `b` equal 7 and 6 respectively after executing this code block.

1

- ○

After the first call of `increment()`, where `a` is defined, `a` equals 7 and `count` equals 5.

2

- ○

In the end, `count` will equal 10.

3

- ○

In the last expression, the value of `count` was actually changed because of the explicit assignment.

R you functional?

100xp

Now that you've acquired some skills in defining functions with different types of arguments and return values, you should try to create more advanced functions. As you've noticed in the previous exercises, it's perfectly possible to add control-flow constructs, loops and even other functions to your function body.

Remember our social media example? The vectors `linkedin` and `facebook` are already defined in the workspace so you can get your hands dirty straight away. As a first step, you will be writing a function that can interpret a single value of this vector. In the next exercise, you will write another function that can handle an entire vector at once.

- Finish the function definition for `interpret()`, that interprets the number of profile views on a single day:
  - The function takes one argument, `num_views`.
  - If `num_views` is greater than 15, the function prints out "You're popular!" to the console and returns `num_views`.
  - Else, the function prints out "Try to be more visible!" and returns 0.
- Finally, call the `interpret()` function twice: on the first value of the `linkedin` vector and on the second element of the `facebook` vector.

you functional?

100xp

Now that you've acquired some skills in defining functions with different types of arguments and return values, you should try to create more advanced functions. As you've noticed in the previous exercises, it's perfectly possible to add control-flow constructs, loops and even other functions to your function body.

Remember our social media example? The vectors `linkedin` and `facebook` are already defined in the workspace so you can get your hands dirty straight away. As a first step, you will be writing a function that can interpret a single value of this vector. In the next exercise, you will write another function that can handle an entire vector at once.

Instructions

- Finish the function definition for `interpret()`, that interprets the number of profile views on a single day:
  - The function takes one argument, `num_views`.
  - If `num_views` is greater than 15, the function prints out "You're popular!" to the console and returns `num_views`.
  - Else, the function prints out "Try to be more visible!" and returns 0.
- Finally, call the `interpret()` function twice: on the first value of the `linkedin` vector and on the second element of the `facebook` vector

```
# The linkedin and facebook vectors have already been created for you


# Define the interpret function

interpret <- function(num_views) {

  if (num_views > 15) {

print("You're popular!")

return(num_views)


  } else {


print("Try to be more visible!")

return(0)

  }

}


# Call the interpret function twice

interpret(linkedin[1])
```

interpret(facebook[2])

R you functional? (2)

100xp

A possible implementation of the `interpret()` function is already available in the editor. In this exercise you'll be writing another function that will use the `interpret()` function to interpret *all* the data from your daily profile views inside a vector. Furthermore, your function will return the sum of views on popular days, if asked for. A `for` loop is ideal for iterating over all the vector elements. The ability to return the sum of views on popular days is something you can code through a function argument with a default value.

Instructions

Finish the template for the `interpret_all()` function:

- Make `return_sum` an optional argument, that is `TRUE` by default.
- Inside the `for` loop, iterate over all `views`: on every iteration, add the result of `interpret(v)` to `count`. Remember that `interpret(v)` returns `v` for popular days, and `0` otherwise. At the same time, `interpret(v)` will also do some printouts.
- Finish the `if` construct:
  - If `return_sum` is `TRUE`, return `count`.
  - Else, return `NULL`.

Call this newly defined function on both `linkedin` and `facebook`.

R you functional? (2)

100xp

A possible implementation of the `interpret()` function is already available in the editor. In this exercise you'll be writing another function that will use the `interpret()` function to interpret *all* the data from your daily profile views inside a vector. Furthermore, your function will return the sum of views on popular days, if asked for. A `for` loop is ideal for iterating over all the vector elements. The ability to return the sum of views on popular days is something you can code through a function argument with a default value.

Instructions

Finish the template for the `interpret_all()` function:

- Make `return_sum` an optional argument, that is `TRUE` by default.
- Inside the `for` loop, iterate over all `views`: on every iteration, add the result of `interpret(v)` to `count`. Remember that `interpret(v)` returns `v` for popular days, and `0` otherwise. At the same time, `interpret(v)` will also do some printouts.
- Finish the `if` construct:
  - If `return_sum` is `TRUE`, return `count`.
  - Else, return `NULL`.

Call this newly defined function on both `linkedin` and `facebook`.

```
# The linkedin and facebook vectors have already been created for you

linkedin <- c(16, 9, 13, 5, 2, 17, 14)

facebook <- c(17, 7, 5, 16, 8, 13, 14)


# The interpret() can be used inside interpret_all()

interpret <- function(num_views) {

  if (num_views > 15) {

    print("You're popular!")

    return(num_views)

  } else {

    print("Try to be more visible!")

    return(0)

  }

}


# Define the interpret_all() function



# views: vector with data to interpret

# return_sum: return total number of views on popular days?

interpret_all <- function(views, return_sum =TRUE) {

  count <- 0


  for (v in views) {

count <-interpret(v) + count

  }


  if (return_sum) {
```

```
return(count)
  } else {
 return(NULL)
  }
}
```

```
# Call the interpret_all() function on both linkedin and facebook

interpret_all(linkedin)
interpret_all(facebook)
```

Load an R Package

100xp

There are basically two extremely important functions when it comes down to R packages:

- `install.packages()`, which as you can expect, installs a given package.
- `library()` which loads packages, i.e. attaches them to the search list on your R workspace.

To install packages, you need administrator privileges. This means that `install.packages()` will thus not work in the DataCamp interface. However, almost all CRAN packages are installed on our servers. You can load them with `library()`.
In this exercise, you'll be learning how to load the `ggplot2` package, a powerful package for data visualization. You'll use it to create a plot of two variables of the `mtcars` data frame. The data has already been prepared for you in the workspace.
Before starting, execute the following commands in the console:

- `search()`, to look at the currently attached packages and
- `qplot(mtcars$wt, mtcars$hp)`, to build a plot of two variables of the `mtcars` data frame.

An error should occur, because you haven't loaded the `ggplot2` package

- To fix the error you saw in the console, load the `ggplot2` package.
- Now, retry calling the `qplot()` function with the same arguments.
- Finally, check out the currently attached packages again.

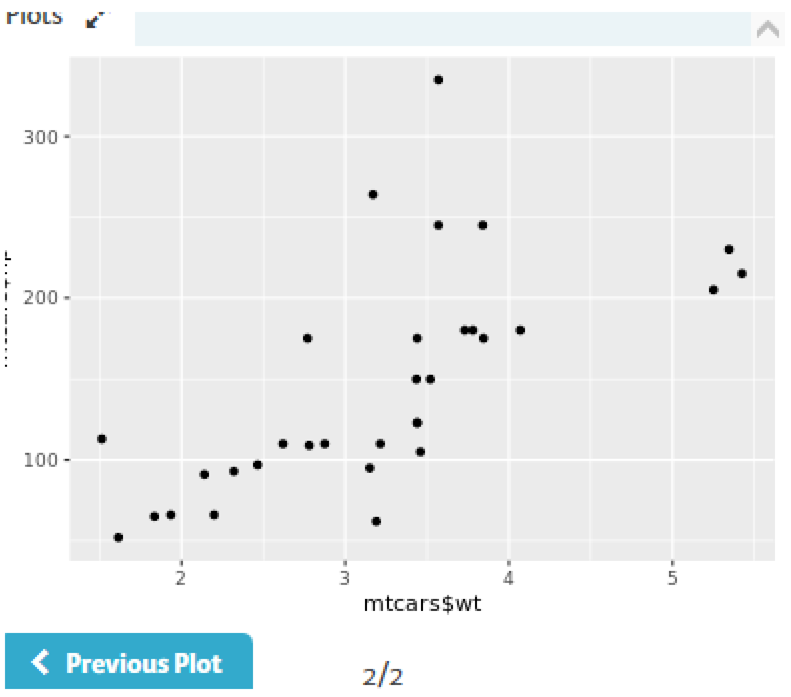
```
# Load the ggplot2 package
library(ggplot2)
```

# Retry the qplot() function

qplot(mtcars$wt, mtcars$hp)

# Check out the currently attached packages again

search()



Different ways to load a package

50xp

The `library()` and `require()` functions are not very picky when it comes down to argument types: both `library(rjson)` and `library("rjson")` work perfectly fine for loading a package.
Have a look at some more code chunks that (attempt to) load one or more packages:

```
# Chunk 1
library(data.table)
require(rjson)

# Chunk 2
```

```
library("data.table")
require(rjson)

# Chunk 3
library(data.table)
require(rjson, character.only = TRUE)

# Chunk 4
library(c("data.table", "rjson"))
```

Select the option that lists all of the chunks that do not generate an error. The console on the right is yours to experiment in.

Lapply:--apply the function to all elements of a list/vector

Before you go about solving the exercises below, have a look at the documentation of the `lapply()` function. The Usage section shows the following expression:

```
lapply(X, FUN, ...)
```

To put it generally, `lapply` takes a vector or list `X`, and applies the function `FUN` to each of its members. If `FUN` requires additional arguments, you pass them after you've specified `X` and `FUN` (`...`). The output of `lapply()` is a list, the same length as `X`, where each element is the result of applying `FUN` on the corresponding element of `X`.

Now that you are truly brushing up on your data science skills, let's revisit some of the most relevant figures in data science history. We've compiled a vector of famous mathematicians/statisticians and the year they were born. Up to you to extract some information!

Instructions

- Have a look at the `strsplit()` calls, that splits the strings in `pioneers` on the `:` sign. The result, `split_math` is a list of 4 character vectors: the first vector element represents the name, the second element the birth year.
- Use `lapply()` to convert the character vectors in `split_math` to lowercase letters: apply `tolower()` on each of the elements in `split_math`. Assign the result, which is a list, to a new variable `split_low`.
- Finally, inspect the contents of `split_low` with `str()`.

# The vector pioneers has already been created for you

pioneers <- c("GAUSS:1777", "BAYES:1702", "PASCAL:1623", "PEARSON:1857")

# Split names from birth year

split_math <- strsplit(pioneers, split = ":")

```
# Convert to lowercase strings: split_low


split_low <- lapply(split_math, tolower)

# Take a look at the structure of split_low

str(split_low)
```

Use lapply with your own function

100xp

As Filip explained in the instructional video, you can use `lapply()` on your own functions as well. You just need to code a new function and make sure it is available in the workspace. After that, you can use the function inside `lapply()` just as you did with base R functions.

In the previous exercise you already used `lapply()` once to convert the information about your favorite pioneering statisticians to a list of vectors composed of two character strings. Let's write some code to select the names and the birth years separately.

The sample code already includes code that defined `select_first()`, that takes a vector as input and returns the first element of this vector.

Instructions

- Apply `select_first()` over the elements of `split_low` with `lapply()` and assign the result to a new variable `names`.
- Next, write a function `select_second()` that does the exact same thing for the second element of an inputted vector.
- Finally, apply the `select_second()` function over `split_low` and assign the output to the variable years

```
# Code from previous exercise:

pioneers <- c("GAUSS:1777", "BAYES:1702", "PASCAL:1623", "PEARSON:1857")

split <- strsplit(pioneers, split = ":")

split_low <- lapply(split, tolower)



# Write function select_first()

select_first <- function(x) {

  x[1]
```

}

# Apply select_first() over split_low: names

names <-lapply(split_low , select_first)

# Write function select_second()

select_second <- function(x) {

  x[2]

}

# Apply select_second() over split_low: years

years <-lapply(split_low , select_second)

lapply and anonymous functions

100xp

Writing your own functions and then using them inside `lapply()` is quite an accomplishment! But defining functions to use them only once is kind of overkill, isn't it? That's why you can use so-called **anonymous functions** in R.

Previously, you learned that functions in R are objects in their own right. This means that they aren't automatically bound to a name. When you create a function, you can use the assignment operator to give the function a name. It's perfectly possible, however, to not give the function a name. This is called an anonymous function:

```
# Named function
triple <- function(x) { 3 * x }

# Anonymous function with same implementation
function(x) { 3 * x }
```

```
# Use anonymous function inside lapply()
lapply(list(1,2,3), function(x) { 3 * x })
```

Instructions

- Transform the first call of `lapply()` such that it uses an anonymous function that does the same thing.
- In a similar fashion, convert the second call of `lapply` to use an anonymous version of the `select_second()` function.
- Remove both the definitions of `select_first()` and `select_second()`, as they are no longer useful.

