Lab 09: Regular [Expression] Show – SOLUTIONS

PSTAT 100, Summer Session A 2025 with Ethan P. Marzban

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

July 24, 2025

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 Friday, July 25, 2025**.
- 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

Welcome to another PSTAT 100 Lab! In this lab, we will cover the following:

- Regular Expressions / Wrangling Text using `R`

Introduction

In this lab, we'll explore some of the ways `R` can be used to wrangle text data. We'll frequently be referring to the script of Episode 2, titled "Just Set Up the Chairs" from the show *Regular Show*. As a bit of background, *Regular Show* is a show that aired on Cartoon Network from 2010 until 2017, and maintains an impressive rating of 8.6 on IMdB (the *Internet Movie Database*). We've selected the script from a relatively early episode, which shouldn't contain any major spoilers for the show (in case you are inclined to watch it!). The script is stored in a file called `script.txt`, located in the `data/` subfolder.

This chunk below reads in the script, and assigns the result to a vector called `script`:

```
script <- readLines("data/script.txt")
```

Here are the first few entires of the `script` vector:

```
script %>% head()
```

```
[1] "Benson: Alright, listen up. We've got this birthday party today, so...lots to do...lots to
[2] ""
[3] "Muscle Man & Hi Five Ghost: Yes! Uh!"
[4] ""
[5] "(Muscle Man and Hi Five Ghost high-five each other)"
[6] ""
```

Using `stringr::str_subset()`

One of the packages included in the `tidyverse` is the **`stringr`** package. This package provides a plethora of functions useful when it comes to analyzing text data. The first function we'll explore is the `str_subset()` function.

From the relevant helpfile, "`str_subset()` returns all elements of string where there's at least one match to pattern." For example,

```
sample_text <- c("My", "cat", "and", "your", "cat", "could", "be", "friends")
sample_text %>% str_subset("cat")
```

```
[1] "cat" "cat"
```

! Question 1

Using the `str_subset()` function and the `length()` function, calculate mow many lines of dialogue Rigby (a character from *Regular Show*) has in the script. Assign your answer to a variable called `rigby_num_lines`.

Solution:

```
## replace this line with your code
rigby_num_lines <- script %>% str_subset("Rigby:") %>% length()
```

Answer Check:

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

```
All tests passed!
```

Another useful function is the `str_count()` function. Whereas `str_subset()` displays all subsets containing a match to the `pattern`, `str_count()` displays the number of matches in each string. For example:

```
## counts the number of 'r's in each string:
c("cat", "dog", "rabbit", "parrot") %>%
  str_count("r")
```

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

! Question 2

Recompute the number of lines Rigby has in the script, this time using `str_count()`. Assign your answer to a variable called `rigby_num_lines2`, and check that it agrees with your answer to Question 1 above.

Solution:

```
## replace this line with your code
rigby_num_lines2 <- script %>% str_count("Rigby:") %>% sum()
```

Answer Check:

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

```
All tests passed!
```

The logical "or" (`|`) works with strings as well! For example,

```
## counts the number of times either 'a' or 'r' (or both) appear
c("cat", "dog", "rabbit", "parrot") %>%
  str_count("a|r")
```

```
[1] 1 0 2 3
```

! Question 3

How many times does Mordecai (another character in the show) say the word "dude"? Keep in mind that `R` is case-sensitive. Assign your answer to a variable called `mordecai_dude_count`

Solution:

```
## replace this line with your code
mordecai_dude_count <- script %>%
  str_subset("Mordecai:") %>%      ## look only at Mordecai's dialogue
  str_count("dude|Dude") %>%       ## account for potential capitalization
  sum()                            ## aggregate
```

Answer Check:

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

```
All tests passed!
```

Regular Expressions

Most of the functions from the `stringr` package, like `str_subset()`, contain a "`pattern`" argument which is used for matching. For example, `str_subset(string, pattern)` extracts only subsets of `string` that contain `pattern`. Up until now, we've been using a full string as the `pattern` - we often call such a string a **literal**. Essentially, a literal character is one that is matched directly, with no special meaning - most characters are like this. There do, however, exist **metacharacters** which have special meaning in the context of text analysis.

These metacharacters exist within the universe of **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, though are 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.

Metacharacters

Wildcard

First, let's discuss some common metacharacters. The first metacharacter we'll cover is the period (`.`) - this is used to match *any single* character, except a new line. For this reason, we often call this the **wildcard** character. For example:

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

```
[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>
```

Note that this is **not** returning only words/phrases with a period - again, this is because the period is a metacharacter! If we wanted to **escape** the special meaning assocaited with the period (i.e. to match a literal period), we can use `\\.`

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

```
[1] | one<.>world
[2] | one<.>species
```

> **Tip**
>
> The `str_view()` function works similarly to `str_subset()`, but in addition to only returning strings that match the desired `pattern` it also highlighs the `pattern` using angled brackets (`<` , `>`).

