Strings

Working with text in R

Federica Fusi

MSCA, UIC

Updated: 2021-04-06

Working with strings

Strings

Strings represent character vectors or, in other words, text.

Text can mean different units of analysis:

- letters,
- words,
- sentences,
- paragraphs,
- n-grams,
- pages,
- chapters,
- documents....

You need to estalish your unit of analysis as you would for a numeric dataset.

Strings in R

Strings in R are surrounded by "" and can be stored in vectors as we did for other variables.

```
string1 <- c("a", "apple", "I am eating an apple")
```

Strings are manipulated with the package stringr (tidyverse).

Functions generally start with str_, so if you start typing "str_" you will see a list of all functions that you might need to work with strings.

Length

We can start with a simple function, which returns the lenght of each character elements within a string.

Note the difference in the output between str_ functions and traditional R functions.

```
str_length(string1) #Return the length of each character element
```

```
## [1]  1  5 20
```

```
length(string1) #Return the lenght of the string vector (e.g., how many
```

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

Combine strings

Before, I used the familiar c function to combine multiple character elements within a vector.

str_c will combine multiple character elements into one unique string.

As before, note the difference between the two functions by looking at the separator "" and how each element is either separated into multiple elements or put together in one unique string.

```
str1 <- str_c("I", "eat", "an", "apple")
str1
```

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

```
str2 <- c("I", "eat", "an", "apple")
str2
```

```
## [1] "I"     "eat"   "an"    "apple"
```

Combine strings

With str_c, you can use the argument sep to decide how to separate each element

```
str_c("I", "eat", "an", "apple", sep = " ")
```

```
## [1] "I eat an apple"
```

```
str_c("I", "eat", "an", "apple", sep = ",")
```

```
## [1] "I,eat,an,apple"
```

```
str_c("I", "eat", "an", "apple", sep = ".")
```

```
## [1] "I.eat.an.apple"
```

Combine strings

It is also possible to vectorize a string such that the same prefix and suffix are repeated across a set of words.

This can be quite helpful for longitudinal datasets.

```
string_2 = str_c("Year", c("2020", "2021", "2022"), "Pop")

string_2
```

```
## [1] "Year2020Pop" "Year2021Pop" "Year2022Pop"
```

```
string_2bis = str_c("Year_", c("2020", "2021", "2022"), "_Pop")

string_2bis
```

```
## [1] "Year_2020_Pop" "Year_2021_Pop" "Year_2022_Pop"
```

Collapse into one string

If you want to collapse all elements into one single string, you can use the argument collapse which combine multiple elements into one single string. You can specify the separators.

```
str2
```

```
## [1] "I"     "eat"   "an"    "apple"
```

```
str_c(str2, collapse = " ")
```

```
## [1] "I eat an apple"
```

```
str_c(str2, collapse = ",")
```

```
## [1] "I,eat,an,apple"
```

Subsetting

Another interesting function is str_sub which allows you to subset character elements. The structure of the function is the following str_sub(X, start, end).

```
str2
```

```
## [1] "I"     "eat"   "an"    "apple"
```

```
str_sub(str2, 1, 2)  # string, start, end
```

```
## [1] "I"  "ea" "an" "ap"
```

If you use negative numbers, they count backwards from end, such that -1 represents the last letter of a word.

```
str_sub(str2, -3, -1)
```

```
## [1] "I"   "eat" "an"  "ple"
```

str_ functions

There are several other str_ functions, some of which are presented here. I will leave you to explore them as you need them - see some examples in the next slides.

Function	Description
str_c()	string concatenation
str_length()	number of characters
str_sub()	extracts substrings
str_dup()	duplicates characters
str_trim()	removes leading and trailing whitespace
str_pad()	pads a string
str_wrap()	wraps a string
str_sort()	sorts a vector by alphabetical order

Lower and upper case

We have already seen a few functions to make a string in lower or upper case (e.g., tolower). These functions do the same within the tidyverse system.

```
str3 = c("APPLE", "PEAR", "BANANA")
str4 = str_to_lower(str3)
str4
```

```
## [1] "apple"  "pear"   "banana"
```

```
str3 = str_to_upper(str4)
str3
```

```
## [1] "APPLE"  "PEAR"   "BANANA"
```

Lower and upper case

The function str_to_title will capitalize only the first letter of each word.

```
str_to_title(str3)
```

```
## [1] "Apple"  "Pear"   "Banana"
```

```
str_to_title(str1)
```

