Data Analytic using R

Lecture 5 – String manipulation and Web scraping

String manipulation

String is very common data type and a lot of data stored as simple text. There are a lot of functions that make automatic string processing to convert semi-structured data into structured data, e.g. `read.csv()`. Still there are many situations when we need to do text manipulations manually, e.g. processing of unstructured data.

As it is common for R, base package has a number of very capable functions and there are many packages that try to do the same job better. Let's review some of those functions – the most popular ones. In particular, we consider the package `stringr` from the `tidyverse` family.

`base` package functions

String splitting

We can split a string by any character or a combination of characters.

```r
# test string
test <- "The quick brown fox jumps over the lazy dog."

# split by space
strsplit(test, split = " ")
```

```
## [[1]]
## [1] "The"   "quick" "brown" "fox"   "jumps" "over"  "the"   "lazy"  "dog."
```

Very important to remember that any object in R is a vector. Hence, variable `test` is a vector of length 1 containing one string. The result of applying function `strsplit()` to that vector is a list where every element is a vector of strings after splitting the original string.

```r
# multiple strings vector
test <- c("The quick brown",  "fox jumps over", "the lazy dog.")

# split by space
strsplit(test, split = " ")
```

```
## [[1]]
## [1] "The"   "quick" "brown"
##
## [[2]]
## [1] "fox"   "jumps" "over"
##
## [[3]]
## [1] "the"  "lazy" "dog."
```

Input vector of strings is a character vector with three elements – split result is a list with three elements as well. Each element is a character vector.

This behavior is the only possible approach to deal with vectorised data but it requires extra actions when you want to extract some particular element from the split data.

```
test <- "The quick brown fox jumps over the lazy dog."
res <- strsplit(test, split = " ")

# get the first word from the above sentence
res[[1]][1]
```

```
## [1] "The"
```

```
# get first two words
head(res[[1]], 2)
```

```
## [1] "The"   "quick"
```

```
# get last word
tail(res[[1]], 1)
```

```
## [1] "dog."
```

Splitting can be done by any character or string. Delimiter or split element disappears, it can not be included in any string resulted from splitting.

```
test <- "The quick brown fox jumps over the lazy dog."

# split by letter "o"
strsplit(test, split = "o")
```

```
## [[1]]
## [1] "The quick br"   "wn f"           "x jumps "       "ver the lazy d"
## [5] "g."
```

```
# split by string "fox"
strsplit(test, split = "fox")
```

```
## [[1]]
## [1] "The quick brown "          " jumps over the lazy dog."
```

```
# interesting result from splitting by nothing - by empty string
strsplit(test, split = "")
```

```
## [[1]]
##  [1] "T" "h" "e" " " "q" "u" "i" "c" "k" " " "b" "r" "o" "w" "n" " " "f" "o" "x"
## [20] " " "j" "u" "m" "p" "s" " " "o" "v" "e" "r" " " "t" "h" "e" " " "l" "a" "z"
## [39] "y" " " "d" "o" "g" "."
```

Important parameter in the function `strsplit()` is `fixed`. Its default value is `FALSE` which means splitting will be done using *regular expressions*. Some other applications called them “wild cards”. We discuss them later. Here is a simple example

```r
test <- "The quick brown fox. Fox jumps over the lazy dog."

# splitting by a combination of two characters
# any character followed by "o". Full stop symbol stays as it is.
strsplit(test, split = ".o")
```

```
## [[1]]
## [1] "The quick b"   "wn "           "x. "           "x jumps"
## [5] "ver the lazy " "g."
```

If you want to split by full stop `.` (and some other special characters), you have to change parameter `fixed = TRUE`. This way `strsplit()` forgets about regular expressions and use split elements as they are

```r
test <- "The quick brown fox. Fox jumps over the lazy dog."

# splitting by a full stop
strsplit(test, split = ".", fixed = TRUE)
```

```
## [[1]]
## [1] "The quick brown fox"          " Fox jumps over the lazy dog"
```

Concatenation of strings

As we can split the string, we should be able to combine it back. HEre is a function `paste()`

```r
# several string objects combined by underscore
paste("a", "b", "c", sep = "_")
```

```
## [1] "a_b_c"
```

```r
# numerical vector will be converted to string as paste results only in strings
test <- c(1, 2, 3)

# concatenate every element of the vector
paste("aa", test, sep = "_")
```

```
## [1] "aa_1" "aa_2" "aa_3"
```

```r
# concatenate all elements of the vector in one string
paste(test, collapse = "_")
```

```
## [1] "1_2_3"
```

```r
# concatenate every element of the vector and then collapse them together
paste("aa", test, sep = "_", collapse = " ")
```

```
## [1] "aa_1 aa_2 aa_3"
```

```r
# parameters sep and collapse can be any strings
paste(test, "dog", sep = " nice ")
```

```
## [1] "1 nice dog" "2 nice dog" "3 nice dog"
```