# Definition of split_low

pioneers <- c("GAUSS:1777", "BAYES:1702", "PASCAL:1623", "PEARSON:1857")

split <- strsplit(pioneers, split = ":")

split_low <- lapply(split, tolower)

# Transform: use anonymous function inside lapply

names <- lapply(split_low, function(x) {

  x[1]

})

# Transform: use anonymous function inside lapply

years <- lapply(split_low, function(x) {

  x[2]

})

Use lapply with additional arguments

100xp

In the video, the `triple()` function was transformed to the `multiply()` function to allow for a more generic approach. `lapply()` provides a way to handle functions that require more than one argument, such as the `multiply()` function:

```
multiply <- function(x, factor) {
  x * factor
}
lapply(list(1,2,3), multiply, factor = 3)
```

On the right we've included a generic version of the select functions that you've coded earlier: `select_el()`. It takes a vector as its first argument, and an index as its second argument. It returns the vector's element at the specified index.

Instructions

Use `lapply()` twice to call `select_el()` over all elements in `split_low`: once with the `index` equal to 1 and a second time with the index equal to 2. Assign the result to `names` and `years`, respectively.

# Definition of split_low

pioneers <- c("GAUSS:1777", "BAYES:1702", "PASCAL:1623", "PEARSON:1857")

split <- strsplit(pioneers, split = ":")

split_low <- lapply(split, tolower)

# Generic select function

select_el <- function(x, index) {

  x[index]

}

# Use lapply() twice on split_low: names and years

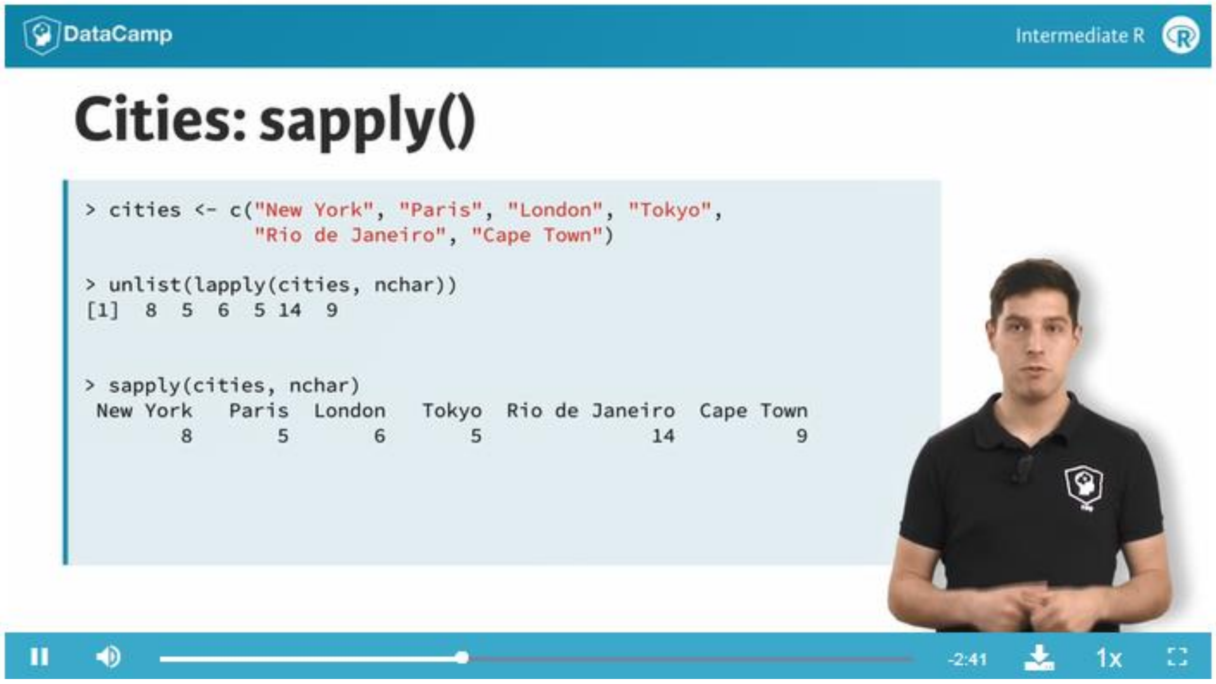
names <- lapply(split_low,select_el, index = 1)

years <-lapply(split_low,select_el,index =2)

Sapply :--gives matrix representation

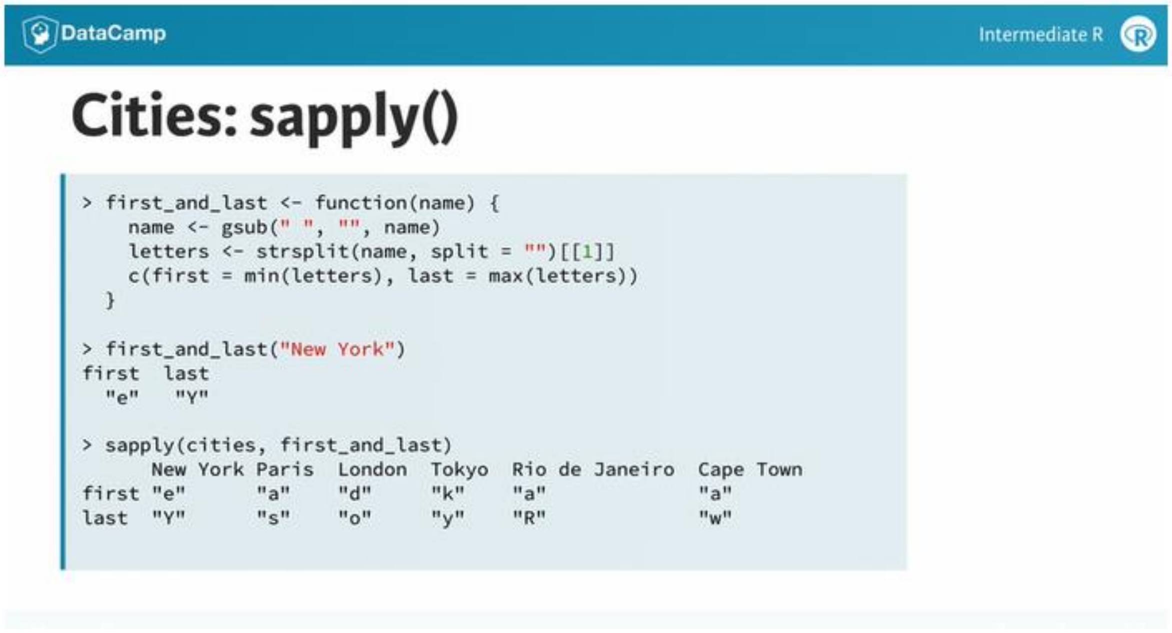
sapply

50xp



Cities: sapply()

```
> cities <- c("New York", "Paris", "London", "Tokyo",
              "Rio de Janeiro", "Cape Town")

> unlist(lapply(cities, nchar))
[1]  8  5  6  5 14  9

> sapply(cities, nchar)
 New York    Paris   London    Tokyo  Rio de Janeiro   Cape Town
        8        5        6        5              14           9
```



Cities: sapply()

```
> first_and_last <- function(name) {
    name <- gsub(" ", "", name)
    letters <- strsplit(name, split = "")[[1]]
    c(first = min(letters), last = max(letters))
  }

> first_and_last("New York")
first  last
  "e"   "Y"

> sapply(cities, first_and_last)
      New York Paris  London  Tokyo  Rio de Janeiro  Cape Town
first "e"      "a"    "d"     "k"    "a"             "a"
last  "Y"      "s"    "o"     "y"    "R"             "w"
```

How to use sapply

100xp

You can use `sapply()` similar to how you used `lapply()`. The first argument of `sapply()` is the list or vector `X` over which you want to apply a function, `FUN`. Potential additional arguments to this function are specified afterwards (`...`):

```
sapply(X, FUN, ...)
```

In the next couple of exercises, you'll be working with the variable `temp`, that contains temperature measurements for 7 days. `temp` is a list of length 7, where each element is a vector of length 5, representing 5 measurements on a given day. This variable has already been defined in the workspace: type `str(temp)` to see its structure.

Instructions

- Use `lapply()` to calculate the minimum (built-in function `min()`) of the temperature measurements for every day.
- Do the same thing but this time with `sapply()`. See how the output differs.
- Use `lapply()` to compute the the maximum (`max()`) temperature for each day.
- Again, use `sapply()` to solve the same question and see how `lapply()` and `sapply()` differ.

temp has already been defined in the workspace

# Use lapply() to find each day's minimum temperature

lapply(temp, min)

# Use sapply() to find each day's minimum temperature

sapply(temp,min)

# Use lapply() to find each day's maximum temperature

lapply(temp, max)

# Use sapply() to find each day's maximum temperature

sapply(temp, max)

sapply with your own function

100xp

Like `lapply()`, `sapply()` allows you to use self-defined functions and apply them over a vector or a list:

```
sapply(X, FUN, ...)
```

Here, `FUN` can be one of R's built-in functions, but it can also be a function you wrote. This self-written function can be defined before hand, or can be inserted directly as an anonymous function.

Instructions

- Finish the definition of `extremes_avg()`: it takes a vector of temperatures and calculates the average of the minimum and maximum temperatures of the vector.
- Next, use this function inside `sapply()` to apply it over the vectors inside `temp`.
- Use the same function over `temp` with `lapply()` and see how the outputs differ.

```
# temp is already defined in the workspace


# Finish function definition of extremes_avg

extremes_avg <- function(list1) {

  ( min(list1) + max(list1) )/ 2

}


# Apply extremes_avg() over temp using sapply()

sapply(temp,extremes_avg)


# Apply extremes_avg() over temp using lapply()

lapply(temp, extremes_avg)
```

sapply with function returning vector

100xp

In the previous exercises, you've seen how `sapply()` simplifies the list that `lapply()` would return by turning it into a vector. But what if the function you're applying over a list or a vector returns a vector of length greater than 1? If you don't remember from the video, don't waste more time in the valley of ignorance and head over to the instructions!

Instructions

- Finish the definition of the `extremes()` function. It takes a vector of numerical values and returns a vector containing the minimum and maximum values of a given vector, with the names "min" and "max", respectively.
- Apply this function over the vector `temp` using `sapply()`.
- Finally, apply this function over the vector `temp` using `lapply()` as well.

```
# temp is already available in the workspace



# Create a function that returns min and max of a vector: extremes

extremes <- function(x) {

  c(min = min(x), max = max(x))

}



# Apply extremes() over temp with sapply()

sapply(temp, extremes)



# Apply extremes() over temp with lapply()

lapply(temp, extremes)
```

sapply can't simplify, now what?

100xp

It seems like we've hit the jackpot with `sapply()`. On all of the examples so far, `sapply()` was able to nicely simplify the rather bulky output of `lapply()`. But, as with life, there are things you can't simplify. How does `sapply()` react?

We already created a function, `below_zero()`, that takes a vector of numerical values and returns a vector that only contains the values that are strictly below zero.

Instructions

- Apply `below_zero()` over `temp` using `sapply()` and store the result in `freezing_s`.
- Apply `below_zero()` over `temp` using `lapply()`. Save the resulting list in a variable `freezing_l`.
- Compare `freezing_s` to `freezing_l` using the `identical()` function.

```
# temp is already prepared for you in the workspace
# Definition of below_zero()

below_zero <- function(x) {

  return(x[x < 0])

}


# Apply below_zero over temp using sapply(): freezing_s

freezing_s <- sapply(temp,below_zero)


# Apply below_zero over temp using lapply(): freezing_l

freezing_l <- lapply(temp,below_zero)


# Are freezing_s and freezing_l identical?

identical(freezing_s,freezing_l)
```

sapply with functions that return NULL

100xp

You already have some apply tricks under your sleeve, but you're surely hungry for some more, aren't you? In this exercise, you'll see how `sapply()` reacts when it is used to apply a function that returns `NULL` over a vector or a list.

