CONTENT

Learning Objectives

Bookmark this page

Learning Objectives

In this course you will learn about:

- The basics of R
- Writing your own R scripts
- How to use R to solve problems related to movies data
- The fundamentals of R Syntax
- Vectors, lists, matrix, arrays and dataframes
- Reading and writing data in R

Learning Objectives

Bookmark this page

Learning Objectives

In this course you will learn about:

- The basics of R
- Writing your own R scripts
- How to use R to solve problems related to movies data
- The fundamentals of R Syntax
- Vectors, lists, matrix, arrays and dataframes
- Reading and writing data in R

1. R Basics

Basic math with R

name	year	length_min	genre	average_rating	cost_millions	foreign	age_restriction
Fight Club	1999	139	Drama	8.9	63	0	18
Star Wars	1977	121	Action	8.7	11	0	10
Interstellar	2014	169	Adventure	8.6	165	0	10
Jumanji	1995	104	Fantasy	6.9	65	0	12

- Total time:

```
139 + 121
```

Variables

name	year	length_min	genre	average_rating	cost_millions	foreign	age_restriction
Fight Club	1999	139	Drama	8.9	63	0	18
Star Wars	1977	121	Action	8.7	11	0	10
Interstellar	2014	169	Adventure	8.6	165	0	10
Jumanji	1995	104	Fantasy	6.9	65	0	12

- Variable assignment:

```
x <- 139 + 121
x
[1] 260

y <- x / 60
y
[1] 4.333333
```

Variables In R
Variables are typically assigned using <- but can also be assigned using =, as in x <- 1 or x = 1.

One important note is that the "equals sign" can also be used to assign values to variables,

One important note is that the "equals sign" can also be used to assign values to variables,

BIG DATA UNIVERSITY

Variable reassignment

- Create x:

```
x <- 139 + 121
x
[1] 260
```

- Overwrite x:

```
x <- x / 60
x
[1] 4.333333
```

Variables In R
To remove variable from memory use **rm**(my_variable) command.

Variable Names

- Total time:

```
total <- 139 + 121
total
```

```
[1] 260
```

- Total hours:

```
total_hr <- total / 60
total_hr
```

Order of Operations

```
total <- 139 + 121
total
```

```
total_hr <- total / 60
total_hr
```

```
[1] 4.333333
```

```
total_hr <- (139 + 121) / 60
total_hr
```

Strings in R

```
movie <- "Toy Story"
movie
```

```
[1] "Toy Story"
```

Start of transcript. Skip to the end.

Hello, and welcome. In this video, we will show you how to perform basic mathematical operations in R, how to assign values to variables, and how to create strings.

Here we have some data that contains a set of movies, along with some relevant information.

As an example, let's check whether or not we have time to watch "Fight Club" and "Star Wars" in four hours. Based on the chart, the respective lengths are 139 and 121 minutes. To figure out the total time, all we need to do is write "139 + 121", and we see that the answer is 260 minutes.

But we need to know how many hours this will take.

So we'll use the forward slash to divide 260 minutes by 60 minutes per hour.

As you can see, these two movies take a little longer than four hours.

Besides addition and division, R provides operators for subtraction, multiplication, and exponentiation.

You can see the operator symbols in the chart here.

If you'd like to save a calculation and use it later, you can store a value in a variable.

So let's add the run times again, but this time we'll store the result in a variable called "x".
You can see how that works in the code snippet here. If we then take a look at our new variable "x",
we see that the value 260 has indeed been stored.
From now on, we can perform mathematical operations on the variable "x", so let's divide it by 60 and store the result in a variable called "y".
As you can see, we get the same answer as before.
One important note is that the "equals sign" can also be used to assign values to variables, although it's not quite as common.
You can also reassign variables that you've already used. So again let's assign a value to "x" by adding 139 and 121. If we look at the output, we'll see the value 260 from before.
Overwriting "x" is very simple. All we need to do is use the variable assignment operator again, and if we look at the output, we see that the original value has been replaced. Since variables take up memory, it's good practice to clean them up when you no longer need them. To do this, you can use the "rm" command, while passing in the variable that you want to remove.
Whenever you write code, it's important to use meaningful variable names.
The variable name "x" from before isn't very descriptive and may be hard to understand.
So when computing the total time, we can instead use a variable name like "total", which is more descriptive of the variable's value. When computing the total hours, we can use another variable name with "underscore hr" appended, so that the variable can be differentiated.
It's important to know the order of operations, since some operators have a higher precedence.
In our previous examples, we've been performing the addition and division operations in different statements, which gives us the result we'd expect. But we can combine them into a single statement.
Notice however that we've added the parentheses around the addition portion.
This is because the division operator has a higher precedence than the addition operator.
Without the parentheses, R would first divide 121 by 60, and then add the result to 139.
The parentheses ensure that we get the correct answer.
In R, a string is simply a sequence of one or more characters.
To create a string, simply enclose the characters within matching single quotes, or matching double quotes.
Like numbers, strings can also be assigned to variables.
By now, you should understand the basic mathematical operations provided by R, as well as how to work with variables and create strings.
Thank you for watching this video.
End of transcript. Skip to the start.

>> Lab:

Table of Contents

Estimated Time Needed: **15 min**

About the Dataset

Which movie should you watch next?

Let's say each of your friends tells you their favorite movies. You do some research on the movies and put it all into a table. Now you can begin exploring the dataset, and asking questions about the movies. For example, you can check if movies from some certain genres tend to get better ratings. You can check how the production cost for movies changes across years, and much more.

Movies dataset

The table gathered includes one row for each movie, with several columns for each movie characteristic:

- **name** - Name of the movie
- **year** - Year the movie was released
- **length_min** - Length of the movie (minutes)
- **genre** - Genre of the movie
- **average_rating** - Average rating on IMDB
- **cost_millions** - Movie's production cost (millions in USD)
- **foreign** - Is the movie foreign (1) or domestic (0)?
- **age_restriction** - Age restriction for the movie

name	year	length_min	genre	average_rating	cost_millions	foreign	age_restriction
Toy Story	1995	81	Animation	8.3	30	0	0
Akira	1998	125	Animation	8.1	10.4	1	14
The Breakfast Club	1985	97	Drama	7.9	1	0	14
The Artist	2011	100	Romance	8	15	1	12
Modern Times	1936	87	Comedy	8.6	1.5	0	10
Fight Club	1999	139	Drama	8.9	63	0	18
City of God	2002	130	Crime	8.7	3.3	1	18
The Untouchables	1987	119	Drama	7.9	25	0	14
Star Wars	1977	121	Action	8.7	11	0	10
American Beauty	1999	122	Drama	8.4	15	0	14
Room	2015	118	Drama	8.3	13	1	14
Dr. Strangelove	1964	94	Comedy	8.5	1.8	1	10
The Ring	1998	95	Horror	7.3	1.2	1	18
Monty Python and the Holy Grail	1975	91	Comedy	8.3	0.4	1	18
High School Musical	2006	98	Comedy	5.2	4.2	0	0
Shaun of the Dead	2004	99	Horror	8	6.1	1	18
Taxi Driver	1976	113	Crime	8.3	1.3	1	14
The Shawshank Redemption	1994	142	Crime	9.3	25	0	16
Interstellar	2014	169	Adventure	8.6	165	0	10
Casino	1995	178	Biography	8.2	50	0	18
The Goodfellas	1990	145	Biography	8.7	25	0	14
Blue is the Warmest Colour	2013	179	Romance	7.8	4.5	1	18
Black Swan	2010	108	Thriller	8	13	0	16

Simple Math in R

Let's say you want to watch *Fight Club* and *Star Wars: Episode IV (1977)*, back-to-back. Do you have enough time to **watch both movies ir hours?** Let's try using simple math in R.

What is the **total movie length** for Fight Club and Star Wars (1977)?

- **Fight Club**: 139 min
- **Star Wars: Episode IV**: 121 min

```
139 + 121
```

260

Great! You've determined that the total number of movie play time is **260 min**.

What is 260 min in hours?

```
260 / 60
```

4.33333333333333

Well, it looks like it's **over 4 hours**, which means you can't watch *Fight Club* and *Star Wars (1977)* back-to-back if you only have 4 hours available!

[Tip] Simple math in R

You can do a variety of mathematical operations in R including:
addition: **2 + 2**
subtraction: **5 - 2**
multiplication: **3 * 2**
division: **4 / 2**
exponentiation: **4 ** 2** or **4 ^ 2 **

Variables in R

We can also **store** our output in **variables**, so we can use them later on. For example:

3]:
```
x <- 139 + 121
```

To return the value of `x`, we can simply run the variable as a command:

4]:
```
x
```

260

We can also perform operations on `x` and save the result to a **new variable**:

5]:
```
y <- x / 60
y
```

4.33333333333333

If we save something to an **existing variable**, it will **overwrite** the previous value:

6]:
```
x <- x / 60
x
```

4.33333333333333

It's good practice to use **meaningful variable names**, so you don't have to keep track of what variable is what:

```
total <- 139 + 121
total
```

260

```
total_hr <- total / 60
total_hr
```

4.33333333333333

You can put this all into a single expression, but remember to use **round brackets** to add together the movie lengths first, before dividing by 60.

```
total_hr <- (139 + 121) / 60
total_hr
```

4.33333333333333

[Tip] Variables in R

As you just learned, you can use **variables** to store values for repeated use. Here are some more **characteristics of variables in R**:

- variables store the output of a block of code
- variables are typically assigned using **
- once created, variables can be removed from memory using **rm(**my_variable**)**

Vectors in R

What if we want to know the **movie length in *hours***, not minutes, for *Toy Story* and for *Akira*?

- **Toy Story (1995)**: 81 min
- **Akira (1998)**: 125 min

```
c(81, 125) / 60
```

1.35 · 2.08333333333333

As you see above, we've applied a single math operation to both of the items in `c(81, 125)`. You can even assign `c(81, 125)` to a variable before performing an operation.

```
ratings <- c(81, 125)
ratings / 60
```

1.35 · 2.08333333333333

What we just did was create vectors, using the combine function `c()`. The `c()` function takes multiple items, then combines them into a **vector**.

It's important to understand that **vectors** are used everywhere in R, and vectors are easy to use.

It's important to understand that **vectors** are used everywhere in R, and vectors are easy to use.

```
c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
c(1:10)
```

1 · 2 · 3 · 4 · 5 · 6 · 7 · 8 · 9 · 10

1 · 2 · 3 · 4 · 5 · 6 · 7 · 8 · 9 · 10

```
c(10:1) # 10 to 1
```

10 · 9 · 8 · 7 · 6 · 5 · 4 · 3 · 2 · 1

[Tip] # Comments

Did you notice the **comment** after the **c(10:1)** above? Comments are very useful in describing your code. You can create your own comments by using the **#** symbol and writing your comment after it. R will interpret it as a comment, not as code.

Strings in R

R isn't just about numbers -- we can also have strings too. For example:

```
movie <- "Toy Story"
movie
```

'Toy Story'

In R, you can identify **character strings** when they are wrapped with **matching double (") or single (') quotes**.

Let's create a **character vector** for the following **genres**:

- Animation
- Comedy
- Biography
- Horror
- Romance
- Sci-fi

```
genres <- c("Animation", "Comedy", "Biography", "Horror", "Romance", "Sci-fi")
genres
```

'Animation' · 'Comedy' · 'Biography' · 'Horror' · 'Romance' · 'Sci-fi'

Introduction to Vectors

name	year	length_min
Toy Story	1995	81
Akira	1998	125
The Breakfast Club	1985	97

```
c(81, 125) / 60
```

```
[1] 1.350000 2.083333
```

```
ratings <- c(81, 125)
ratings / 60
```

More on vectors

```
c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
c(1:10)
```

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

```
c(10:1) # 10 to 1
```

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

 # comments

- Used to describe your code.
- Created using # symbol.
- Not interpreted as code.

Types of vectors

- Numeric vector:

```
c(1985, 1999, 2010, 2002)
```

```
[1] 1985 1999 2010 2002
```

- Character vector:

```
c("Toy Story", "Akira", "The Artist", "City of God")
```

```
[1] "Toy Story"   "Akira"       "The Artist"  "City of God"
```

Types of vectors

- Numeric vector
- Character Vector
- **Logical Vector:**

```
1997 > 2000
```

```
[1] FALSE
```

```
movie_ratings
```

```
 [1] 7.3 8.5 8.3 7.5 6.9 5.2 8.2 8.0 7.9 9.3
```

```
movie_ratings > 7.5
```

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

You can see how that would look in the output here.

Summarizing the Factors

```
genre_vector <- c("Comedy","Animation","Crime","Comedy","Animation")
```

```
genre_factor <- factor(genre_vector)
```

```
summary(genre_vector)
```

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

```
summary(genre_factor)
```

Ordered Factor

```
movielength_vector <- c("Very short", "Short", "Medium","Short",
                        "Long","Very short", "Very Long")
```

```
[1] "Very short" "Short"      "Medium"     "Short"      "Long"
[6] "Very short" "Very Long"
```

```
mvlength_factor <- factor(movielength_vector,
                          ordered = TRUE ,
                          levels = c("Very short" , "Short" , "Medium",
                                     "Long","Very Long"))
mvlength_factor
```

```
[1] Very short Short      Medium     Short      Long       Very short Very Long

Levels: Very short < Short < Medium < Long < Very Long
```

If you take a look at the output, you can see that this information has been successfully encoded.

Cognitive Class Home Page
Cognitive Class: RP0101EN R 101
Learning Paths
Courses
Badges ▼
Business
Help
Dashboard for: rafasolis1984

Course , current location
Discussion
Resources
Progress

Course Module 1 - R basics Vectors and Factors (4:50) Vectors and Factors (4:50)

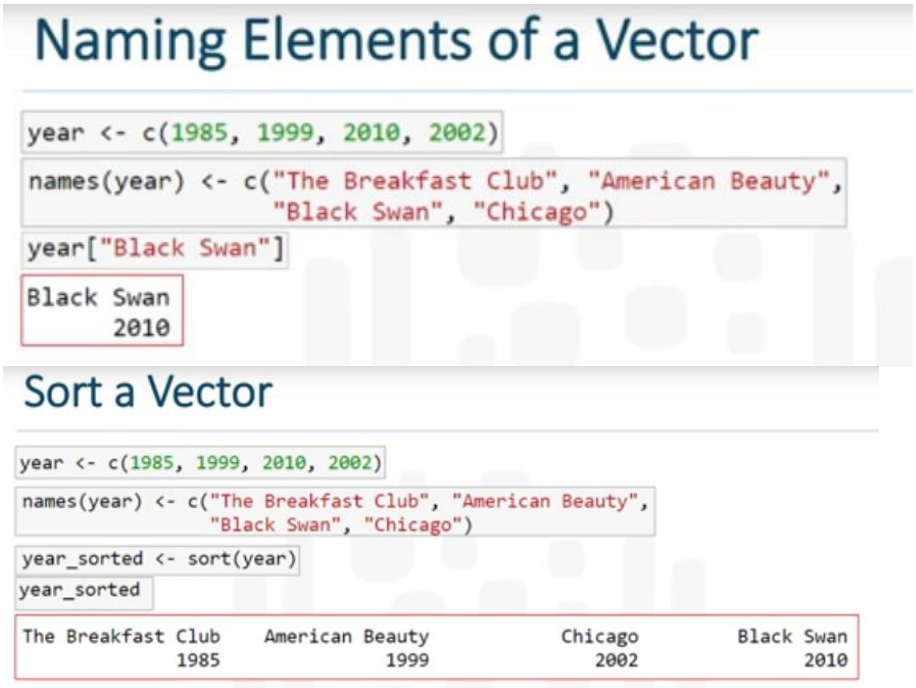
Vectors and Factors (4:50)
Vectors and Factors (4:50)
By now, you should understand how to create various types of vectors,
4:37 / 4:50

Press UP to enter the speed menu then use the UP and DOWN arrow keys to navigate the different speeds, then press ENTER to change to the selected speed.

Click on this button to mute or unmute this video or press UP or DOWN buttons to increase or decrease volume level.
Maximum Volume.
Video transcript

Start of transcript. Skip to the end.
Hello, and welcome. In this video, we will show you how to create vectors and factors in the R programming language.
A vector is a one-dimensional array of objects,
and it's a simple tool to store your data. In R, there is no restriction
on the type or number of elements that a vector can contain. Let's create a vector that contains the run times of
that contains the run times of "Toy Story" and "Akira".
To do this, we simply use the "c" command, and inside the parentheses, we'll write the two values that we want
to include in our vector. For purposes of illustration, we'll also divide the vector by 60.
This division operation will be applied to each element individually.
We can also assign a vector to a variable, which will give us the same result.
It's easy to create a vector that contains a sequence of elements. So let's say
you wanted a vector that contains the numbers from one to ten, in order.
You could type all the values out, or you could simply write "one", then a colon, then "ten".
The result will be the same. If you want the numbers in decreasing order,
simply switch the 1 and the 10. And you can see how that would look. As a quick note,
take a look at the "pound sign" to the right of the "c" command. In R,
this symbol is used to write comments in your code.
Everything to the right of the symbol is for the programmer only, and will not be interpreted.
So we just saw that we can create vectors that hold numerical data, like a set of years.
But we can also create a character vector, if we want to store a list of strings.
For example, we can create a vector that holds titles, which would look like this.
One additional vector type we'll touch on is the logical vector.
Logical data is simply a "True" or "False", typically created via a comparison operator.
For example, if we type "1997 greater than 2000", we get "False" as the output.
These operators can be applied to vectors.
So suppose we have a vector that contains movie ratings,
which looks like this. If we run "movie ratings greater than 7.5",
R will check each element in the vector to see if the value is greater than 7.5.
If so, the resulting value will be true, and if not, the value will be false.
You can see how that would look in the output here.
In R, factors are variables that can take on a limited number of values.
These variables are often referred to as categorical variables. The other extreme would be a continuous variable,
which can take on an infinite number of value
Let's encode the vector we have here as a factor.
If you look at the output, you'll notice that there are 3 levels, which correspond to the
3 unique categories of the vector: Animation, comedy, and crime.
The summary function is useful when working with factors. If we apply

“summary” on the original vector, you’ll notice that it simply provides some basic information
about the vector’s structure and content. But if we apply it on the factor,
the output shows us the number of occurrences of each of the component categories.
There are two types of categorical variables
A nominal categorical variable does not have any implied order.
The previous examples with movie genres were nominal, since there is no clear ordering between categories
like “Animation” and “comedy”. Ordinal variables, on the other hand, do have an ordering.
So consider the vector here. It should be apparent that certain categories are “less” than others,
and we’d want to encode this information into a factor.
To do this, we need to pass two additional arguments to the “factor” function.
We need to specify that “ordered = TRUE”, and then we need to specify a vector with the levels.
Notice that the categories in the vector are listed in the order that we want,
from “lowest” to “highest” so to speak.
If you take a look at the output, you can see that this information has been successfully encoded.
By now, you should understand how to create various types of vectors,
as well as how to use the “factor” function.
Thank you for watching this video.
End of transcript. Skip to the start.



Smallest and Largest Number

```
year <- c(1985, 1999, 2010, 2002)
```

```
min(year)
```

```
[1] 1985
```

```
max(year)
```

```
[1] 2010
```

Vector Summary

```
cost_2014 <- c(8.6, 8.5, 8.1)
summary(cost_2014)
```

```
 Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
 8.10    8.30    8.50    8.40    8.55    8.60 
```

Vector Index

```
cost_2014 <- c(8.6, 8.5, 8.1)
cost_2014 [1:3]
```

```
[1] 8.6 8.5 8.1
```

```
titles <- c("Black Swan", "Casino", "City of God", "Jumanji", "Toy Story")
titles[-1]
```

```
titles <- c("Black Swan", "Jumanji", "City of God", "Toy Story", "Casino")
titles[8]
```

```
[1] NA
```

```
cost_2014 <- c(8.6, 8.5, 8.1)
cost_2014 > 8.3
```

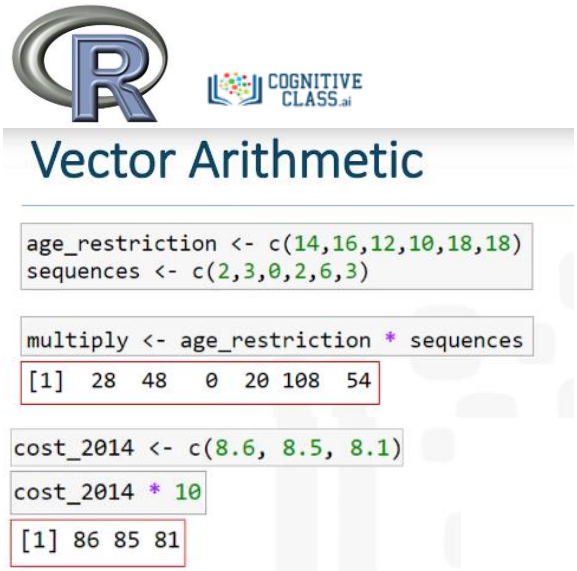
```
[1]  TRUE  TRUE FALSE
```

```
cost_2014[cost_2014 > 8.3]
```

```
[1] 8.6 8.5
```

Missing Values

```
age_restric <- c(14, 12, 10, NA, 18, NA)
```



In this video, we will demonstrate several useful operations you can apply to vectors in the R programming language.
Suppose we have a vector that contains the production years of four movies.
It may be difficult to remember which movies these years belong to, so we can use the ?names?
function to map a title to each year.
Now we can call our ?year?
vector while passing in a title in the brackets.
As you can see in the output, the year and title are now mapped together.
Finding the number of elements of a vector is simple.
All you need to do is call the "length"
function, while passing in the vector as input.
As expected, the vector has a length of 4.
In order to sort a vector, we can simply call the "sort"
function, while passing in the vector as input.
This will sort the movies in ascending order timewise, from oldest to newest.
Notice that since we used the "names"
function, the titles appear in the output along with the year.
Finding the smallest and largest numbers in a vector is an easy task as well.
To find the smallest number, simply use the ?min?
function while passing the vector as input.
For the largest number, use the "max"
function instead.
We see that the min and max are 1985, and 2010, respectively.
Now suppose we have a vector that contains the cost of three movies, in millions, and we want to compute the average.
One way to do this is by using the "sum"
function to calculate the sum of all the vector elements, and then dividing the result by 3.
A simpler alternative is to use the "mean"
function instead.
Both of these methods will produce the same result.
Another useful tool is the "summary"
function.
This function will display descriptive statistics about the vector like the min and max, median, mean, and 1st and 3rd quartiles.
In order to retrieve an element of a vector, you simply need to write the vectors name, followed by the desired index in square brackets.

So this piece of code would retrieve the second element.
You can also use a vector inside the square brackets.
In this case, we're telling R to retrieve the second and third elements.
You can also retrieve the elements of a vector by specifying a range.
In this example, we retrieve the elements from index 1 to index 3.
Putting a negative number as the vector index will remove that particular element from the output.
So for example, a "-1" will remove the element that was in index "1".
The vector that you see here only has 5 elements.
If you try to access an index beyond 5, R will report a missing value by returning the symbol "NA".
Now going back to our cost vector, let's apply a logical operation on each of the elements.
As we?d expect, the first two elements are true, while the third element is false.
In order to retrieve the elements from the vector that are true, you can write the conditional inside of the square brackets.
Notice that the only elements that are output are the elements that were true from the previous
step.
As we mentioned before, R handles missing values through the symbol "NA".
Missing values come up all the time in real world datasets for various reasons.
For example, if these numbers represent age restrictions for a movie, then the indices with missing values might be movies that have an unknown age restriction.
An important thing to remember about vectors is that all arithmetic operations are performed element-wise.
So let's multiply these two vectors together and take a look at the output.
You'll notice that the first element, 28, is simply the product of the first element of vector 1, and the first element of vector 2.
48 is the product of the second elements, and so on.
In this example, we multiply a vector by a number instead, but the concept is the same.
Each element is multiplied by 10.
By now, you should have a good idea of how to access the elements of a vector and how to apply useful operations.
Thank you for watching this video.
End of transcript. Skip to the start.

>> Lab:

Table of Contents

About the Dataset

You have received many movie recomendations from your friends and compiled all of the recommendations into a table, with information about each movie.

This table has one row for each movie and several columns.

- **name** - The name of the movie
- **year** - The year the movie was released
- **length_min** - The lenght of the movie in minutes
- **genre** - The genre of the movie
- **average_rating** - Average rating on Imdb
- **cost_millions** - The movie's production cost in millions
- **sequences** - The amount of sequences
- **foreign** - Indicative of whether the movie is foreign (1) or domestic (0)
- **age_restriction** - The age restriction for the movie

Here's what the data looks like:

name	year	length_min	genre	average_rating	cost_millions	foreign	age_restriction
Toy Story	1995	81	Animation	8.3	30	0	0
Akira	1998	125	Animation	8.1	10.4	1	14
The Breakfast Club	1985	97	Drama	7.9	1	0	14
The Artist	2011	100	Romance	8	15	1	12
Modern Times	1936	87	Comedy	8.6	1.5	0	10
Fight Club	1999	139	Drama	8.9	63	0	18
City of God	2002	130	Crime	8.7	3.3	1	18
The Untouchables	1987	119	Drama	7.9	25	0	14
Star Wars	1977	121	Action	8.7	11	0	10
American Beauty	1999	122	Drama	8.4	15	0	14
Room	2015	118	Drama	8.3	13	1	14
Dr. Strangelove	1964	94	Comedy	8.5	1.8	1	10
The Ring	1998	95	Horror	7.3	1.2	1	18
Monty Python and the Holy Grail	1975	91	Comedy	8.3	0.4	1	18
High School Musical	2006	98	Comedy	5.2	4.2	0	0
Shaun of the Dead	2004	99	Horror	8	6.1	1	18
Taxi Driver	1976	113	Crime	8.3	1.3	1	14
The Shawshank Redemption	1994	142	Crime	9.3	25	0	16
Interstellar	2014	169	Adventure	8.6	165	0	10

Vectors

Vectors are strings of numbers, characters or logical data (one-dimension array). In other words, a vector is a simple tool to store your grouped data.