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

Sort

str_sort is used to sort a series of strings in alphabetical order

```
str2
```

```
## [1] "I"     "eat"   "an"    "apple"
```

```
str3
```

```
## [1] "APPLE"  "PEAR"   "BANANA"
```

```
str_sort(str2)
```

```
## [1] "an"    "apple" "eat"   "I"
```

```
str_sort(str3)
```

```
## [1] "APPLE"  "BANANA" "PEAR"
```

Sort

It is also possible to reverse the order and specify where NA should be placed.

```
str_sort(str3, decreasing = T)
```

```
## [1] "PEAR"   "BANANA" "APPLE"
```

```
str5 <- c("banana", NA, "melon", "watermelon", NA)

str_sort(str5, na_last = T) # NAs go at the end
```

```
## [1] "banana"     "melon"      "watermelon" NA           NA
```

```
str_sort(str5, na_last = F) #NAs go at the beginning
```

```
## [1] NA           NA           "banana"     "melon"      "watermelon"
```

```
str_sort(str5, na_last = NA) #NAs are dropped
```

```
## [1] "banana"     "melon"      "watermelon"
```

Trim

If we want to trim leading and trailing whitespace in R, we can use the str_trim function as shown below.

```
str5 <- c("banana ", " melon", " watermelon     ")

str_trim(str5, side = "left") # remove whitespacing on the left side onl
```

```
## [1] "banana "         "melon"           "watermelon     "
```

```
str_trim(str5, side = "right") #remove whitespacing on the right side or
```

```
## [1] "banana"      " melon"      " watermelon"
```

```
str_trim(str5, side = "both") #remove whitespacing on both sides
```

```
## [1] "banana"     "melon"      "watermelon"
```

Pattern matching

Patterns

Working with text often entails describing and identifying text patterns.

A *pattern* is a regular expression (often called a regex) - i.e., a set of symbol that describe a text pattern.

Pattern matching can be complicated - we are going to explore some basics today and next week. Consider it might require a moment to become familiar with the language.

We will use str_view() and str_view_all() to visualize how different patterns are found within a set of words.

Simple pattern matching

The easiest match involves a determined set of one or more letters.

```
str <- c("pizza", "tacos", "hamburger", "hot dog")

#str_view(str, "a")

#str_view(str, "ham")
```

Note that this is case-sensitive

```
#str_view(str, "A")
```

And eager to please, meaning that it will select only the first match unless we specify _all

```
#str_view(str, "r")

#str_view(str, "0")

#str_view_all(str, "r")
```

Matching multiple words

As in traditional logical statements, we can use | (OR) to create more complex pattern matching.

```
str <- c("pizza", "tacos", "hamburger", "hot dog")

#str_view(str, "a|o") #Find a and o

#str_view_all(str, "a|o") #Find a and o

#str_view(str, "pizza|tacos") # Matches with pizza and tacos
```

You'll often find a similar syntax when you want to extract multiple values.

```
match_words <- c("pizza", "tacos")

#str_view(str, paste0(match_words, collapse = "|"))

paste0(match_words, collapse = "|") # see what paste0 does
```

```
## [1] "pizza|tacos"
```

Metacharacters

Metacharacters are characters that take on a special meaning and allow more complex matching patterns.

There are few challenges when creating patterns:

- Writing the right pattern
- Match what you want
- Match only what you want. You don't want to specify a pattern that is overly permissive.

Metacharacter

Metacharacter	Symbol	Meaning
the dot	.	Wildcard - any character
the backslash	\\	Escape symbol
the dollar sign	$	End of string
the caret or hat	^	Beginning of string OR exclusion
the star or asterisk	*	Repetition
the plus sign	+	Repetition
the question mark	?	Repetition
left or opening bracket	[	Groups of characters
right or closing bracket	]	Groups of character
left or opening brace	{	Precise count
right or closing brace	}	Precise count
the dash, hyphen or minus sign	-	To...from...

The wildcart

The . is a wildcard - it means that it matches any character.

Let's see a few examples.

```
#str_view(str, ".a.")
```

The above code matches any three characters, whose middle letter is an a.

We can also match any pairs of characters, whose last letter is a.

```
#str_view(str, ".a")
```

The wildcard can even be matched to a whitespace.

```
#str_view(str, "t.d")
```

Escape character

What if you want to match an actual .?

You need to use an escape character, which in R corresponds to \\.