A function `print_info()`, that takes a vector and prints the average of this vector, has already been created for you. It uses the `cat()` function.

Instructions

- Apply `print_info()` over the contents of `temp` with `sapply()`.
- Repeat this process with `lapply()`. Do you notice the difference?

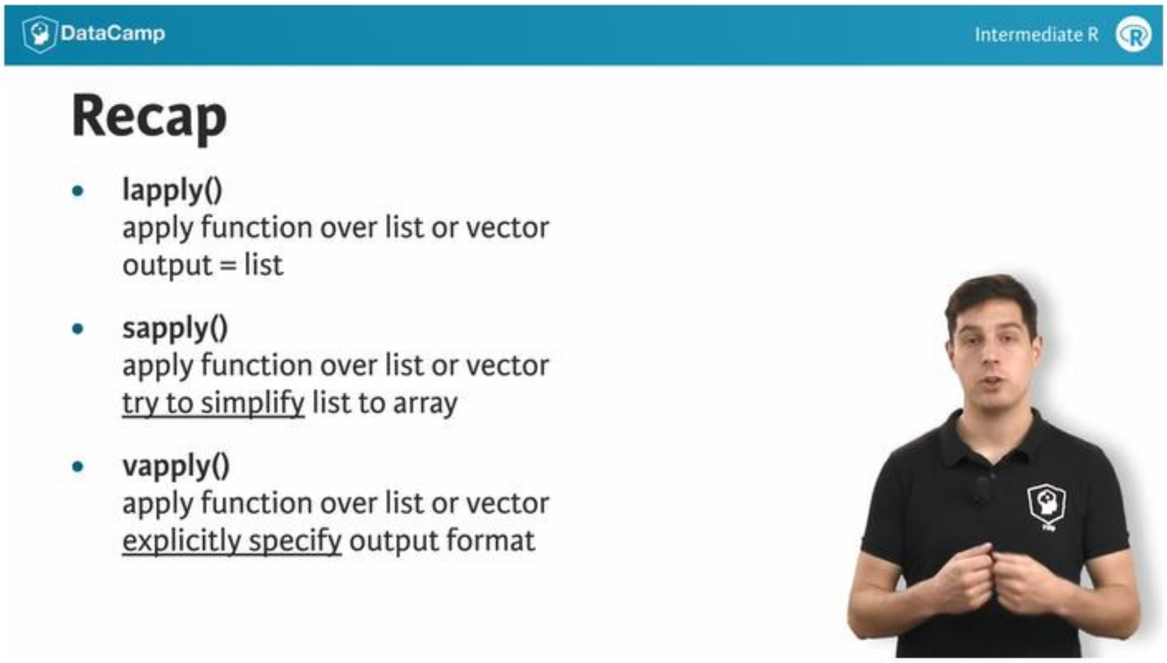
```
# temp is already available in the workspace


# Definition of print_info()

print_info <- function(x) {

  cat("The average temperature is", mean(x), "\n")
```

}

```
# Apply print_info() over temp using sapply()
sapply(temp, print_info)

# Apply print_info() over temp using lapply()
lapply(temp,print_info)
```

vapply

50xp

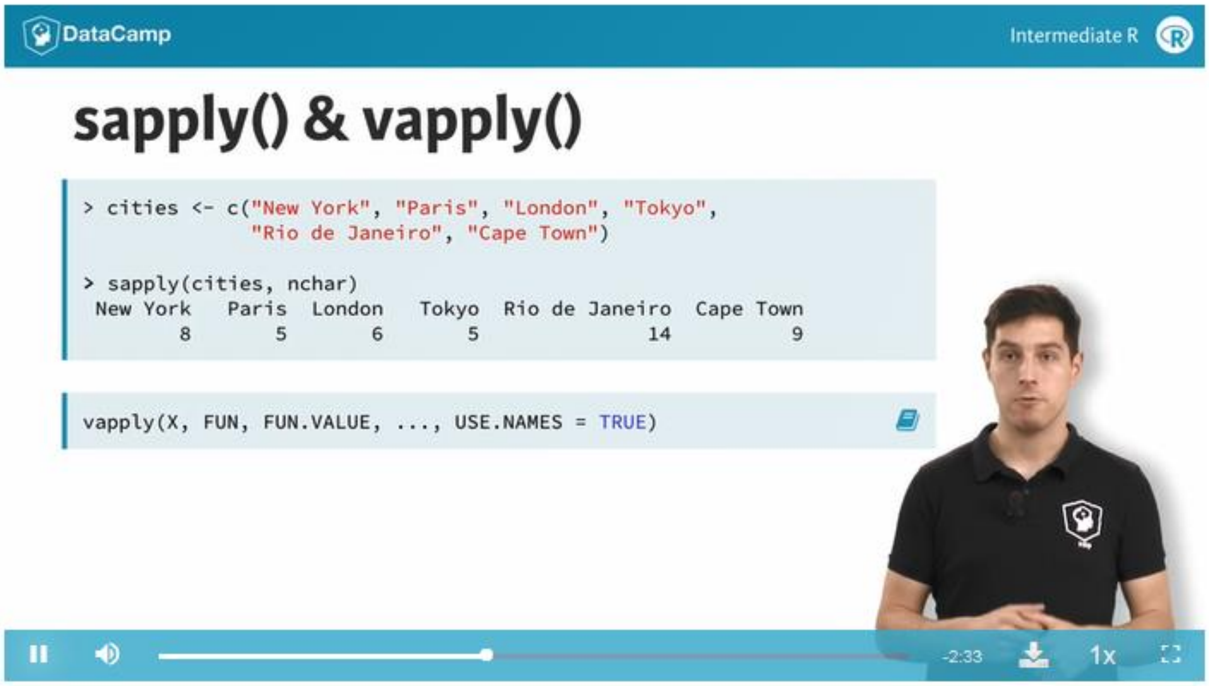


Recap

- **lapply()**
  apply function over list or vector
  output = list
- **sapply()**
  apply function over list or vector
  try to simplify list to array
- **vapply()**
  apply function over list or vector
  explicitly specify output format

vapply

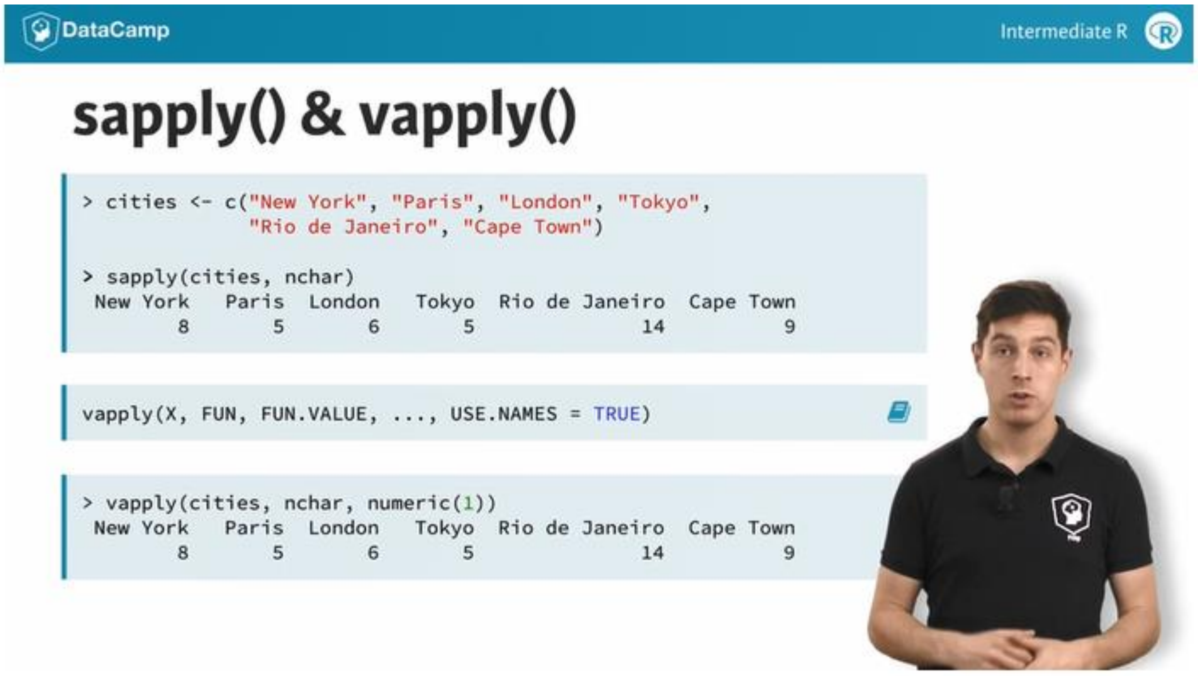
50xp



sapply() & vapply()

```
> cities <- c("New York", "Paris", "London", "Tokyo",
              "Rio de Janeiro", "Cape Town")

> sapply(cities, nchar)
 New York    Paris   London    Tokyo  Rio de Janeiro   Cape Town
        8        5        6        5              14           9
```

```
vapply(X, FUN, FUN.VALUE, ..., USE.NAMES = TRUE)
```



sapply() & vapply()

```
> cities <- c("New York", "Paris", "London", "Tokyo",
              "Rio de Janeiro", "Cape Town")

> sapply(cities, nchar)
 New York    Paris   London    Tokyo  Rio de Janeiro   Cape Town
        8        5        6        5              14           9
```

```
vapply(X, FUN, FUN.VALUE, ..., USE.NAMES = TRUE)
```

```
> vapply(cities, nchar, numeric(1))
 New York    Paris   London    Tokyo  Rio de Janeiro   Cape Town
        8        5        6        5              14           9
```

Use vapply

100xp

Before you get your hands dirty with the third and last apply function that you'll learn about in this intermediate R course, let's take a look at its syntax. The function is called `vapply()`, and it has the following syntax:

```
vapply(X, FUN, FUN.VALUE, ..., USE.NAMES = TRUE)
```

Over the elements inside `X`, the function `FUN` is applied. The `FUN.VALUE` argument expects a template for the return argument of this function `FUN`. `USE.NAMES` is `TRUE` by default; in this case `vapply()` tries to generate a named array, if possible.

For the next set of exercises, you'll be working on the `temp` list again, that contains 7 numerical vectors of length 5. We also coded a function `basics()` that takes a vector, and returns a named vector of length 3, containing the minimum, mean and maximum value of the vector respectively.

Instructions

- Apply the function `basics()` over the list of temperatures, `temp`, using `vapply()`. This time, you can use `numeric(3)` to specify the `FUN.VALUE` argument.

```
# temp is already available in the workspace


# Definition of basics()

basics <- function(x) {

  c(min = min(x), mean = mean(x), max = max(x))

}


# Apply basics() over temp using vapply()

vapply(temp,basics,numeric(3))
```

Use vapply (2)

100xp

So far you've seen that `vapply()` mimics the behavior of `sapply()` if everything goes according to plan. But what if it doesn't?

In the video, Filip showed you that there are cases where the structure of the output of the function you want to apply, `FUN`, does not correspond to the template you specify in `FUN.VALUE`. In that case, `vapply()` will throw an error that informs you about the misalignment between expected and actual output.