In R, you create a vector with the combine function **c()**. You place the vector elements separated by a comma between the brackets. Vectors will be very useful in the future as they allow you to apply operations on a series of data easily.

Note that the items in a vector must be of the same class, for example all should be either number, character, or logical.

Numeric, Character and Logical Vectors

Let's say we have four movie release dates (1985, 1999, 2015, 1964) and we want to assign them to a single variable, `release_year`. This means we'll need to create a vector using `c()`.

Using numbers, this becomes a **numeric vector**.

```
release_year
```

1985 · 1999 · 2015 · 1964

What if we use quotation marks? Then this becomes a **character vector**.

```
# Create genre vector and assign values to it
titles <- c("Toy Story", "Akira", "The Breakfast Club")
titles
```

'Toy Story' · 'Akira' · 'The Breakfast Club'

There are also **logical vectors**, which consist of TRUE's and FALSE's. They're particular important when you want to check its contents

```
titles == "Akira" # which item in `titles` is equal to "Akira"?
```

FALSE · TRUE · FALSE

[Tip] TRUE and FALSE in R

Did you know? R only recognizes `TRUE`, `FALSE`, `T` and `F` as special values for true and false. That means all other spellings, including *True* and *true*, are not interpreted by R as logical values.

Vector Operations

Adding more elements to a vector

You can add more elements to a vector with the same **`c()`** function you use the create vectors:

```
release_year <- c(1985, 1999, 2015, 1964)
release_year
```

1985 · 1999 · 2015 · 1964

```
release_year <- c(release_year, 2016:2018)
release_year
```

1985 · 1999 · 2015 · 1964 · 2016 · 2017 · 2018

Length of a vector

How do we check how many items there are in a vector? We can use the **length()** function:

```
release_year
length(release_year)
```

1985 · 1999 · 2015 · 1964 · 2016 · 2017 · 2018

7

Head and Tail of a vector

We can also retrieve just the **first few items** using the **head()** function:

```
head(release_year) #first six items
```

1985 · 1999 · 2015 · 1964 · 2016 · 2017

```
head(release_year, n = 2) #first n items
```

1985 · 1999

```
head(release_year, 2)
```

1985 · 1999

We can also retrieve just the **last few items** using the **tail()** function:

We can also retrieve just the **last few items** using the **tail()** function:

```
tail(release_year) #last six items
```

1999 · 2015 · 1964 · 2016 · 2017 · 2018

```
tail(release_year, 2) #last two items
```

2017 · 2018

Sorting a vector

We can also sort a vector:

```
sort(release_year)
```

1964 · 1985 · 1999 · 2015 · 2016 · 2017 · 2018

We can also **sort in decreasing order**:

```
sort(release_year, decreasing = TRUE)
```

2018 · 2017 · 2016 · 2015 · 1999 · 1985 · 1964

But if you just want the minimum and maximum values of a vector, you can use the `min()` and `max()` functions

```
min(release_year)
max(release_year)
```

1964

2018

Average of Numbers

If you want to check the average cost of movies produced in 2014, what would you do? Of course, one way is to add all the numbers together, then divide by the number of movies:

```
cost_2014 <- c(8.6, 8.5, 8.1)

# sum results in the sum of all elements in the vector
avg_cost_2014 <- sum(cost_2014)/3
avg_cost_2014
```

8.4

You also can use the mean function to find the average of the numeric values in a vector:

```
mean_cost_2014 <- mean(cost_2014)
mean_cost_2014
```

8.4

Giving Names to Values in a Vector

Suppose you want to remember which year corresponds to which movie.

With vectors, you can give names to the elements of a vector using the **names()* * function:

```
#Creating a year vector
release_year <- c(1985, 1999, 2010, 2002)

#Assigning names
names(release_year) <- c("The Breakfast Club", "American Beauty", "Black Swan", "Chicago")

release_year
```

The Breakfast Club: 1985 **American Beauty:** 1999 **Black Swan:** 2010 **Chicago:** 2002

Now, you can retrieve the values based on the names:

```
release_year[c("American Beauty", "Chicago")]
```

American Beauty: 1999 **Chicago:** 2002

Note that the values of the vector are still the years. We can see this in action by adding a number to the first item:

Note that the values of the vector are still the years. We can see this in action by adding a number to the first item:

```
release_year[1] + 100 #adding 100 to the first item changes the year
```

The Breakfast Club: 2085

And you can retrieve the names of the vector using `names()`

```
names(release_year)[1:3]
```

'The Breakfast Club' · 'American Beauty' · 'Black Swan'

Summarizing Vectors

You can also use the **"summary"** function for simple descriptive statistics: minimum, first quartile, mean, third quartile, maximum:

```
summary(cost_2014)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
   8.10    8.30    8.50    8.40    8.55    8.60 
```

Using Logical Operations on Vectors

A vector can also be comprised of `TRUE` and `FALSE`, which are special **logical values** in R. These boolean values are used used to indicate whether a condition is true or false.

Let's check whether a movie year of 1997 is older than (**greater in value than**) 2000.

```
movie_year <- 1997
movie_year > 2000
```

FALSE

You can also make a logical comparison across multiple items in a vector. Which movie release years here are "greater" than 2014?

```
movies_years <- c(1998, 2010, 2016)
movies_years > 2014
```

FALSE · FALSE · TRUE

We can also check for **equivalence**, using `==`. Let's check which movie year is equal to 2015.

```
movies_years == 2015 # is equal to 2015?
```

FALSE · FALSE · FALSE

If you want to check which ones are **not equal** to 2015, you can use `!=`

```
movies_years != 2015
```

TRUE · TRUE · TRUE

[Tip] Logical Operators in R

You can do a variety of logical operations in R including:

- Checking equivalence: **1 = = 2**
- Checking non-equivalence: **TRUE != FALSE**
- Greater than: **100 > 1**
- Greater than or equal to: **100 > = 1**
- Less than: **1 < 2**
- Less than or equal to: **1

Subsetting Vectors

What if you wanted to retrieve the second year from the following **vector of movie years**?

```
movie_years <- c(1985, 1999, 2002, 2010, 2012)
movie_years
```

1985 · 1999 · 2002 · 2010 · 2012

To retrieve the **second year**, you can use square brackets `[]`:

```
movie_years[2] #second item
```

1999

To retrieve the **third year**, you can use:

```
movie_years[3]
```

2002

And if you want to retrieve **multiple items**, you can pass in a vector:

```
movie_years[c(1,3)] #first and third items
```

1985 · 2002

Retrieving a vector without some of its items

To retrieve a vector without an item, you can use negative indexing. For example, the following returns a vector slice **without the first item**.

```
titles <- c("Black Swan", "Jumanji", "City of God", "Toy Story", "Casino")
titles[-1]
```

'Jumanji' · 'City of God' · 'Toy Story' · 'Casino'

You can save the new vector using a variable:

```
new_titles <- titles[-1] #removes "Black Swan", the first item
new_titles
```

'Jumanji' · 'City of God' · 'Toy Story' · 'Casino'

** Missing Values (NA)**

Sometimes values in a vector are missing and you have to show them using NA, which is a special value in R for "Not Available". For example, if you don't know the age restriction for some movies, you can use NA.

```
age_restric <- c(14, 12, 10, NA, 18, NA)
age_restric
```

14 · 12 · 10 · <NA> · 18 · <NA>

[Tip] Checking NA in R

You can check if a value is NA by using the **is.na()** function, which returns TRUE or FALSE.

- Check if NA: **is.na(NA)**
- Check if not NA: **!is.na(2)**

Subsetting vectors based on a logical condition

What if we want to know which movies were created after year 2000? We can simply apply a logical comparison across all the items in a vector:

```
release_year > 2000
```

The Breakfast Club: FALSE **American Beauty:** FALSE **Black Swan:** TRUE **Chicago:** TRUE

To retrieve the actual movie years after year 2000, you can simply subset the vector using the logical vector within **square brackets "[]"**:

```
release_year[movie_years > 2000] #returns a vector for elements that returned TRUE for the condition
```

Black Swan: 2010 **Chicago:** 2002 **3:** <NA>

As you may notice, subsetting vectors in R works by retrieving items that were TRUE for the provided condition. For example, `year[year > 2000]` can be verbally explained as: *"From the vector `year`, return only values where the values are TRUE for `year > 2000`"*.

You can even manually write out TRUE or T for the values you want to subset:

```
release_year
release_year[c(T, F, F, F)] #returns the values that are TRUE
```

The Breakfast Club: 1985 **American Beauty:** 1999 **Black Swan:** 2010 **Chicago:** 2002

The Breakfast Club: 1985

Factors

Factors are variables in R which take on a limited number of different values; such variables are often refered to as **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. For example, the height of a tree is a continuous variable, but the titles of books would be a categorical variable.

One of the most important uses of factors is in statistical modeling; since categorical variables enter into statistical models differently than continuous variables, storing data as factors insures that the modeling functions will treat such data correctly.

Let's start with a ***vector*** of genres:

Let's start with a ***vector*** of genres:

```
genre_vector <- c("Comedy", "Animation", "Crime", "Comedy", "Animation")
genre_vector
```

'Comedy' · 'Animation' · 'Crime' · 'Comedy' · 'Animation'

As you may have noticed, you can theoretically group the items above into three categories of genres: *Animation*, *Comedy* and *Crime*. In R-terms, we call these categories **"factor levels"**.

The function **factor()** converts a vector into a factor, and creates a factor level for each unique element.

```
genre_factor <- as.factor(genre_vector)
levels(genre_factor)
```

'Animation' · 'Comedy' · 'Crime'

Summarizing Factors

When you have a large vector, it becomes difficult to identify which levels are most common (e.g., "How many 'Comedy' movies are there?").

To answer this, we can use **summary()**, which produces a **frequency table**, as a named vector.

```
summary(genre_factor)
```

Animation: 2 **Comedy:** 2 **Crime:** 1

And recall that you can sort the values of the table using **sort()**.

```
sort(summary(genre_factor)) #sorts values by ascending order
```

Crime: 1 **Animation:** 2 **Comedy:** 2

Ordered factors

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

A **nominal variable** is a categorical variable for names, without an implied order. This means that it is impossible to say that 'one is better or larger than the other'. For example, consider **movie genre** with the categories *Comedy*, *Animation*, *Crime*, *Comedy*, *Animation*. Here, there is no implicit order of low-to-high or high-to-low between the categories.

In contrast, **ordinal variables** do have a natural ordering. Consider for example, **movie length** with the categories: *Very short*, *Short* , *Medium*, *Long*, *Very long*. Here it is obvious that *Medium* stands above *Short*, and *Long* stands above *Medium*.

```
movie_length <- c("Very Short", "Short", "Medium","Short", "Long",
                  "Very Short", "Very Long")
movie_length
```

'Very Short' · 'Short' · 'Medium' · 'Short' · 'Long' · 'Very Short' · 'Very Long'

`movie_length` should be converted to an ordinal factor since its categories have a natural ordering. By default, the function factor() transforms `movie_length` into an unordered factor.

To create an **ordered factor**, you have to add two additional arguments: `ordered` and `levels`.

- `ordered`: When set to `TRUE` in `factor()`, you indicate that the factor is ordered.
- `levels`: In this argument in `factor()`, you give the values of the factor in the correct order.

```
movie_length_ordered <- factor(movie_length, ordered = TRUE ,
                               levels = c("Very Short" , "Short" , "Medium",
                                          "Long","Very Long"))
movie_length_ordered
```

Very Short · Short · Medium · Short · Long · Very Short · Very Long

▶ **Levels**:

Now, lets look at the summary of the ordered factor, factor_mvlength_vector:

```
summary(movie_length_ordered)
```

Very Short: 2 **Short:** 2 **Medium:** 1 **Long:** 1 **Very Long:** 1

Vectors in R can be which of the following types?

- ◯ Logical
- ◯ Numeric
- ◯ Character
- ◉ All of the above ✔

What would be the output in R given: `c(1,2) == 1 ?`

- ◯ `FALSE TRUE`
- ◉ `TRUE FALSE` ✔
- ◯ `FALSE FALSE`
- ◯ `TRUE TRUE`

How would you retrieve the items larger than 5 (as in 15 and 10) from the following vector: `costs <- c(3, 15, 3, 10)?`

- ◯ `costs[15,10]`
- ◯ `costs[c(15,10)]`
- ◯ `costs(costs > 5)`
- ◉ `costs[costs > 5]` ✔
- ◯ `costs > 5`

2. Data Structures in R

Learning Objectives

Bookmark this page

Learning Objectives

In this lesson you will learn about:

- Arrays & Matrices
- Lists
- Dataframes

Arrays

```
movie_vector <- c("Akira", "Toy Story", "Room", "The Wave", "Whiplash",
                  "Star Wars", "The Ring", "The Artist", "Jumanji")
```

```
movie_array <- array(movie_vector, dim = c(4,3))
```

```
     [,1]         [,2]          [,3]
[1,] "Akira"      "Whiplash"    "Jumanji"
[2,] "Toy Story"  "Star Wars"   "Akira"
[3,] "Room"       "The Ring"    "Toy Story"
[4,] "The Wave"   "The Artist"  "Room"
```

Accessing an Array

```
     [,1]         [,2]          [,3]
[1,] "Akira"      "Whiplash"    "Jumanji"
[2,] "Toy Story"  "Star Wars"   "Akira"
[3,] "Room"       "The Ring"    "Toy Story"
[4,] "The Wave"   "The Artist"  "Room"
```

```
movie_array[1,2]
```

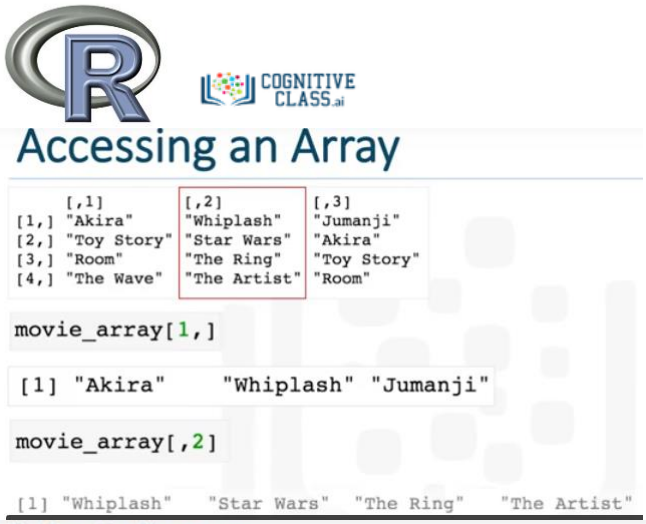
```
[1] "Whiplash"
```

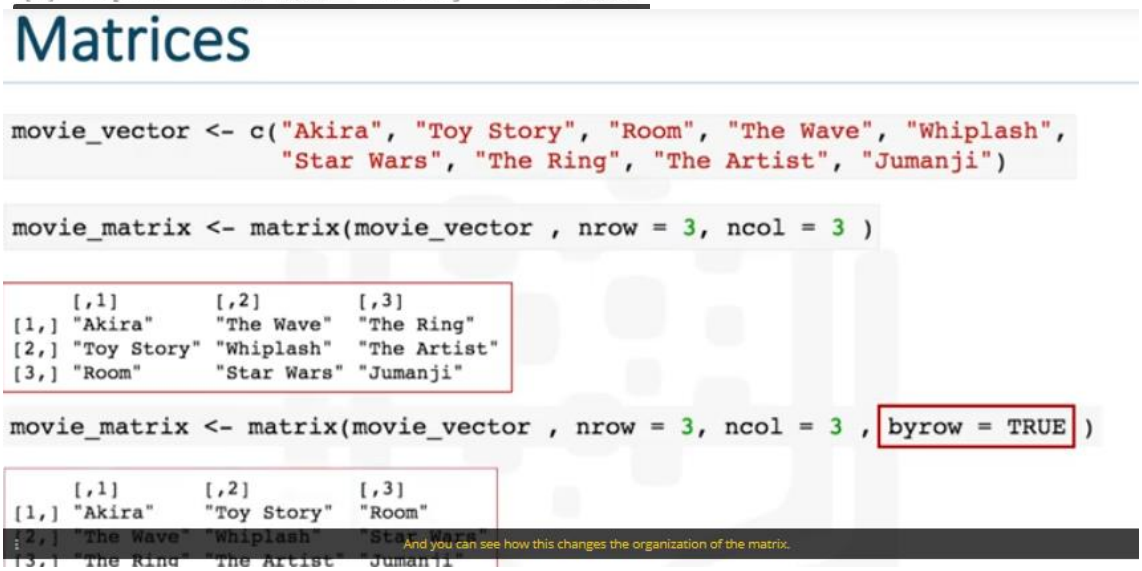
Accessing an Array

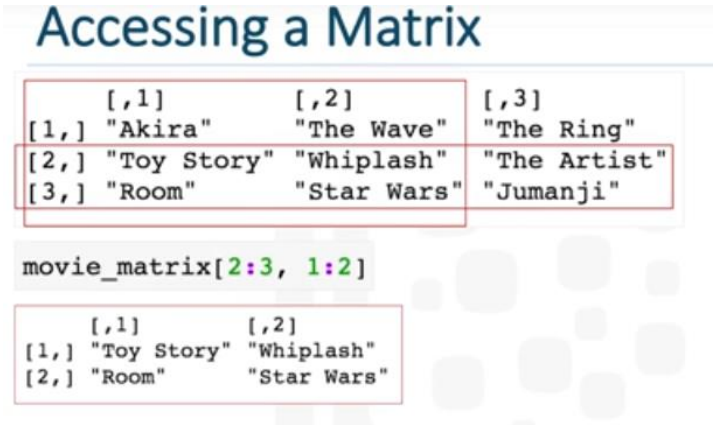
```
     [,1]         [,2]          [,3]
[1,] "Akira"      "Whiplash"    "Jumanji"
[2,] "Toy Story"  "Star Wars"   "Akira"
[3,] "Room"       "The Ring"    "Toy Story"
[4,] "The Wave"   "The Artist"  "Room"
```

```
movie_array[1,]
```

```
[1] "Akira"     "Whiplash"  "Jumanji"
```







In this video, we will show you how to work with arrays and matrices in the R programming language.
An array is a structure that contains data of the same type, whether that's strings, or characters, or integers.
Arrays can be multi-dimensional as well, so the data can be contained in multiple rows and columns.
In order to create an array, we?ll first create a vector using the ?c? command.
Afterwards, we need to use the "array" function.
The first argument is the vector we just created.

The second argument is a vector that specifies the dimension.
In this case, our array will have four rows, and three columns.
Notice that the elements start to repeat at row 2, column 3.
This is because our original vector only had 9 elements, but the array size is four times three, which is twelve.
To access an element in an array, we need to specify the element's index.
So for example, say we wanted to extract "Whiplash".
"Whiplash" is in the first row, and the second column.
All we need to do is specify the row and column in square brackets, to get that particular element.
We can also extract an entire row from the matrix.
To do so, simply type the row you want in the square brackets, but leave the column empty.
As you can see, the output is the entire first row.
The same can be done for the columns.
Leave the row empty, and type the column you want to extract.
In this case, we've extracted column two of the array.
A matrix is similar in structure to an array.
The main difference is that a matrix must be two dimensional.
Starting with the "movie vector" from before, we can use the "matrix" function while specifying
three arguments.
The first argument is simply the vector, while the next two arguments are the number of rows and columns, respectively.
So in our case, we're building a three by three matrix.
One thing to note is that by default, the matrix is arranged by columns, rather than by rows.
We can change this by adding the parameter byrow = TRUE?.
And you can see how this changes the organization of the matrix.
You can access certain subsets of the matrix with the proper notation.
Consider the piece of code here.
This is telling R to extract the values that are in rows 2 to 3, and in the columns from 1 to 2.
The output is the subset of the array whose elements fall into this row and column range.
By now, you should understand how to create an array or a matrix, and how to access the data.
Thank you for watching this video.
End of transcript. Skip to the start.

>> Lab:

About the Dataset

Imagine you got many movie recomendations from your friends and compiled all of the recomendations in a table, with specific info a each movie.

The table has one row for each movie and several columns

- **name** - The name of the movie
- **year** - The year the movie was released
- **length_min** - The lenght of the movie in minutes
- **genre** - The genre of the movie
- **average_rating** - Average rating on Imdb
- **cost_millions** - The movie's production cost in millions
- **sequences** - The amount of sequences
- **foreign** - Indicative of whether the movie is foreign (1) or domestic (0)
- **age_restriction** - The age restriction for the movie

name	year	length_min	genre	average_rating	cost_millions	foreign	age_restriction
Toy Story	1995	81	Animation	8.3	30	0	0
Akira	1998	125	Animation	8.1	10.4	1	14
The Breakfast Club	1985	97	Drama	7.9	1	0	14
The Artist	2011	100	Romance	8	15	1	12
Modern Times	1936	87	Comedy	8.6	1.5	0	10
Fight Club	1999	139	Drama	8.9	63	0	18
City of God	2002	130	Crime	8.7	3.3	1	18
The Untouchables	1987	119	Drama	7.9	25	0	14
Star Wars	1977	121	Action	8.7	11	0	10
American Beauty	1999	122	Drama	8.4	15	0	14
Room	2015	118	Drama	8.3	13	1	14
Dr. Strangelove	1964	94	Comedy	8.5	1.8	1	10
The Ring	1998	95	Horror	7.3	1.2	1	18
Monty Python and the Holy Grail	1975	91	Comedy	8.3	0.4	1	18
High School Musical	2006	98	Comedy	5.2	4.2	0	0
Shaun of the Dead	2004	99	Horror	8	6.1	1	18
Taxi Driver	1976	113	Crime	8.3	1.3	1	14
The Shawshank Redemption	1994	142	Crime	9.3	25	0	16

What is an Array?

An array is a structure that holds values grouped together, like a 2 x 2 table of 2 rows and 2 columns. Arrays can also be **multidimensional**, such as a 2 x 2 x 2 array.

What is the difference between an array and a vector?

Vectors are always one dimensional like a single row of data. On the other hand, an array can be multidimensional (stored as rows and columns). The "dimension" indicates how many rows of data there are.

Let's create a 4 x 3 array (4 rows, 3 columns)

The example below is a vector of 9 movie names, hence the data type is the same for all the elements.

```
#lets first create a vector of nine movies
movie_vector <- c("Akira", "Toy Story", "Room", "The Wave", "Whiplash",
                  "Star Wars", "The Ring", "The Artist", "Jumanji")
movie_vector
```

'Akira' · 'Toy Story' · 'Room' · 'The Wave' · 'Whiplash' · 'Star Wars' · 'The Ring' · 'The Artist' · 'Jumanji'

To create an array, we can use the **array()** function.

```
movie_array <- array(movie_vector, dim = c(4,3))
movie_array
```

A matrix: 4 × 3 of type chr

Akira	Whiplash	Jumanji
Toy Story	Star Wars	Akira
Room	The Ring	Toy Story
The Wave	The Artist	Room

Note that **arrays are created column-wise**. Did you also notice that there were only 9 movie names, but the array was 4 x 3? The original **vector doesn't have enough elements** to fill the entire array (that should have 3 x 4 = 12 elements). So R simply fills rest of the empty values by going back to the beginning of the vector and starting again ("Akira", "Toy story", "Room" in this case).

We also needed to provide `c(4,3)` as a second *argument* to specify the number of rows (4) and columns (3) that we wanted.

[Tip] What is an "argument"? How are "arguments" different from "_parameters_"?
Arguments and parameters are terms you will hear constantly when talking about **functions**. - The _**parameters**_ are the input variables used in a function, like **dim** in the function **array()**. - The _**arguments**_ refer to the _values_ for those parameters that a function takes as inputs, like **c(4,3)**
We actually don't need to write out the name of the parameter (dim) each time, as in: `array(movie_vector, c(4,3))` As long as we write the arguments out in the correct order, R can interpret the code.
Arguments in a function may sometimes need to be of a **specific type**. For more information on each function, you can open up the help file by running the function name with a ? beforehand, as in: `?array`

Array Indexing

Let's look at our array again:

```
movie_array
```

A matrix: 4 × 3 of type chr

Akira	Whiplash	Jumanji
Toy Story	Star Wars	Akira
Room	The Ring	Toy Story
The Wave	The Artist	Room

To access an element of an array, we should pass in **[row, column]** as the row and column number of that element. For example, here we retrieve **Whiplash** from row 1 and column 2:

```
movie_array[1,2] #[row, column]
```

'Whiplash'

To display all the elements of the first row, we should put 1 in the row and nothing in the column part. Be sure to keep in the comma after the `1`.

```
movie_array[1,]
```

'Akira' · 'Whiplash' · 'Jumanji'

Likewise, you can get the elements by column as below.

```
movie_array[,2]
```

'Whiplash' · 'Star Wars' · 'The Ring' · 'The Artist'

To get the dimension of the array, **dim()** should be used.

```
dim(movie_array)
```

4 · 3

We can also do math on arrays. Let's create an array of the lengths of each of the nine movies used earlier.

```
length_vector <- c(125, 81, 118, 81, 106, 121, 95, 100, 104)
length_array <- array(length_vector, dim = c(3,3))
length_array
```

A matrix: 3 × 3
of type dbl

125	81	95
81	106	100
118	121	104

Let's add 5 to the array, to account for a 5-min bathroom break:

```
length_array + 5
```

A matrix: 3 × 3
of type dbl

130	86	100
86	111	105
123	126	109

Tip: Performing operations on objects, like adding 5 to an array, does not change the object. **To change the object, we would need to assign the new result to itself**.

Using Logical Conditions to Subset Arrays

Which movies can I finish watching in two hours? Using a logical condition, we can check which movies are less than 2 hours long.

```
mask_array <- length_array > 120
mask_array
```

A matrix: 3 × 3 of type lgl

TRUE	FALSE	FALSE
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE

Using this array of TRUEs and FALSEs, we can subset the array of movie names:

```
x_vector <- c("Akira", "Toy Story", "Room", "The Wave", "Whiplash",
              "Star Wars", "The Ring", "The Artist", "Jumanji")
x_array <- array(x_vector, dim = c(3,3))

x_array[mask_array]
```

'Akira' · 'Star Wars'

What is a Matrix?