Pattern matching and replacement

There is a family of functions to find and replace a given patter in a string or a character vector with multiple elements.

```r
test <- c("The quick brown fox jumps over the lazy dog.", "Hard working dog studies R.")

# check if pattern included in the elements of the character vector
grep(pattern = "dog", x = test)  # yes, dog is in elements 1 and 2
```

```
## [1] 1 2
```

```r
grep(pattern = "fox", x = test)  # yes, fox is in element 1
```

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

```r
grep(pattern = "lion", x = test) # no, there is no lion anywhere
```

```
## integer(0)
```

```r
# the same check as above but output is a boolean for each element of the vector
grepl(pattern = "dog", x = test)
```

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

```r
grepl(pattern = "fox", x = test)
```

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

```r
grepl(pattern = "lion", x = test)
```

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

```r
# replacement of the first matching
# only first character "o" replaced by "OOO"
sub(pattern = "o", replacement = "OOO",  x = test)
```

```
## [1] "The quick brOOOwn fox jumps over the lazy dog."
## [2] "Hard wOOOrking dog studies R."
```

```
# replacement of the all matches
# all characters "o" replaced by "OOO"
gsub(pattern = "o", replacement = "OOO",  x = test)
```

```
## [1] "The quick brOOOwn fOOOx jumps OOOver the lazy dOOOg."
## [2] "Hard wOOOrking dOOOg studies R."
```

More advanced functions to search for pattern matches

```
test <- c("The quick brown fox jumps over the lazy dog.", "Hard working dog studies R.")

# find location of the first pattern matching
# this is location 13 for first string and location 7 for second
regexpr(pattern = "o", text = test)
```

```
## [1] 13  7
## attr(,"match.length")
## [1] 1 1
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
```

```
# find all locations of the pattern matching
# these are location 13,18,27,42 for first string and locations 7,15 for second
gregexpr(pattern = "o", text = test)
```

```
## [[1]]
## [1] 13 18 27 42
## attr(,"match.length")
## [1] 1 1 1 1
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
##
## [[2]]
## [1]  7 15
## attr(,"match.length")
## [1] 1 1
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
```

```
# work the same as regexpr above but returns a list
regexec(pattern = "o", text = test)
```

```
## [[1]]
## [1] 13
## attr(,"match.length")
## [1] 1
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
##
## [[2]]
## [1] 7
## attr(,"match.length")
## [1] 1
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
```

All above functions can find/replace the exact match or they can work with *regular expressions*.

Regular expressions

Regular expressions are extremely useful for extraction information for any text. Regular expressions work [almost] the same way in many programming languages. Here is a link https://regexone.com/ with relatively simple interactive exercises to help you learn regular expressions. It is easy to learn basics but it is hard to master your skill of regular expressions as this is an extremely powerful tool. Below are some examples. Students are strongly encouraged to google for help if they want to get really complex matching rules.

```r
test <- "The quick brown fox jumps over the lazy dog."

# find "a" or "e" or "i" or "o" or "u", that is
# find any vowel and replace it by nothing
# result is a string of consonants only
gsub(pattern = "[aeiou]", replacement = "", x = test)
```

```
## [1] "Th qck brwn fx jmps vr th lzy dg."
```

```r
# another example
test <- "Your mark is 97 points."

# find one or more repetition of any numbers
temp_matches <- gregexpr(pattern = "[[:digit:]]+", text = test)

# extract information according to above object
temp_numbers <- regmatches(x = test, m = temp_matches)
temp_numbers   # check the result
```

```
## [[1]]
## [1] "97"
```

```r
# convert character to numbers
as.numeric(unlist(temp_numbers))
```

```
## [1] 97
```

```r
# more complex example
test <- "You're 79 years old. Your mark is 97 points."

# extract points only but ignore age
temp_matches <- gregexpr(pattern = "[[:digit:]]+.points", text = test)
regmatches(x = test, m = temp_matches)
```

```
## [[1]]
## [1] "97 points"
```

```r
# extract any piece of string that starts from a number and finishes by "s".
temp_matches <- gregexpr(pattern = "[0-9]+\\s\\S+s", text = test)
regmatches(x = test, m = temp_matches)
```

```
## [[1]]
## [1] "79 years"  "97 points"
```

```r
# one more example, assume that you have a list of files
file_list <- c("file_record_transcript.pdf", "file_07241999.pdf", "testfile_fake.pdf",
               "file.zip", "file_0.pdf.tmp")

# extract only file names that start with "file" and finish with ".pdf"
file_list[grepl(pattern = "^(file.+)\\.pdf$", x = file_list)]
```

```
## [1] "file_record_transcript.pdf" "file_07241999.pdf"
```