An escape character removes the special meaning from a metacharacter, so that it gets back to its original meaning.

```
str1 <- c("pizza.", ".tacos", "hamburger.", "hot.dog")

# The wildcard can be any character
#str_view(str1, "a.")

# In this case, it looks for the exact match
#str_view(str1, "a\\.")

# Matching all periods.
#str_view(str1, "\\.")
```

Anchors

Anchors are used to set whether you are searching for a given pattern in a given position within the string.

By default, regular expressions are matched at any part of a string. So, if you want to match any letter at the beginning or end of a string, you need to use:

- ^ to anchor the search at the start of the string
- $ to anchor the search at the end of the string

If you begin with power (^), you end up with money ($)

Anchors

The start of a string:

```
#str_view(str, "t")

#str_view(str, "^t")
```

The end of a string:

```
#str_view(str, "a")

#str_view(str, "a$")
```

Anchors

To have an exact match, you need to use both symbols at the beginning and the end.

See the examples below:

```
str2 <- c("pizza with cheese", "pizza", "pizza with pepperoni", "pizzeri

#str_view(str2, "pizza")

#str_view(str2, "^pizza$")

#str_view(str2, "pizz")

#str_view(str2, "^pizz$")
```

What would you do if you wanted to match the actual dollar symbol?

```
str3 <- c("$3", "$4", "Euro5", "8")
```

Escaping anchors

```
str3 <- c("$3", "$4", "Euro5", "8")

#str_view(str3, "\\$")

#See what happens if the character is not escaped
# str_view(str3, "$")
```

Note that some metacharacters such as `$ . | ? * + ( ) [ {` can also be escaped by using `[]`.

```
str3 <- c("$3", "$4", "Euro5", "8")

#str_view(str3, "[$]")
```

Groups of characters

There are some special pattern symbols that help you to match more than one character but with more precision than . which matches anything.

- [abc] matches a, b, or c, or any other specified groups of characters
- [^abc] matches anything except a, b, or c, or any other specified groups of characters
- [a-z] matches all letters between a and z
- [0-9] matches all numbers between 0 and 9
- \d matches any digit.
- \s matches any whitespace (e.g. space, tab, newline).

Some examples

```
pns <- c('pan', 'pen', 'pin', 'pon', 'pun', 'paun')

#str_view(pns, "p[aeiou]n")
```

Note that it matches only each vowel individually not in pair. In fact, the last word has no matching.

Other examples:

```
triplets <- c('123', 'abc', '1er4', 'ABC', ':-)')

# Matches any one number in a string
#str_view(triplets, "[0-9]")

# Matches any three numbers in a row in a string
#str_view(triplets, "[0-9][0-9][0-9]")
```

Escaping characters

Something important to remember: since you are matching a string, you are still putting everything in between " ".

Also, again, remember that if you insert a metacharacter like . into the range, it is escaped (e.g., it becomes a regular character) with the exceptions of the closing bracket], the dash -, the caret ^, and the backslash.

If you want those later to be literal characters, you need to escape them with \

Groups vs exact match

Look at the different results that you obtain if you use or not use the [].

```
str = c("banana", "apple", "peach")

# Match the syllabus na
#str_view(str, "na")

# Match any letters n or a.
# So in the case of banana it matches to a because it comes first!
#str_view(str, "[na]")
```

Excluding characters

Sometimes, it is helpful to indicate the characters to exclude instead of the characters to include. You can use the symbol `^` to do so.

Wasn't `^` used to indicate the beginning of a string?

Yes, it can be used for both purposes. Pay attention to the position of the symbol `^` (caret) in the following examples.

```
str = c("banana", "apple", "peach")

# Include letters a or b
#str_view(str, '[ab]')

# Exclude letters a or b
#str_view(str, '[^ab]')

# Include letters a or b only at the beginning of a string
#str_view(str, '^[ab]')

# Include initial letters except if they are a or b
#str_view(str, '^[^ab]')
```

Quick exercise

Write a matching pattern to accomplish the following:

- Identify words starting with a vowel
- Identify words of more than 4 letters (e.g., words with at least 5 letters)
- Identify a letter followed by a number

```
exercise_str = c("banana12", "apple324", "peach1", "bananaaa", "anne", "
```

Quick exercise - Solutions

```
str_view(exercise_str, "^[aeiou]")
```

banana12

apple324

peach1

bananaaa

anne

dog

Repetition

Repetition

Repetition symbols are helpful when you want to match the same character more than once.