Matrices are a subtype of arrays. A matrix **must** have 2 dimensions, whereas arrays are more flexible and can have, 1, 2 or more dimensions

To create a matrix out of a vector , you can use **matrix()**, which takes in an argument for the vector, an argument for the number of rows and another for the number of columns.

```
movie_matrix <- matrix(movie_vector, nrow = 3, ncol = 3)
```

```
movie_matrix
```

A matrix: 3 × 3 of type chr

Akira	The Wave	The Ring
Toy Story	Whiplash	The Artist
Room	Star Wars	Jumanji

Accessing elements of a matrix

As with arrays, you can use **[row, column]** to access elements of a matrix. To retrieve "Akira", you should use [1,1] as it lies in the first row and first column.

```
movie_matrix[1,1]
```

'Akira'

To get data from a certain range, the folowing code can help. This takes the elements from rows 2 to 3, and from columns 1 to 2.

```
movie_matrix[2:3, 1:2]
```

A matrix: 2 × 2 of type chr

Toy Story	Whiplash
Room	Star Wars

Concatenation function

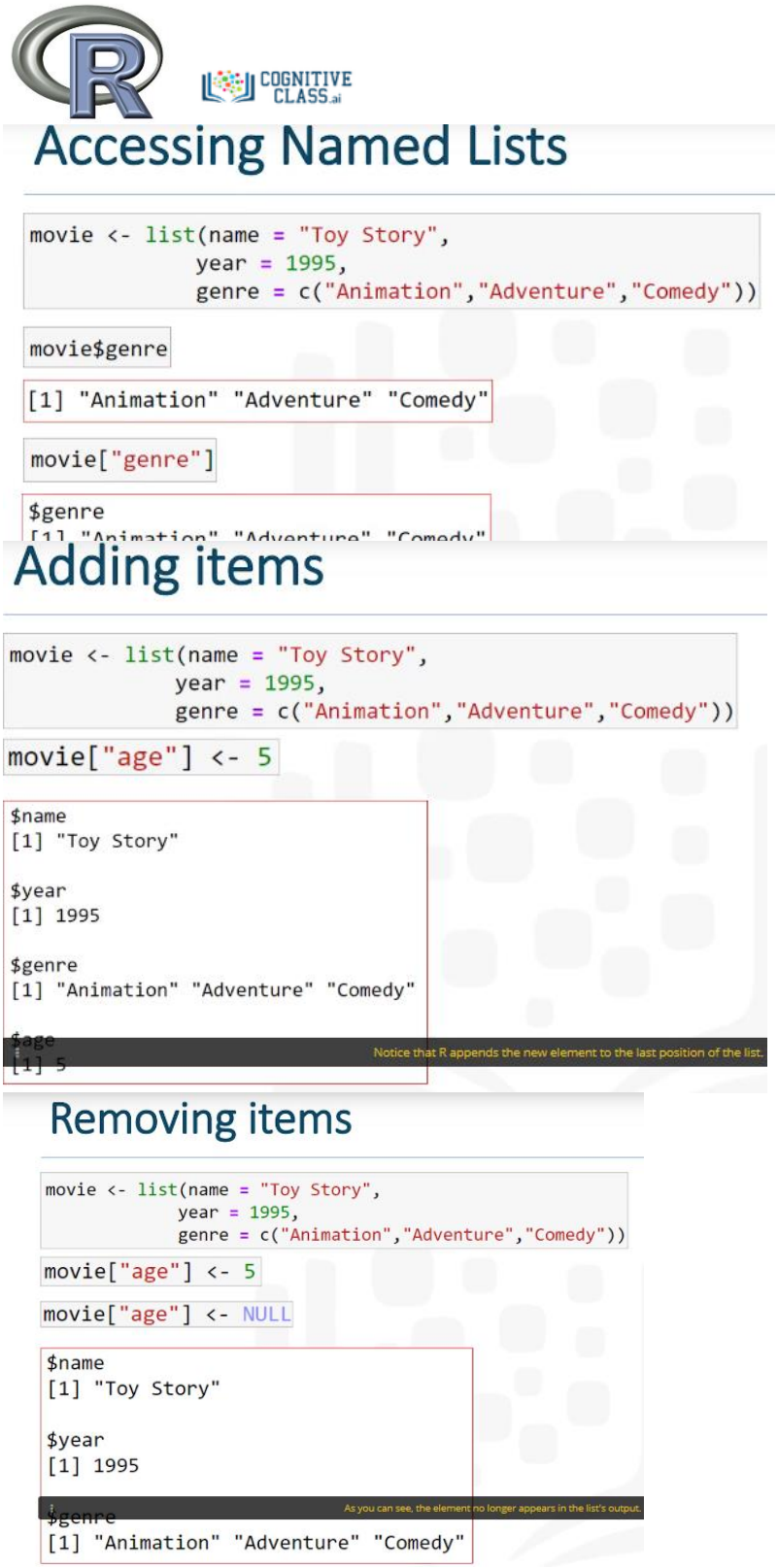
A concatenation function is used to combine two vectors into one vector. It combines values of both vectors.
Lets create a new vector for the upcoming movies as upcoming_movie and add them to the movie_vector to create a new_vector of movies.

```
upcoming_movie <- c("Fast and Furious 8", "xXx: Return of Xander Cage", "Suicide Squad")
```

```
new_vector <- c(movie_vector, upcoming_movie)
```

```
new_vector
```

'Akira' · 'Toy Story' · 'Room' · 'The Wave' · 'Whiplash' · 'Star Wars' · 'The Ring' · 'The Artist' · 'Jumanji' · 'Fast and Furious 8' · 'xXx: Return of Xander Cage' · 'Suicide Squad'



In this video, we will show you how to create a list in the R programming language, and how to access and modify the list elements.
In R, a list is a collection of objects, similar to a vector.
But unlike a vector, the elements inside of a list can differ in terms of data type.
So consider the piece of code here.
We use the "list" function and specify the elements that we want to include.
Notice that the first element is a string, the second is a number, and the third is a vector.

Accessing an element of a list is simple.
So for example, say we wanted to access element two, which is the value 1995.
All we need to do is type the index of the element inside the square brackets.
This will retrieve the desired element.
We can also access all the elements within a given range by specifying the start and end points.
Here, we access all elements between index 2 and index 3.
You can also provide each of the individual variables with a name.
This improves the list's organization and makes everything a bit easier to read.
In order to access a list element by name, you can use the "dollar sign" symbol, followed by the name of the element you want to access.
Alternatively, you can provide a string that contains the name, inside the square brackets.
This is similar to how you'd access an element by its numerical index.
But notice that both methods produce the same result.
You can add a new element to the list by using the variable assignment operator.
Simply provide the name of the new element inside the square brackets.
Notice that R appends the new element to the last position of the list.
To modify a value of the list, simply use the variable assignment operator to overwrite an existing value.
If you'd like to remove an element from the list, all you need to do is assign a "NULL" value to it.
As you can see, the element no longer appears in the list's output.
By now, you should understand how to create a list, as well as how to add, modify, and remove items.
Thank you for watching this video.
End of transcript. Skip to the start.

Data Frames (3:41)

Accessing Data Frames

```
                 name year
1           Toy Story 1995
2               Akira 1998
3 The Breakfast Club 1985
4           The Artist 2011
5         Modern Times 1936
6           Fight Club 1999
7          City of God 2002
8   The Untouchables 1987
```

```
movies$name
```

```
[1] Toy Story           Akira               The Breakfast Club The Artist
[5] Modern Times        Fight Club          City of God        The Untouchables
8 Levels: Akira City of God Fight Club Modern Times ... Toy Story
```

So in this case, we will retrieve all the names of the data frame.

Accessing Data Frames

```
                 name year
1           Toy Story 1995
2               Akira 1998
3 The Breakfast Club 1985
4           The Artist 2011
5         Modern Times 1936
6           Fight Club 1999
7          City of God 2002
8   The Untouchables 1987
```

```
                 name
1           Toy Story
2               Akira
3 The Breakfast Club
4           The Artist
5         Modern Times
6           Fight Club
7          City of God
8   The Untouchables
```

```
movies[1]
```

Accessing Data Frames

```
                 name year
1           Toy Story 1995
2               Akira 1998
3 The Breakfast Club 1985
4           The Artist 2011
5         Modern Times 1936
6           Fight Club 1999
7          City of God 2002
8   The Untouchables 1987
```

```
movies[1,2]
```

```
[1] 1995
```

Data Frame Structure

```
                  name year
1          Toy Story 1995
2              Akira 1998
3 The Breakfast Club 1985
4         The Artist 2011
5       Modern Times 1936
6         Fight Club 1999
7        City of God 2002
8   The Untouchables 1987
```

```
str(movies)
```

```
'data.frame':   8 obs. of  2 variables:
 $ name: chr  "Toy Story" "Akira" "The Breakfast Club" "The Artist" ...
 $ year: num  1995 1998 1985 2011 1936 ...
```

In the output, we can see the number of objects and variables, as well as the variables' type

Head and Tail

```
                  name year
1          Toy Story 1995
2              Akira 1998
3 The Breakfast Club 1985
4         The Artist 2011
5       Modern Times 1936
6         Fight Club 1999
7        City of God 2002
8   The Untouchables 1987
```

```
head(movies)
```

```
                  name year
1          Toy Story 1995
2              Akira 1998
3 The Breakfast Club 1985
4         The Artist 2011
5       Modern Times 1936
6         Fight Club 1999
```

```
tail(movies)
```

```
                  name year
3 The Breakfast Club 1985
4         The Artist 2011
5       Modern Times 1936
6         Fight Club 1999
7        City of God 2002
8   The Untouchables 1987
```

Inserting a new row

```
                  name year
1          Toy Story 1995
2              Akira 1998
3 The Breakfast Club 1985
4         The Artist 2011
5       Modern Times 1936
6         Fight Club 1999
7        City of God 2002
8   The Untouchables 1987
```

```
                  name year length
1          Toy Story 1995     81
2              Akira 1998    125
3 The Breakfast Club 1985     97
4         The Artist 2011    100
5       Modern Times 1936     87
6         Fight Club 1999    139
7        City of God 2002    130
8   The Untouchables 1987    119
9    Dr. Strangelove 1964     94
```

```
movies <- rbind(movies, c(name="Dr. Strangelove", year=1964, length=94))
```

Deleting Rows

	name	year	length
1	Toy Story	1995	81
2	Akira	1998	125
3	The Breakfast Club	1985	97
4	The Artist	2011	100
5	Modern Times	1936	87
6	Fight Club	1999	139
7	City of God	2002	130
8	The Untouchables	1987	119
9	Dr. Strangelove	1964	94

	name	year	length
1	Toy Story	1995	81
2	Akira	1998	125
3	The Breakfast Club	1985	97
4	The Artist	2011	100
5	Modern Times	1936	87
6	Fight Club	1999	139
7	City of God	2002	130
8	The Untouchables	1987	119

```
movies <- movies[-9,]
```

Deleting Columns

	name	year	length
1	Toy Story	1995	81
2	Akira	1998	125
3	The Breakfast Club	1985	97
4	The Artist	2011	100
5	Modern Times	1936	87
6	Fight Club	1999	139
7	City of God	2002	130
8	The Untouchables	1987	119

	name	year
1	Toy Story	1995
2	Akira	1998
3	The Breakfast Club	1985
4	The Artist	2011
5	Modern Times	1936
6	Fight Club	1999
7	City of God	2002
8	The Untouchables	1987

```
movies["length"] <- NULL
```

In this video, we will show you how to create a data frame in the R programming language, as well as how to access data and make changes.
A data frame is a type of structure that contains correlated information.
So for example, a data frame would be a great structure for storing these movie titles along with their corresponding years.
We need to use the "data.frame" function, and each argument is a vector that represents one of the columns.
Each vector should contain data of the same type, so in our case, we have one vector with numbers, and one vector with strings.
The variables of a data frame can be accessed using the "dollar sign" symbol.
So in this case, we will retrieve all the names of the data frame.
In addition to the "dollar sign" symbol, we can access a column by specifying the column number inside of the square brackets.
So in this case we again retrieve the names column since it's the first column of the data frame.
Individual elements can be accessed as well, rather than entire columns.
So for example, say we wanted to access the element in row 1, column 2.
All we need to do is write the row and column numbers inside the square brackets, like so.
And as you can see, we get the desired output.
To get some information about the data frame's structure, you can pass the data frame as an argument to the "str" function.
In the output, we can see the number of objects and variables, as well as the variables' type and content.

The "head" and "tail" function can be used to look at the beginning and end of a data frame, respectively.
The "head" function will display the first six elements, as you can see here.
The "tail" function will similarly show us the last six elements of the data frame.
If you want to insert a new column into the data frame, simply specify the new column?s name in square brackets, and assign to it a vector that contains the individual values.
You can see here that we?ve added a new column for "length".
In order to insert a new row, we need to use the "rbind" function and assign the output back to our data frame.
The arguments are the data frame itself, as well as a vector that contains values for all the columns.
As you can see we've increased the number of elements in the data frame from 8 to 9.
You can delete rows from a data frame by using negative numbers inside the square brackets.
You can see here that we use -9 for the row, and we leave the column blank.
The output will be a data frame that only has rows 1 through 8, and we then have to reassign it to the original data frame variable.
If you look at the result, you?ll notice that the ninth movie has been removed.
In order to delete a column, like the length column here for example, all we need to do is assign a NULL value, like so.
As you can see, this will remove that column from the data frame.
By now, you should understand how to create and access a data frame, as well as how to make modifications.
Thank you for watching this video.
End of transcript. Skip to the start.

>> Lab:

The table has one row for each movie and several columns

- **name** - The name of the movie
- **year** - The year the movie was released
- **length_min** - The lenght of the movie in minutes
- **genre** - The genre of the movie
- **average_rating** - Average rating on Imdb
- **cost_millions** - The movie's production cost in millions
- **sequences** - The amount of sequences
- **foreign** - Indicative of whether the movie is foreign (1) or domestic (0)
- **age_restriction** - The age restriction for the movie

Part of the dataset can be seen below

name	year	length_min	genre	average_rating	cost_millions	foreign	age_restriction
Toy Story	1995	81	Animation	8.3	30	0	0
Akira	1998	125	Animation	8.1	10.4	1	14
The Breakfast Club	1985	97	Drama	7.9	1	0	14
The Artist	2011	100	Romance	8	15	1	12
Modern Times	1936	87	Comedy	8.6	1.5	0	10
Fight Club	1999	139	Drama	8.9	63	0	18
City of God	2002	130	Crime	8.7	3.3	1	18
The Untouchables	1987	119	Drama	7.9	25	0	14
Star Wars	1977	121	Action	8.7	11	0	10
American Beauty	1999	122	Drama	8.4	15	0	14
Room	2015	118	Drama	8.3	13	1	14
Dr. Strangelove	1964	94	Comedy	8.5	1.8	1	10
The Ring	1998	95	Horror	7.3	1.2	1	18
Monty Python and the Holy Grail	1975	91	Comedy	8.3	0.4	1	18
High School Musical	2006	98	Comedy	5.2	4.2	0	0
Shaun of the Dead	2004	99	Horror	8	6.1	1	18
Taxi Driver	1976	113	Crime	8.3	1.3	1	14

Lists

First of all, we're gonna take a look at lists in R. A list is a sequenced collection of different objects of R, like vectors, numbers, characters, other lists as well, and so on. You can consider a list as a container of correlated information, well structured and easy to read. A list accepts items of different types, but a vector (or a matrix, which is a multidimensional vector) doesn't. To create a list just type **list()** with your content inside the parenthesis and separated by commas. Let's try it!

```
movie <- list("Toy Story", 1995, c("Animation", "Adventure", "Comedy"))
```

In the code above, the variable movie contains a list of 3 objects, which are a string, a numeric value, and a vector of strings. Easy, eh? Now let's print the content of the list. We just need to call its name.

```
movie
```

1. 'Toy Story'
2. 1995
3. 'Animation' · 'Adventure' · 'Comedy'

A list has a sequence and each element of a list has a position in that sequence, which starts from 1. If you look at our previous example, you can see that each element has its position represented by double square brackets "**[[ ]]**".

Accessing items in a list

It is possible to retrieve only a part of a list using the **single _square** bracket operator_ "**[ ]**". This operator can be also used to get a single element in a specific position. Take a look at the next example:

The index number 2 returns the second element of a list, if that element exists:

```
movie[2]
```

1. 1995

Or you can select a part or interval of elements of a list. In our next example we are retrieving the 1st, 2nd, and 3rd elements:

```
movie[2:3]
```

1. 1995
2. 'Animation' · 'Adventure' · 'Comedy'

It looks a little confusing, but lists can also store names for its elements.

Named lists

The following list is a named list:

```
movie <- list(name = "Toy Story",
              year = 1995,
              genre = c("Animation", "Adventure", "Comedy"))
```

Let me explain that: the list **movie** has some named objects within it. **name**, for example, is an object of type **character**, **year** is an object of type **number**, and **genre** is a vector with objects of type **character**.

Now take a look at this list. This time, it's full of information and well organized. It's clear what each element means. You can see that the elements have different types, and that's ok because it's a list.

```
movie
```

$name	'Toy Story'
$year	1995
$genre	'Animation' · 'Adventure' · 'Comedy'

You can also get separated information from the list. You can use **listName$selectorName**. The *dollar-sign operator* **$** will give you the block of data that is related to selectorName.

Let's get the genre part of our movies list, for example.

```
movie$genre
```

'Animation' · 'Adventure' · 'Comedy'

Another way of selecting the genre column:

```
movie["genre"]
```

$genre =
'Animation' · 'Adventure' · 'Comedy'

You can also use numerical selectors like an array. Here we are selecting elements from 2 to 3.

```
movie[2:3]
```

$year	1995
$genre	'Animation' · 'Adventure' · 'Comedy'

The function **class()** returns the type of a object. You can use that function to retrieve the type of specific elements of a list:

```
class(movie$name)
class(movie$foreign)
```

'character'
'NULL'

Adding, modifying, and removing items

Adding a new element is also very easy. The code below adds a new field named **age** and puts the numerical value 0 into it. In this case we use the double square brackets operator, because we are directly referencing a list member (and we want to change its content).

```
movie[["age"]] <- 5
movie
```

$name	'Toy Story'
$year	1995
$genre	'Animation' · 'Adventure' · 'Comedy'
$age	5

In order to modify, you just need to reference a list member that already exists, then change its content.

```
movie[["age"]] <- 6
# Now it's 6, not 5
movie
```

$name 'Toy Story'
$year 1995
$genre 'Animation' · 'Adventure' · 'Comedy'
$age 6

And removing is also easy! You just put ***NULL***, which means missing value/data, into it.

```
movie[["age"]] <- NULL
movie
```

$name 'Toy Story'
$year 1995
$genre 'Animation' · 'Adventure' · 'Comedy'

Concatenating lists

Concatenation is the proccess of puting things together, in sequence. And yes, you can do it with lists. Just call the function ***c()***. Take a look at the next example:

```
# We split our previous list in two sublists
movie_part1 <- list(name = "Toy Story")
movie_part2 <- list(year = 1995, genre = c("Animation", "Adventure", "Comedy"))

# Now we call the function c() to put everything together again
movie_concatenated <- c(movie_part1, movie_part2)

# Check it out
movie_concatenated
```

$name 'Toy Story'
$year 1995
$genre 'Animation' · 'Adventure' · 'Comedy'

Lists are really handy for organizing different types of elements in R, and also easy to use. Additionally, lists are also important since this type of data structure is essential to create data frames, our next covered topic.

DataFrames

A DataFrame is a structure that is used for storing data tables. Underneath it all, a data frame is a list of vectors of same length, exactly like a table (each vector is a column). We call a function called **data.frame()** to create a data frame and pass vector, which are our columns, as arguments. It is required to name the columns that will compose the data frame.

```
movies <- data.frame(name = c("Toy Story", "Akira", "The Breakfast Club", "The Artist",
                              "Modern Times", "Fight Club", "City of God", "The Untouchables"),
                     year = c(1995, 1998, 1985, 2011, 1936, 1999, 2002, 1987),
                     stringsAsFactors=F)
```

Let's print its content of our recently created data frame:

```
movies
```

A data.frame: 8 × 2

name	year
<chr>	<dbl>
Toy Story	1995
Akira	1998
The Breakfast Club	1985
The Artist	2011
Modern Times	1936
Fight Club	1999

We can also use the **"$"** selector to get some type of information. This operator returns the content of a specific column of a data frame (that's why we have to choose a name for each column).

```
movies$name
```

'Toy Story' · 'Akira' · 'The Breakfast Club' · 'The Artist' · 'Modern Times' · 'Fight Club' · 'City of God' · 'The Untouchables'

You retrieve data using numeric indexing, like in lists:

```
# This returns the first (1st) column
movies[1]
```

A data.frame: 8 × 1

name
<chr>
Toy Story
Akira
The Breakfast Club
The Artist

The function called **str()** is one of most useful functions in R. With this function you can obtain textual information about an object. In this case, it delivers information about the objects whitin a data frame. Let's see what it returns:

```
str(movies)
```

```
'data.frame':   8 obs. of  2 variables:
 $ name: chr  "Toy Story" "Akira" "The Breakfast Club" "The Artist" ...
 $ year: num  1995 1998 1985 2011 1936 ...
```

It shouws this data frame has 8 observations, for 2 columns, so called **name** and **year**. The "name" column is a factor with 8 levels and "year" is a numerical column.

The class() function works for data frames as well. You can use it to determine the type of a column of a data frame.

```
class(movies$year)
```

'numeric'

You can use numerical selectors to reach information inside the table.

```
movies[1,2] #1-Toy Story, 2-1995
```

1995

The ***head()*** function is very useful when you have a large table and you need to take a peek at the first elements. This function returns the first 6 values of a data frame (or event a list).

```
head(movies)
```

A data.frame: 6 × 2

	name	year
	<chr>	<dbl>
1	Toy Story	1995
2	Akira	1998
3	The Breakfast Club	1985
4	The Artist	2011
5	Modern Times	1936
6	Fight Club	1999

Similar to the previous function, ***tail()*** returns the last 6 values of a data frame or list.

```
tail(movies)
```

A data.frame: 6 × 2

	name	year
	<chr>	<dbl>
3	The Breakfast Club	1985
4	The Artist	2011
5	Modern Times	1936
6	Fight Club	1999
7	City of God	2002
8	The Untouchables	1987

Now let's try to add a new column to our data frame with the length of each movie in minutes.

```
movies['length'] <- c(81, 125, 97, 100, 87, 139, 130, 119)
movies
```

A data.frame: 8 × 3

name	year	length
<chr>	<dbl>	<dbl>
Toy Story	1995	81
Akira	1998	125
The Breakfast Club	1985	97
The Artist	2011	100
Modern Times	1936	87
Fight Club	1999	139
City of God	2002	130
The Untouchables	1987	119

A new column was included into our data frame with just one line of code. We just needed to add a vector to data frame, then it will be our new column.

Now let's try to add a new movie to our data set.

```
movies <- rbind(movies, c(name="Dr. Strangelove", year=1964, length=94))
movies
```

A data.frame: 9 × 3

name	year	length
<chr>	<chr>	<chr>
Toy Story	1995	81
Akira	1998	125
The Breakfast Club	1985	97
The Artist	2011	100
Modern Times	1936	87
Fight Club	1999	139
City of God	2002	130
The Untouchables	1987	119
Dr. Strangelove	1964	94

Remember, you can't add a list with more variables than the data frame, and vice-versa.

We don't need this movie anymore, so let's delete it. Here we are deleting row 12 by assigning to itself the movies dataframe without the 12th row.

```
movies <- movies[-12,]
movies
```

A data.frame: 9 × 3

	name	year	length
	<chr>	<chr>	<chr>
1	Toy Story	1995	81
2	Akira	1998	125
3	The Breakfast Club	1985	97
4	The Artist	2011	100
5	Modern Times	1936	87
6	Fight Club	1999	139
7	City of God	2002	130
8	The Untouchables	1987	119
9	Dr. Strangelove	1964	94

To delete a column you can just set it as ***NULL***.

```
movies[["length"]] <- NULL
movies
```

A data.frame: 9 × 2

	name	year
	<chr>	<chr>
1	Toy Story	1995
2	Akira	1998
3	The Breakfast Club	1985
4	The Artist	2011

That is it! You learned a lot about data frames and how easy it is to work with them.