As you could see, some special characters above are used with a special meaning. For example `.` means any character, `+` means one or more repetitions of the previous character. As a result, you can not use `.` or `+` to split the character or to search for these symbols.

One way to solve this problem is to use parameter `fixed = TRUE`. Another way is to use *escape characters* defined by \ in front of them. Escape characters in R works a little bit different to other programming languages. If you need \ character in other language, you have to put two of them in R – \\

Have a look on examples below:

```r
strsplit("1.2.3", split = ".")
```

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

```r
strsplit("1.2.3", split = "\\.")
```

```
## [[1]]
## [1] "1" "2" "3"
```

```r
strsplit("1.2.3", split = ".", fixed = TRUE)
```

```
## [[1]]
## [1] "1" "2" "3"
```

```r
strsplit("1+2+3", split = "+")
```

```
## [[1]]
## [1] "1" "+" "2" "+" "3"
```

```r
strsplit("1+2+3", split = "\\+")
```

```
## [[1]]
## [1] "1" "2" "3"
```

There are several escape characters related to text formatting. There are part of the text but they are not normally visible.

```r
# tabulation
temp <- paste(c(1,2,3), collapse = "\t")
print(temp)
```

```
## [1] "1\t2\t3"
```

```r
cat(temp)
```

```
## 1    2   3
```

```r
# end of line
temp <- paste(c(1,2,3), collapse = "\n")
print(temp)
```

```
## [1] "1\n2\n3"
```

```r
cat(temp)
```

```
## 1
## 2
## 3
```

```r
# double quotation mark
temp <- paste(c(1,2,3), collapse = "\"")
print(temp)
```

```
## [1] "1\"2\"3"
```

```r
cat(temp)
```

```
## 1"2"3
```

stringr package

Package `stringr` does the same job as functions above. It might be more efficient for some operations. It is more convenient to use for others. Also, it is a part of `tidyverse` package family, hence it supports piping and compatible with `dplyr` and `purrr`. And it has a nice cheat sheet – https://stringr.tidyverse.org/

Quick reminder: cheat sheet is a brief introduction in to most popular functions. There are way more functions available – don't forget to check the help file.

```
library(stringr)
```

Split and concatenate characters

```
test <- c("The quick brown fox jumps over the lazy dog.", "Hard working dog studies R.")

# split by space - result is a list as before
str_split(string = test, pattern = " ")
```

```
## [[1]]
## [1] "The"   "quick" "brown" "fox"   "jumps" "over"  "the"   "lazy"  "dog."
##
## [[2]]
## [1] "Hard"    "working" "dog"     "studies" "R."
```

```
# split for N number of elements - result is a matrix
str_split_fixed(string = test, pattern = " ", n = 4)
```

```
##      [,1]   [,2]      [,3]    [,4]
## [1,] "The"  "quick"   "brown" "fox jumps over the lazy dog."
## [2,] "Hard" "working" "dog"   "studies R."
```

```
# an alternative syntax for the above functions using piping
test %>% str_split(" ")
```

```
## [[1]]
## [1] "The"   "quick" "brown" "fox"   "jumps" "over"  "the"   "lazy"  "dog."
##
## [[2]]
## [1] "Hard"    "working" "dog"     "studies" "R."
```

```
# concatenation for multiple characters
str_c("I", "love", "R", sep = "_")
```

```
## [1] "I_love_R"
```

```
# concatenation for each element of the vector
str_c(c(1,2,3), "test", sep = " ")
```

```
## [1] "1 test" "2 test" "3 test"
```

```
# concatenation for all elements of the vector in to one character
str_c(c(1,2,3), collapse = " = ")
```

```
## [1] "1 = 2 = 3"
```

```
# concatenate strings and expressions
name = "Joe"
age = 40
str_glue("My name is {name}, I'm {age} years old.",
         "Square root of {age} equals {sqrt(age)}.", .sep = "\n")
```

```
## My name is Joe, I'm 40 years old.
## Square root of 40 equals 6.32455532033676.
```

