Lab 03: String Cheese SOLUTIONS

PSTAT 100: Spring 2024 (Instructor: Ethan P. Marzban)

MEMBER 1 (NetID 1) MEMBER 2 (NetID 2)
MEMBER 3 (NetID 3)

April 28, 2024

Required Packages

```
library(ottr)        # for checking test cases (i.e. autograding)
library(pander)      # for nicer-looking formatting of dataframe outputs
library(tidyverse)   # for graphs, data wrangling, etc.
```

Logistical Details

Logistical Details

- This lab is due by **11:59pm on Wednesday, May 1, 2024**.
- Collaboration is allowed, and encouraged!
  - If you work in groups, list ALL of your group members' names and NetIDs (not Perm Numbers) in the appropriate spaces in the YAML header above.
  - Please delete any "MEMBER X" lines in the YAML header that are not needed.
  - No more than 3 people in a group, please.
- Ensure your Lab properly renders to a `.pdf`; non-`.pdf` submissions will not be graded and will receive a score of 0.
- Ensure all test cases pass (test cases that have passed will display a message stating `"All tests passed!"`)

Lab Overview and Objectives

In this lab, we will discuss:

- Basic handling of strings and **regular expressions**

Regular Expressions

Regular expressions (often abbreviated as **regex**, pronounced either with a hard 'g' or a soft 'g') are essentially a sequence of symbols and characters designed to aid in the identification of particular patterns in strings. They are an integral part of the analysis of text, and thankfully can be (relatively) easily managed using tools from the `stringr` and `stringi` packages (both a part of the `tidyverse`). With that said, wrangling strings is sometimes considered to be the bane of data scientists' existences. We'll only be scratching the surface in this lab, but if you are planning on pursuing a career in data science I encourage you to read more on this topic.

First, let's explore the `str_view()` function. This function is used to extract out all elements of a character vector that match some pattern (where the pattern is specified using regular expressions). For example:

```
str_view(
  string = c("blackberry", "blueberry", "apple"),
  pattern = "berry"
)
```

```
[1] | black<berry>
[2] | blue<berry>
```

! Question 1

The vector called `fruit` (stored in the `stringr` package, which was loaded when we loaded the tidyverse) contains a character vector of 80 different fruits. Extract the fruits whose name contains the string "berry", and store this in a variable called `berries`.

Solution:

```
## replace this line with your code

berries <- str_view(fruit, "berry")
```

Answer Check:

```
# DO NOT EDIT THIS LINE
invisible({check("tests/q1.R")})
```

```
All tests passed!
```

Now, when dealing with regular expressions, there are two types of characters to be aware of: **literal characters** (which are letters/numbers that match exactly) and **metacharacters** (which are letters/numbers that have special meaning, and are not matched exactly *a priori*). For example:

```
str_view(
  c("one.world", "one.species", "together"),
  ".")
```

```
[1] | <o><n><e><.><w><o><r><l><d>
[2] | <o><n><e><.><s><p><e><c><i><e><s>
[3] | <t><o><g><e><t><h><e><r>
```

Clearly, something's wrong - `str_view()` is not returning only the words/characters with periods in them. This is because the period, `.`, is a metacharacter. Specifically, where regular expressions are concerned, `.` will match with *any* single character. For example:

```
str_view(
  c("cat", "dog", "orangutan"),
  "a."
)
```

```
[1] | c<at>
[3] | or<an>gut<an>
```

will return only the characters containing an `a` followed by another character (notice how `"dog"` was not returned, as it does not contain an `"a"`.)

! Question 2

Extract only the fruits in the `fruit` vector whose names contain a "w". Store this in a variable called `w_fruits`.

Solution:

```
## replace this line with your code

w_fruits <- str_view(fruit, "w.")
```

Answer Check:

```
# DO NOT EDIT THIS LINE
invisible({check("tests/q2.R")})
```

```
All tests passed!
```

We can also combine periods:

```
str_view(
  c("a", "aa", "aba", "abbba", "abc"),
  "a.a"
```

```
)
```

```
[3] | <aba>
```

! Question 3

Extract only the fruits in the `fruit` vector whose names contain a "a", followed by two characters, followed by an "i". Store this in a variable called `ai_fruits`.

Solution:

```
## replace this line with your code

ai_fruits <- str_view(fruit, "a..i")
```

Answer Check:

```
# DO NOT EDIT THIS LINE
invisible({check("tests/q3.R")})
```

```
All tests passed!
```

Quantifiers can be used to control the number of times a pattern can match:

- `?`: 0 times or 1 time
- `+`: 1 or more times
- `*`: 0 or more times

For example:

```
x <- c("a", "ab", "abb")

# matches an "a", optionally followed by a "b"
str_view(x, "ab?")
```

```
[1] | <a>
[2] | <ab>
[3] | <ab>b
```

```
# matches an "a", followed by at least one "b"
str_view(x, "ab+")
```

```
[2] | <ab>
[3] | <abb>
```

```r
# matches an "a", followed by any number of "b"s
str_view(x, "ab*")
```

```
[1] | <a>
[2] | <ab>
[3] | <abb>
```