Below we create a list for a student and his info. Select all the correct options can we use to retrieve his courses?

```
john <- list("studentid" = 9, "age" = 18, "courses" = c("Data Science 101", "Data Science Methodology"))
```

- ◯ `john["courses"]`
- ◯ `john[3]`
- ◯ `john$courses`
- ◉ All the above options are correct ✔

Select the correct code from the following options which produces the following result?

```
  student id
1    john  1
2    mary  2
```

- ◉ `data.frame("student" = c("john", "mary"), "id" = c(1, 2))` ✔
- ◯ `array("student" = c("john", "mary"), "id" = c(1, 2))`
- ◯ `data.frame(c("john", "mary"), c(1, 2))`
- ◯ `data.frame(student = c(john, mary), id = c(1, 2))`
- ◯ `list("student" = c("john", "mary"), "id" = c(1, 2))`

3. R programming fundamentals

Learning Objectives

In this lesson you will learn about:

- Conditions and loops
- Functions in R
- Objects and Classes
- Debugging

Conditional Statements

name	year
Toy Story	1995
Akira	1998
The Breakfast Club	1985
The Artist	2011
Modern Times	1936
Fight Club	1999
City of God	2002
The Untouchables	1987
Star Wars Episode IV	1977
American Beauty	1999
Room	2015
Dr. Strangelove	1964
The Ring	1998
Monty Python and the Holy Grail	1975
High School Musical	2006
Shaun of the Dead	2004

In other words, we need to examine the datase

If Statements

```
movie_year <- 2002

if(movie_year > 2000){

    print('Movie year is greater than 2000')

}
```

If Statements

```
movie_year <- 1997

if(movie_year > 2000){

    print('Movie year is greater than 2000')

} else {

    print('Movie year is not greater than 2000')

}
```

```
[1] "Movie year is not greater than 2000"
```

Logical Operators

```
1995 < 1987
```

```
[1] FALSE
```

```
20016 >= 2015
```

```
[1] TRUE
```

True False

```
"Toy Story" == "Toy Story" & 1995 < 1987
```

```
[1] FALSE
```

Comparison and Logical Operators

Operator	Meaning
==	Is equal to?
!=	Is not equal to?
>	Greater than?
<	Less than?
>=	Greater than or equal to?
<=	Less than or equal to?
&	And
|	Or
!	Not

You can look here for an overview of the comparison and logical operators in R. The "or" operator

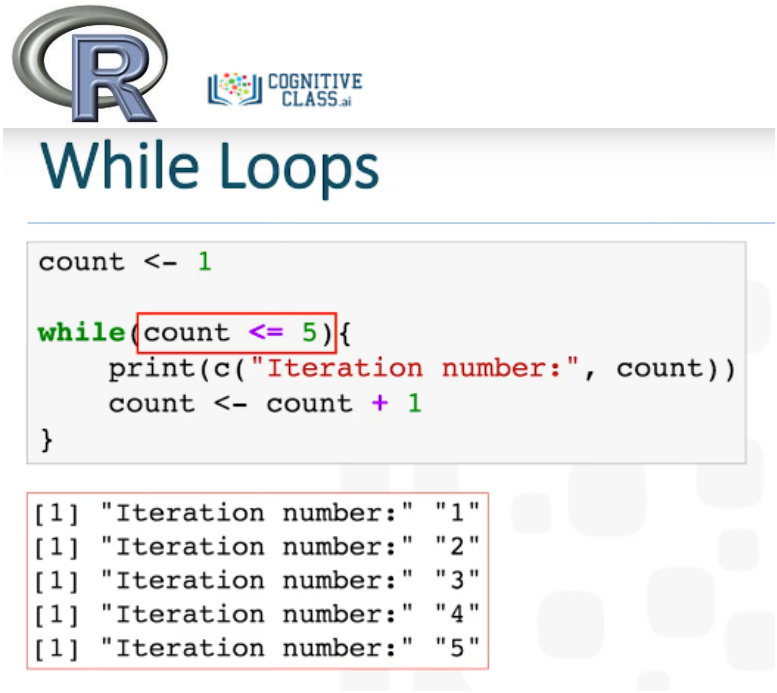
For Loops

```
years <- c(1995, 1998, 1985, 2011, 1936, 1999)

for (yr in years) {
    print(yr)
}
```

```
[1] 1995
[1] 1998
[1] 1985
[1] 2011
[1] 1936
[1] 1999
```

```
for (yr in years) {

    if(yr < 1980) {
        print("Old movie")
    } else {
        print("Not that old")
    }

}
```

We can also combine the "for" loop with the "if else" block from before.



In this video, we will show you how to utilize conditional statements, "for" loops, and "while" loops, in the R programming language.
Suppose that you have a group of movies in a dataset, like the one you see here, and you want to only select the movies released after the year 2000.
To do this, we're going to have to rely on a conditional statement.
In other words, we need to examine the dataset row by row, and check if the year value is greater than 2000 or not.
In R, we can use an 'if" statement to accomplish this.
Notice the syntax here.
Inside the parentheses we have the conditional statement.
If this statement is true, then we execute the code inside the curly braces.
Since 2002 is greater than 2000, we do see the printed statement in the output.
You can also add an "else" block to your "if" statement.
The code inside the "else" block will only execute if the conditional in the "if" statement is false.
Notice that since 1997 is less than 2000, the print statement inside the "else" block is executed.
Logical operators are used to compare two values, and the output is either "true" or "false".
We've already seen the "greater than" operator, but of course there's a "less than" operator as well.
Putting an "equals sign" after the "greater than" sign will make a "greater than or equal to" operator.
The same is true for the "less than" sign as well.
In R, a single "equals sign is used for variable assignment, so to check if two values are equal, we need to use a double "equals sign".
To check if two values are not equal, we write an "exclamation point" followed by an "equals sign".
We can also combine multiple conditional statements together.
Notice the ampersand that separates the statement on the left from the statement on the right.
This is the "and" operator, which will return true only if both of the connecting statements are true.
So let's see what the output should be.

The statement on the left is "true" since both strings are the same, but the statement on the right is false.
So the final output of the "and" statement will be "false".
You can look here for an overview of the comparison and logical operators in R. The "or" operator
works similarly to the "and" operator, except that to return true, only one statement needs to be true, rather than both.
The "not" operator is used to negate a Boolean value from true to false, or vice versa.
And the "in" operator checks if one operand is contained in the other.
"For" loops can be used to cycle through all the values in a vector.
Take a look at the code snippet here.
This "for" loop will run once for each item of the "years" vector, in order.
At each iteration, we can access the current value using the "yr" variable that we defined.
By printing this variable at each iteration, we end up outputting the entire vector.
We can also combine the "for" loop with the "if else" block from before.
Notice at each iteration of the "for" loop, we check if the movie year is less than 1980, and we change our output depending on the result.
You can see how that works in the output.
"While" loops work a bit differently.
A "while" loop will continue to execute so long as the condition inside the parentheses remains true.
Notice that the condition is controlled by a variable called "count", so as long as "count" is less than or equal to five, the loop will continue to run.
Since "count" starts at 1, and is incremented by 1 for every iteration of the loop, we'd expect the loop to run 5 times.
And that's exactly what we see in the output.
Keep in mind that this conditional statement was relatively controlled, but "while" loops are especially useful when you don't know how many times the loop will need to run.
By now, you should understand the structure and purpose of conditional statements, for loops, and while loops.

>> Lab:

The table has one row for each movie and several columns

- **name** - The name of the movie
- **year** - The year the movie was released
- **length_min** - The lenght of the movie in minutes
- **genre** - The genre of the movie
- **average_rating** - Average rating on Imdb
- **cost_millions** - The movie's production cost in millions
- **sequences** - The amount of sequences
- **foreign** - Indicative of whether the movie is foreign (1) or domestic (0)
- **age_restriction** - The age restriction for the movie

You can see part of the dataset below

name	year	length_min	genre	average_rating	cost_millions	foreign	age_restriction
Toy Story	1995	81	Animation	8.3	30	0	0
Akira	1998	125	Animation	8.1	10.4	1	14
The Breakfast Club	1985	97	Drama	7.9	1	0	14
The Artist	2011	100	Romance	8	15	1	12
Modern Times	1936	87	Comedy	8.6	1.5	0	10
Fight Club	1999	139	Drama	8.9	63	0	18
City of God	2002	130	Crime	8.7	3.3	1	18
The Untouchables	1987	119	Drama	7.9	25	0	14

Control statements

Control statements are ways for a programmer to control what pieces of the program are to be executed at certain times. The syntax of control statements is very similar to regular english, and they are very similar to logical decisions that we make all the time.

Conditional statements and **Loops** are the control statements that are able to change the execution flow. The expected execution flow is that each line and command should be executed in the order they are written. Control statements are able to change this, allowing you to skip parts of the code or to repeat blocks of code.

Conditional Statements

We often want to check a conditional statement and then do something in response to that condition being true or false.

If Statements

If statements are composed of a conditional check and a block of code that is executed if the check results in **true**. For example, assume we want to check a movie's year, and print something if it greater than 2000:

```
movie_year = 2002

# If Movie_Year is greater than 2000...
if(movie_year > 2000){

    # ...we print a message saying that it is greater than 2000.
    print('Movie year is greater than 2000')

}
```

```
[1] "Movie year is greater than 2000"
```

Notice that the code in the above block **{}** will only be executed if the check results in **true**.

You can also add an `else` block to `if` block -- the code in `else` block will only be executed if the check results in **false**.

Syntax:

```
if (condition) {

    # do something

} else {

    # do something else

}
```

Tip: This syntax can be spread over multiple lines for ease of creation and legibility.

Let's create a variable called `Movie_Year` and attribute it the value 1997. Additionally, let's add an `if` statement to check if the value stored in `Movie_Year` is greater than 2000 or not -- if it is, then we want to output a message saying that Movie_Year is greater than 2000, if not, then we output a message saying that it is not greater than 2000.

```
movie_year = 1997

# If Movie_Year is greater than 2000...
if(movie_year > 2000){

    # ...we print a message saying that it is greater than 2000.
    print('Movie year is greater than 2000')

}else{ # If the above conditions were not met (Movie_Year is not greater than 2000)...

    # ...then we print a message saying that it is not greater than 2000.
    print('Movie year is not greater than 2000')

}
```

```
[1] "Movie year is not greater than 2000"
```

Feel free to change `movie_year`'s value to other values -- you'll see that the result changes based on it!

To create our conditional statements to be used with `if` and `else`, we have a few tools:

Comparison operators

When comparing two values you can use this operators

- equal: `==`
- not equal: `!=`
- greater/less than: `>``
- greater/less than or equal: `>=``

Logical operators

Sometimes you want to check more than one condition at once. For example you might want to check if one condition **and** other condition are true. Logical operators allow you to combine or modify conditions.

- and: `&`
- or: `|`
- not: `!`

Let's try using these operators:

```
movie_year = 1997

# If Movie_Year is BOTH less than 2000 AND greater than 1990 -- both conditions have to be true! -- ...
if(movie_year < 2000 & movie_year > 1990 ) {
    # ...then we print this message.
    print('Movie year between 1990 and 2000')
}

# If Movie_Year is EITHER greater than 2010 OR less than 2000 -- any of the conditions have to be true! -- ...
if(movie_year > 2010 | movie_year < 2000 ) {
    # ...then we print this message.
    print('Movie year is not between 2000 and 2010')
}
```

```
[1] "Movie year between 1990 and 2000"
[1] "Movie year is not between 2000 and 2010"
```

Tip: All the expressions will return the value in Boolean format -- this format can only house two values: true or false!

Subset

Sometimes, we don't want an entire dataset -- maybe in a dataset of people we want only people with age 18 and over, or in the movies dataset, maybe we want only movies that were created after a certain year. This means we want a **subset** of the dataset. In R, we can do this by utilizing the `subset` function.

Suppose we want a subset of the `movies_Data` data frame composed of movies from a given year forward (e.g. year 2000) if a selected variable is **recent**, or from that given year back if we select **old**. We can quite simply do that in R by doing this:

```
decade = 'recent'

# If the decade given is recent...
if(decade == 'recent' ){
    # Subset the dataset to include only movies after year 2000.
    subset(movies_data, year >= 2000)
} else { # If not...
    # Subset the dataset to include only movies before 2000.
    subset(movies_data, year < 2000)
}
```

A data.frame: 13 × 8

	name	year	length_min	genre	average_rating	cost_millions	foreign	age_restriction
	<fct>	<int>	<int>	<fct>	<dbl>	<dbl>	<int>	<int>
4	The Artist	2011	100	Romance	8.0	15.0	1	12
7	City of God	2002	130	Crime	8.7	3.3	1	18
11	Room	2015	118	Drama	8.3	13.0	1	14
15	High School Musical	2006	98	Comedy	5.2	4.2	0	0
16	Shaun of the Dead	2004	99	Horror	8.0	6.1	1	18
19	Interstellar	2014	169	Adventure	8.6	165.0	0	10
22	Blue is the Warmest Colour	2013	179	Romance	7.8	4.5	1	18
23	Black Swan	2010	108	Thriller	8.0	13.0	0	16
25	The Wave	2008	107	Thriller	7.6	5.5	1	16
26	Whiplash	2014	106	Drama	8.5	3.3	1	12

Loops

Sometimes, you might want to repeat a given function many times. Maybe you don't even know how many times you want it to execute, but have an idea like `once for every row in my dataset`. Repeated execution like this is supplemented by **loops**. In R, there are two main loop structures, `for` and `while`.

The `for` loop

The `for` loop structure enables you to execute a code block once for every element in a given structure. For example, it would be like saying **execute this once for every row in my dataset**, or "execute this once for every element in this column bigger than 10". `for` loops are a very useful structure that make the processing of a large amount of data very simple.

Let's try to use a `for` loop to print all the years present in the `year` column in the `movies_Data` data frame. We can do that like this:

```
# Get the data for the "year" column in the data frame.
years <- movies_data['year']

# For each value in the "years" variable...
# Note that "val" here is a variable -- it assumes the value of one of the data points in "years"!
for (val in years) {
    # ...print the year stored in "val".
    print(val)
}
```

```
 [1] 1995 1998 1985 2011 1936 1999 2002 1987 1977 1999 2015 1964 1998 1975 2006
[16] 2004 1976 1994 2014 1995 1990 2013 2010 1985 2008 2014 2014 1995 2004 2002
```

The `while` loop

As you can see, the `for` loop is useful for a controlled flow of repetition. However, what if we don't know when we want to stop the loop? What if we want to keep executing a code block until a certain threshold has been reached, or maybe when a logical expression finally results in an expected fashion?

The `while` loop exists as a tool for repeated execution based on a condition. The code block will keep being executed until the given logical condition returns a `False` boolean value.

Let's try using `while` to print the first five movie names of our dataset. It can be done like this:

```
# Creating a start point.
iteration = 1

# We want to repeat until we reach the sixth operation -- but not execute the sixth time.
# While iteration is less or equal to five...
while (iteration <= 5) {

    print(c("This is iteration number:",as.character(iteration)))

    # ...print the "name" column of the iteration-th row.
    print(movies_data[iteration,]$name)

    # And then, we increase the "iteration" value -- so that we actually reach our stopping condition
    # Be careful of infinite while loops!
    iteration = iteration + 1
}
```

```
[1] "This is iteration number:" "1"
[1] Toy Story
30 Levels: Akira American Beauty Back to the Future ... Whiplash
[1] "This is iteration number:" "2"
[1] Akira
30 Levels: Akira American Beauty Back to the Future ... Whiplash
[1] "This is iteration number:" "3"
[1] The Breakfast Club
30 Levels: Akira American Beauty Back to the Future ... Whiplash
[1] "This is iteration number:" "4"
[1] The Artist
30 Levels: Akira American Beauty Back to the Future ... Whiplash
[1] "This is iteration number:" "5"
[1] Modern Times
30 Levels: Akira American Beauty Back to the Future ... Whiplash
```

Applying Functions to Vectors

One of the most common uses of loops is to **apply a given function to every element in a vector of elements**. Any of the loop structures can do that, however, R conveniently provides us with a very simple way to do that: By inferring the operation.

R is a very smart language when it comes to element-wise operations. For example, you can perform an operation on a whole list by utilizing that function directly on it. Let's try that out:

```
# First, we create a vector...
my_list <- c(10,12,15,19,25,33)

# ...we can try adding two to all the values in that vector.
my_list + 2

# Or maybe even exponentiating them by two.
my_list ** 2

# We can also sum two vectors element-wise!
my_list + my_list
```

$12 \cdot 14 \cdot 17 \cdot 21 \cdot 27 \cdot 35$

$100 \cdot 144 \cdot 225 \cdot 361 \cdot 625 \cdot 1089$

$20 \cdot 24 \cdot 30 \cdot 38 \cdot 50 \cdot 66$

R makes it very simple to operate over vectors -- anything you think should work will probably work. Try to mess around with vectors and see what you find out!

This is the end of the **Loops and Conditional Execution in R** notebook. Hopefully, now you know how to manipulate the flow of your code to your needs. Thank you for reading this notebook, and good luck on your studies.

Functions in R

- Function: reusable block of code

COGNITIVE CLASS.ai

Pre-defined functions

```
ratings <- c(8.7, 6.9, 8.5)
mean(ratings)
```

```
[1] 8.033333
```

```
sort(ratings)
```

```
[1] 6.9 8.5 8.7
```

```
sort(ratings, decreasing = TRUE)
```

User-defined functions

```
printHelloWorld <- function(){
    print("Hello World")
}
```

```
printHelloWorld()
```

```
[1] "Hello World"
```

```
add <- function(x, y) {
    x + y
}
add(3, 4)
```

The "add" function here takes in two nu

Returning values explicitly

```
add <- function(x, y){
    return(x + y)
}
add(3, 4)
```

```
[1] 7
```

COGNITIVE CLASS.ai

If / Else in functions

```
isGoodRating <- function(rating){
    if(rating < 7){
        return("NO")
    }else{
        return("YES")
    }
}
```

```
isGoodRating(6)
```

```
[1] "NO"
```

```
isGoodRating(9.5)
```

"YES" inside the "else" bl

```
[1] "YES"
```

Default input values in functions

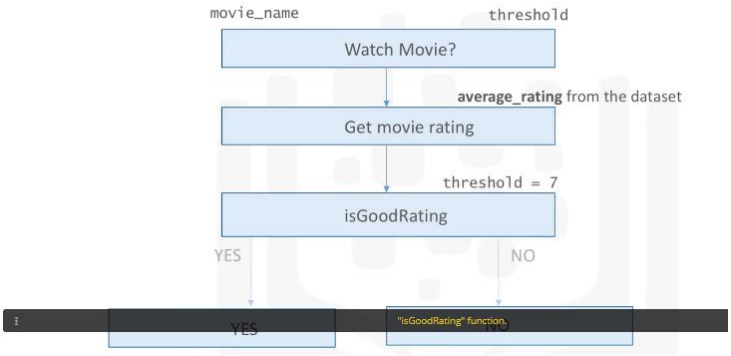
```
isGoodRating <- function(rating, threshold = 7){
    if(rating < threshold){
        return("NO")
    }else{
        return("YES")
    }
}
```

```
isGoodRating(8)
```

```
[1] "YES"
```

```
isGoodRating(8, threshold = 8.5)
```

Deciding a movie to watch



Using functions within functions

my_data

name	year	length_min	genre	average_rating	cost_millions	foreign	age_restriction
Toy Story	1995	81	Animation	8.3	30	0	0
Akira	1998	125	Animation	8.1	10.4	1	14
The Breakfast Club	1985	97	Drama	7.9	1	0	14
The Artist	2011	100	Romance	8	15	1	12
Modern Times	1936	87	Comedy	8.6	1.5	0	10
Fight Club	1999	139	Drama	8.9	63	0	18

```
watchMovie <- function(moviename, my_threshold = 7){
    rating <- my_data[my_data[,1] == moviename,"average_rating"]
    isGoodRating(rating, threshold = my_threshold)
}
```

```
watchMovie("Akira")
```

```
[1] "YES"
```

```
watchMovie("Akira", 8)
```

```
[1] "YES"
```

Global and local variables

```
myFunction <- function(){
    y <<- 3.14
    temp <- 'Hello World'
    return(temp)
}
myFunction()
```

```
[1] "Hello World"
```

```
y
```

```
[1] 3.14
```

You'll notice that "temp" uses the standard variable assignment operator, but the operator

```
temp
```

```
Error in eval(expr, envir, enclos): object 'temp' not found
```

In this video, we're going to show you how to work with functions in the R programming language.
A function is a block of code that can be re-used in different parts of a program.
Generally speaking, this can be broken down into pre-defined functions, and user-defined functions.
Pre-defined functions are the functions that are already defined for you, whether they're built in to R or provided in a separate package.
Let's take a look at a few common functions.
Here, we define a small vector of movie ratings, and we use the built-in "mean" function to find the average, which you can see in the output.
The "sort" function is used to order the elements of a vector.
Notice that the elements are arranged in ascending order.
This is the default behavior.
If you'd like to sort in descending order, you need to add the parameter "decreasing = TRUE" when calling the function.
As we mentioned, you can define your own functions as well.
In this code snippet, we create a function called "printHelloWorld" that, when run, simply prints "hello world" to the screen.

So if we call this function by name, we get the expected output.
Our functions can take in arguments as well.
The "add" function here takes in two numbers, and produces the sum as output.
The "return" statement can be used to explicitly output a value from the function.
When "return" is encountered, anywhere in the function, the corresponding value will be output and the function will exit.
Keep in mind that if the function lacks a return statement, then R will automatically return the value of the last evaluated expression.
The "return" statement is particularly useful when you need an "if else" block, since the final output value will be dependent on some condition.
Take a look at the function "isGoodRating" that we've defined.
This function takes in a movie rating and runs through a conditional block.
So the function will return a different value depending on the input.
For example, if we pass in 6, the "if" statement will be "true" and "NO" will be returned.
But if we pass in 9.5, the "if" statement will be false and the function will return "YES" inside the "else" block.
If you wish, you can set a default value for a function argument.
Notice in the function definition we've added "threshold = 7".
Now when we call this function, we only need to specify the "rating" parameter.
R will automatically use the value of 7 for threshold.
However, you can also override the default value, like so.
Now since 8 is less than the specified value of 8.5, the function will return "NO" as its output.
To increase the complexity of an application, you can use functions within other functions.
Let's first see the main function we're going to build at a high level.
The function will receive a movie name and a threshold, and it will determine whether to watch the movie by outputting "YES" or "NO".
In order to do this, the function will check a database for the movie's average rating.
This rating, along with the default threshold of 7, will be passed to the previously defined "isGoodRating" function.
The final output of the function will depend on the output of "isGoodRating".
So based on the dataset you see here, let's take a look at the entire function's structure.
The function is called "watchMovie", and it takes in the movie's name, as well as a rating threshold.
This line is responsible for pulling the movie's rating from the database.
It does this by finding a matching name in the "name" column, and then finding the "average rating" value in the corresponding row.
In the next line of the function, we pass this rating to the "isGoodRating" function.
"my threshold" is passed as well, but it has the same default value of 7.
So if we pass in the movie name "Akira", the rating is 8.1, which is greater than the default threshold of 7.
So the output will be "YES".
We can also override the threshold, but 8.1 is still greater than 8, so once again we get "YES" as output.
Variables can be defined inside of functions, but there are a few important considerations.
The function here, when run, will simply output "hello world".
But take a closer look at the variables "y" and "temp".
If you try to access "y" outside of the function, you'll notice the output is 3.14.
But if you try to access "temp", we get an error instead.
This is because there is a difference in how these variables are defined.
You'll notice that "temp" uses the standard variable assignment operator, but the operator

for "y" has an extra "left arrow".
This means that "y" is defined as a global variable, so it can be accessed outside the function.
"temp" on the other hand is local, so it can only be accessed inside the function.
By now, you should understand how to work with both pre-defined and user-defined functions,
how to nest functions within other functions, and how to create global variables.

>> Lab:

Table of Contents

About the Dataset

Imagine you got many movie recomendations from your friends and compiled all of the recomendations in a table, with specific info about each movie.

The table has one row for each movie and several columns

- **name** - The name of the movie
- **year** - The year the movie was released
- **length_min** - The lenght of the movie in minutes
- **genre** - The genre of the movie
- **average_rating** - Average rating on Imdb
- **cost_millions** - The movie's production cost in millions
- **sequences** - The amount of sequences
- **foreign** - Indicative of whether the movie is foreign (1) or domestic (0)
- **age_restriction** - The age restriction for the movie

You can see part of the dataset below

name	year	length_min	genre	average_rating	cost_millions	foreign	age_restriction
Toy Story	1995	81	Animation	8.3	30	0	0
Akira	1998	125	Animation	8.1	10.4	1	14
The Breakfast Club	1985	97	Drama	7.9	1	0	14
The Artist	2011	100	Romance	8	15	1	12

Lets first download the dataset that we will use in this notebook:

```
# code to download the dataset
download.file("https://ibm.box.com/shared/static/n5ay5qadfe7e1nnsv5s01oe1x62mq51j.csv", destfile="movies-db.csv")
```

What is a Function?

A function is a re-usable block of code which performs operations specified in the function.

There are two types of functions :

- **Pre-defined functions**
- **User defined functions**

Pre-defined functions are those that are already defined for you, whether it's in R or within a package. For example, `sum()` is a pre-defined function that returns the sum of its numeric inputs.

User-defined functions are custom functions created and defined by the user. For example, you can create a custom function to print **Hello World**.

Pre-defined functions

There are many pre-defined functions, so let's start with the simple ones.

Using the `mean()` function, let's get the average of these three movie ratings:

- **Star Wars (1977)** - rating of 8.7
- **Jumanji** - rating of 6.9
- **Back to the Future** - rating of 8.5

```
ratings <- c(8.7, 6.9, 8.5)
mean(ratings)
```

8.03333333333333

We can use the `sort()` function to sort the movies rating in *ascending order*.

```
sort(ratings)
```

6.9 · 8.5 · 8.7

[Tip] How do I learn more about the pre-defined functions in R?

We will be introducing a variety of **pre-defined functions** to you as you learn more about R. There are just too many functions, so there's no way we can teach them all in one sitting. But if you'd like to take a quick peek, here's a short reference card for some of the commonly-used pre-defined functions: https://cran.r-project.org/doc/contrib/Short-refcard.pdf

User-defined functions

Functions are very easy to create in R:

```
printHelloWorld <- function(){
    print("Hello World")
}
printHelloWorld()
```

```
[1] "Hello World"
```

To use it, simply run the function with `()` at the end:

```
printHelloWorld()
```

```
[1] "Hello World"
```

But what if you want the function to provide some **output** based on some **inputs**?

```
add <- function(x, y) {
    x + y
}
add(3, 4)
```

7

As you can see above, you can create functions with the following syntax to take in inputs (as its arguments), then provide some output.

```
f <- function(<arguments>) {
Do something
Do something
return(some_output)
}
```

Explicitly returning outputs in user-defined functions

In R, the last line in the function is automatically inferred as the output the function.

You can also explicitly tell the function to return an output.

```
add <- function(x, y){
    return(x + y)
}
add(3, 4)
```

7

It's good practice to use the `return()` function to explicitly tell the function to return the output.