It is not a problem to produce the same result as `str_glue()` by using a humble `paste()`, however `str_glue()` does it easier. Just one more example for a data pipeline.

```
# top 6 rows from mtcars data set
head(mtcars) %>% str_glue_data("{rownames(.)} has {hp} hp")
```

```
## Mazda RX4 has 110 hp
## Mazda RX4 Wag has 110 hp
## Datsun 710 has 93 hp
## Hornet 4 Drive has 110 hp
## Hornet Sportabout has 175 hp
## Valiant has 105 hp
```

Pattern matching

Package `stringr` has a number of functions for pattern matching similar to `grep()` and `regexpr()` including the use of regular expressions. For example:

```
test <- c("apple", "orange")

# detect if a pattern "p" in the string
str_detect(string = test, pattern = "p")
```

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

```
# find indexes of elements with matching patterns
str_which(test, "p")
```

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

```
# count how many times pattern was matched
str_count(test, "p")
```

```
## [1] 2 0
```

```r
# locate the first position of pattern matches
str_locate(test, "p")  # result is matrix of indexes
```

```
##      start end
## [1,]     2   2
## [2,]    NA  NA
```

```r
# locate all positions of matching
str_locate_all(test, "p") # result is a list of matrices
```

```
## [[1]]
##      start end
## [1,]     2   2
## [2,]     3   3
##
## [[2]]
##      start end
```

```r
# locate position all vowels by using regular expression
str_locate_all(test, "[aeiou]")
```

```
## [[1]]
##      start end
## [1,]     1   1
## [2,]     5   5
##
## [[2]]
##      start end
## [1,]     1   1
## [2,]     3   3
## [3,]     6   6
```

All above functions help us detect and locate matching and then we can use these information to extract matching from the data. However, there is not need to do that manually. There are special functions that locate and extract data:

```r
test <- c("apple", "orange")

# extract the first vowel from each element
str_extract(test, "[aeiou]")
```

```
## [1] "a" "o"
```

```r
# extract all vowels from each element
str_extract_all(test, "[aeiou]")
```

```
## [[1]]
## [1] "a" "e"
##
## [[2]]
## [1] "o" "a" "e"
```

```r
# subset only elements that have the match
str_subset(test, "p")
```

```
## [1] "apple"
```

```r
# select characters by index from start to end
str_sub(test, start = 1, end = 3)
```

```
## [1] "app" "ora"
```

```r
# extract first match as a matrix
str_match(test, "[aeiou]")
```

```
##      [,1]
## [1,] "a"
## [2,] "o"
```

```r
# extract all matching as a list of matrices
str_match_all(test, "[aeiou]")
```

```
## [[1]]
##      [,1]
## [1,] "a"
## [2,] "e"
##
## [[2]]
##      [,1]
## [1,] "o"
## [2,] "a"
## [3,] "e"
```

It is possible to mutate strings. Well, strings are immutable but it is possible to create new strings with required changes.

```r
test <- c("apple", "orange")

# change all characters to upper case
str_to_upper(test)
```

```
## [1] "APPLE"  "ORANGE"
```

```r
# change all character to lower case
str_to_lower("TEst")
```

```
## [1] "test"
```

```r
# change the string to title case - first letter of each word in upper case
str_to_title("i love r.")
```

```
## [1] "I Love R."
```

```r
# replace the first vowel by _
str_replace(test, pattern = "[aeiou]", replacement = "_")
```

```
## [1] "_pple"  "_range"
```

```r
# replace the all vowels by _
str_replace_all(test, pattern = "[aeiou]", replacement = "_")
```

```
## [1] "_ppl_"  "_r_ng_"
```

Please check the cheat sheet referenced before for the most popular `stringr` functions and look through help file for the package as there are many other functions that might be useful. For example,

```r
test <- "         This text has        way too         many spaces.            "

# remove all vowels
str_remove_all(test, pattern = "[aeiou]")
```

```
## [1] "         Ths txt hs        wy t         mny spcs.            "
```

```r
# clean up leading and trailing white spaces from the text
str_trim(test)
```

```
## [1] "This text has        way too         many spaces."
```

```r
# clean up all excessive spaces
str_squish(test)
```

```
## [1] "This text has way too many spaces."
```