We can also use curly braces `{` and `}` to specify the number of matches exactly:

- `{n}`: exactly n repetitions
- `{n,}`: n or more repetitions
- `{n, m}`: between n and m repetitions

```r
y <- c("ab", "abb", "abbb", "abbbb", "abbbbb")

# will match only with a double "b"
str_view(y, "b{2}")
```

```
[2] | a<bb>
[3] | a<bb>b
[4] | a<bb><bb>
[5] | a<bb><bb>b
```

```r
# will match with a sequence of two or more consecutive "b"s
str_view(y, "b{2,}")
```

```
[2] | a<bb>
[3] | a<bbb>
[4] | a<bbbb>
[5] | a<bbbbb>
```

```r
# will match with a sequence of between 2 and 4 consecutive "b"s
str_view(y, "b{2,4}")
```

```
[2] | a<bb>
[3] | a<bbb>
[4] | a<bbbb>
[5] | a<bbbb>b
```

! Question 4

Extract only the fruits in the `fruit` vector whose names contain a sequence of at least two consecutive "r"s. Store this in a variable called `two_r_fruits`.

Solution:

```
## replace this line with your code

two_r_fruits <- str_view(fruit, "r{2}")
```

Answer Check:

```
# DO NOT EDIT THIS LINE
invisible({check("tests/q4.R")})
```

```
All tests passed!
```

Character classes can be used to match a set of characters (i.e. identify strings that contain at least one of the elements in a set of characters). For example:

```
z <- c("abb", "def", "cde")

# match strings that contain either a "c" or an "f"
str_view(z, "[cf]")
```

```
[2] | de<f>
[3] | <c>de
```

! Question 5

Extract only the fruits in the `fruit` vector whose names contain a vowel, followed by a "b", followed by another vowel. Store this in a variable called `vowel_b_vowel_fruit`.

Solution:

```
## replace this line with your code

vowel_b_vowel_fruit <- str_view(fruit, "[aeiou]b[aeiou]")
```

Answer Check:

```
# DO NOT EDIT THIS LINE
invisible({check("tests/q5.R")})
```

```
All tests passed!
```

We can negate classes using the `^`. For example:

```
# match with "non-vowel" "y" "non-vowel"
str_view(fruit, "[^aeiou]y[^aeiou]")
```

```
[13] | cana<ry >melon
[47] | <lyc>hee
[61] | p<hys>alis
```

Perhaps somewhat confusingly, the caret (`^`) is also used as a part of what is known as an **anchor**, which is used to specify the start and end of a string. Specifically, `^` is used to specify the start of a string and `$` is used to specify the end. For example:

```
z <- c("aba", "ab", "ba")

# start with an "a"
str_view(z, "^a")
```

```
[1] | <a>ba
[2] | <a>b
```

```
# ends with an "a"
str_view(z, "a$")
```

```
[1] | ab<a>
[3] | b<a>
```

! Question 6

Extract only the fruits in the `fruit` vector whose names start with a vowel or end with a vowel (**hint:** look up what the `|` symbol is and how it can be used here). Store this in a variable called `vowel_vowel_fruit`.

Solution:

```
## replace this line with your code

vowel_vowel_fruit <- str_view(fruit, "^[aeiou]|[aeiou]$")
```

Answer Check:

```
# DO NOT EDIT THIS LINE
invisible({check("tests/q6.R")})
```

```
All tests passed!
```

Two additional important metacharacters are `\\b` (which matches word boundaries) and `\\B` (which matches boundaries that have either both word or non-word character on either side).

```
w <- c("Two words", "oneword")
str_view(w, "\\b")
```

```
[1] | <>Two<> <>words<>
[2] | <>oneword<>
```

```
str_view(w, "\\B")
```

```
[1] | T<>w<>o w<>o<>r<>d<>s
[2] | o<>n<>e<>w<>o<>r<>d
```

! Question 7

Extract only the fruits in the `fruit` vector whose names consist of two words. Store this in a variable called `two_word_fruits`.

Solution:

```
## replace this line with your code

two_word_fruits <- str_view(fruit, ".\\b.")
```

Answer Check:

```
# DO NOT EDIT THIS LINE
invisible({check("tests/q7.R")})
```

```
All tests passed!
```

A couple of miscellaneous notes:

- It is important to note that regular expressions are case-sensitive:

```
str_view(c("ab", "Ab"), "^a")
```

```
[1] | <a>b
```

- Sometimes, it will be necessary to match a metacharacter literally. The way we do this is using a double backslash \\:

```
str_view(c("hi^bye", "hi"), "\\^")
```

```
[1] | hi<^>bye
```

Submission Details

1) Check that all of your tables, plots, and code outputs are rendering correctly in your final `.pdf`.
2) Check that you passed all of the test cases (on questions that have autograders). You'll know that you passed all tests for a particular problem when you get the message "All tests passed!".
3) Submit **ONLY** your `.pdf` to Gradescope. Make sure to match pages to your questions - we'll be lenient on the first few labs, but after a while failure to match pages will result in point penalties.