Using IF/ELSE statements in functions

The `return()` function is particularly useful if you have any IF statements in the function, when you want your output to be dependent on some condition:

```
isGoodRating <- function(rating){
    #This function returns "NO" if the input value is less than 7. Otherwise it returns "YES".

    if(rating < 7){
        return("NO") # return NO if the movie rating is less than 7

    }else{
        return("YES") # otherwise return YES
    }
}

isGoodRating(6)
isGoodRating(9.5)
```

NO'

Setting default argument values in your custom functions

You can a set a default value for arguments in your function. For example, in the `isGoodRating()` function, what if we wanted to create a threshold for what we consider to be a good rating?

Perhaps by default, we should set the threshold to 7:

```
isGoodRating <- function(rating, threshold = 7){
    if(rating < threshold){
        return("NO") # return NO if the movie rating is less than the threshold
    }else{
        return("YES") # otherwise return YES
    }
}

isGoodRating(6)
isGoodRating(10)
```

'NO'
'YES'

Notice how we did not have to explicitly specify the second argument (threshold), but we could specify it. Let's say we have a higher standard for movie ratings, so let's bring our threshold up to 8.5:

```
isGoodRating(8, threshold = 8.5)
```

'NO'

Great! Now you know how to create default values. **Note that** if you know the order of the arguments, you do not need to write out the argument, as in:

```
isGoodRating(8, 8.5) #rating = 8, threshold = 8.5
```

'NO'

Using functions within functions

Using functions within functions is no big deal. In fact, you've already used the `print()` and `return()` functions. So let's try making our `isGoodRating()` more interesting.

Let's create a function that can help us decide on which movie to watch, based on its rating. We should be able to provide the name of the movie, and it should return **NO** if the movie rating is below 7, and **YES** otherwise.

First, let's read in our movies data:

```
my_data <- read.csv("movies-db.csv")
head(my_data)
```

A data.frame: 6 × 8

	name	year	length_min	genre	average_rating	cost_millions	foreign	age_restriction
	<fct>	<int>	<int>	<fct>	<dbl>	<dbl>	<int>	<int>
1	Toy Story	1995	81	Animation	8.3	30.0	0	0
2	Akira	1998	125	Animation	8.1	10.4	1	14
3	The Breakfast Club	1985	97	Drama	7.9	1.0	0	14
4	The Artist	2011	100	Romance	8.0	15.0	1	12

```
# Within myData, the row should be where the first column equals "Akira"
# AND the column should be "average_rating"

akira <- my_data[my_data$name == "Akira", "average_rating"]
akira

isGoodRating(akira)
```

8.1
'YES'

Now, let's put this all together into a function, that can take any **moviename** and return a **YES** or **NO** for whether or not we should watch it.

```
watchMovie <- function(data, moviename){
    rating <- data[data["name"] == moviename,"average_rating"]
    return(isGoodRating(rating))
}

watchMovie(my_data, "Akira")
```

'YES'

Make sure you take the time to understand the function above. Notice how the function expects two inputs: `data` and `moviename`, and so when we use the function, we must also input two arguments.

_But what if we only want to watch really good movies? How do we set our rating threshold that we created earlier? _
Here's how:

```
watchMovie <- function(data, moviename, my_threshold){
    rating <- data[data$name == moviename,"average_rating"]
    return(isGoodRating(rating, threshold = my_threshold))
}
```

Now our watchMovie takes three inputs: **data**, **moviename** and **my_threshold**

```
watchMovie(my_data, "Akira", 7)
```

'YES'

What if we want to still set our default threshold to be 7?
Here's how we can do it:

```
watchMovie <- function(data, moviename, my_threshold = 7){
    rating <- data[data[,1] == moviename,"average_rating"]
    return(isGoodRating(rating, threshold = my_threshold))
}

watchMovie(my_data,"Akira")
```

'YES'

While the **watchMovie** is easier to use, I can't tell what the movie rating actually is. How do I make it *print* what the actual movie rating is, before giving me a response? To do so, we can simply add in a **print** statement before the final line of the function.

We can also use the built-in `paste()` function to concatenate a sequence of character strings together into a single string.

```
watchMovie <- function(moviename, my_threshold = 7){
    rating <- my_data[my_data[,1] == moviename,"average_rating"]

    memo <- paste("The movie rating for", moviename, "is", rating)
    print(memo)

    return(isGoodRating(rating, threshold = my_threshold))
}

watchMovie("Akira")
```

```
[1] "The movie rating for Akira is 8.1"
'YES'
```

Just note that the returned output is actually the resulting value of the function:

```
x <- watchMovie("Akira")
```

```
[1] "The movie rating for Akira is 8.1"
```

```
print(x)
```

```
[1] "YES"
```

Global and local variables

So far, we've been creating variables within functions, but did you notice what happens to those variables outside of the function?

Let's try to see what **memo** returns:

```
watchMovie <- function(moviename, my_threshold = 7){
    rating <- my_data[my_data[,1] == moviename,"average_rating"]

    memo <- paste("The movie rating for", moviename, "is", rating)
    print(memo)

    isGoodRating(rating, threshold = my_threshold)
}

watchMovie("Akira")
```

```
[1] "The movie rating for Akira is 8.1"
'YES'
```

```
memo
```

```
Error in eval(expr, envir, enclos): object 'memo' not found
Traceback:
```

We got an error: `object 'memo' not found`. **Why?**

It's because all the variables we create in the function remain within the function. In technical terms, this is a **local variable**, meaning that the variable assignment does not persist outside the function. The `memo` variable only exists within the function.

But there is a way to create **global variables** from within a function -- where you can use the global variable outside of the function. It is typically *not* recommended that you use global variables, since it may become harder to manage your code, so this is just for your information.

To create a **global variable**, we need to use this syntax:

> `x <<- 1`

Here's an example of a global variable assignment:

```
myFunction <- function(){
    y <<- 3.14
    return("Hello World")
    }
myFunction()
```

```
'Hello World'
```

```
y #created only in the myFunction function
```

```
3.14
```

Objects and Classes (3:25)

Objects in R

- Example: x <- 5
- Object classes
  - Numeric
  - Character
  - Logical
  - Integer

Object Classes - Numeric

```
average_rating <- 8.3
```

```
[1] 8.3
```

```
class(average_rating)
```

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

Object Classes - Character

```
movies <- c("Toy Story", "Akira")
```

```
[1] "Toy Story" "Akira"
```

```
class(movies)
```

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

Object Classes - Logical

```
logical_vector <- c(TRUE, FALSE, FALSE, TRUE, TRUE)
```

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

```
class(logical_vector)
```

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

Object Classes – Numeric to Integer

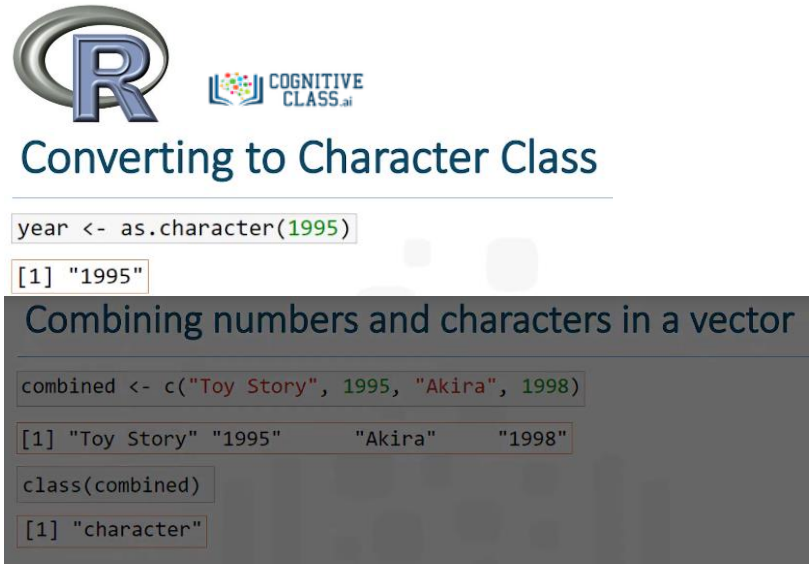
```
age_restriction <- c(12, 10, 18, 18)
```

```
[1] 12 10 18 18
```

```
class(age_restriction)
```

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

```
integer_vector <- as.integer(age_restriction)
```



In this video, we will provide an overview of the various object classes in the R programming language.
In R, an object is a data structure that has attributes, and methods that act on those attributes.
Take a look at the example here.
In this case, the variable "x" references an object that carries the value "5" somewhere in memory.
The object's attribute that specifies its type is known as its "class".
There are several classes in R that are used frequently.
The "numeric" class is used for real numbers.
The "character" class is used for string values.
The "logical" class is for "true false" values.
And the "integer" class is for integers, as you'd expect.
Let's take a look at these classes in more detail.
Let's create a variable to hold the decimal value, "8.3".
In R, objects that hold decimal values are "numeric" by default.
We can use the "class" function to check the object's type.
As we mentioned, the "character" class is used for strings, or text.
Here we've created an array of two strings, which you can see in the output.
If we run the "class" function, we see that the object's type is "character".
The "logical" class is for Boolean values, which are either "true" or "false".
So we can create a vector of "true false" values, and when we run the "class" function, we see that the object type is "logical".
Integers are the numbers that can be written without a fractional component.
But it's important to remember what we mentioned before about "numeric" being the default.
To demonstrate this, let's create a vector of integers, and use the "class" function to see the object type.
As you can see, the type is "numeric" rather than integer.
In order to convert this data into the "integer" class, we need to use the "as.integer" function.
We'll pass in our vector and assign it to a new vector called "integer vector".
If we check our new vector's type with the "class function", you can see that we've successfully converted our data to the "integer" type.
As a side note, there is also the "as.numeric" function for converting back to the "numeric" class.
If you want to convert a number into the "character" class, you can use the "as.character" function.
Notice that the input to the function is the number "1995".

When we look at the output, we can see the number is now a string.
Vectors are meant to only store data of the same type.
Look at this code snippet here where we try to create a vector that combines character types with numeric types.
Based on the output, we see that R automatically converts the numeric types in to character types.
And if we pass this vector into the "class" function, we can confirm that this is the case.
By now, you should have an understanding of objects and classes, as well as how to use the "class" function.

>> Lab:

The table has one row for each movie and several columns

- **name** - The name of the movie
- **year** - The year the movie was released
- **length_min** - The lenght of the movie in minutes
- **genre** - The genre of the movie
- **average_rating** - Average rating on Imdb
- **cost_millions** - The movie's production cost in millions
- **sequences** - The amount of sequences
- **foreign** - Indicative of whether the movie is foreign (1) or domestic (0)
- **age_restriction** - The age restriction for the movie

Part of the table can be seen here

name	year	length_min	genre	average_rating	cost_millions	foreign	age_restriction
Toy Story	1995	81	Animation	8.3	30	0	0
Akira	1998	125	Animation	8.1	10.4	1	14
The Breakfast Club	1985	97	Drama	7.9	1	0	14
The Artist	2011	100	Romance	8	15	1	12
Modern Times	1936	87	Comedy	8.6	1.5	0	10
Fight Club	1999	139	Drama	8.9	63	0	18

What is an Object?

Everthing that you manipulate in R, literally every entity in R, is considered an **object**. In real life, we think of an object as something that we can hold and look at. R objects are a lot like that. For example, vector is one of the objects in R.

An object in R has different kinds of properties or attributes. One of the attributes in objects is called the **class** of the object. The **class** of an object is the data type of this object. For instance, the class of vector can be numeric if it's composed of numeric values or character if it's composed of string values. The various classes (data types) of objects in R are important for data science programming.

Class

The most common classes (data types) of objects in R are:

- numeric (real numbers)
- character
- integer
- logical (True/False)
- complex

If you want to know about the data type of your values, you can use the **"class()"** function and add the variables' name to it. Let's create a variable from the average rating of some movies and then find which data types they belong to:

```
movie_rating <- c(8.3, 5.2, 9.3, 8.0) # create a vector from average ratings
movie_rating # print the variable
```

8.3 · 5.2 · 9.3 · 8

To check what is the data type, let's use **class()**

```
class(movie_rating) # show the variable's data type
```

'numeric'

As you see, the **class()** function shows that the data type of values in the vector is **numeric**.

Tip: A vector can only contain objects of the same class. However, a list can have different data types.

Numeric

Decimal values are called numerics. They are the default computational data type in R. In the example below, If we assign a decimal value to a variable average_rating, then average_rating will be of numeric type.

```
average_rating <- 8.3       # assign a decimal value
average_rating
```

8.3

Using **class** to check the data type results in **numeric**

```
class(average_rating)
```

'numeric'

Character

A character object is used to represent string values in R, strings are simply text values.

```
movies <-c("Toy Story", "Akira", "The Breakfast Club", "The Artist")
movies
class(movies)
```

'Toy Story' · 'Akira' · 'The Breakfast Club' · 'The Artist'

'character'

If numbers and texts are combined in a vector, everything is converted to the class **character**. Let's make a vector from combined movie names and their production year, then find the data type for the vector

```
combined <- c("Toy Story", 1995, "Akira", 1998)
combined
class(combined)
```

'Toy Story' · '1995' · 'Akira' · '1998'

'character'

When you simply enter numbers into R, they will be saved as class **numeric** by default. For example in the following vector, even though numbers are integers, they are stored as numeric type in R:

```
movie_length <- c(80, 110, 90, 80) # create a vector from movie length
movie_length # print the variable
class(movie_length)
```

80 · 110 · 90 · 80

numeric'

Integer

An integer is a number that can be written without a fractional component. For example, 21, 4, 0, and −2048 are integers, while 9.75, 5 ½, and √2 are not. In R, when you create a variable from the mentioned numbers, they are not going to be stored as integer data type. In order to get the integer class we need to convert the variable type from numeric to integer using **as.integer()** function. Let's create a vector and check if the data type is numeric.

```
age_restriction <- c(12, 10, 18, 18) # create a vector from age restriction
age_restriction # print the vector

class(age_restriction)
```

12 · 10 · 18 · 18

'numeric'

```
integer_vector <- as.integer(age_restriction)
class(integer_vector)
```

'integer'

Logical

The logical class contains True/False values (Boolean values). Let's create a vector with logical values and check its class:

```
logical_vector <- c(T,F,F,T,T) # creating the vector
class(logical_vector)
```

'logical'

A logical value is often created via comparison between variables. In the below example, we will compare the length of movies **Toy Story and Akira**.

```
length_Akira <- 125
length_ToyStory <- 81
```

If we assign the result of the compare statement to a variable the variable will have FALSE if the statement was false, and TRUE if the statement is true.

```
x <- length_ToyStory > length_Akira      # is ToyStory larger than akira?
x
```

FALSE

```
x <- length_Akira > length_ToyStory # is akira larger than ToyStory?
x # print the logical value
```

TRUE

The resulting variable is of type logical

```
class(x)       # print the class name of x
```

'logical'

Complex

A complex number is a number that can be expressed in the form a + bi, where a and b are real numbers and i is the imaginary unit.

```
z = 8 + 6i     # create a complex number
z
```

8+6i

```
class(z)
```

'complex'

Converting One Class to Another

We can convert (coerce) one data type to another if we desire. For example, we can convert objects from other data types into character values with the **"as.character()"** function. In the following example, we convert numeric value into character:

```
year <- as.character(1995) # convert integer into character data type
year                       # print the value of year in character data type
```

'1995'

As we mentioned before, in order to create an integer variable in R, we can use the **"as.integer()"** function. In the following example, even though the number is an integer data type, R saves the number as numeric data type by default. So you need to change the number to integer later if it is necessary.

```
Length_ToyStory <- 81
class(81)
```

'numeric'

```
length_ToyStory <- as.integer(81)
class(length_ToyStory)       # print the class name of length_ToyStory
```

'integer'

Difference between Class and Mode

For a simple vector, the class and mode of the vector are the same thing: the data type of the values inside the vector (character, numeric, integer, etc). However, in some of other objects such as matrix, array, data frame, and list, class and mode means different things.

In those mentioned objects, the **class()** function shows the type of the data structure. What does that mean? The class of matrix will be **matrix** regardless of what data types are **inside** the matrix. The same applies to list, array and data frame.

Mode on the other hand, determines what types of data can be found within the object and how that values are stored. So, you need to use the **mode()** function to find the data type of values inside a matrix (character, numeric, integer, etc).

So, in addition to the classes such as numeric, character, integer, logical, and complex, we have other classes such as matix, array, dataframe, and list

Matrix

Let's create a matrix storing the genre for each movie. Then, we will find the class and mode of the created matrix to see which information we will get from them.

First, let's check the effect of class and mode on a **vector**.

```
movies <- c("Toy Story", "Akira", "The Breakfast Club", "The Artist") # creating two vectors
genre <- c("Animation/Adventure/Comedy", "Animation/Adventure/Comedy", "Comedy/Drama", "Comedy/Drama")

class(movies)
mode(movies)
```

'character'
'character'

As you see in the above, for the vector the class and mode shows the data type of values. Now lets create a matrix from these two vectors.

```
movies_genre <- cbind(movies, genre)
movies_genre
```

A matrix: 4 × 2 of type chr

movies	genre
Toy Story	Animation/Adventure/Comedy
Akira	Animation/Adventure/Comedy
The Breakfast Club	Comedy/Drama
The Artist	Comedy/Drama

Now **class()** shows that the data type is **matrix**.

```
class(movies_genre)
```

'matrix'

And **mode** shows the data type of the elements of the matrix

```
mode(movies_genre)
```

'character'

For the matrix, the **class()** shows how the values are stored and shown in R, in this case, in a matrix. However, **mode()** shows the data type of values in the matrix. In the above example we have made a matrix filled with **character** values.

Array

A slightly more complicated version of the **matrix** data type is the **array** data type. The **array** data type can still only have one data type inside of it, but the set of data types it can store is larger. In addition to the data types an array can store matrices as its elements. In the following, we are going to create the array from integer number (1 to 12) and then compare the class and mode in an array:

```
sample_array <- array(1:12, dim = c(3, 2, 2)) # create an array with dimensions 3 x 2 x 2
sample_array
class(sample_array)
mode(sample_array)
```

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

'array'
'numeric'

So, the array's class is **array** and its mode is **numeric**.

Data Frame

Data frames are similar to arrays but they have certain advantages over arrays. Data frames allow you to associate each row and each column with a name of your choosing and allow **each column** of the data frame to have a **different data type** if you like. Let's create a data frame from the movie names, year and their length:

```
Name <- c("Toy Story", "Akira", "The Breakfast Club", "The Artist")
Year <- c(1995, 1998, 1985, 2011)
Length <- c(81, 125, 97, 100)
RowNames = c("Movie 1", "Movie 2", "Movie 3", "Movie 4")

sample_DataFrame <- data.frame(Name, Year, Length, row.names=RowNames)
sample_DataFrame

class(sample_DataFrame)
```

A data.frame: 4 × 3

	Name	Year	Length
	<fct>	<dbl>	<dbl>
Movie 1	Toy Story	1995	81
Movie 2	Akira	1998	125
Movie 3	The Breakfast Club	1985	97
Movie 4	The Artist	2011	100

List

The final data type that we are going to go over is **list**. Lists are similar to vectors, but they can contain multiple data types. For example:

```
sample_List = list("Star Wars", 8.7, TRUE)
sample_List

class(sample_List)

mode(sample_List)

mode(sample_List[[3]])
```

1. 'Star Wars'
2. 8.7
3. TRUE

'list'
'list'
'logical'

As you see, we have character, numeric, and logical data types in the list. The data type of third element in the list is logical as the "mode()" function shows us. The **mode** for the entire list is **list**, it could show the type of all the elements, since they don't all have the same data type.

Attributes

Objects have one or more **attributes** that can modify how R thinks about the object. Imagine you have a bowl of pasta and cheese. If you add spice, you change it to something with new taste. Different spices makes different dishes.

Attributes are like spice. You can change any individual attribute of object with the **attr()** function. You also can use the **attribute()** function to return a list of all of the attributes currently defined for that object.

For example in the following code, we will create a vector from the average ratings (8.3, 8.1, 7.9, 8) and costs of four movies (30, 10.4, 1, 15), an then we change the **dim** attribute of the vector. R will now treat z as it were a 4-by-2 matrix.

```
z <- c(8.3, 8.1, 7.9, 8, 30, 10.4, 1, 15)
z
attr(z, "dim") <- c(4,2)
z
```

8.3 · 8.1 · 7.9 · 8 · 30 · 10.4 · 1 · 15

A matrix: 4 × 2 of type dbl

8.3	30.0
8.1	10.4
7.9	1.0
8.0	15.0

Now, we can find the class and mode of the above matrix.

```
class(z)
mode(z)
```

'matrix'

'numeric'

Debugging (3:34)

Producing an error

```
"a" + 10
```

```
Error in "a" + 10: non-numeric argument to
binary operator
```

Error halts execution

```
for(i in 1:3){
    print(i + "a")
    }
```

```
Error in i + "a": non-numeric argument to
binary operator
```

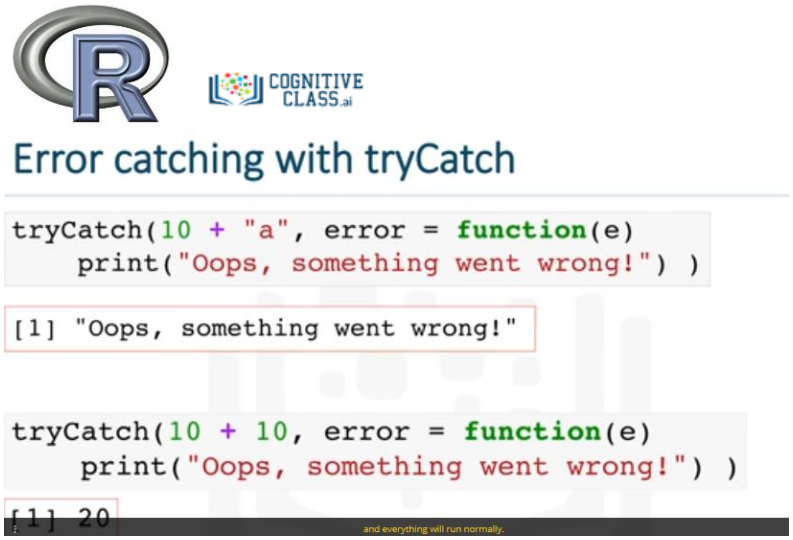
Error catching with tryCatch

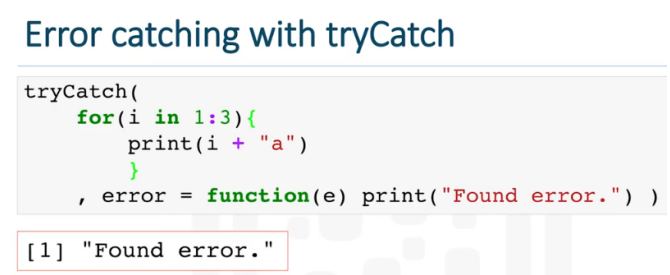
```
tryCatch(10 + 10)
```

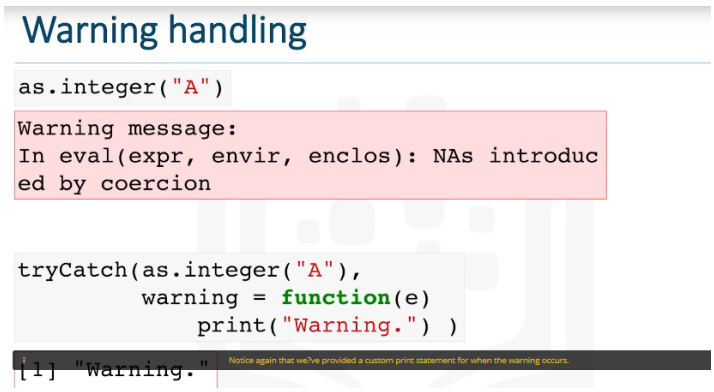
```
[1] 20
```

```
tryCatch("a" + 10)
```

```
Error in "a" + 10: non-numeric argument
to binary operator
```







In this video, we're going to show you how to deal with bugs in the R programming language.
If you try to perform an invalid operation, R will alert you with the appropriate error message.
Consider the piece of code here.
Notice that we're trying to add a character type and a number together.
When run, we see in the error output that one of the arguments is non-numeric.
An important thing to note is that an error will halt the code's execution.
If you look at this code, you'll see that we have a for loop that, under normal circumstances, would run three times.
But when the error is encountered, error message is output and the code stops immediately.
The previous two examples were simple, but these errors can often be difficult to locate in a large body of code.
The process of finding the source of these programming bugs and fixing them, is known as debugging.
If you know that an error might occur, you can catch the error while it?s happening to avoid the halting of your script.

The way to do this is by using a "tryCatch" statement.
A "tryCatch" statement will run the code normally, assuming that there are no errors involved.
You can see an example of normal execution here.
But if we go back to our invalid addition statement, the "tryCatch" will alert that
this is invalid.
At first it may seem like nothing new has happened, but "tryCatch" has a special way
of dealing with these types of issues.
Pay close attention to the syntax here.
Notice that we've added a custom print statement inside the parentheses.
If the "tryCatch" statement detects an error, it will run this code rather than printing
out the default error message.
However, if everything is okay, like when we added "10 + 10", no error will be thrown,
and everything will run normally.
You can even return a value, rather than just running a print statement.
Notice here that when the error is thrown, we simply return a concatenated string of
the two values.
It?s worth re-examining our previous example with the "for"
loop.
The code inside the "tryCatch" statement will run until an error is found.
But after the first error is found, the "tryCatch" will not resume execution of the main code
inside the parentheses.
This is why our custom message is only printed once, rather than three times.
A "tryCatch" statement will also allow you to catch warnings.
Unlike an error, a warning will not halt the code, but it generally suggests that something
undesirable is happening.
For example, if you look here, you can see that we're passing in the letter "A" to the
"as.integer"
function.
When this happens, R provides the warning displayed here.
The syntax for catching a warning is similar to the syntax from before.
Notice again that we?ve provided a custom print statement for when the warning occurs.
By now, you should understand how to catch errors and warnings, as well as how to run
your own code when they occur.