Web scraping using rvest

To do data analysis we need data. There are many ways to collect data and one of them is web scraping – reading HTML pages directly from the web and extracting useful information from these pages. We can read the page into a character vector and then try to work through HTML/CSS code manually using string manipulation functions.

```r
# load HTML page
my_data <- readLines("https://www.imdb.com/title/tt0076759/")
```

```
## Warning in readLines("https://www.imdb.com/title/tt0076759/"): incomplete final
## line found on 'https://www.imdb.com/title/tt0076759/'
```

```r
# check some lines from the character vector
my_data[7:17]
```

```
##  [1] "var ue_csm = window,"
##  [2] "    ue_hob = +new Date();"
##  [3] "(function(d){var e=d.ue=d.ue||{},f=Date.now||function(){return+new Date};e.d=function(b){retur
##  [4] ""
##  [5] ""
##  [6] "    var ue_err_chan = 'jserr';"
##  [7] "(function(d,e){function h(f,b){if(!(a.ec>a.mxe)&&f){a.ter.push(f);b=b||{};var c=f.logLevel||b.
##  [8] "pec:0,ts:0,erl:[],ter:[],mxe:50,startTimer:function(){a.ts++;setInterval(function(){d.ue&&a.pe
##  [9] ""
## [10] ""
## [11] "var ue_id = 'DT6E0MAXAJ8SWR4TPPGQ',"
```

Obviously this is not the easy way to go. Nowadays most web pages are computer generated, so they have standard structures and can be processed automatically. And there is a package for that - `rvest`. We still have to do some manual work to figure out what is useful information on the web page but then we can repeat the same data extraction procedure on the other web pages from the same web server.

Web scraping for one page

Open the web page of interest in the web browser and get to its HTML code. If you use Google Chrome, use right-click on the page and select "Inspect"; or go to settings, select "More tools" and then "Developer tools"; or just press `Ctrl + Shift + I`. The same instructions would work for Microsoft Edge, Opera, Brave and some other browsers as they based the same engine Chromium. Internet Explorer has menu "Inspect element" by right-click on the page. Either way, you need to see the code and its connection to web page elements.

First step is to load the web page:

```
# load the library
library(rvest)
```

```
## Warning: package 'rvest' was built under R version 4.2.1
```

```
# load the web page
html <- read_html("https://www.imdb.com/title/tt0088763/")

# the result is xml document
html
```

```
## {html_document}
## <html lang="en-US" xmlns:og="http://opengraphprotocol.org/schema/" xmlns:fb="http://www.facebook.com
## [1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset=UTF-8 ...
## [2] <body>\n<div>    <img height="1" width="1" style="display:none;visibility ...
```

In fact, the resulted variable is almost the same type of document as the character string before but we are not going to decode the entire document as there are way too much everything. We don't need most of that information. We just need to extract only small number of objects/values. For example, we need the movie title. Go through the hierarchical structure of the HTML/CSS code in the right window and look for the code related to the movie title on the web page.

Every tag in HTML/CSS code is a node and we can address that node by its name or its class.

```r
# ask for the node h1
html %>% html_elements("h1")
```

```
## {xml_nodeset (1)}
## [1] <h1 textlength="18" data-testid="hero-title-block__title" class="sc-b73cd ...
```

Result is xml document with the content of tag “h1”. Above code is very easy. The result might be difficult to use as there might be multiple tags “h1” in the document, so you will get them all. To be more precised, we can use elements of the path to that tag. For example,

```r
# ask for the node h1 located within the object of class title_wrapper
html %>% html_elements(".sc-80d4314-0.fjPRnj h1")
```

```
## {xml_nodeset (1)}
## [1] <h1 textlength="18" data-testid="hero-title-block__title" class="sc-b73cd ...
```

Please note the dot in front of class name. This is an indication that it is a class and not the tag. “Strange” combination of letters and numbers is a class name from the code two levels up in the hierarchy `<div class="sc-80d4314-0 fjPRnj">`. Class name can not have spaces, so space should be replaced by dot.

To get the actual text instead of xml code, we use function `html_text()`

```r
# get the text from selected node and trim white spaces
html %>% html_elements(".sc-80d4314-0.fjPRnj h1") %>% html_text(trim = TRUE)
```

```
## [1] "Back to the Future"
```

Movie year is inside code in the same black as a title.

```r
# select the node from class "sc-80d4314-0.fjPRnj" from tag "a"
html %>% html_elements(".sc-80d4314-0.fjPRnj a")
```

```
## {xml_nodeset (4)}
## [1] <a class="ipc-link ipc-link--baseAlt ipc-link--inherit-color sc-8c396aa2- ...
## [2] <a class="ipc-link ipc-link--baseAlt ipc-link--inherit-color sc-8c396aa2- ...
## [3] <a class="ipc-button ipc-button--single-padding ipc-button--center-align- ...
## [4] <a class="ipc-button ipc-button--single-padding ipc-button--center-align- ...
```

There are multiple hyperlinks () in that block. There are many ways to extract a piece of information we need.