- * indicates that the preceding item will be matched 0 or more times
- + indicates that the preceding item will be matched 1 or more times
- ? indicates that the preceding item is optional and will be matched at most 1 time

Let's see this with a quick example with the same string we used above:

```
exercise_str = c("banana12", "apple324", "peach1", "bananaaa", "anne", "
#str_view_all(exercise_str, "na*") # zero or more a, so it matches both
#str_view_all(exercise_str, "na+") # 1 or more a
#str_view_all(exercise_str, "na?") # zero or maximum one "a"
```

Repetition

You can also specify the number of matches precisely:

- {n}: exactly n
- {n,}: n or more
- {,m}: at most m
- {n,m}: between n and m

```
exercise_str = c("banana12", "apple324", "peach1", "bananaaa", "anne", "
#Identify words of more than 4 letters (e.g., words with at least 5 lett
#str_view_all(exercise_str, ".{4,}")
# Words with exactly 4 letters
#str_view_all(exercise_str, ".{4}")
#str_view_all(exercise_str, "na{1}") # at most one one "a"
#str_view_all(exercise_str, "[0-9]{2}") # at least two numbers
```

Tools

Tools

Now that you know how to identify a pattern... How would you use this skill for real-life applications?

There are several functions that you can use:

- to determine which strings match a pattern (str_detect)
- to count the number of matches within a string (str_count)
- to find position of a matching pattern (str_locate)
- to extract the content of a match (str_subset or str_extract)
- to even replace matching patterns with new values (str_replace)

Applying these functions with pattern matching is what allows us to work with strings. Let's see a few examples using the dataset "fruit" included in the stringr package.

str_detect()

str_detect() identifies whether a pattern exists within a set of strings. Return a logical vector that says TRUE is the pattern is identified and FALSE if it is not identified within the string.

Let's see, for instance, which common words start with "t".

```
str_detect(fruit, "^t")
```

Detect:

- Fruit names that start with t
- Fruit names that end with a vowel

Solutions

```
# How many fuirt names start with t?
sum(str_detect(fruit, "^t"))
```

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

```
# What proportion of fruit names end with a vowel?
mean(str_detect(fruit, "[aeiou]$"))
```

```
## [1] 0.35
```

str_subset

Subset a string vector according to a given pattern... Compared to str_detect, it doesn't return a logical vector but the actual subset of strings containing the patterns.

We can, for instance, subset all strings that start with a, e, or i.

```
str_subset(fruit, "^[aei]")
```

Or we can subset all strings that do not start with a, e, or i by using the argument negate = T

```
str_subset(fruit, "^[aei]", negate = T)
```

Subsetting

Note that, in general, we would use filter within a pipe to obtain this result

```
fruit2 = as_tibble(fruit)

fruit2 %>%
  filter(str_detect(value, "^[aei]"))
```

```
## # A tibble: 5 x 1
##   value
##   <chr>
## 1 apple
## 2 apricot
## 3 avocado
## 4 eggplant
## 5 elderberry
```

str_count

str_count counts how many matches there are in a string and returns a set of 1 if the string is a match and 0 is the string is not a match. This can be used to count how many matches occur on average.

```
# Count how many words begin with a, e, or i

# str_count(words, "^[aei]")

sum(str_count(fruit, "^[aei]"))

# Count how many vowels are contained within each word
# str_count(words, "[aeiou]")

# You can count the average number of occurences across all words
mean(str_count(fruit, "[aeiou]"))
```

str_extract

str_extract is used to extract the actual text of a match (and only the text of the matching pattern, not the entire string).

Note that extract_all extracts only the first match. If you want to extract all matches, you need to use str_extract_all.

Extract all matching with "ta".

```
str_extract(fruit, "ta")
```

str_replace

str_replace allows you to replace matches with new strings. If you want to replace multiple words, you can use str_replace_all.

Just as an example, let's replace all "a" with a *.

```
str_replace(fruit, "l", "*")
```

str_locate

str_locate is used to locate the start and ending position of each match

```
str_locate(fruit, "a")
```

Quick exercise

```
names <- c("University of Arizona", "Malcolm X College", "Harry S Truman
# Extract all strings representing a University institution
# Extract the location of a each university - i.e., all strings contain
```

Solutions

```
str_extract(names, ".*University.*")
str_subset(names, "University")

location <- c("Arizona", "Illinois", "Oregon", "Phoenix", "Ohio")
str_extract(names, paste0(location, collapse = "|"))
```