Instructions

- Inspect the code on the right and try to run it. If you haven't changed anything, an error should pop up. That's because `vapply()` still expects `basics()` to return a vector of length 3. The error message gives you an indication of what's wrong.
- Try to fix the error by editing the `vapply()` command.

```
# temp is already available in the workspace


# Definition of the basics() function

basics <- function(x) {

  c(min = min(x), mean = mean(x), median = median(x), max = max(x))

}


# Fix the error:

vapply(temp, basics, numeric(4))
```

From sapply to vapply

100xp

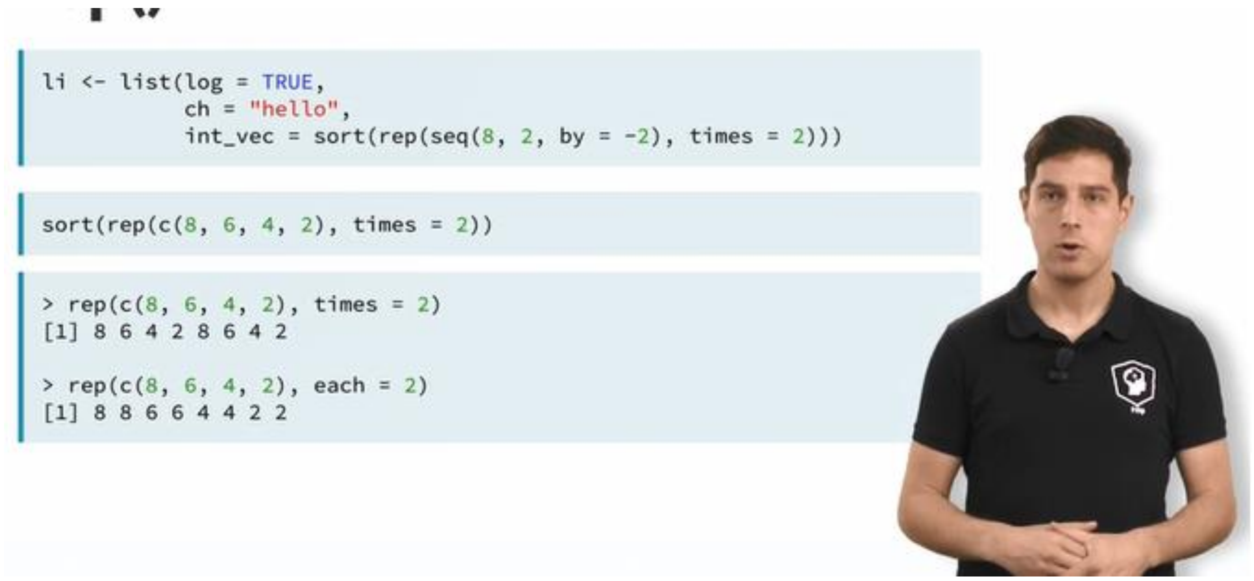
As highlighted before, `vapply()` can be considered a more robust version of `sapply()`, because you explicitly restrict the output of the function you want to apply. Converting your `sapply()` expressions in your own R scripts to `vapply()` expressions is therefore a good practice (and also a breeze!).

Instructions

Convert all the `sapply()` expressions on the right to their `vapply()` counterparts. Their results should be exactly the same; you're only adding robustness. You'll need the templates `numeric(1)` and `logical(1)`.

```
# temp is already defined in the workspace


# Convert to vapply() expression

vapply(temp, max,numeric(1))


# Convert to vapply() expression
```

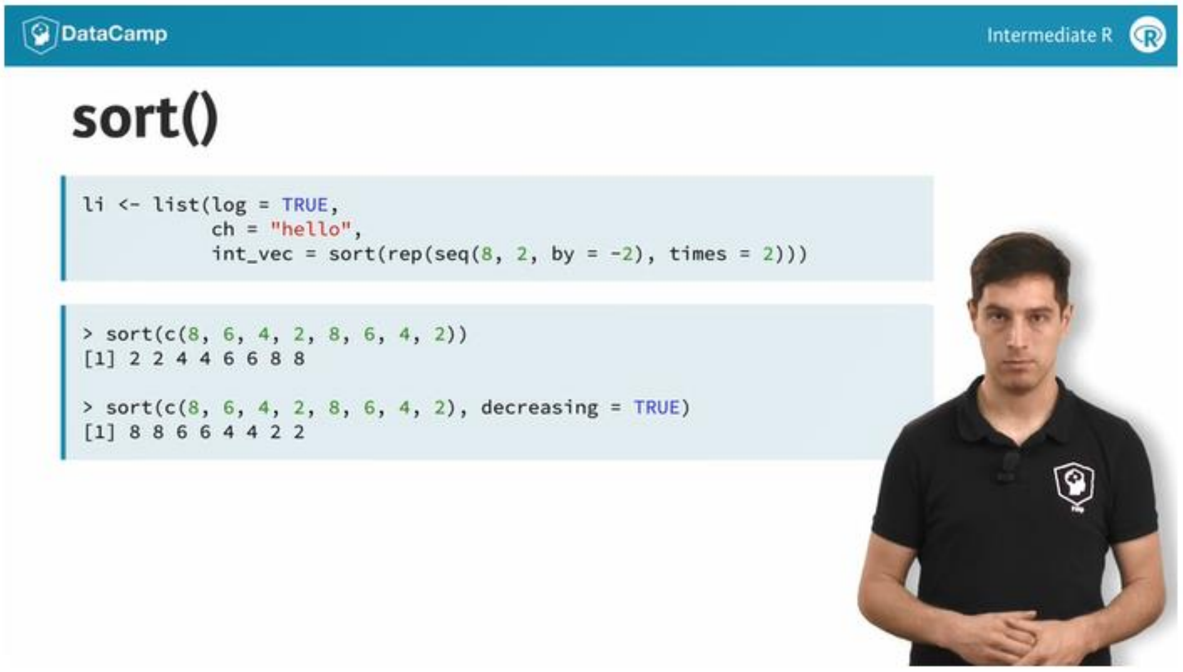
vapply(temp, function(x, y) { mean(x) > y }, y = 5,logical(1))



```
li <- list(log = TRUE,
           ch = "hello",
           int_vec = sort(rep(seq(8, 2, by = -2), times = 2)))
```

```
sort(rep(c(8, 6, 4, 2), times = 2))
```

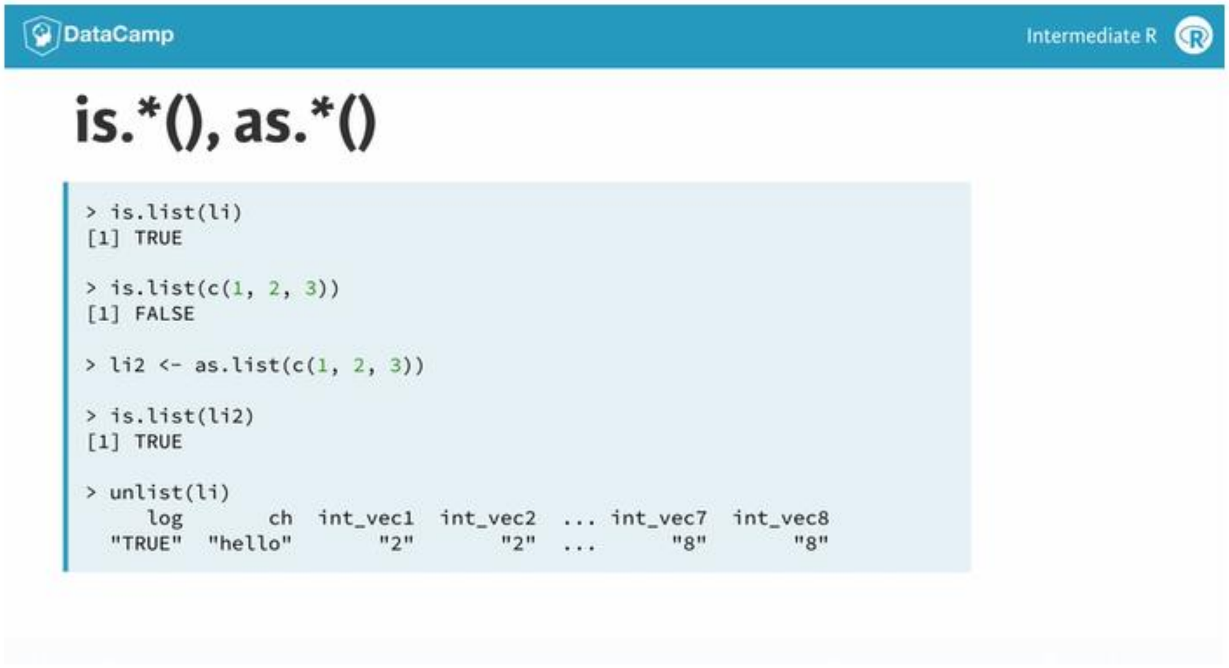
```
> rep(c(8, 6, 4, 2), times = 2)
[1] 8 6 4 2 8 6 4 2

> rep(c(8, 6, 4, 2), each = 2)
[1] 8 8 6 6 4 4 2 2
```



sort()

```
li <- list(log = TRUE,
           ch = "hello",
           int_vec = sort(rep(seq(8, 2, by = -2), times = 2)))
```

```
> sort(c(8, 6, 4, 2, 8, 6, 4, 2))
[1] 2 2 4 4 6 6 8 8

> sort(c(8, 6, 4, 2, 8, 6, 4, 2), decreasing = TRUE)
[1] 8 8 6 6 4 4 2 2
```

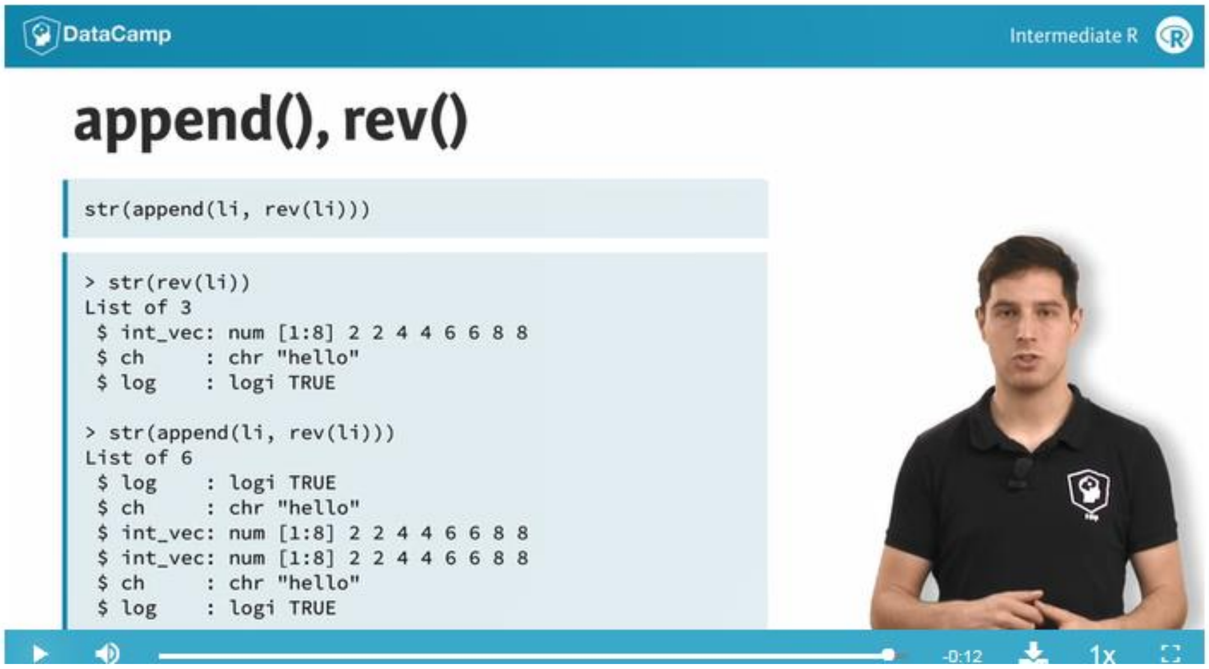

DataCamp Intermediate R

is.*(), as.*()

```
> is.list(li)
[1] TRUE

> is.list(c(1, 2, 3))
[1] FALSE

> li2 <- as.list(c(1, 2, 3))

> is.list(li2)
[1] TRUE

> unlist(li)
     log       ch  int_vec1  int_vec2  ... int_vec7  int_vec8
  "TRUE"  "hello"       "2"       "2"  ...      "8"       "8"
```

Useful Functions

50xp


DataCamp Intermediate R

append(), rev()

```
str(append(li, rev(li)))
```

```
> str(rev(li))
List of 3
 $ int_vec: num [1:8] 2 2 4 4 6 6 8 8
 $ ch     : chr "hello"
 $ log    : logi TRUE

> str(append(li, rev(li)))
List of 6
 $ log    : logi TRUE
 $ ch     : chr "hello"
 $ int_vec: num [1:8] 2 2 4 4 6 6 8 8
 $ int_vec: num [1:8] 2 2 4 4 6 6 8 8
 $ ch     : chr "hello"
 $ log    : logi TRUE
```

Mathematical utilities

100xp

Have another look at some useful math functions that R features:

- `abs()`: Calculate the absolute value.
- `sum()`: Calculate the sum of all the values in a data structure.
- `mean()`: Calculate the arithmetic mean.
- `round()`: Round the values to 0 decimal places by default. Try out `?round` in the console for variations of `round()` and ways to change the number of digits to round to.