First, add more information from CSS hierarchy as we did above

```r
# select the node from class "sc-80d4314-0.fjPRnj" from tag "a"
html %>% html_elements(".sc-80d4314-0.fjPRnj .ipc-inline-list__item a") %>% html_text(trim = TRUE)
```

```
## [1] "1985" "PG"
```

There are still two elements, but both of them are useful and we can use indexing to extract each of them individually.

Second method, to use indexing directly on XML object, which is almost the same as a list in R.

```r
# select the node the same way as before
(html %>% html_elements(".sc-80d4314-0.fjPRnj a"))[[1]] %>% html_text(trim = TRUE)
```

```
## [1] "1985"
```

We got a year. Please pay attention to round brackets before indexing `[[1]]`.

Release year is a character as everything on the HTML page but we can convert it to number

```r
(html %>% html_elements(".sc-80d4314-0.fjPRnj a"))[[1]] %>%
  html_text(trim = TRUE) %>% as.numeric()
```

```
## [1] 1985
```

What is other useful information for data analysis of movies in the international movies data base (IMDB)? Rating, length of the movie, genres, movie plot. Let's try to extract them.

```r
# Rating is in the node of class "sc-7ab21ed2-1.jGRxWM" - there are two copies of it again
# will fix it later
html %>% html_elements(".sc-7ab21ed2-1.jGRxWM ") %>% html_text(trim = TRUE) %>% as.numeric()
```

```
## [1] 8.5 8.5
```

```r
# Length of the movie
length <- (html %>% html_elements(".sc-80d4314-2.iJtmbR li"))[[3]] %>% # three elements again
  html_text(trim = TRUE)           # we select third one - result is "1h 56m"
# use regular expressions to extract number and convert everything to minutes
  h <- as.numeric(sub("h.*", "", length))       # hours
  m <- as.numeric(sub("m.*", "", sub(".*h ", "", length))) # minutes
  sum(c(h * 60, m), na.rm = TRUE)     # result in minutes
```

```
## [1] 116
```

```r
# Genres are in the individual hyperlinks with tag "a", there are several nodes
# inside class "ipc-chip-list__scroller"
html %>% html_elements(".ipc-chip-list__scroller a") %>%
  html_text(trim = TRUE)  %>%     # result is a character vector
  str_flatten(",")                # combine all elements together
```

```
## [1] "Adventure,Comedy,Sci-Fi"
```

There is another way to extract data by using custom attributes of HTML tags, which could work for some cases above too. For example, movie plot is inside tag `span` with attribute `data-testid="plot-l"`. Using just tag `span` is counterproductive – there are too many of them. However, that attribute is unique for movie plot.

```r
# Movie plot
html %>% html_elements('span[data-testid="plot-l"]') %>%
  html_text(trim = TRUE)
```

```
## [1] "Marty McFly, a 17-year-old high school student, is accidentally sent 30 years into the past in a
```

Beware of a correct use of quotation marks. Attribute value is in double quotation, hence we have to use single quotation the entire parameter.