>> Lab:

What is debugging and error handling?

What do you get when you try to add `"a" + 10`? An error!

```
"a" + 10
```

```
Error in "a" + 10: non-numeric argument to binary operator
Traceback:
```

And what happens to your code if an error occurs? It halts!

```
for(i in 1:10){
    #for every number, i, in the sequence of 1,2,3:
    print(i + "a")
    }
```

```
Error in i + "a": non-numeric argument to binary operator
Traceback:

1. print(i + "a")
```

These are very simple examples, and the sources of the errors are easy to spot. But when it's embedded in a large chunk of code with many parts, it can be difficult to identify *when*, *where*, and *why* an error has occurred. This process of identifying the source of the error and fixing it is called **debugging**.

Error Catching

If you know an error may occur, the best way to handle the error is to `catch` the error while it's happening, so it doesn't prevent the script from halting at the error.

No error:

```
tryCatch(10 + 10)
```

```
20
```

Error:

```
tryCatch("a" + 10) #Error
```

```
Error in "a" + 10: non-numeric argument to binary operator
Traceback:

1. tryCatch("a" + 10)
2. tryCatchList(expr, classes, parentenv, handlers)
```

Error Catching with `tryCatch`:

`tryCatch` first *tries* to run the code, and if it works, it executes the code normally. **But if it results in an error**, you can define what to do instead.

```
#If tryCatch detects it will cause an error, print a message instead. Overall, no error is generated and the code continued to run successfully.

tryCatch(10 + "a",
         error = function(e) print("Oops, something went wrong!") ) #No error
```

```
[1] "Oops, something went wrong!"
```

```
#If error, return "10a" without an error

x <- tryCatch(10 + "a", error = function(e) return("10a") ) #No error
x
```

```
'10a'
```

```
tryCatch(
    for(i in 1:3){
        #for every number, i, in the sequence of 1,2,3:
        print(i + "a")
        }
    , error = function(e) print("Found error.") )
```

```
[1] "Found error."
```

Warning Catching

Aside from **errors**, there are also **warnings**. Warnings do not halt code, but are displayed when something is perhaps not running the way a user expects.

```
as.integer("A") #Converting "A" into an integer warns the user that the value is converted to NA
```

```
Warning message in eval(expr, envir, enclos):
“NAs introduced by coercion”
```

```
<NA>
```

If needed, you can also use `tryCatch` to catch the warnings as they occur, without producing the warning message:

```
tryCatch(as.integer("A"), warning = function(e) print("Warning.") )
```

```
[1] "Warning."
```

Scaling R with big data

As you learn more about R, if you are interested in exploring platforms that can help you run analyses at scale, you might want to sign up for a free account on IBM Watson Studio, which allows you to run analyses in R with two Spark executors for free.

What output will the following produce?

```
chance_precipitation <- 0.80
if( chance_precipitation > 0.5 ) {
    print("Bring an umbrella") } else {
        print("Don't bring an umbrella")}
```

- ○ `"Thunderstorm warning"`
- ○ `"Don't bring an umbrella"`
- ◉ `"Bring an umbrella"` ✔
- ○ Some sort of error

Which of the following statements are true?

- ☑ Using `return()` when writing a function is optional when you just want the result of the last line in the function to be the output of the function.
- ☐ Using `return()` when writing a function is necessary even when you just want the result of the last line in the function to be the output of the function.
- ☑ Using `return()` is useful when you want to produce outputs based on different conditions.
- ☐ Using `return()` serves no purpose when you want to produce outputs based on different conditions.

✔

Which of the following would you use to check the class of the object, `myobject` ?

- ◉ `class(myobject)` ✔
- ○ `type(myobject)`
- ○ `class(object)`
- ○ `class[myobject]`

4. Working with data in R

Learning Objectives

In this lesson you will learn about:

- Reading CSV and Excel Files
- Reading text files
- Writing and saving data objects to file in R

Reading CSV Files

name	year	length_min	genre	average_rating	cost_millions	foreign	age_restriction
Toy Story	1995	81	Animation	8.3	30	0	0
Akira	1998	125	Animation	8.1	10.4	1	14
The Breakfast Club	1985	97	Drama	7.9	1	0	14
The Artist	2011	100	Romance	8	15	1	12
Modern Times	1936	87	Comedy	8.6	1.5	0	10
Fight Club	1999	139	Drama	8.9	63	0	18
City of God	2002	130	Crime	8.7	3.3	1	18
The Untouchables	1987	119	Drama	7.9	25	0	14
Star Wars	1977	121	Action	8.7	11	0	10
American Beauty	1999	122	Drama	8.4	15	0	14
Room	2015	118	Drama	8.3	13	1	14
Dr. Strangelove	1964	94	Comedy	8.5	1.8	1	10

```
read.csv("/file_path/movies-db.csv")
```

Reading Excel Files

```
install.packages("readxl")
```

```
library(readxl)
```

```
read_excel("/file_path/movies-db.xls")
```

Accessing Data from Dataset

```
my_data <- read.csv("/file_path/movies-db.csv")
my_data
```

```
                                  name year length_min      genre
1                            Toy Story 1995         81  Animation
2                                 Akira 1998        125  Animation
3                    The Breakfast Club 1985         97      Drama
4                           The Artist 2011        100    Romance
5                         Modern Times 1936         87     Comedy
6                           Fight Club 1999        139      Drama
7                         City of God 2002        130      Crime
8                     The Untouchables 1987        119      Drama
9                           Star Wars 1977        121     Action
10                    American Beauty 1999        122      Drama
11                               Room 2015        118      Drama
12                    Dr. Strangelove 1964         94     Comedy
13                           The Ring 1998         95     Horror
14 Monty Python and the Holy Grail 1975         91     Comedy
15                 High School Musical 2006         98     Comedy
16                   Shaun of the Dead 2004         99     Horror
17                          Taxi Driver 1976        113      Crime
18          The Shawshank Redemption 1994        142      Crime
19                       Interstellar 2014        169  Adventure
20                             Casino 1995        178  Biography
21                  The goodfellas 1990        145  Biography
22      Blue is the warmest colour 2013        179    Romance
```

You can see here that after assigning the data to a variable name, we've used the variable

Accessing Data from Dataset

my_data

	name	year	length_min	genre
	Toy Story	1995	81	Animation
	Akira	1998	125	Animation
	The Breakfast Club	1985	97	Drama
	The Artist	2011	100	Romance
	Modern Times	1936	87	Comedy
	Fight Club	1999	139	Drama
	City of God	2002	130	Crime
	The Untouchables	1987	119	Drama
	Star Wars	1977	121	Action
	American Beauty	1999	122	Drama
	Room	2015	118	Drama
	Dr. Strangelove	1964	94	Comedy
	The Ring	1998	95	Horror
	Monty Python and the Holy Grail	1975	91	Comedy

```
my_data['name']
```

```
 [1] Toy Story
 [2] Akira
 [3] The Breakfast Club
 [4] The Artist
 [5] Modern Times
 [6] Fight Club
 [7] City of God
 [8] The Untouchables
 [9] Star Wars
[10] American Beauty
[11] Room
[12] Dr. Strangelove
```

Accessing Data from Dataset

my_data

	name	year	length_min	genre
	Toy Story	1995	81	Animation
	Akira	1998	125	Animation
	The Breakfast Club	1985	97	Drama
	The Artist	2011	100	Romance
	Modern Times	1936	87	Comedy
	Fight Club	1999	139	Drama
	City of God	2002	130	Crime
	The Untouchables	1987	119	Drama
	Star Wars	1977	121	Action
	American Beauty	1999	122	Drama
	Room	2015	118	Drama
	Dr. Strangelove	1964	94	Comedy
	The Ring	1998	95	Horror
	Monty Python and the Holy Grail	1975	91	Comedy

```
my_data[1,]
```

```
      name year length_min      genre average_rating cost_millions foreign
1 Toy Story 1995         81 Animation             8.3            30       0
  age_restriction
1               0
```

This will retrieve the selected row with all of the columns of the dataset.

Accessing Data from Dataset

my_data

	name	year	length_min	genre
	Toy Story	1995	81	Animation
	Akira	1998	125	Animation
	The Breakfast Club	1985	97	Drama
	The Artist	2011	100	Romance
	Modern Times	1936	87	Comedy
	Fight Club	1999	139	Drama
	City of God	2002	130	Crime
	The Untouchables	1987	119	Drama
	Star Wars	1977	121	Action
	American Beauty	1999	122	Drama
	Room	2015	118	Drama
	Dr. Strangelove	1964	94	Comedy
	The Ring	1998	95	Horror
	Monty Python and the Holy Grail	1975	91	Comedy

```
my_data[1, c("name","length_min")]
```

```
      name length_min
1 Toy Story         81
```

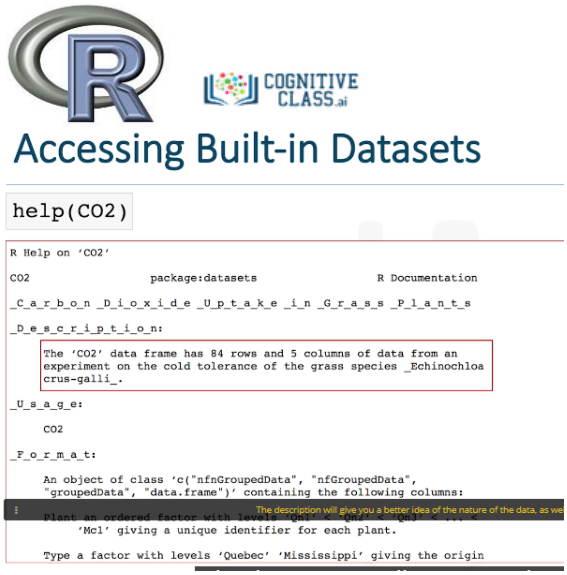
As you can see in the output, this ensures that we only extract the desired columns.

Accessing Built-in Datasets

```
data()
```

```
R data sets

Data sets in package '.':

Countries
Foodsupply
mortality
movies-db
Population
recipes

Data sets in package 'datasets':

AirPassengers            Monthly Airline Passenger Numbers 1949-1960
BJsales                  Sales Data with Leading Indicator
BJsales.lead (BJsales)
                         Sales Data with Leading Indicator
BOD                      Biochemical Oxygen Demand
CO2                      Carbon Dioxide Uptake in Grass Plants
ChickWeight              Weight versus age of chicks on different diets
DNase                    Elisa assay of DNase
EuStockMarkets           Daily Closing Prices of Major European Stock
                         Indices, 1991-1998
Formaldehyde             Determination of Formaldehyde
```

The output returns all the datasets, each with a sm



In this video, we will show you how to read CSV files, Excel files, and built-in datasets using the R programming language.
A common file format for structured data is CSV, which stands for comma separated values.
CSV files store the data in a table format, and in each row, every column value is separated by a delimiter.
As the name would suggest, this delimiter is traditionally a comma.
In order to read a CSV data file, all we need to do is call the "read.csv" function while passing in the path to the file.
We can also use R to read in XLS files, which is the file format of an Excel spreadsheet.
But unlike CSV, R does not have a native function for reading Excel files.
So to add this functionality, we're going to have to run the "install.packages" function.
Once a package is installed, it does not need to be installed again unless it is uninstalled.
Whenever you use a library that is not native to R, you have to it into the R environment by calling the "library" function.
After you've done so, reading an Excel file is as simple as calling "read_excel" and passing in the path to the file.
In order to make use of the data that we're reading, we need to assign the output to a variable.
By default, R will structure the data as a data frame, which provides us with a lot of tools and flexibility.
You can see here that after assigning the data to a variable name, we've used the variable name to get a preview of the data.
The snippet you're seeing here shows the first four columns.
Once we have the dataset loaded in a variable, we can start accessing its elements.
So for example, say we wanted to access the "name" column, highlighted here.
To do so, we can directly reference the column name inside the square brackets.
In this output snippet, you can see we get the elements of this particular column.
You can also retrieve an entire row from a dataset, like the one highlighted here.
Inside the square brackets, simply type the row number you want to access, followed by a comma, leaving the column blank.
This will retrieve the selected row with all of the columns of the dataset.
It may be the case that you want to retrieve a row, but you're only interested in a select number of columns, like the ones highlighted here.

So we can access that row like before, but instead of leaving the column blank, we'll use the "c" function to form a vector with the columns we're interested in.
As you can see in the output, this ensures that we only extract the desired columns.
We've been working with data that we read in from a file, but R actually provides a number of built-in datasets that we can use.
To see the available datasets, all we need to do is call the "data" function.
The output returns all the datasets, each with a small description included.
Everything in the "datasets" package is built-in.
Take a look at the "CO2" dataset here.
We're going to use this dataset to quickly show another utility function that R provides.
R provides documentation for each of the datasets, which we can access by calling the "help" function along with the dataset's name.
You can see that this provides a lot of information.
The description will give you a better idea of the nature of the data, as well as the dataset's size.
Since this dataset is built-in, we don't need to import it or load it in order to start accessing the data.
We can immediately start referencing it by name since R has already prepared the dataset.
So you can start to see what the CO2 data looks like in the output.
By now, you should understand how to read and access CSV files, excel files, and built-in datasets.

Reading Text (.txt) files in R (2:40)

Reading text files into R

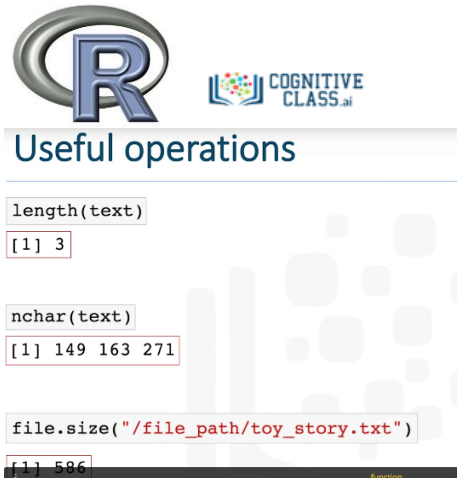
/file_path/toy_story.txt

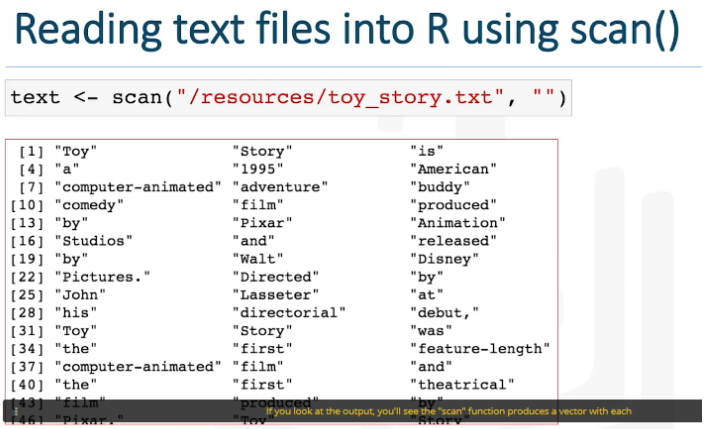
Toy Story is a 1995 American computer-animated adventure buddy comedy film produced by Pixar Animation Studios and released by Walt Disney Pictures.
Directed by John Lasseter at his directorial debut, Toy Story was the first feature-length computer-animated film and the first theatrical film produced by Pixar.
Toy Story follows a group of anthropomorphic toys who pretend to be lifeless whenever humans are present, and focuses on the relationship between Woody, a former cowboy doll (voiced by Tom Hanks), and Buzz Lightyear, an

Reading text files into R using readLines()

```
text <- readLines("/file_path/toy_story.txt")
```

```
[1] "Toy Story is a 1995 American computer-animated adventure buddy comedy film produced by P
ixar Animation Studios and released by Walt Disney Pictures. "

[2] "Directed by John Lasseter at his directorial debut, Toy Story was the first feature-leng
th computer-animated film and the first theatrical film produced by Pixar. "

[3] "Toy Story follows a group of anthropomorphic toys who pretend to be lifeless whenever hu
mans are present, and focuses on the relationship between Woody, a pullstring cowboy doll (vo
iced by Tom Hanks), and Buzz Lightyear, an astronaut action figure (voiced by Tim Allen). "
```





In this video, we're going to show you how to read in a text file using the R programming language.
We'll also introduce several functions that provide useful information about the file.
In order to demonstrate how to read these files, we're going to work with an example, which has the file path shown here.
You can also take a quick look at the file's contents.
Our example file contains three lines of text from Wikipedia's page on the 1995 movie "Toy Story".
One function that we can use to read the file is "readLines".
Notice that the argument to the "readLines" function is the file path from before.
The "text" variable receives a character vector, containing one item for each of the lines in the file.
Keep in mind that a "line" is not the same thing as a sentence.
Instead, lines are broken up by individual line breaks, which essentially form new paragraphs.
Let's now take a look at a few functions we can apply to gain information about the file.
In order to count the number of lines in the dataset, we can pass our "text" variable to the "length" function.
The "length" function will simply count up the number of elements in the vector, which in this case, is 3.
The "nchar" function will count up the number of characters in each line of our character vector.
You can see the result of that function here.
Each value includes letters and numbers, as well as symbols.
If you'd like to know the size of your file, you can pass the file path to the "file.size" function.
The output here is listed in bytes, which tells us that our example file has a size of 586 bytes.

If you'd like to read a text file by word, rather than by line, you can use the "scan" function while following the syntax you see here.
Again, the first argument is simply a file path.
And the second argument is just an empty string, but it is needed for our purposes.
If you look at the output, you'll see the "scan" function produces a vector with each individual word as an element.
By now, you should understand the functions that allow you to work with text files in the R programming language.

In this video, we'll show you how to export data from the R programming language into various file formats.
Simple structures like strings and matrices can easily be saved in a text file.
Consider the sample matrix here.
In order to save this matrix to a file, we're going to make use of the "write" function.
The "file" argument is simply the target file that we're going to save the data to.
The "ncolumns" argument specifies the number of columns that the data will be separated into.

And the final argument specifies the separator between the elements.
In our case, the elements will be separated by a space character.
When working with something a bit more complicated, like a data frame, you may want to save the
structure to a CSV file.
To do so, you can use the "write.csv"
function.
Let's take a quick look at the arguments.
The first argument is simply the data frame that you'd like to write to a file.
The second argument is the path for the file you'd like to output.
And "row.names" can be used to specify whether or not the file will display the name for each row.
You can omit column names as well, but it's a bit trickier.
You need to use the "write.table" function, while making sure to add the parameter "col.names
= FALSE".
Also notice the last parameter here.
This ensures that all the values will be separated by commas, making it a true a CSV file.
You can save your data frames into an excel file, but you?ll need to rely on an external package, like the "xlsx" package here.
If you've never installed the package before, run the "install.packages" function first.
The "library" function will load the package into the R environment so that it can be used.
Once you've done so, you can call the "write.xlsx" function to write the data to an excel file.
The syntax is similar to the csv function from before, except we need to specify the sheet name.
You can save and load R objects by using the "RData" format, along with the "save" function.
Take a look at the function's syntax.
The "list" argument takes in a vector, where each element is a variable name of the object you want to save.
The "file" argument simply specifies the file to write to.
The "safe" argument specifies whether or not you want the save to be performed atomically.
If this is false, there's a chance that the existing file could be damaged if the save fails.
By now, you should understand how to export R data into text files, CSV files, excel files, and RData files.

>> Lab:

Table of Contents

Movies dataset

Here we have a dataset that includes one row for each movie, with several columns for each movie characteristic:

- **name** - Name of the movie
- **year** - Year the movie was released
- **length_min** - Length of the movie (minutes)
- **genre** - Genre of the movie
- **average_rating** - Average rating on IMDB
- **cost_millions** - Movie's production cost (millions in USD)
- **foreign** - Is the movie foreign (1) or domestic (0)?
- **age_restriction** - Age restriction for the movie

name	year	length_min	genre	average_rating	cost_millions	foreign	age_restriction
Toy Story	1995	81	Animation	8.3	30	0	0
Akira	1998	125	Animation	8.1	10.4	1	14
The Breakfast Club	1985	97	Drama	7.9	1	0	14
The Artist	2011	100	Romance	8	15	1	12
Modern Times	1936	87	Comedy	8.6	1.5	0	10
Fight Club	1999	139	Drama	8.9	63	0	18
City of God	2002	130	Crime	8.7	3.3	1	18

Download the Data

We've made it easy for you to get the data, which we've hosted online. Simply run the code cell below (Shift + Enter) to download the data to your current folder.

```
# Download datasets

# CSV file
download.file("https://ibm.box.com/shared/static/n5ay5qadfe7e1nnsv5s01oe1x62mq51j.csv",
              destfile="movies-db.csv")

# XLS file
download.file("https://ibm.box.com/shared/static/nx0ohd9sq0iz3p871zg8ehc1m39ibpx6.xls",
              destfile="movies-db.xls")
```

If you ran the cell above, you have now downloaded the following files to your current folder:

> movies-db.csv
> movies-db.xls

Reading CSV Files

What are CSV files?

Let's read data from a CSV file. CSV (Comma Separated Values) is one of the most common formats of structured data you will find. These files contain data in a table format, where in each row, columns are separated by a delimiter -- traditionally, a comma (hence comma-separated values).

Usually, the first line in a CSV file contains the column names for the table itself. CSV files are popular because you do not need a particular program to open it.

Reading CSV files in R

In the `movies-db.csv` file, the first line of text is the header (names of each of the columns), followed by rows of movie information.

To read CSV files into R, we use the core function `read.csv`.

`read.csv` easy to use. All you need is the filepath to the CSV file. Let's try loading the file using the filepath to the `movies-db.csv` file we downloaded earlier:

```
# Load the CSV table into the my_data variable.
my_data <- read.csv("movies-db.csv")
my_data
```

A data.frame: 30 × 8

A data.frame: 30 × 8

name	year	length_min	genre	average_rating	cost_millions	foreign	age_restriction
<fct>	<int>	<int>	<fct>	<dbl>	<dbl>	<int>	<int>
Toy Story	1995	81	Animation	8.3	30.0	0	0
Akira	1998	125	Animation	8.1	10.4	1	14
The Breakfast Club	1985	97	Drama	7.9	1.0	0	14
The Artist	2011	100	Romance	8.0	15.0	1	12
Modern Times	1936	87	Comedy	8.6	1.5	0	10
Fight Club	1999	139	Drama	8.9	63.0	0	18
City of God	2002	130	Crime	8.7	3.3	1	18
The Untouchables	1987	119	Drama	7.9	25.0	0	14
Star Wars Episode IV	1977	121	Action	8.7	11.0	0	10
American Beauty	1999	122	Drama	8.4	15.0	0	14
Room	2015	118	Drama	8.3	13.0	1	14

The data was loaded into the `my_data` variable. But instead of viewing all the data at once, we can use the `head` function to take a look at only the top six rows of our table, like so:

[3]:
```
# Print out the first six rows of my_data
head(my_data)
```

A data.frame: 6 × 8

	name	year	length_min	genre	average_rating	cost_millions	foreign	age_restriction
	<fct>	<int>	<int>	<fct>	<dbl>	<dbl>	<int>	<int>
1	Toy Story	1995	81	Animation	8.3	30.0	0	0
2	Akira	1998	125	Animation	8.1	10.4	1	14
3	The Breakfast Club	1985	97	Drama	7.9	1.0	0	14
4	The Artist	2011	100	Romance	8.0	15.0	1	12
5	Modern Times	1936	87	Comedy	8.6	1.5	0	10
6	Fight Club	1999	139	Drama	8.9	63.0	0	18

Additionally, you may want to take a look at the **structure** of your newly created table. R provides us with a function that summarizes an entire table's properties, called `str`. Let's try it out.

]:
```
# Prints out the structure of your table.
str(my_data)
```

```
'data.frame':   30 obs. of  8 variables:
 $ name           : Factor w/ 30 levels "Akira","American Beauty",..: 29 1 21 20 14 10 8 27 18 2 ...
 $ year           : int  1995 1998 1985 2011 1936 1999 2002 1987 1977 1999 ...
 $ length_min     : int  81 125 97 100 87 139 130 119 121 122 ...
 $ genre          : Factor w/ 12 levels "Action","Adventure",..: 3 3 7 10 5 7 6 7 1 7 ...
 $ average_rating : num  8.3 8.1 7.9 8 8.6 8.9 8.7 7.9 8.7 8.4 ...
 $ cost_millions  : num  30 10.4 1 15 1.5 63 3.3 25 11 15 ...
 $ foreign        : int  0 1 0 1 0 0 1 0 0 0 ...
 $ age_restriction: int  0 14 14 12 10 18 18 14 10 14 ...
```

When we loaded the file with the `read.csv` function, we had to only pass it one parameter -- the **path** to our desired file.

If you're using Data Scientist Workbench, it is simple to find the path to your uploaded file. In the **Recent Data** section in the sidebar on the right, you can click the arrow to the left of the filename to see extra options -- one of these commands should be **Insert Path**, which automatically copies the path to your file into Jupyter Notebooks.

Reading Excel Files

Reading XLS (Excel Spreadsheet) files is similar to reading CSV files, but there's one catch -- R does not have a native function to read them. However, thankfully, R has an extremely large repository of user-created functions, called *CRAN*. From there, we can download a library package to make us able to read XLS files.