As a data scientst in training, you've estimated a regression model on the sales data for the past six months. After evaluating your model, you see that the training error of your model is quite regular, showing both positive and negative values. The error values are already defined in the workspace on the right (`errors`).

Instructions

Calculate the sum of the absolute rounded values of the training errors. You can work in parts, or with a single one-liner. There's no need to store the result in a variable, just have R print it.

Ctrl+H

Take Hint (-30xp)

```
The errors vector has already been defined for you

errors <- c(1.9, -2.6, 4.0, -9.5, -3.4, 7.3)



# Sum of absolute rounded values of errors

sum(round(abs(errors)))
```

Find the error

100xp

We went ahead and included some code on the right, but there's still an error. Can you trace it and fix it?

In times of despair, help with functions such as `sum()` and `rev()` are a single command away; simply use `?sum` and `?rev` in the console.

Instructions

Fix the error by *including* code on the last line. Remember: you want to call `mean()` only once!

```
# Don't edit these two lines
```

vec1 <- c(1.5, 2.5, 8.4, 3.7, 6.3)

vec2 <- rev(vec1)

# Fix the error

mean(c(abs(vec1), abs(vec2)))

Data Utilities

100xp

R features a bunch of functions to juggle around with data structures::

- `seq()`: Generate sequences, by specifying the `from`, `to`, and `by` arguments.
- `rep()`: Replicate elements of vectors and lists.
- `sort()`: Sort a vector in ascending order. Works on numerics, but also on character strings and logicals.
- `rev()`: Reverse the elements in a data structures for which reversal is defined.
- `str()`: Display the structure of any R object.
- `append()`: Merge vectors or lists.
- `is.*()`: Check for the class of an R object.
- `as.*()`: Convert an R object from one class to another.
- `unlist()`: Flatten (possibly embedded) lists to produce a vector.

Remember the social media profile views data? Your LinkedIn and Facebook view counts for the last seven days are already defined as lists on the right.

Instructions

- Convert both `linkedin` and `facebook` lists to a vector, and store them as `li_vec` and `fb_vec` respectively.
- Next, append `fb_vec` to the `li_vec` (Facebook data comes last). Save the result as `social_vec`.
- Finally, sort `social_vec` *from high to low*. Print the resulting vector.

# The linkedin and facebook lists have already been created for you

linkedin <- list(16, 9, 13, 5, 2, 17, 14)

facebook <- list(17, 7, 5, 16, 8, 13, 14)

# Convert linkedin and facebook to a vector: li_vec and fb_vec

li_vec <- unlist(linkedin)

```
fb_vec <- unlist(facebook)


# Append fb_vec to li_vec: social_vec

social_vec <- append(li_vec, fb_vec)


# Sort social_vec

sort(social_vec, decreasing = TRUE)
```

Find the error (2)

100xp

Just as before, let's switch roles. It's up to you to see what unforgivable mistakes we've made. Go fix them!

Instructions

Correct the expression. Make sure that your fix still uses the functions `rep()` and `seq()`.

```
# Fix me

rep(seq(1, 7, by = 2), times = 7)
```

Beat Gauss using R

200xp

There is a popular story about young Gauss. As a pupil, he had a lazy teacher who wanted to keep the classroom busy by having them add up the numbers 1 to 100. Gauss came up with an answer almost instantaneously, 5050. On the spot, he had developed a formula for calculating the sum of an arithmetic series. There are more general formulas for calculating the sum of an arithmetic series with different starting values and increments. Instead of deriving such a formula, why not use R to calculate the sum of a sequence?

Instructions

- Using the function `seq()`, create a sequence that ranges from 1 to 500 in increments of 3. Assign the resulting vector to a variable `seq1`.
- Again with the function `seq()`, create a sequence that ranges from 1200 to 900 in increments of -7. Assign it to a variable `seq2`.

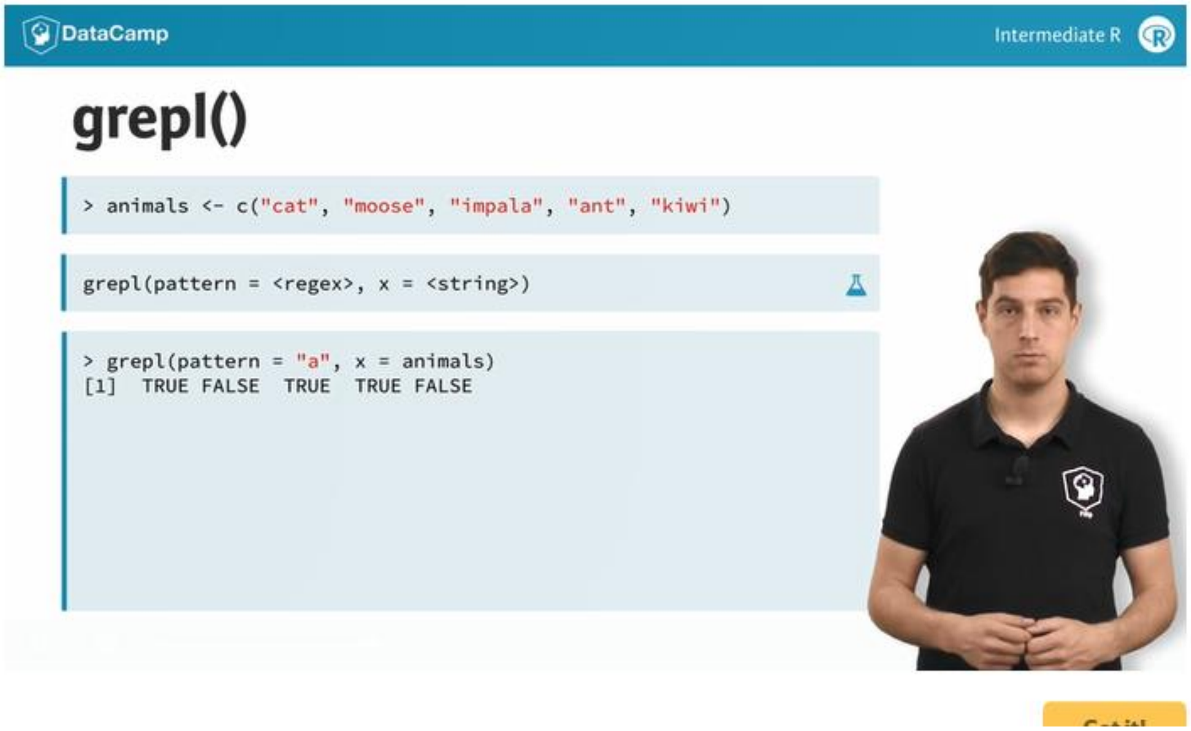
- Calculate the total sum of the sequences, either by using the `sum()` function twice and adding the two results, or by first concatenating the sequences and then using the `sum()` function once. Print the result to the console.

```
# Create first sequence: seq1

seq1 <- seq(1,500, by =3)


# Create second sequence: seq2


seq2 <-seq(1200,900, by =-7)

# Calculate total sum of the sequences

sum(seq1) + sum(seq2)
```

Regular expressions:-

Regular Expressions

50xp

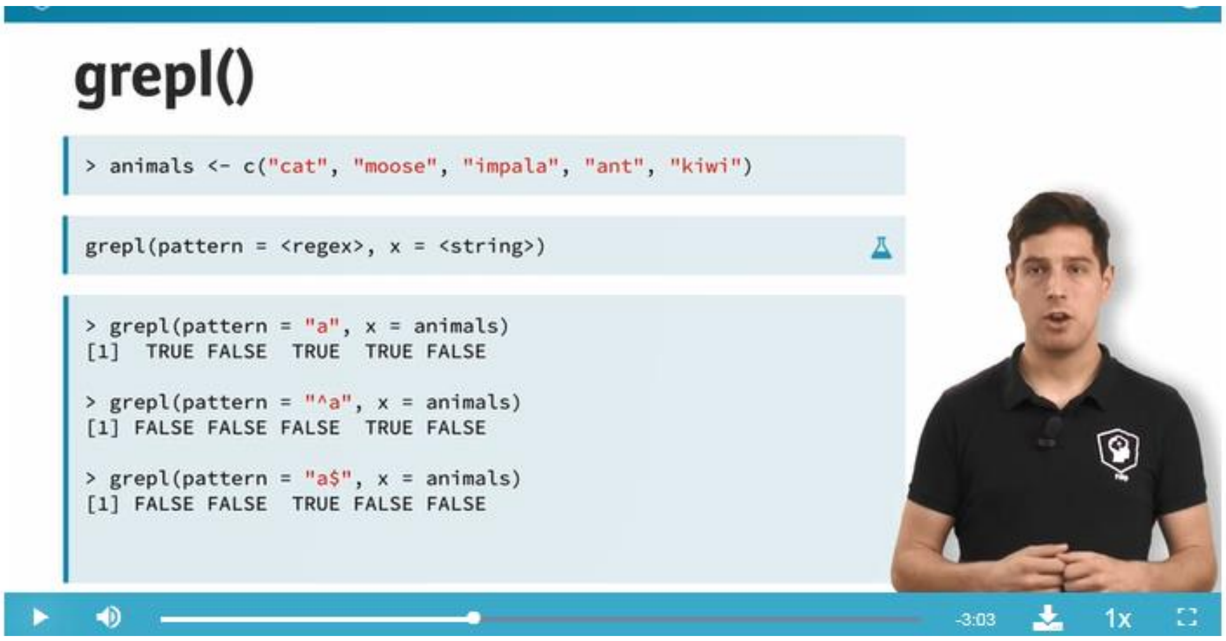


grepl()

```
> animals <- c("cat", "moose", "impala", "ant", "kiwi")
```

```
grepl(pattern = <regex>, x = <string>)
```

```
> grepl(pattern = "a", x = animals)
[1]  TRUE FALSE  TRUE  TRUE FALSE
```

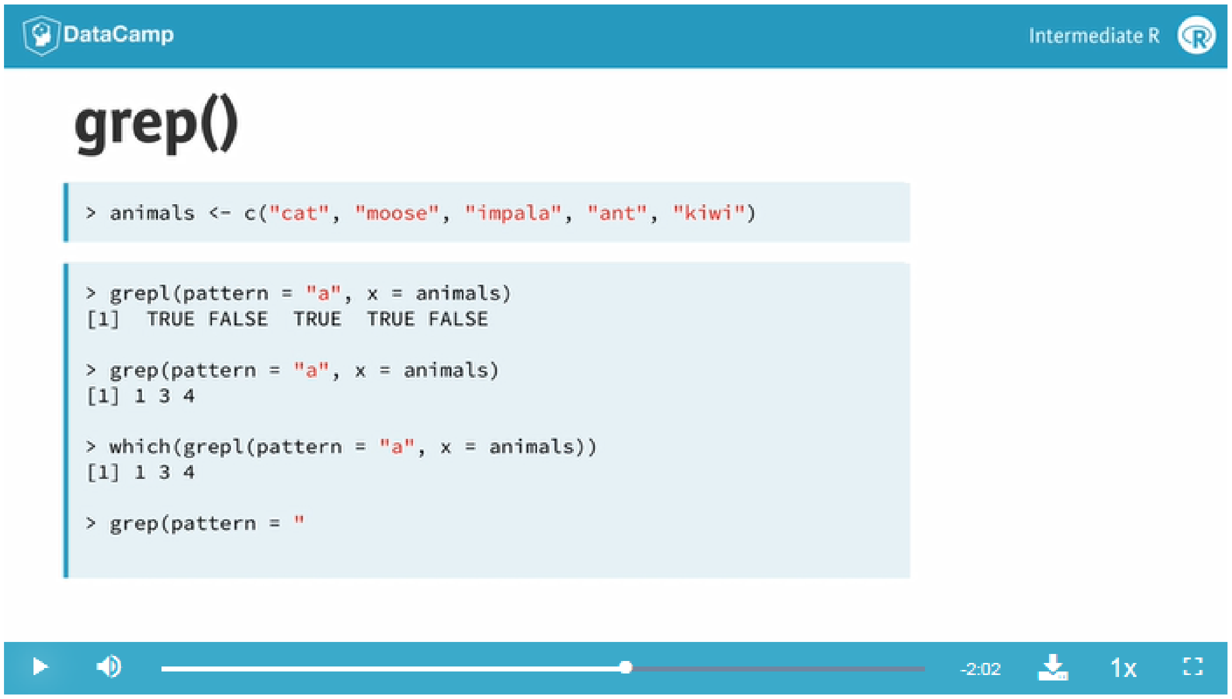


grepl()

```
> animals <- c("cat", "moose", "impala", "ant", "kiwi")
```

```
grepl(pattern = <regex>, x = <string>)
```

```
> grepl(pattern = "a", x = animals)
[1]  TRUE FALSE  TRUE  TRUE FALSE

> grepl(pattern = "^a", x = animals)
[1] FALSE FALSE FALSE  TRUE FALSE

> grepl(pattern = "a$", x = animals)
[1] FALSE FALSE  TRUE FALSE FALSE
```