Let's assume that we have all required elements. As we plan to apply the above code to multiple pages, we should have a function that takes a web page URL as an input and outputs all required pieces of information as a data frame.

```r
scrap_movie <- function(url){
  # function to scrape the web page

  # load web page
  html <- read_html(url)

  # get movie title
  movie_title <- html %>% html_elements("h1") %>% html_text(trim = TRUE)

  # get year
  movie_year <- (html %>% html_elements(".sc-80d4314-0.fjPRnj a"))[[1]] %>%
                html_text(trim = TRUE) %>% as.numeric()

  # get movie rating - take first value only
  movie_rating <- (html %>% html_elements(".sc-7ab21ed2-1.jGRxWM ") %>%
    html_text(trim = TRUE) %>% as.numeric())[1]

  # get movie length
  length <- (html %>% html_elements(".sc-80d4314-2.iJtmbR li"))[[3]] %>% html_text(trim = TRUE)
  h <- as.numeric(sub("h.*", "", length))                     # hours
  m <- as.numeric(sub("m.*", "", sub(".*h ", "", length)))    # minutes
  movie_length <- sum(c(h * 60, m), na.rm = TRUE)             # result in minutes

  # get genres
  movie_genres <- html %>% html_elements(".ipc-chip-list__scroller a") %>%
  html_text(trim = TRUE) %>% str_flatten(",")

  # get movie plot
  movie_plot <- html %>% html_elements('span[data-testid="plot-l"]') %>%
    html_text(trim = TRUE)

  # store results together as one-line dataframe
  res <- data.frame(movie_title, movie_year, movie_rating, movie_length,
                    movie_genres, movie_plot)

  return(res)
}

# test the function
scrap_movie("https://www.imdb.com/title/tt0088763/")
```

```
##           movie_title movie_year movie_rating movie_length
## 1 Back to the Future       1985          8.5          116
##               movie_genres
## 1 Adventure,Comedy,Sci-Fi
```

```
##
## 1 Marty McFly, a 17-year-old high school student, is accidentally sent 30 years into the past in a ti
```

The function works OK for the movie we used before. We can try it on any other movie or web page from IMDB.

```
# movie code is different by one 63 -> 64
scrap_movie("https://www.imdb.com/title/tt0088764/")
```

```
##      movie_title movie_year movie_rating movie_length movie_genres movie_plot
## 1 Backfischliebe       1985            6          100        Drama
```

No error messages, no major problems. However, movie plot is missing – if you check the web-page, you see there is no movie plot there.

Web scraping for multiple pages

For data analysis we need more data than one or two movies, we need MORE… For example, we can select "Top Rated Movies" in the IMDB menu and get to this page: https://www.imdb.com/chart/top/?ref_=nv_mv_250 . There are 250 movies, let's get them all. There are two approaches:

1. Get information from this page only: title, year, rating
2. Get URL for each title and check their individual pages to get all available information.

We try both of them at the same time. Again, we need to inspect HTML/CSS code to understand the structure of the page.

```
# load the web page
url <- "https://www.imdb.com/chart/top/?ref_=nv_mv_250"
html <- read_html(url)

html
```

```
## {html_document}
## <html xmlns:og="http://ogp.me/ns#" xmlns:fb="http://www.facebook.com/2008/fbml">
## [1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset=UTF-8 ...
## [2] <body id="styleguide-v2" class="fixed">\n            <img height="1" widt ...
```

There is a very long way through the hierarchy of nodes (HTML/CSS tags) to find the useful information – one long table with 250 movies. Hopefully we don't need the full path and table class is unique.

```
df <- html %>% html_node(".chart") %>%     # beware space in the class name
  html_table()                             # extract the table from the node

head(df)
```

```
## # A tibble: 6 x 5
##   ``    `Rank & Title`                          `IMDb Rating` `Your Rating` ``
##   <lgl> <chr>                                           <dbl> <chr>         <lgl>
## 1 NA    "1.\n      The Shawshank Redemption\n~           9.2 "12345678910~ NA
## 2 NA    "2.\n      The Godfather\n        (19~           9.2 "12345678910~ NA
## 3 NA    "3.\n      The Dark Knight\n        (~           9   "12345678910~ NA
## 4 NA    "4.\n      The Godfather: Part II\n  ~           9   "12345678910~ NA
## 5 NA    "5.\n      12 Angry Men\n         (195~           8.9 "12345678910~ NA
## 6 NA    "6.\n      Schindler's List\n        ~           8.9 "12345678910~ NA
```

```
names(df)
```

```
## [1] ""              "Rank & Title" "IMDb Rating"  "Your Rating"  ""
```

We got the table that needs some cleaning but it is a data frame and you know how to deal with it. Packages `dplyr` and `tydir` will be very helpful.

```
suppressMessages(library(tidyverse))

# clean up te dat frame
df <- df %>% select(c(2,3)) %>%                    # get only two meaningful columns
  separate(1, into = c("None", "Title", "Year"), sep = "\n ") %>%  # split the column
  select(-None, Rating = 4) %>%    # remove first column and rename the last one
  mutate(Year = Year %>% str_remove_all("[:punct:]") %>% as.numeric()) # clean Year

# check the result
head(df)
```

```
## # A tibble: 6 x 3
##   Title                           Year Rating
##   <chr>                          <dbl>  <dbl>
## 1 "     The Shawshank Redemption"  1994    9.2
## 2 "     The Godfather"             1972    9.2
## 3 "     The Dark Knight"           2008    9
## 4 "     The Godfather: Part II"    1974    9
## 5 "     12 Angry Men"              1957    8.9
## 6 "     Schindler's List"          1993    8.9
```