To download a package, we use the `install.packages` function. Once installed, you do not need to install that same library ever again, unless, of course, you uninstall it.

```
# Download and install the "readxl" library
install.packages("readxl")
```

```
Updating HTML index of packages in '.Library'
Making 'packages.html' ... done
```

Whenever you are going to use a library that is not native to R, you have to load it into the R environment after you install it. In other words, you need to install once only, but to use it, you must load it into R for every new session. To do so, use the `library` function, which loads up everything we can use in that library into R.

```
# Load the "readxl" library into the R environment.
library(readxl)
```

Now that we have our library and its functions ready, we can move on to actually reading the file. In `readxl`, there is a function called `read_excel`, which does all the work for us. You can use it like this:

```
# Read data from the XLS file and attribute the table to the my_excel_data variable.
my_excel_data <- read_excel("movies-db.xls")
```

Since `my_excel_data` is now a dataframe in R, much like the one we created out of the CSV file, all of the native R functions can be applied to it, like `head` and `str`.

```
# Prints out the structure of your table.
# Tells you how many rows and columns there are, and the names and type of each column.
# This should be the very same as the other table we created, as they are the same dataset.
str(my_excel_data)
```

```
tibble [30 × 8] (S3: tbl_df/tbl/data.frame)
 $ name           : chr [1:30] "Toy Story" "Akira" "The Breakfast Club" "The Artist" ...
 $ year           : num [1:30] 1995 1998 1985 2011 1936 ...
 $ length_min     : num [1:30] 81 125 97 100 87 139 130 119 121 122 ...
 $ genre          : chr [1:30] "Animation" "Animation" "Drama" "Romance" ...
 $ average_rating : num [1:30] 8.3 8.1 7.9 8 8.6 8.9 8.7 7.9 8.7 8.4 ...
 $ cost_millions  : num [1:30] 30 10.4 1 15 1.5 63 3.3 25 11 15 ...
 $ foreign        : num [1:30] 0 1 0 1 0 0 1 0 0 0 ...
 $ age_restriction: num [1:30] 0 14 14 12 10 18 18 14 10 14 ...
```

Much like the `read.csv` function, `read_excel` takes as its main parameter the **path** to the desired file.

[Tip] A **Library** is basically a collection of different classes and functions which are used to perform some specific operations. You can install and use libraries to add more functions that are not included on the core R files. For example, the **readxl** library adds functions to read data from excel files.

It's important to know that there are many other libraries too which can be used for a variety of things. There are also plenty of other libraries to read Excel files -- readxl is just one of them.

Accessing Rows and Columns

Whenever we use functions to read tabular data in R, the default method of structuring this data in the R environment is using Data Frames -- R' primary data structure. Data Frames are extremely versatile, and R presents us many options to manipulate them.

Suppose we want to access the "name" column of our dataset. We can directly reference the column name on our data frame to retrieve this data like this:

```
# Retrieve a subset of the data frame consisting of the "name" columns
my_data['name']
```

A data.frame: 30 × 1

name
<fct>
Toy Story
Akira
The Breakfast Club
The Artist
Modern Times
Fight Club
City of God
The Untouchables

Another way to do this is by using the `$` notation which at the output will provide a vector:

```
# Retrieve the data for the "name" column in the data frame.
my_data$name
```

Toy Story · Akira · The Breakfast Club · The Artist · Modern Times · Fight Club · City of God · The Untouchables · Star Wars Episode IV · American Beauty · Room · Dr. Strangelove · The Ring · Monty Python and the Holy Grail · High School Musical · Shaun of the Dead · Taxi Driver · The Shawshank Redemption · Interstellar · Casino · The Goodfellas · Blue is the Warmest Colour · Black Swan · Back to the Future · The Wave · Whiplash · The Grand Hotel Budapest · Jumanji · The Eternal Sunshine of the Spotless Mind · Chicago

▸ **Levels**:

You can also do the same thing using **double square brackets**, to get a vector of `names` column.

```
my_data[["name"]]
```

Toy Story · Akira · The Breakfast Club · The Artist · Modern Times · Fight Club · City of God · The Untouchables · Star Wars Episode IV · American Beauty · Room · Dr. Strangelove · The Ring · Monty Python and the Holy Grail · High School Musical · Shaun of the Dead · Taxi Driver · The Shawshank Redemption · Interstellar · Casino · The Goodfellas · Blue is the Warmest Colour · Black Swan · Back to the Future · The Wave · Whiplash · The Grand Hotel Budapest · Jumanji · The Eternal Sunshine of the Spotless Mind · Chicago

▸ **Levels**:

▾ **Levels**:
'Akira' · 'American Beauty' · 'Back to the Future' · 'Black Swan' · 'Blue is the Warmest Colour' · 'Casino' · 'Chicago' · 'City of God' · 'Dr. Strangelove' · 'Fight Club' · 'High School Musical' · 'Interstellar' · 'Jumanji' · 'Modern Times' · 'Monty Python and the Holy Grail' · 'Room' · 'Shaun of the Dead' · 'Star Wars Episode IV' · 'Taxi Driver' · 'The Artist' · 'The Breakfast Club' · 'The Eternal Sunshine of the Spotless Mind' · 'The Goodfellas' · 'The Grand Hotel Budapest' · 'The Ring' · 'The Shawshank Redemption' · 'The Untouchables' · 'The Wave' · 'Toy Story' · 'Whiplash'

Similarly, any particular row of the dataset can also be accessed. For example, to get the first row of the dataset with all column values, we can use:

```
# Retrieve the first row of the data frame.
my_data[1,]
```

A data.frame: 1 × 8

	name	year	length_min	genre	average_rating	cost_millions	foreign	age_restriction
	<fct>	<int>	<int>	<fct>	<dbl>	<dbl>	<int>	<int>
1	Toy Story	1995	81	Animation	8.3	30	0	0

The first value before the comma represents the **row** of the dataset and the second value (which is blank in the above example) represents the **column** of the dataset to be retrieved. By setting the first number as 1 we say we want data from row 1. By leaving the column blank we say we want all the columns in that row.

We can specify more than one column or row by using `c`, the **concatenate** function. By using `c` to concatenate a list of elements, we tell R that we want these observations out of the data frame. Let's try it out.

```r
# Retrieve the first row of the data frame, but only the "name" and "length_min" columns.
my_data[1, c("name","length_min")]
```

A data.frame: 1 × 2

	name	length_min
	<fct>	<int>
1	Toy Story	81

Accessing Built-in Datasets in R

R provides various built-in datasets for users to utilize for different purposes. To know which datasets are available, R provides a simple function -- `data` -- that returns all of the present datasets' names with a small description beside them. The ones in the `datasets` package are all inbuilt.

```r
# Displays a list of the inbuilt datasets. Opens in a new "window".
data()
```

Data sets

A data.frame: 104 × 3

Package	Item	Title
<chr>	<chr>	<chr>
datasets	AirPassengers	Monthly Airline Passenger Numbers 1949-1960
datasets	BJsales	Sales Data with Leading Indicator
datasets	BJsales.lead (BJsales)	Sales Data with Leading Indicator
datasets	BOD	Biochemical Oxygen Demand
datasets	CO2	Carbon Dioxide Uptake in Grass Plants
datasets	ChickWeight	Weight versus age of chicks on different diets
datasets	DNase	Elisa assay of DNase
datasets	EuStockMarkets	Daily Closing Prices of Major European Stock Indices, 1991-1998

Average Heights and Weights for American Women

Description

This data set gives the average heights and weights for American women aged 30–39.

Usage

```
women
```

Format

A data frame with 15 observations on 2 variables.

`[,1]`	`height`	numeric	Height (in)
`[,2]`	`weight`	numeric	Weight (lbs)

Details

The data set appears to have been taken from the American Society of Actuaries *Build and Blood Pressure Study* for some (unknown to us) year.

The World Almanac notes: "The figures represent weights in ordinary indoor clothing and shoes, and heights with shoes".

Source

The World Almanac and Book of Facts, 1975.

References

McNeil, D. R. (1977) *Interactive Data Analysis*. Wiley.

Examples

```
require(graphics)
plot(women, xlab = "Height (in)", ylab = "Weight (lb)",
     main = "women data: American women aged 30-39")
```

[Package *datasets* version 3.5.1]

Since the datasets listed are inbuilt, you do not need to import or load them to use them. If you reference them by their name, R already has the data frame ready.

```
women
```

A data.frame: 15 × 2

height <dbl>	weight <dbl>
58	115
59	117
60	120
61	123

What does CSV stand for, when talking about tabular data files?

- ◯ Column-sorted values
- ◉ Comma-separated values ✔
- ◯ Commonly-spaced values
- ◯ Column-separated values
- ◯ None of the above

Which of the following are true?

- ☑ `read.csv()` can be used to read in CSV files
- ☑ You need to install libraries, such as the "readxl" library, to read Excel files into R
- ☑ You can load specified datasets or list the available datasets using `data()`
- ☑ You can write to a variety of filetypes, including .txt, .csv, .xls, .xlsx, and .Rdata.

✔

To get the number of characters in a character vector, `char_vec`, what function can you use?

- [x] `nchar(char_vec)`
- [ ] `numberOfCharacters[char_vec]`
- [ ] `char_vec.nchar()`
- [ ] `length(char_vec)`

5. Strings and Dates in R

Learning Objectives

In this lesson you will learn about:

- String operations in R
- Regular Expressions
- Dates in R

Basic String Operations

```
summary <- readLines("/file_path/The_Artist.txt")
summary
```

```
[1] "The Artist is a 2011 French romantic comedy-drama in the style of a
black-and-white silent film. It was written, directed, and co-edited by M
ichel Hazanavicius, produced by Thomas Langmann and starred Jean Dujardin
and Bérénice Bejo. The story takes place in Hollywood, between 1927 and 1
932, and focuses on the relationship of an older silent film star and a r
ising young actress as silent cinema falls out of fashion and is replaced
by the talkies."
[2] ""
[3] "The Artist received highly positive reviews from critics and won man
y accolades. Dujardin won the Best Actor Award at the 2011 Cannes Film Fe
stival, where the film premiered. The film was nominated for six Golden G
lobes, the most of any 2011 film, and won three: Best Motion Picture – Mu
sical or Comedy, Best Original Score, and Best Actor – Motion Picture Mus
ical or Comedy for Dujardin. In January 2012, the film was nominated for
twelve BAFTAs, the most of any film from 2011 and won seven, including Be
st Film, Best Director and Best Original Screenplay for Hazanavicius, and
```

Basic String Operations

```
nchar(summary[1])
```

```
450
```

Basic String Operations

```
toupper(summary[1])
```

```
[1] "THE ARTIST IS A 2011 FRENCH ROMANTIC COMEDY-DRAMA IN THE STYLE
OF A BLACK-AND-WHITE SILENT FILM. IT WAS WRITTEN, DIRECTED, AND CO-E
DITED BY MICHEL HAZANAVICIUS, PRODUCED BY THOMAS LANGMANN AND STARRE
D JEAN DUJARDIN AND BÉRÉNICE BEJO. THE STORY TAKES PLACE IN HOLLYWOO
D, BETWEEN 1927 AND 1932, AND FOCUSES ON THE RELATIONSHIP OF AN OLDE
R SILENT FILM STAR AND A RISING YOUNG ACTRESS AS SILENT CINEMA FALLS
OUT OF FASHION AND IS REPLACED BY THE TALKIES."
```

Basic String Operations

```
tolower(summary[1])
```

```
[1] "the artist is a 2011 french romantic comedy-drama in the style
of a black-and-white silent film. it was written, directed, and co-e
dited by michel hazanavicius, produced by thomas langmann and starre
d jean dujardin and bérénice bejo. the story takes place in hollywoo
d, between 1927 and 1932, and focuses on the relationship of an olde
r silent film star and a rising young actress as silent cinema falls
out of fashion and is replaced by the talkies."
```

Basic String Operations

```
chartr(" ", "-", summary[1])
```

```
[1] "The-Artist-is-a-2011-French-romantic-comedy-drama-in-the-style-
of-a-black-and-white-silent-film.-It-was-written,-directed,-and-co-e
dited-by-Michel-Hazanavicius,-produced-by-Thomas-Langmann-and-starre
d-Jean-Dujardin-and-Bérénice-Bejo.-The-story-takes-place-in-Hollywoo
d,-between-1927-and-1932,-and-focuses-on-the-relationship-of-an-olde
r-silent-film-star-and-a-rising-young-actress-as-silent-cinema-falls
-out-of-fashion-and-is-replaced-by-the-talkies."
```

Basic String Operations

```
char_list <- strsplit(summary[1], " ")
word_list <- unlist(char_list)
word_list
```

```
 [1] "The"           "Artist"          "is"              "a"
 [5] "2011"          "French"          "romantic"        "comedy-drama"
 [9] "in"            "the"             "style"           "of"
[13] "a"             "black-and-white" "silent"          "film."
[17] "It"            "was"             "written,"        "directed,"
[21] "and"           "co-edited"       "by"              "Michel"
[25] "Hazanavicius," "produced"        "by"              "Thomas"
[29] "Langmann"      "and"             "starred"         "Jean"
[33] "Dujardin"      "and"             "Bérénice"        "Bejo."
[37] "The"           "story"           "takes"           "place"
[41] "in"            "Hollywood,"      "between"         "1927"
[45] "and"           "1932,"           "and"             "focuses"
[49] "on"            "the"             "relationship"    "of"
[53] "an"            "older"           "silent"          "film"
[57] "star"          "and"             "a"               "rising"
[61] "young"         "actress"         "as"              "silent"
[65] "cinema"        "falls"           "out"             "of"
[69] "fashion"       "and"             "is"              "replaced"
[73] "by"            "the"             "talkies."
```

since in the original string, each word is

Basic String Operations

```
paste(sorted_list, collapse = " ")
```

```
[1] "1927 1932, 2011 a a a actress an and and and and and and and Artist
as Bejo. Bérénice between black-and-white by by by cinema co-edited comed
y-drama directed, Dujardin falls fashion film film. focuses French Hazana
vicius, Hollywood, in in is is It Jean Langmann Michel of of of older on
out place produced relationship replaced rising romantic silent silent si
lent star starred story style takes talkies. the the the The The Thomas w
as written, young"
```

Basic String Operations

```
sub_string <- substr(summary[1], start = 4, stop = 50)
sub_string
```

```
[1] " Artist is a 2011 French romantic comedy-drama "
```

```
trimws(sub_string)
```

```
[1] "Artist is a 2011 French romantic comedy-drama"
```

Basic String Operations

```
summary[1]
```

```
[1] "The Artist is a 2011 French romantic comedy-drama in the style of a
black-and-white silent film. It was written, directed, and co-edited by M
ichel Hazanavicius, produced by Thomas Langmann and starred Jean Dujardin
and Bérénice Bejo. The story takes place in Hollywood, between 1927 and 1
932, and focuses on the relationship of an older silent film star and a r
ising young actress as silent cinema falls out of fashion and is replaced
by the talkies."
```

```
library(stringr)
str_sub(summary[1], -8, -1)
```

```
[1] "talkies."
```

In this video, we're going to show you several functions for manipulating strings in the R programming language.
To begin, we're going to read in text from a file in order to demonstrate the string operations.
The "summary" variable will hold a vector with 3 elements that represent the 3 lines of the file.
You can see what those lines look like here.
If you'd like to count the number of characters in a string, you can use the "nchar" function while passing the string in as an argument.
As you can see the first element of our vector is a string with 450 characters.
If you want to convert all the characters of a string into upper case, you can use the "toupper" function.
You can see how that would look here.
Similarly, you can use the "tolower" function if you'd like to convert all the string's characters to lower case.
In order to replace a specific set of characters in a string, you can use the "chartr" function.
The first argument is a string that contains the characters you'd like to replace.
The second argument is a string that holds the replacement characters.
And the last argument is the string you'd like to apply the operation on.
Based on the output, you can see how the function replaced every space character with a hyphen.
The "strsplit" function allows you to break apart a string by using some kind of expression.
Take a look at the syntax here.
The first argument is the string to split, and the second argument is the expression that we want to use to perform the split.
In this case, we're breaking up the string by the space character.
This will return a list, so to form a character vector, we'll need to use the "unlist" function.
When you look at the output, you'll notice that the vector has one element per word, since in the original string, each word is separated by a space character.
The "word list" vector we just created can be sorted as well, by passing it into the "sort" function.
Doing so will order the elements alphabetically.
We can also concatenate the elements of a character vector by using the "paste" function.


The "collapse" argument determines the string value that will separate the individual elements.
In our case, we'll simply separate them by a single space character.
The output here is a single string representing our alphabetically sorted list.
To isolate a specific portion of a string, you can use the "substr" function.
Simply pass in the start and end indices of the segment you want, and this contiguous segment will be output.
But notice that this particular substring has a leading and a trailing space character.
To get rid of them, we can use the "trimws" function, which removes all whitespace from the beginning and end of a string.
There may be times that you want to form a substring by counting back from the last position.
So for example, you may want the last 8 characters, like the ones you see here.
To do this, you'll need to use the "str_sub" function from the "stringr" library.
Notice how the start and end point arguments are negative in this case.
So the start point is the eighth character from the last point of the string, and the end point is the index of the last character.
You can see in the output that this successfully retrieved the last eight characters.
By now, you should understand how to modify characters in a string, how to split a string into a vector, and how to retrieve specific substrings.

>> Lab:

About the Dataset

In this module, we are going to use **The_Artist.txt** file. This file contains text data which is basically summary of the **The Artist** movie and we are going to perform various operations on this data.

This is how our data look like.

The Artist is a 2011 French romantic comedy-drama in the style of a black-and-white silent film. It was written, directed, and co-edited by Michel Hazanavicius, produced by Thomas Langmann and starred Jean Dujardin and Bérénice Bejo. The story takes place in Hollywood, between 1927 and 1932, and focuses on the relationship of an older silent film star and a rising young actress as silent cinema falls out of fashion and is replaced by the talkies.

The Artist received highly positive reviews from critics and won many accolades. Dujardin won the Best Actor Award at the 2011 Cannes Film Festival, where the film premiered. The film was nominated for six Golden Globes, the most of any 2011 film, and won three: Best Motion Picture – Musical or Comedy, Best Original Score, and Best Actor – Motion Picture Musical or Comedy for Dujardin. In January 2012, the film was nominated for twelve BAFTAs, the most of any film from 2011 and won seven, including Best Film, Best Director and Best Original

Reading Text Files

To read text files in R, we can use the built-in R function **readLines()**. This function takes **file path** as the argument and read the whole file.

Let's read the **The_Artist.txt** file and see how it looks like.

```
my_data <- readLines("The_Artist.txt")
my_data
```

'The Artist is a 2011 French romantic comedy-drama in the style of a black-and-white silent film. It was written, directed, and co-edited by Michel Hazanavicius, produced by Thomas Langmann and starred Jean Dujardin and Bérénice Bejo. The story takes place in Hollywood, between 1927 and 1932, and focuses on the relationship of an older silent film star and a rising young actress as silent cinema falls out of fashion and is replaced by the talkies.' ·
'' .
'The Artist received highly positive reviews from critics and won many accolades. Dujardin won the Best Actor Award at the 2011 Cannes Film Festival, where the film premiered. The film was nominated for six Golden Globes, the most of any 2011 film, and won three: Best Motion Picture – Musical or Comedy, Best Original Score, and Best Actor – Motion Picture Musical or Comedy for Dujardin. In January 2012, the film was nominated for twelve BAFTAs, the most of any film from 2011 and won seven, including Best Film, Best Director and Best Original Screenplay for Hazanavicius,

Tip: If you got an error message here, make sure that you run the code cell above first to download the dataset.

So, we get a character vector which has three elements and these elements can be accessed as we access array.

Let's check the length of **my_data** variable

```
length(my_data)
```

5

Length of **my_data** variable is **5** which means it contains 5 elements.

Similarly, we can check the size of the file by using the **file.size()** method of R and it takes **file path** as argument and returns the number of bytes. By executing code block below, we will get **1065** at the output, which is the size of the file **in bytes**.

```
file.size("/resources/data/The_Artist.txt")
```

<NA>

There is another method **scan()** which can be used to read **.txt** files. The Difference in **readLines()** and **scan()** method is that, **readLines()** is used to read text files line by line whereas **scan()** method read the text files word by word.

scan() method takes two arguments. First is the **file path** and second argument is the string expression according to which we want to separate the words. Like in example below, we pass an empty string as the separator argument.

```
my_data1 <- scan("The_Artist.txt", "")
my_data1
```

'The' · 'Artist' · 'is' · 'a' · '2011' · 'French' · 'romantic' · 'comedy-drama' · 'in' · 'the' · 'style' · 'of' · 'a' · 'black-and-white' · 'silent' · 'film.' · 'It' · 'was' · 'written,' · 'directed,' · 'and' · 'co-edited' · 'by' · 'Michel' · 'Hazanavicius,' · 'produced' · 'by' · 'Thomas' · 'Langmann' · 'and' · 'starred' · 'Jean' · 'Dujardin' · 'and' · 'Bérénice' · 'Bejo.' · 'The' · 'story' · 'takes' · 'place' · 'in' · 'Hollywood,' · 'between' · '1927' · 'and' · '1932,' · 'and' · 'focuses' · 'on' · 'the' · 'relationship' · 'of' · 'an' · 'older' · 'silent' · 'film' · 'star' · 'and' · 'a' · 'rising' · 'young' · 'actress' · 'as' · 'silent' · 'cinema' · 'falls' · 'out' · 'of' · 'fashion' · 'and' · 'is' · 'replaced' · 'by' · 'the' · 'talkies.' · 'The' · 'Artist' · 'received' · 'highly' · 'positive' · 'reviews' · 'from' · 'critics' · 'and' · 'won' · 'many' · 'accolades.' · 'Dujardin' · 'won' · 'the' · 'Best' ·

```
length(my_data1)
```

177

String Operations

There are many string operation methods in R which can be used to manipulate the data. We are going to use some basic string operations on the data that we read before.

nchar()

The first function is **nchar()** which will return the total number of characters in the given string. Let's find out how many characters are there in the first element of **my_data** variable.

```
nchar(my_data[1])
```

450

toupper()

Now, sometimes we need the whole string to be in Upper Case. To do so, there is a function called **toupper()** in R which takes a string as input and provides the whole string in upper case at output.

```
toupper(my_data[3])
```

'THE ARTIST RECEIVED HIGHLY POSITIVE REVIEWS FROM CRITICS AND WON MANY ACCOLADES. DUJARDIN WON THE BEST ACTOR AWARD AT THE 2011 CANNES FILM FESTIVAL, WHERE THE FILM PREMIERED. THE FILM WAS NOMINATED FOR SIX GOLDEN GLOBES, THE MOST OF ANY 2011 FILM, AND WON THREE: BEST MOTION PICTURE – MUSICAL OR COMEDY, BEST ORIGINAL SCORE, AND BEST ACTOR – MOTION PICTURE MUSICAL OR COMEDY FOR DUJARDIN. IN JANUARY 2012, THE FILM WAS NOMINATED FOR TWELVE BAFTAS, THE MOST OF ANY FILM FROM 2011 AND WON SEVEN, INCLUDING BEST FILM, BEST DIRECTOR AND BEST ORIGINAL SCREENPLAY FOR HAZANAVICIUS, AND BEST ACTOR FOR DUJARDIN.'

In above code block, we convert the third element of the character vector in upper case.

tolower()

Similarly, **tolower()** method can be used to convert whole string into lower case. Let's convert the same string that we convert in upper case, into lower case.

```
tolower(my_data[3])
```

'the artist received highly positive reviews from critics and won many accolades. dujardin won the best actor award at the 2011 cannes film festival, where the film premiered. the film was nominated for six golden globes, the most of any 2011 film, and won three: best motion picture – musical or comedy, best original score, and best actor – motion picture musical or comedy for dujardin. in january 2012, the film was nominated for twelve baftas, the most of any film from 2011 and won seven, including best film, best director and best original screenplay for hazanavicius, and best actor for dujardin.'

We can clearly see the difference between the outputs of last two methods.

chartr()

`what if we want to replace any characters in given string?` This operation can also be performed in R using **chartr()** method which takes three arguments. The first argument is the characters which we want to replace in string, second argument is the new characters and the last argument is the string on which operation will be performed.

Let's replace **white spaces** in the string with the **hyphen ("-") sign** in the first element of the **my_data** variable.

```
chartr(" ", "-", my_data[1])
```

'The-Artist-is-a-2011-French-romantic-comedy-drama-in-the-style-of-a-black-and-white-silent-film.-It-was-written,-directed,-and-co-edited-by-Michel-Hazanavicius,-produced-by-Thomas-Langmann-and-starred-Jean-Dujardin-and-Bérénice-Bejo.-The-story-takes-place-in-Hollywood,-between-1927-and-1932,-and-focuses-on-the-relationship-of-an-older-silent-film-star-and-a-rising-young-actress-as-silent-cinema-falls-out-of-fashion-and-is-replaced-by-the-talkies.'

strsplit()

Previously, we learned that we can read file word by word using **scan()** function. `But what if we want to split the given string word by word?`

This can be done using **strsplit()** method. Let's split the string according to the white spaces.

```
character_list <- strsplit(my_data[1], " ")
word_list <- unlist(character_list)
word_list
```

'The' · 'Artist' · 'is' · 'a' · '2011' · 'French' · 'romantic' · 'comedy-drama' · 'in' · 'the' · 'style' · 'of' · 'a' · 'black-and-white' · 'silent' · 'film.' · 'It' · 'was' · 'written,' · 'directed,' · 'and' · 'co-edited' · 'by' · 'Michel' · 'Hazanavicius,' · 'produced' · 'by' · 'Thomas' · 'Langmann' · 'and' · 'starred' · 'Jean' · 'Dujardin' · 'and' · 'Bérénice' · 'Bejo.' · 'The' · 'story' · 'takes' · 'place' · 'in' · 'Hollywood,' · 'between' · '1927' · 'and' · '1932,' · 'and' · 'focuses' · 'on' · 'the' · 'relationship' · 'of' · 'an' · 'older' · 'silent' · 'film' · 'star' · 'and' · 'a' · 'rising' · 'young' · 'actress' · 'as' · 'silent' · 'cinema' · 'falls' · 'out' · 'of' · 'fashion' · 'and' · 'is' · 'replaced' · 'by' · 'the' · 'talkies.'