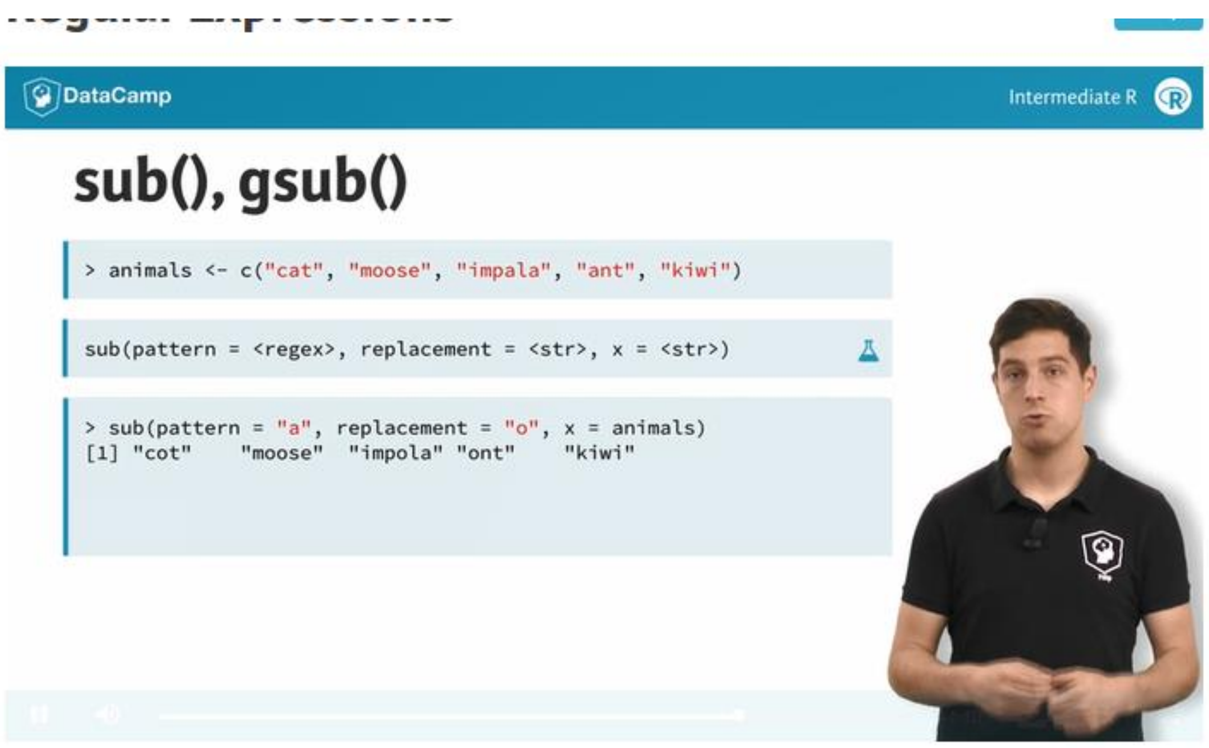


DataCamp Intermediate R

grep()

```
> animals <- c("cat", "moose", "impala", "ant", "kiwi")
```

```
> grepl(pattern = "a", x = animals)
[1]  TRUE FALSE  TRUE  TRUE FALSE

> grep(pattern = "a", x = animals)
[1] 1 3 4

> which(grepl(pattern = "a", x = animals))
[1] 1 3 4

> grep(pattern = "
```



DataCamp Intermediate R

sub(), gsub()

```
> animals <- c("cat", "moose", "impala", "ant", "kiwi")
```

```
sub(pattern = <regex>, replacement = <str>, x = <str>)
```

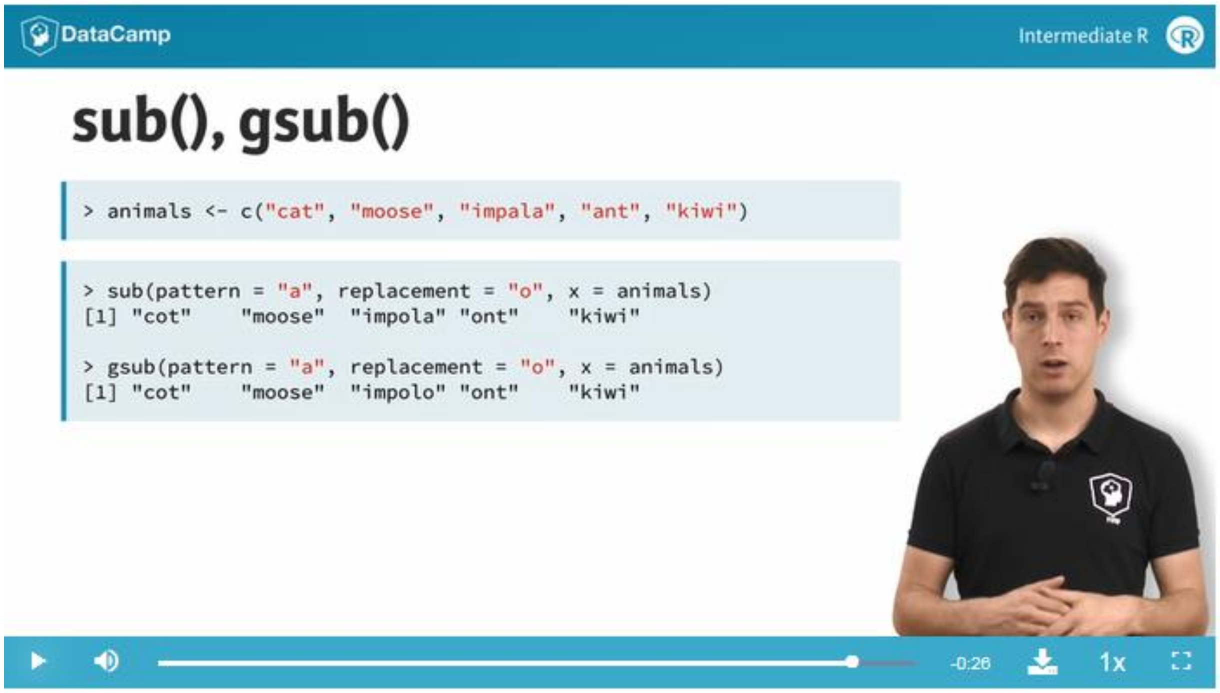
```
> sub(pattern = "a", replacement = "o", x = animals)
[1] "cot"    "moose"  "impola" "ont"    "kiwi"
```

Sub only does for first occurrence

Gsub for the entire set

Regular Expressions

50xp



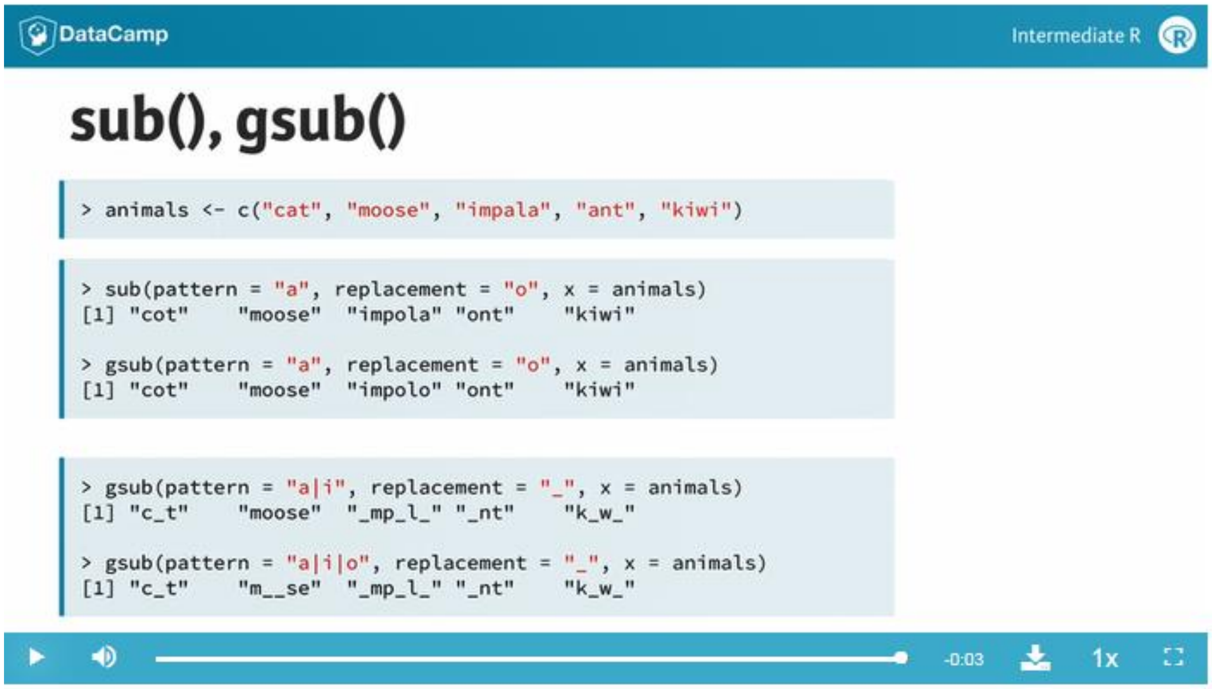
sub(), gsub()

```
> animals <- c("cat", "moose", "impala", "ant", "kiwi")
```

```
> sub(pattern = "a", replacement = "o", x = animals)
[1] "cot"    "moose"  "impola" "ont"    "kiwi"

> gsub(pattern = "a", replacement = "o", x = animals)
[1] "cot"    "moose"  "impolo" "ont"    "kiwi"
```

Regular Expressions

50xp



sub(), gsub()

```
> animals <- c("cat", "moose", "impala", "ant", "kiwi")
```

```
> sub(pattern = "a", replacement = "o", x = animals)
[1] "cot"    "moose"  "impola" "ont"    "kiwi"

> gsub(pattern = "a", replacement = "o", x = animals)
[1] "cot"    "moose"  "impolo" "ont"    "kiwi"
```

```
> gsub(pattern = "a|i", replacement = "_", x = animals)
[1] "c_t"    "moose"  "_mp_l_" "_nt"    "k_w_"

> gsub(pattern = "a|i|o", replacement = "_", x = animals)
[1] "c_t"    "m__se"  "_mp_l_" "_nt"    "k_w_"
```

grepl & grep

100xp

In their most basic form, regular expressions can be used to see whether a pattern exists inside a character string or a vector of character strings. For this purpose, you can use:

- `grepl()`, which returns `TRUE` when a pattern is found in the corresponding character string.
- `grep()`, which returns a vector of indices of the character strings that contains the pattern.

Both functions need a `pattern` and an `x` argument, where `pattern` is the regular expression you want to match for, and the `x` argument is the character vector from which matches should be sought.
In this and the following exercises, you'll be querying and manipulating a character vector of email addresses! The vector `emails` has already been defined in the editor on the right so you can begin with the instructions straight away!