Depending on your goal there might be different strategies for web-scraping. The first approach is to get the limited data but work with one page only. For the second approach, we don't need movie titles and ratings but only a list of URLs for all 250 movies. There are multiple hyperlinks for each row – we need only the one from column with class="titleColumn".

```
#
urls <- html %>% html_node(".chart") %>%     # get the node with table
  html_elements(".titleColumn a") %>%     # get all tag "a" nodes in column "titleColumn"
  html_attr("href")      # extract value of attribute "href" that contains a hyperlink

# check the length of the vector
length(urls)     # 250 as we expected
```

```
## [1] 250
```

```
# check first two elements
urls[1:2]
```

```
## [1] "/title/tt0111161/?pf_rd_m=A2FGELUUNOQJNL&pf_rd_p=1a264172-ae11-42e4-8ef7-7fed1973bb8f&pf_rd_r=H
## [2] "/title/tt0068646/?pf_rd_m=A2FGELUUNOQJNL&pf_rd_p=1a264172-ae11-42e4-8ef7-7fed1973bb8f&pf_rd_r=H
```

Two important considerations for the above results:

1. All links are "relative" to the main page, we need to convert them to absolute links by adding web server address
2. All links are too long, we don't really need all that stuff. We need only the part before the first question mark.

Package `stringr` helps to solve both problems.

```
urls <- urls %>% word(1, sep = "\\?") %>%     # select the beginning of the url
  str_c("https://www.imdb.com", .)           # add IMDB address to each url

# check the top 6 links
urls[1:6]
```

```
## [1] "https://www.imdb.com/title/tt0111161/"
## [2] "https://www.imdb.com/title/tt0068646/"
## [3] "https://www.imdb.com/title/tt0468569/"
## [4] "https://www.imdb.com/title/tt0071562/"
## [5] "https://www.imdb.com/title/tt0050083/"
## [6] "https://www.imdb.com/title/tt0108052/"
```

Now we are ready to scrape all 250 movies – it will take some time. You don't want to do it multiple times. Test your code on 2-3 links first; if everything works fine – download all movies and save them as RData file to use for the analysis later.

```
# test for three movies only and use "purrr" functionality and custom function prepared
# apply custom function to every url and combine results
ddf <- urls[1:3] %>% map_dfr(scrap_movie)

# check the result
ddf
```

```
##                movie_title movie_year movie_rating movie_length
## 1 The Shawshank Redemption       1994          9.3          142
## 2            The Godfather       1972          9.2          175
## 3          The Dark Knight       2008          9.0          152
##        movie_genres
## 1             Drama
## 2       Crime,Drama
## 3 Action,Crime,Drama
##
## 1                                                                        Two imprisoned men bond ove
## 2                                        The aging patriarch of an organized crime dynasty in postw
## 3 When the menace known as the Joker wreaks havoc and chaos on the people of Gotham, Batman must acc
```

```
# check variables
names(ddf)
```

```
## [1] "movie_title"  "movie_year"   "movie_rating" "movie_length" "movie_genres"
## [6] "movie_plot"
```

Everything looks good, so we are ready to get more data.

Quick reference for `rvest`

There are not so many functions in `rvest`. Here are the most popular/important functions

- `read_html(url)`: scrape HTML content from a given URL
- `html_elements("tag")`: calls node based on HTML/CSS *tag*
- `html_elements('tag[attribute="somename"]')`: calls node based on HTML/CSS *tag* and its attribute
- `html_elements(".class")`: calls node based on CSS *class*
- `html_elements("#id")`: calls node based on div *id*
- `html_text(trim = TRUE)`: strips the HTML tags and extracts only the text
- `html_attrs("href")`: extract value of tag attribute
- `html_table()`: turns HTML tables into data frames

If you need a help you always can google – there are a lot of `rvest` tutorials online.

Really nice presentation/tutorial on `rvest`. Use arrows left and right to go through it. http://www.stat.wmich.edu/naranjo/R/Web-Scraping-isoslides.html

Excellent chapter on using `rvest` in the online book on R. It is a bit outdated now, for example, it uses `html_nodes()` function while it is replaced by `html_elements()`. However, this is a minor inconvenience. It is an excellent book overall. https://jtr13.github.io/cc19/web-scraping-using-rvest.html