In above code block, we separate the string word by word, but **strsplit()** method provides a list at the output which contains all the separated words as single element which is more complex to read. So, to make it more easy to read each word as single element, we used **unlist()** method which converts the list into character vector and now we can easily access each word as a single element.

sort()

Sorting is also possible in R. Let's use **sort()** method to sort elements of the **word_list** character vector in ascending order.

```
sorted_list <- sort(word_list)
sorted_list
```

'1927' · '1932,' · '2011' · 'a' · 'a' · 'a' · 'actress' · 'an' · 'and' · 'and' · 'and' · 'and' · 'and' · 'and' · 'Artist' · 'as' · 'Bejo.' · 'Bérénice' · 'between' · 'black-and-white' · 'by' · 'by' · 'by' · 'cinema' · 'co-edited' · 'comedy-drama' · 'directed,' · 'Dujardin' · 'falls' · 'fashion' · 'film' · 'film.' · 'focuses' · 'French' · 'Hazanavicius,' · 'Hollywood,' · 'in' · 'in' · 'is' · 'is' · 'It' · 'Jean' · 'Langmann' · 'Michel' · 'of' · 'of' · 'of' · 'older' · 'on' · 'out' · 'place' · 'produced' · 'relationship' · 'replaced' · 'rising' · 'romantic' · 'silent' · 'silent' · 'silent' · 'star' · 'starred' · 'story' · 'style' · 'takes' · 'talkies.' · 'the' · 'the' · 'the' · 'The' · 'The' · 'Thomas' · 'was' · 'written,' · 'young'

paste()

Now, we sort all the elements of ** word_list** character vector. Let's use **paste()** function here, which is used to concatenate strings. This method takes two arguments, the strings we want to concatenate and **collapse** argument which defines the separator in the words.

Here, we are going to concatenate all words of **sorted_list** character vector into a single string.

```
paste(sorted_list, collapse = " ")
```

'1927 1932, 2011 a a a actress an and and and and and and and Artist as Bejo. Bérénice between black-and-white by by by cinema co-edited comedy-drama directed, Dujardin falls fashion film film. focuses French Hazanavicius, Hollywood, in in is is It Jean Langmann Michel of of of older on out place produced relationship replaced rising romantic silent silent silent star starred story style takes talkies. the the the The The Thomas was written, young'

substr()

There is another function **substr()** in R which is used to get a sub section of the string.

Let's take an example to understand it more. In example below, we use the **substr()** method and provide it three arguments. First argument is the data string from which we want the sub string. Second argument is the starting point from where function will start reading the string and the third argument is the stopping point till where we want the function to read string.

```
sub_string <- substr(my_data[1], start = 4, stop = 50)
sub_string
```

Artist is a 2011 French romantic comedy-drama '

So, from the character vector, we start reading the first element from 4th position and read the string till 50th position and at the output, we get the resulted string which we stored in **sub_string** variable.

trimws()

As the sub string that we get in code block above, have some white spaces at the initial and end points. So, to quickly remove them, we can use **trimws()** method of R like shown below.

```
trimws(sub_string)
```

'Artist is a 2011 French romantic comedy-drama'

So, at the output, we get the string which does not contain any white spaces at the both ends.

str_sub()

To read string from last, we are using **stringr** library. This library contains **str_sub()** method, which takes same arguments as **sub_stirng** method but read string from last.

Like in the example below, we provide a data string and both starting and end points with negative values which indicates that we are reading string from last.

```
library(stringr)
str_sub(my_data[1], -8, -1)
```

'talkies.'

So, we read string from -1 till -8 and it gives **talkies.** with full stop mark at the output.

The Date Format in R (5:31)

The Date Class

```
"2016-08-15" "2016-09-15" "2016-10-15" "2016-11-15"
```

COGNITIVE CLASS.ai

Oscar-winning Actors dataset

```
bestActors
```

```
            Actor.Name Date.of.Birth
1    Leonardo DiCaprio    153360000
2       Eddie Redmayne    379123200
3  Matthew McConaughey     -5011200
4     Daniel Day-Lewis   -400032000
5         Jean Dujardin     77760000
6           Colin Firth   -293760000
7          Jeff Bridges   -633571200
8             Sean Penn   -295833600
9       Forest Whitaker   -257148800
10       Philip Seymour    -77155200
```

```
str(bestActors)
```

```
Date.of.Birth: int  153360000 37912320
```

that counts the number of seconds since the beginning of January 1, 1970.

that counts the number of seconds since the beginning of January 1, 1970.

BIG DATA UNIVERSITY

Converting UNIX time to Date

```
actors.birthday <- as.POSIXct(bestActors$Date.of.Birth,origin="1970-01-01")
```

YYYY-MM-DD HH:MM:SS

```
actors.birthday <- as.Date(actors.birthday)
```

```
[1] "1974-11-11" "1982-01-06" "1969-11-04" "1957-04-29" "1972-06-19"
[6] "1960-09-10" "1949-12-04" "1960-08-17" "1961-07-15" "1967-07-23"
```

Oscar-winning Actresses dataset

```
bestActresses
```

```
           Actor.Name Date.of.Birth
1        Brie Larson    1989/10/01
2     Julianne Moore    1960/12/03
3     Cate Blanchett    1969/05/14
4  Jennifer Lawrence    1990/08/15
5       Meryl Streep    1949/06/22
6    Natalie Portman    1981/06/09
7     Sandra Bullock    1964/07/26
8       Kate Winslet    1975/10/05
9   Marion Cotillard    1975/09/30
```

```
str(bestActresses)
```

```
Date.of.Birth: Factor w/ 10 levels "1945/07/26","1949/06/2
```

We've run into another issue, since the "date of birth" column holds factors, and not the

Converting Character formats to Date

```
actresses.birthday <- as.Date(bestActresses$Date.of.Birth,"%Y/%m/%d")
```

```
"1989-10-01" "1960-12-03" "1969-05-14" "1990-08-15" "1949-06-22"
"1981-06-09" "1964-07-26" "1975-10-05" "1975-09-30" "1945-07-26"
```

Date String Formatting

```
as.Date("27/06/94", "%d/%m/%y")
```

```
"1994-06-27"
```

```
"%a" = "Three character abbreviated weekday name"
"%A" = "Full weekday name"
"%b" = "Three character abbreviated month name"
"%B" = "Full month name"
"%d" = "Day of the month"
"%m" = "Month of the year"
"%y" = "Two digit year representation"
"%Y" = "Four digit year representation"
```

R allows you to specify the format in detail using the codes here, but by default, R will

Useful Date Functions

```
as.Date("1994/06/27") - as.Date("1959/01/01")
```

```
Time difference of 12961 days
```

```
as.Date("1994/06/27") > as.Date("1959/01/01")
```

```
TRUE
```

```
as.Date("1994/06/27") - 14
```

```
"1994-06-13"
```

In this case, the output

Useful Date Functions

```
Sys.Date()
```

```
"2016-08-15"
```

```
date()
```

```
"Mon Aug 15 15:51:48 2016"
```

```
Sys.time()
```

```
"2016-08-15 19:48:11 UTC"
```

Useful Date Functions

```
weekdays(Sys.Date())
```

```
"Monday"
```

```
months(Sys.Date())
```

```
"August"
```

```
quarters(Sys.Date())
```

```
"Q3"
```

And there's also a "quarters" function, wh

```
julian(Sys.Date())
```

```
[1] 17028
attr(,"origin")
[1] "1970-01-01"
```

```
seq(Sys.Date(),by="month",length.out=4)
```

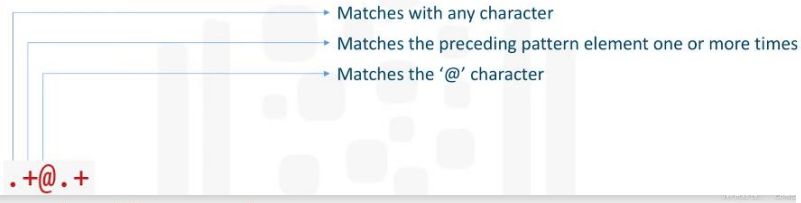
```
"2016-08-15" "2016-09-15" "2016-10-15" "2016-11-15"
```

In this video, we'll show you how to work with calendar dates in the R programming language.
R has a specialized object class for holding dates of a calendar.
The output here may simply look like strings, but R actually represents a date as the number of days since January 1 of 1970.
This particular representation for dates has become a standard across computer environments.
And as a special note, the dates are based off of the Gregorian calendar.
We're going to practice with the date format, but first, let's load up a database of Oscar-Winning actors.
As you can see, the date of birth for each actor is represented in a non-standard format.
By checking the structure of the data frame with the "str" function, we see that the "date of birth" column is of type "int".
As it turns out, this number is represented in the UNIX time format, another standard that counts the number of seconds since the beginning of January 1, 1970.
So we're going to have to reformat these values to work with R's date representation.
In order to do this, we'll use the "as.POSIXct" function.
POSIX is just another date format standard that R utilizes for its Date class structure.
But besides passing in the column we want to convert, we also need to provide the function with a reference point, which will be January 1 of 1970.
The function's output will be a full timestamp, which will have this structure.
Since we're only dealing with calendar dates, we don't need the hours, minutes, and seconds.
So we'll need to use one more function to get to the format we need.
The "as.Date" function takes in a character string or a POSIX timestamp, and converts it into a Date Class object.
You can see the final result in the output here.
Now let's consider another data frame that stores some information about Oscar-Winning Actresses.
Like before, we can pass this data frame to the "str" function in order to see its structure.
We've run into another issue, since the "date of birth" column holds factors, and not the Date class.
So we'll need to convert this column as well.
Once again we'll use "as.Date", which works with both factors and character types.
The additional parameter tells the function to expect the date format to be "Year Month Day", in that order, separated by slashes.
Doing this will successfully convert the column to the Date class.

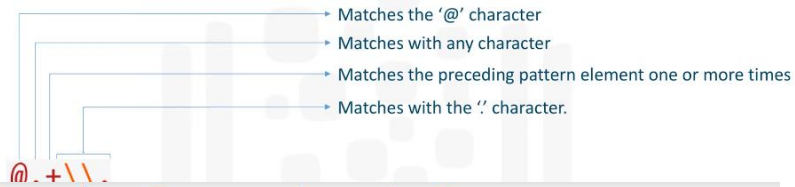
Let's look at another example to understand the date formatting parameter.
The first argument to the function is the string that we want to transform into a date object.
The second argument once again tells R that the first argument has the structure "day month year" separated by slashes.
As you can see here, R successfully parsed our string to form a Date object.
R allows you to specify the format in detail using the codes here, but by default, R will assume the date is "year month day" separated by hyphens or slashes.
Notice that since our second parameter has a lower case "y" for the year, R knew to look for a two-digit year representation.
There are several operations you can use on Date objects.
In this example, we simply subtract one date from another to get the number of days in between.
Here, we use a comparator to see if one date comes after another.
You can also subtract a number from a date.
In this case, the output is a date that is 14 days back in time.
There are also three functions that return the date and time based on the system.
"System Date" returns the current date as a Date class object.
The "date" function returns a character string describing the current date and time.
And "System Time" returns the current time in a POSIX timestamp format.
You can also apply some useful conversions on a Date Class Object.
The "weekdays" function converts the date into a string representing the weekday that the date falls on.
The "months" function works similarly.
And there's also a "quarters" function, which returns a string containing the quarter the date falls into.
If you wish, you can convert your date to the Julian calendar.
And as one last function, you can use the "seq" function to create a sequence of dates based on some reference point and length.
This example starts at the current date, and moves forward in one month increments.
By now, you should understand how to create dates in R, as well as how to apply various useful operations.

Regular Expressions

Regular Expressions



Regular Expressions



Regular Expressions in R

```
grep("@.*",  c("test@testing.com", "not an email", "test2@testing.com"
```

```
[1] 1 3
```

```
grep("@.*",  c("test@testing.com", "not an email", "test2@testing.com"), value=TRUE)
```

```
[1] "test@testing.com"  "test2@testing.com"
```

```
matches <- regexpr("@.*", c("test@testing.com", "not an email", "test2@testing.com"
regmatches(c("test@testing.com", "not an email", "test2@testing.com"), matches)
```

```
[1] "test@newdomain.com"  "not an email"        "test2@newdomain.com"
```

Solving The Problem

	Name	Email
1	John Doe	doej@example.com
2	Jane Doe	jadoe@sample.ca
3	Mark Mann	mmann@example.ca
4	Barry Goode	bgoode@example.com
5	Joe Star	joes@sample.com
6	Susan Quinn	qsus@example.br
7	Alice Erin	erina@example.com
8	Frank Irving	irving@sample.com

We'll then extract the matching substrings, and add them to a new column called "domain".

```
matches <- regexpr("@.+\\.", email_df[, "Email"])
email_df[,'Domain'] = regmatches(email_df[,'Email'], matches)
```

Regular Expression Applications

- Data Extraction
- Cleaning
- Data Analysis
- Data Validation
- Text Mining
- Parsing

In this video, we're going to show you how to work with regular expressions in the R programming language.
In order to demonstrate the regular expression operations, we're going to use the simple data frame here.
As you can see, it contains a few names, and email addresses from different regions.
Suppose our goal is to perform data analysis on each of the domains in the email addresses.
The problem is, some of the email addresses have regional differences, so the url's may differ, like the ones you see here.
So what we need to do is isolate all the characters between the "at sign" and period symbol.
This seems tricky at first since the url's can have variable lengths or strange characters, so regular expressions are perfect for a task like this.
Regular expressions are used to match patterns in strings and text.
Suppose you needed to express the structure of an email address, like the one here, but in the general case.
Basically, an email can be expressed as a set of characters, followed by an "at sign", followed by another set of characters.
What we just described could be written as this regular expression string.
We have the "at sign" in the middle, with text before it, and text after it.
So let's take a closer look at what all these symbols do.
The first symbol in the regular expression string is a period, but this has a special meaning.
A period is like a wild card, that will match any character.
The plus sign is also a special character.
It is used to match the preceding pattern element one or more times.
So a period followed by a plus sign will match any sequence of one or more characters.
The "at sign" simply matches an "at sign" in the string.
Remember that our goal is to isolate text between the "at sign" and the period symbol, but since the period symbol has a special meaning, we'll need to alter our expression

a bit.
Notice in this expression, we've added two backslashes before the period character.
This ensures that the period character itself will be matched in a string, rather than the period acting as a wild card.
So the expression you see here will match an "at sign", followed by one or more characters, followed by a period.
This is exactly what we need for our problem.
R provides several functions that make use of regular expressions.
The first one we'll look at is "grep".
"grep" takes in at least two inputs: The regular expression, and a list of strings you'd like to check for a match.
Notice that this regular expression contains an asterisk, rather than a plus sign.
An asterisk will match zero or more of the previous element, rather than one or more of the previous element.
Other than that subtle difference, their behavior is the same.
The output shows the list positions of the strings that match this regular expression.
You can use the "value" parameter to instruct the function to output the matching strings themselves.
You can also substitute strings found by the regular expression by using the "gsub" function.
The second argument serves as the replacement string.
In our case, all characters after an "at sign" will be replaced by "newdomain.com".
Notice how the second string was unaffected since there was no regular expression match.
In order to extract the matched strings, you can use the "regexpr" function, which is like a more detailed "grep".
This function will find the matching substrings.
We then pass the list of strings and the list of matches to the "regmatches" function, which gives us the desired output.
We're now ready to address our problem.
Let's apply a regular expression to the email column, making sure to isolate everything from the "at sign" to the period.
We'll then extract the matching substrings, and add them to a new column called "domain".
And here is our data frame with the new column.
This structure is now well-suited for data analysis.
We mainly focused on using regular expressions for data extraction, but they're used in a wide variety of areas, like data cleaning and data mining.
They're also used for text parsing, which helps with code compilation, so it's good to know how to work with regular expressions.
By now, you should understand how to use regular expressions to replace and extract patterns in your strings.

>> Lab:

Table of contents

Loading in Data

Let's load in a small list of emails to perform some data analysis and take a look at it.

```
email_df <- read.csv("https://ibm.box.com/shared/static/cbim8daa5vjf5rf4rlz113301vqbu7rk.csv")
email_df
```

A data.frame: 8 × 2

Name	Email
<fct>	<fct>
John Doe	doej@example.com
Jane Doe	jadoe@sample.ca
Mark Mann	mmann@example.ca
Barry Goode	bgoode@example.com
Joe Star	joes@sample.com
Susan Quinn	qsus@example.br
Alice Erin	erina@example.com
Frank Irving	irving@sample.com

So our simple dataset contains a list of names and a list of their corresponding emails. Let's say we want to simply count the frequency of email domains. But several problems arise before we can even attempt this. If we attempt to simply count the email column, we won't end up with what we want since every email is unique. And if we split the string at the '@', we still won't have what we want since even emails with the same domains might have different regional extensions. So how can we easily extract the necessary data in a quick and easy way?

Regular Expressions

Regular Expressions are generic expressions that are used to match patterns in strings and text. A way we can exemplify this is with a simple expression that can match with an email string. But before we write it, let's look at how an email is structured:

`$[test@testing.com](mailto://test@testing.com?cm_mmc=Email_Newsletter-_-Developer_Ed%2BTech-_-WW_WW-_-SkillsNetwork-Courses-IBMDeveloperSkillsNetwork-RP0101EN-SkillsNetwork-20900564&cm_mmca1=000026UJ&cm_mmca2=10006555&cm_mmca3=M12345678&cvosrc=email.Newsletter.M12345678&cvo_campaign=000026UJ&cm_mmc=Email_Newsletter-_-Developer_Ed%2BTech-_-WW_WW-_-SkillsNetwork-Courses-IBMDeveloperSkillsNetwork-RP0101EN-SkillsNetwork-20900564&cm_mmca1=000026UJ&cm_mmca2=10006555&cm_mmca3=M12345678&cvosrc=email.Newsletter.M12345678&cvo_campaign=000026UJ&cm_mmc=Email_Newsletter-_-Developer_Ed%2BTech-_-WW_WW-_-SkillsNetwork-Courses-IBMDeveloperSkillsNetwork-RP0101EN-SkillsNetwork-20900564&cm_mmca1=000026UJ&cm_mmca2=10006555&cm_mmca3=M12345678&cvosrc=email.Newsletter.M12345678&cvo_campaign=000026UJ)$`

So, an email is composed by a string followed by an '@' symbol followed by another string. In R regular expressions, we can express this as:

`$.+@.+$`

Where:

- The '.' symbol matches with any character.
- The '+' symbol repeats the previous symbol one or more times. So, '.+' will match with any string.
- The '@' symbol only matches with the '@' character.

Now, for our problem, which is extracting the domain from an email excluding the regional url code, we need an expression that specifically matches with what we want:

`$@.+\\.$`

Where the `'\.'` symbol specifically matches with the '.' character.

Regular Expressions in R

Now let's look at some R functions that work with R functions.

The grep function below takes in a Regular Expression and a list of strings to search through and returns the positions of where they appear in the list.

```
grep("@.+",  c("test@testing.com" , "not an email", "test2@testing.com"))
```

1 · 3

Grep also has an extra parameter called 'value' that changes the output to display the strings instead of the list positions.

```
grep("@.+",  c("test@testing.com", "not an email", "test2@testing.com"), value=TRUE)
```

'test@testing.com' · 'test2@testing.com'

The next function, 'gsub', is a substitution function. It takes in a Regular Expression, the string you want to swap in with the matches and a list of strings you want to perform the swap with. The code cell below updates valid emails with a new domain:

```
gsub("@.+", "@newdomain.com", c("test@testing.com", "not an email", "test2@testing.com"))
```

'test@newdomain.com' · 'not an email' · 'test2@newdomain.com'

The functions below, 'regexpr' and 'regmatches', work in conjunction to extract the matches found by a regular expression specified in 'regexpr'.

```
matches <- regexpr("@.*", c("test@testing.com", "not an email", "test2@testing.com"))
regmatches(c("test@testing.com", "not an email", "test2@testing.com"), matches)
```

'@testing.com' · '@testing.com'

This function is actually perfect for our problem since we simply need to extract the specific information we want. So let's use it with the Regular Expression we defin above and store the extracted strings in a new column in our dataframe.

```
matches <- regexpr("@.*\\.", email_df[,'Email'])
email_df[,'Domain'] = regmatches(email_df[,'Email'], matches)
```

And this is the resulting dataframe:

```
email_df
```

A data.frame: 8 × 3

Name	Email	Domain
<fct>	<fct>	<chr>
John Doe	doej@example.com	@example.
Jane Doe	jadoe@sample.ca	@sample.
Mark Mann	mmann@example.ca	@example.
Barry Goode	bgoode@example.com	@example.
Joe Star	joes@sample.com	@sample.
Susan Quinn	qsus@example.br	@example.
Alice Erin	erina@example.com	@example.
Frank Irving	irving@sample.com	@sample.

Saving completed Mode: Command Ln 1, Col 1 Regular_Expressior

```
[8]: table(email_df[,'Domain'])
```

```
@example.  @sample.
        5         3
```

How would you combine the individual words from the vector, `hw`, into a single string, "Hello World"?

```
hw <- c("Hello", "World")
```

- (•) `paste(hw, collapse = " ")` ✔
- ( ) `paste("Hello", "World")`
- ( ) `tolower("Hello", "World")`
- ( ) `c(hw[1], hw[2])`
- ( ) None of the above

Submit You have used 1 of 2 attempts Save

✔ Correct (1/1 point)

Review Question 2

1/1 point (graded)

How would you convert the character string "2020-01-01" into a Date object in R?

- (•) `as.Date("2020-01-01")` ✔
- ( ) `convertToDate("2020-01-01")`
- ( ) `date("2020-01-01")`
- ( ) `Sys.Date()`

What does the following regular expression pattern mean?

```
".+@.+"
```

- ( ) Find matches containing an @ symbol where there is one or more characters before the @ symbol, and zero or more characters after the @ symbol.
- ( ) Find matches containing an @ symbol where there is one or more characters before the @ symbol, and at least one character after the @ symbol.
- (•) Find matches containing an @ symbol where there is zero or more characters before the @ symbol, and at least one character after the @ symbol. ✔
- ( ) It's actually a new emoticon.

SUMMARY

R Basics

```
139 + 121
```

```
x <- 139 + 121
```

```
movie <- "Toy Story"
movie
```

```
genre_vector <- c("comedy","comedy","Animation","Animation","Crime")
```

```
genre_factor <- factor(genre_vector)
genre_factor
```

In addition, we explained how you could convert a vector into a factor, which is a type of

Data Structures in R

```
movie_array <- array(movie_vector, dim = c(4,3))
```

```
movie_matrix <- matrix(movie_vector , nrow = 3, ncol = 3 )
```

```
movie <- list("Toy Story", 1995, c("Animation","Adventure","Comedy"))
```

```
movies <- data.frame(name = c("Toy Story", "Akira", "The Breakfast Club", "The Artist",
                              "Modern Times", "Fight Club", "City of God", "The Untouchables"),
                     year = c(1995, 1998, 1985, 2011, 1936, 1999, 2002, 1987))
```

R Programming Fundamentals

```
movie_year <- 1997

if(movie_year > 2000){

    print('Movie year is greater than 2000')

} else {

    print('Movie year is not greater than 2000')

}
```

```
isGoodRating <- function(rating, threshold = 7){
    if(rating < threshold){
        return("NO")
    }else{
        return("YES")
    }
}
```

```
years <- c(1995, 1998, 1985, 2011, 1936, 1999)

for (yr in years) {
    print(yr)
}
```

```
class(average_rating)
```

```
count <- 1

while(count <= 5){
    print(c("Iteration number: ", count))
    count <- count + 1
}
```

```
tryCatch(
    for(i in 1:3){
        print(i + "a")
    , error = function(e) print("Found error.") )
```

In the "debugging" video, we used a "tryCatch" statement to handle errors and warnings.

Course Course Summary R 101 Course Summary (3:04) R 101 Course Summary (3:04)

R 101 Course Summary (3:04)
R 101 Course Summary (3:04)

Thank you for watching, and we hope you enjoyed "R 101".
3:04 / 3:04

Press UP to enter the speed menu then use the UP and DOWN arrow keys to navigate the different speeds, then press ENTER to change to the selected speed.

Click on this button to mute or unmute this video or press UP or DOWN buttons to increase or decrease volume level.
Maximum Volume.
Video transcript

Start of transcript. Skip to the end.
Let's recap what we've covered throughout the course.
In module 1, we showed you how to perform basic math operations, as well as how to assign values to variables.
We also briefly explained how to create a string.
Afterwards, we provided an overview of vectors, as well as several functions and operations that you can apply to them.
In addition, we explained how you could convert a vector into a factor, which is a type of variable that takes on a limited number of values.
Module 2 introduced several essential data types.
Arrays and matrices store objects of the same type.
The difference is that arrays can be multi-dimensional, but a matrix is strictly two-dimensional.
Lists, on the other hand, can hold data of different types.
And a data frame is R's primary structure for storing and working with data sets.
In Module 3, we showed you how to use "if else" statements to control the flow of execution in your program.
We used for loops to cycle through a set of objects, and while loops to perform a set of code while some condition was met.
Afterwards, we showed you how to write your code as a function so that it could be reused, and how to access several functions already built into R. The "class" function allowed us to see whether or not an object was the numeric, character, integer, or logical class.
And we showed you a few ways to convert between these class types.
In the "debugging" video, we used a "tryCatch" statement to handle errors and warnings.
Module 4 covered several functions for importing and exporting data.
We read data into csv, excel, and text files, and then later showed you how to save data back into these file formats.
The RData format allowed us to save multiple R objects to file by referencing variable names.
We also briefly covered a few of R's built-in datasets that can be accessed without any import stage.
Module 5 began with a series of string operations.
We showed you several ways to modify a string's contents, and to access specific substrings.
When discussing the date format, we showed you various methods for forming a date object that R can manipulate.
Afterwards, we introduced a few functions and operations that can be applied to dates.
Lastly, we explained the basics of how regular expressions perform pattern matching, and then we went over the functions that allow you to apply them to strings.
Now that you've completed this course, you should be ready to start writing your own applications in the R programming language.
Thank you for watching, and we hope you enjoyed "R 101".
End of transcript. Skip to the start.

https://courses.cognitiveclass.ai/courses/course-v1:BigDataUniversity+RP0101EN+2016/courseware/02089458e25d4e7d9477c193159c8e9b/8212d0e0ec744234bffd23d538389329/?child=first