Instructions

- Use `grepl()` to generate a vector of logicals that indicates whether these email addressess contain `"edu"`. Print the result to the output.
- Do the same thing with `grep()`, but this time save the resulting indexes in a variable `hits`.
- Use the variable `hits` to select from the `emails` vector only the emails that contain `"edu"`.

```
# The emails vector has already been defined for you

emails <- c("john.doe@ivyleague.edu", "education@world.gov", "dalai.lama@peace.org",

        "invalid.edu", "quant@bigdatacollege.edu", "cookie.monster@sesame.tv")


# Use grepl() to match for "edu"

grepl(pattern = "edu",x = emails)


# Use grep() to match for "edu", save result to hits

hits <-grep(pattern = "edu",x = emails)


# Subset emails using hits

emails[hits]
```

grepl & grep (2)

100xp

You can use the caret, `^`, and the dollar sign, `$` to match the content located in the start and end of a string, respectively. This could take us one step closer to a correct pattern for matching only the ".edu" email addresses from our list of emails. But there's more that can be added to make the pattern more robust:

- `@`, because a valid
- email must contain an at-sign.
- `.*`, which matches any character (.) zero or more times (*). Both the dot and the asterisk are metacharacters. You can use them to match any character between the at-sign and the ".edu" portion of an email address.
- `\\.edu$`, to match the ".edu" part of the email at the end of the string. The `\\` part *escapes* the dot: it tells R that you want to use the `.` as an actual character.

Instructions

- Use `grepl()` with the more advanced regular expression to return a logical vector. Simply print the result.
- Do a similar thing with `grep()` to create a vector of indices. Store the result in the variable `hits`.
- Use `emails[hits]` again to subset the `emails` vector.

```
# The emails vector has already been defined for you

emails <- c("john.doe@ivyleague.edu", "education@world.gov", "dalai.lama@peace.org",

            "invalid.edu", "quant@bigdatacollege.edu", "cookie.monster@sesame.tv")



# Use grepl() to match for .edu addresses more robustly

grepl(pattern = "@.*\\.edu",x = emails)



# Use grep() to match for .edu addresses more robustly, save result to hits

hits <- grep(pattern = "@.*\\.edu" , x = emails)



# Subset emails using hits

emails[hits]
```

sub & gsub

100xp

While `grep()` and `grepl()` were used to simply check whether a regular expression could be matched with a character vector, `sub()` and `gsub()` take it one step further: you can specify a `replacement` argument. If inside the character vector `x`, the regular expression `pattern` is found, the matching element(s) will be replaced with `replacement`.`sub()` only replaces the first match, whereas `gsub()` replaces all matches.
Suppose that `emails` vector you've been working with is an excerpt of DataCamp's email database. Why not offer the owners of the .edu email addresses a new email address on the datacamp.edu domain? This could be quite a powerful marketing stunt: Online education is taking over traditional learning institutions! Convert your email and be a part of the new generation!

Instructions

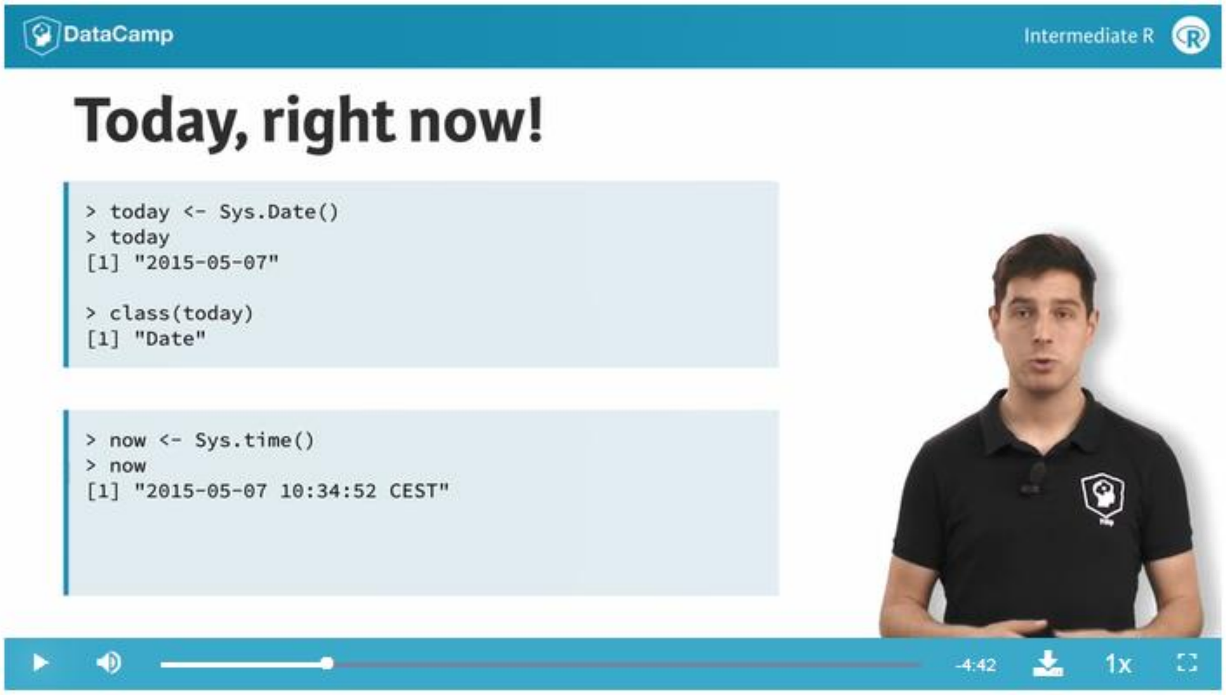
With the advanced regular expression `"@.*\\.edu$"`, use `sub()` to replace the match with `"@datacamp.edu"`. Since there will only be one match per character string, `gsub()` is not necessary here. Inspect the resulting output.

```
# The emails vector has already been defined for you

emails <- c("john.doe@ivyleague.edu", "education@world.gov", "global@peace.org",

            "invalid.edu", "quant@bigdatacollege.edu", "cookie.monster@sesame.tv")



# Use sub() to convert the email domains to datacamp.edu

sub(pattern = "@.*\\.edu$" , replacement = "@datacamp.edu",x = emails)
```

Times and Dates:→

Times and Dates

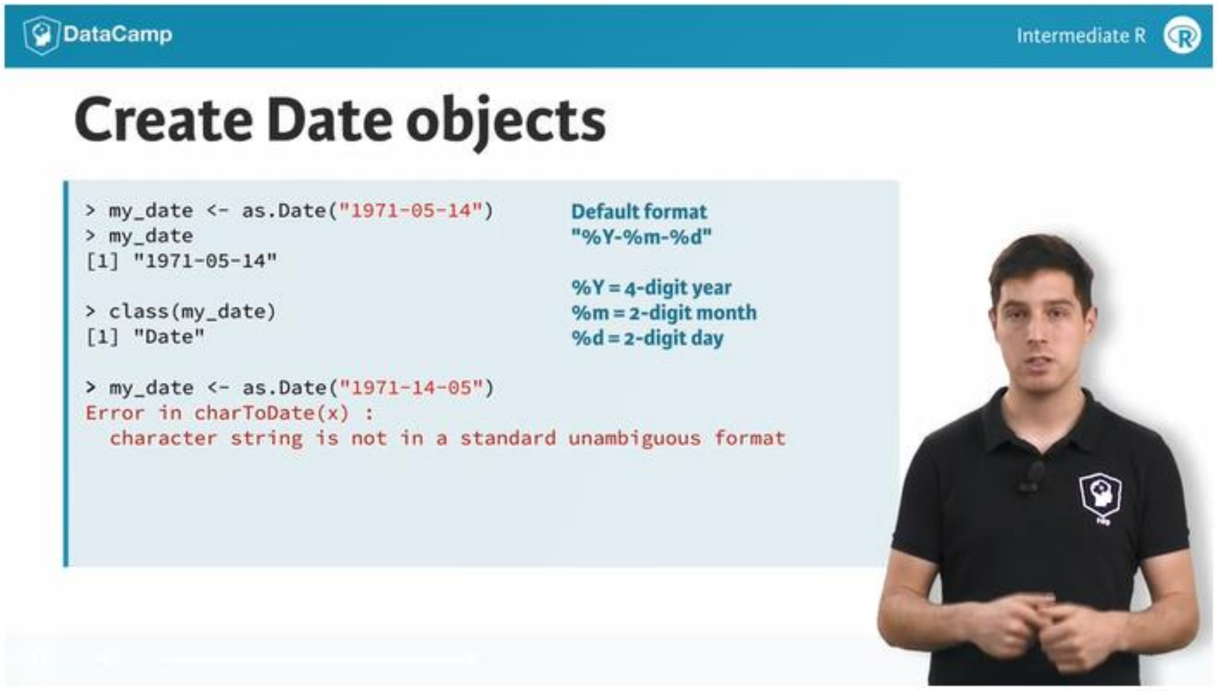
50xp



Today, right now!

```
> today <- Sys.Date()
> today
[1] "2015-05-07"

> class(today)
[1] "Date"
```

```
> now <- Sys.time()
> now
[1] "2015-05-07 10:34:52 CEST"
```

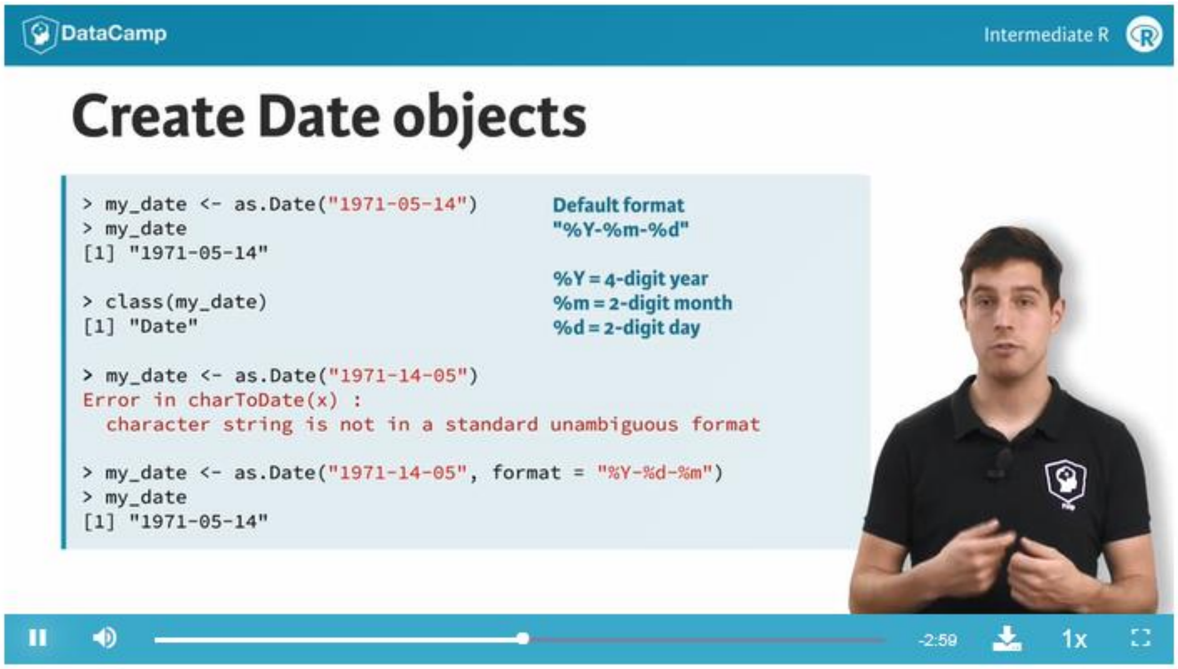


Create Date objects

```
> my_date <- as.Date("1971-05-14")
> my_date
[1] "1971-05-14"

> class(my_date)
[1] "Date"

> my_date <- as.Date("1971-14-05")
Error in charToDate(x) :
  character string is not in a standard unambiguous format
```

Default format
"%Y-%m-%d"

%Y = 4-digit year
%m = 2-digit month
%d = 2-digit day

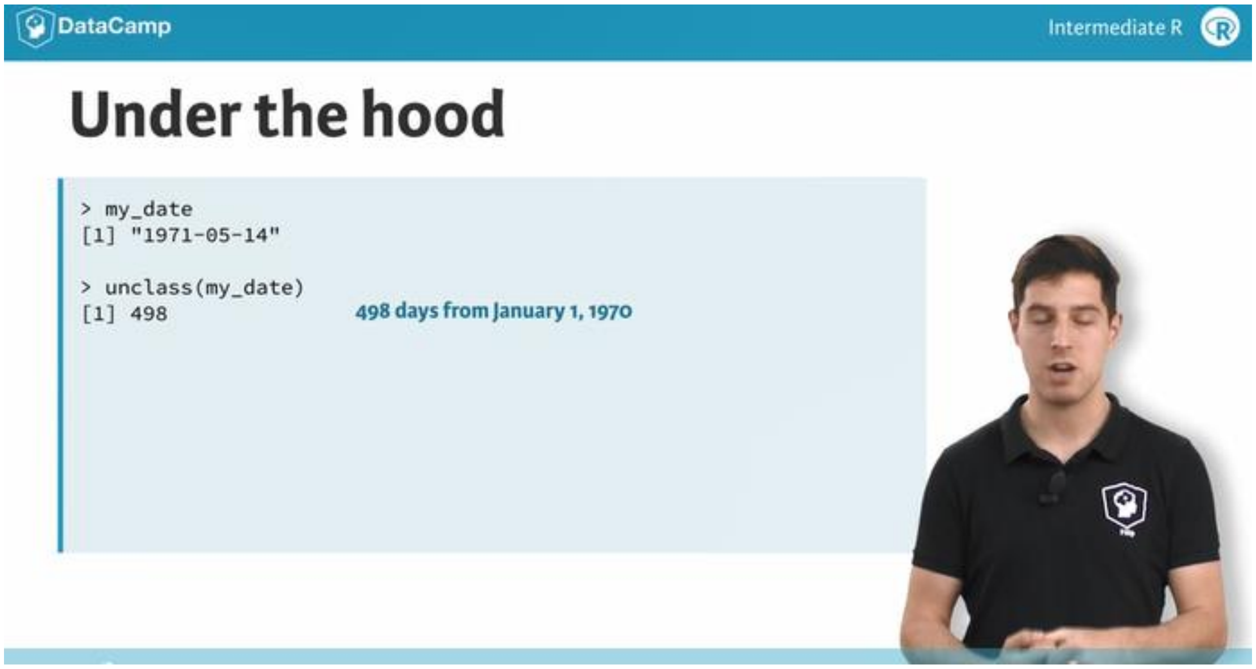


Create Date objects

```
> my_date <- as.Date("1971-05-14")
> my_date
[1] "1971-05-14"

> class(my_date)
[1] "Date"

> my_date <- as.Date("1971-14-05")
Error in charToDate(x) :
  character string is not in a standard unambiguous format

> my_date <- as.Date("1971-14-05", format = "%Y-%d-%m")
> my_date
[1] "1971-05-14"
```

Default format
"%Y-%m-%d"

%Y = 4-digit year
%m = 2-digit month
%d = 2-digit day

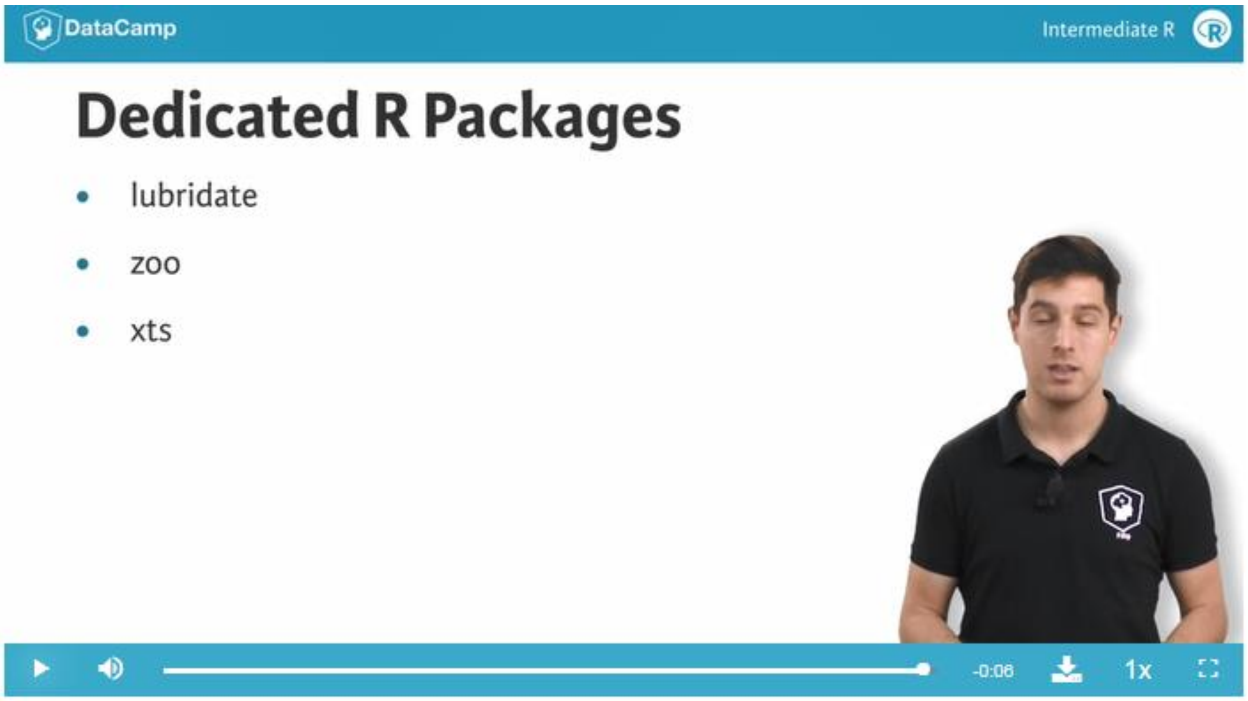


Under the hood

```
> my_date
[1] "1971-05-14"

> unclass(my_date)
[1] 498
```

498 days from January 1, 1970

Time and Date packages



Dedicated R Packages

- lubridate
- zoo
- xts

Right here, right now

100xp

In R, dates are represented by `Date` objects, while times are represented by `POSIXct` objects. Under the hood, however, these dates and times are simple numerical values. `Date` objects store the number of days since the 1st of January in 1970. `POSIXct` objects on the other hand, store the number of seconds since the 1st of January in 1970.

The 1st of January in 1970 is the common origin for representing times and dates in a wide range of programming languages. There is no particular reason for this; it is a simple convention. Of course, it's also possible to create dates and times before 1970; the corresponding numerical values are simply negative in this case.

Instructions

- Ask R for the current date, and store the result in a variable `today`.
- To see what `today` looks like under the hood, call `unclass()` on it.
- Ask R for the current time, and store the result in a variable, `now`.
- To see the numerical value that corresponds to `now`, call `unclass()` on it.

```
# Get the current date: today

today <-Sys.Date()
```

# See what today looks like under the hood

unclass(today)

# Get the current time: now

now <- Sys.time()

# See what now looks like under the hood

unclass(now)

Create and format dates

100xp

To create a `Date` object from a simple character string in R, you can use the `as.Date()` function. The character string has to obey a format that can be defined using a set of symbols (the examples correspond to 13 January, 1982):

- `%Y`: 4-digit year (1982)
- `%y`: 2-digit year (82)
- `%m`: 2-digit month (01)
- `%d`: 2-digit day of the month (13)
- `%A`: weekday (Wednesday)
- `%a`: abbreviated weekday (Wed)
- `%B`: month (January)
- `%b`: abbreviated month (Jan)

The following R commands will all create the same `Date` object for the 13th day in January of 1982:

```
as.Date("1982-01-13")
as.Date("Jan-13-82", format = "%b-%d-%y")
as.Date("13 January, 1982", format = "%d %B, %Y")
```

Notice that the first line here did not need a format argument, because by default R matches your character string to the formats `"%Y-%m-%d"` or `"%Y/%m/%d"`.

In addition to creating dates, you can also convert dates to character strings that use a different date notation. For this, you use the `format()` function. Try the following lines of code:

```
today <- Sys.Date()
format(Sys.Date(), format = "%d %B, %Y")
format(Sys.Date(), format = "Today is a %A!")
```

Instructions

- In the editor on the right, three character strings representing dates have been created. Convert them to dates using `as.Date()`, and assign them to `date1`, `date2`, and `date3` respectively. The code for `date1` is already included.
- Extract useful information from the dates as character strings using `format()`. From the first date, select the weekday. From the second date, select the day of the month. From the third date, you should select the abbreviated month and the 4-digit year, separated by a space.

# Definition of character strings representing dates

str1 <- "May 23, '96"

str2 <- "2012-03-15"

str3 <- "30/January/2006"

# Convert the strings to dates: date1, date2, date3

date1 <- as.Date(str1, format = "%b %d, '%y")

date2 <- as.Date(str2, format = "%Y-%m-%d")

date3 <- as.Date(str3, format = "%d/%B/%Y")

# Convert dates to formatted strings

format(date1, "%A")

format(date2, "%d")

format(date3, "%b%Y")

Create and format times

100xp

Similar to working with dates, you can use `as.POSIXct()` to convert from a character string to a `POSIXct` object, and `format()` to convert from a `POSIXct` object to a character string. Again, you have a wide variety of symbols:

- `%H`: hours as a decimal number (00-23)
- `%I`: hours as a decimal number (01-12)

- `%M`: minutes as a decimal number
- `%S`: seconds as a decimal number
- `%T`: shorthand notation for the typical format `%H:%M:%S`
- `%p`: AM/PM indicator

For a full list of conversion symbols, consult the `strptime` documentation in the console:

```
?strptime
```

Again,`as.POSIXct()` uses a default format to match character strings. In this case, it's `%Y-%m-%d %H:%M:%S`. In this exercise, abstraction is made of different time zones.

Instructions

- Convert two strings that represent timestamps, `str1` and `str2`, to `POSIXct` objects called `time1` and `time2`.
- Using `format()`, create a string from `time1` containing only the minutes.
- From `time2`, extract the hours and minutes as "hours:minutes AM/PM". Refer to the assignment text above to find the correct conversion symbols!

# Definition of character strings representing times

str1 <- "May 23, '96 hours:23 minutes:01 seconds:45"

str2 <- "2012-3-12 14:23:08"

# Convert the strings to POSIXct objects: time1, time2

time1 <- as.POSIXct(str1, format = "%B %d, '%y hours:%H minutes:%M seconds:%S")

time2 <- as.POSIXct(str2)

# Convert times to formatted strings

format(time1, "%M")

format(time2, "%I:%M %p")

Calculations with Dates

100xp

Both `Date` and `POSIXct` R objects are represented by simple numerical values under the hood. This makes calculation with time and date objects very straightforward: R performs the calculations using the underlying numerical values, and then converts the result back to human-readable time information again.
You can increment and decrement `Date` objects, or do actual calculations with them (try it out in the console!):

```
today <- Sys.Date()
today + 1
today - 1

as.Date("2015-03-12") - as.Date("2015-02-27")
```

To control your eating habits, you decided to write down the dates of the last five days that you ate pizza. In the workspace, these dates are defined as five `Date` objects, `day1` to `day5`. The code on the right also contains a vector `pizza` with these 5 `Date` objects.

Instructions

- Calculate the number of days that passed between the last and the first day you ate pizza. Print the result.
- Use the function `diff()` on `pizza` to calculate the differences between consecutive pizza days. Store the result in a new variable `day_diff`.
- Calculate the average period between two consecutive pizza days. Print the result.

# day1, day2, day3, day4 and day5 are already available in the workspace

# Difference between last and first pizza day

day5 -day1

# Create vector pizza

pizza <- c(day1, day2, day3, day4, day5)

# Create differences between consecutive pizza days: day_diff

day_diff <-diff(pizza)

# Average period between two consecutive pizza days

mean(day_diff)