Anchors

Two other common metacharacters are together referred to as **anchors**: `^` and `$`. `^` is used to indicate the start of a string, and `$` is used to indicate the end of a string. For example:

```
c("cat", "canteloupe", "bobcat", "silica", "toucan") %>%
  str_view("^ca")
```

```
[1] | <ca>t
[2] | <ca>nteloupe
```

```
c("cat", "canteloupe", "bobcat", "silica", "toucan") %>%
  str_view("ca$")
```

```
[4] | sili<ca>
```

We can combine the anchors and period to create more sophisticated patterns. For example, suppose we want to find words whose third-to-last and second-to-last characters are `ca`:

```
c("cat", "canteloupe", "bobcat", "silica", "toucan") %>%
  str_view("ca.$")
```

```
[1] | <cat>
[3] | bob<cat>
[5] | tou<can>
```

! Question 4

Part (a)
How many lines in the script contain stage directions? Assign your answer to a variable called `num_stage_dir`. **Hint:** note that stage directions are always enclosed by parentheses.

Solution:

```
## replace this line with your code
num_stage_dir <- script %>% str_subset("\\(") %>% length()
```

Answer Check:

```
# DO NOT EDIT THIS LINE
invisible({check("tests/q4a.R")})
```

```
All tests passed!
```

Part (b)
How many stage directions appear on their own line (as in, not embedded within a larger line of dialogue)? Assign your answer to a variable called `num_stage_dir_sep_line`.

Solution:

```
## replace this line with your code
num_stage_dir_sep_line <- script %>%
  str_subset("^\\(") %>% length()
```

Answer Check:

```
# DO NOT EDIT THIS LINE
invisible({check("tests/q4b.R")})
```

```
All tests passed!
```

Character Classes

Perhaps you noticed, when working on Question 3, parentheses and brackets are also metacharacters.

Square brackets (`[`, `]`) create **character classes**, which 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:

```
c("cat", "canteloupe", "bobcat", "silica", "toucan") %>%
  str_view("[aeiou]$")
```

```
[2] | canteloup<e>
[4] | silic<a>
```

> **Tip**
>
> As we start creating more and more sophisticated regular expressions, it is useful to be able to "read the code in English." For example, the above code chunk can be read as: "extract all strings that end in a vowel."

Now, as was discussed earlier in this lab, regular expressions (and, indeed, `R` in general) are case-sensitive. For example:

```
c("Canteloupe", "cat") %>% str_view("ca")
```

```
[2] | <ca>t
```

Therefore, when dealing with text, there are two options available to us to address this:

1) We could convert everything to lowercase, using a function like `stringr::str_to_lower()`. **Not always recommended**
2) We could use character classes
3) We could specify `ignore_case = TRUE` in most of our commonly-used functions

Though, in practice, we would often opt for the third option, let's investigate the second. For example, to return all words beginning with a `c` followed by an `a`, regardless of case, we might use

```
c("Canteloupe", "CAnteloupe", "cAnteloupe") %>%
  str_view("[cC][aA]")
```

```
[1] | <Ca>nteloupe
[2] | <CA>nteloupe
[3] | <cA>nteloupe
```

! Question 5

How many lines of dialogue in the script end with a vowel (before their final punctuation mark)? Assign your answer to a variable called `num_vowel_end`.

Solution:
The key is to note that a non-stage-direction line of dialogue will always end with either a period, exclaimation point, or question mark.

```
## replace this line with your code
num_vowel_end <- script %>%
  str_subset("[aeiou][\\.\\?\\!]$") %>% length()
```

Answer Check:

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

```
All tests passed!
```

Quantifiers

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>
```

```
# 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 (`{`, `}`) to specify the number of matches exactly:

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

```
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
```

```
# 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>
```

```
# 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 6

Which lines of dialogue in the script end with at least two consecutive vowels (before their final punctuation mark)? Assign your answer to a vector called `rep_vowel_end`.

Solution:

```
## replace this line with your code
rep_vowel_end <- script %>%
  str_subset("[aeiou]{2,}[\\.\\?\\!]$")
```

Answer Check:

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

```
All tests passed!
```

Somewhat confusingly, the caret (^) can also be used to negate a character class.

Tip

A caret outside of a character class is an anchor; a caret inside of a character class is a negation.

For example, to extract all words that do not begin with a vowel, we can use:

```
c("apple", "banana", "carrot", "durian") %>%
  str_view("^[^aeiou]")
```

```
[2] | <b>anana
[3] | <c>arrot
[4] | <d>urian
```

Question 7

How many lines of dialogue end with something other than a vowel before their final punctuation? Exclude stage directions; assign your answer to a variable called `num_end_non_vowel`.

Solution:

To exclude stage directions, we need to ensure the given string doesn't begin with a parenthesis.

```
## replace this line with your code
num_end_non_vowel <-  script %>%
  str_count("^[^\\(].{1,}[^aeiou][\\.\\?\\!]$")  %>% sum()
```

Reading this out loud: starts with a non-parenthesis, followed by at least one character, followed by a punctuation, followed by the end of the string.

Answer Check:

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

```
All tests passed!
```

Submission Details

Congrats on finishing this PSTAT 100 lab! Please carry out the following steps:

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 ALL pages to the ONE question on Gradescope**; failure to do so will incur a penalty of 0.1 points.

Note on 4(a):

```
one <- script %>% str_extract("\\(.*\\)")
two <- script %>% str_extract("\\(")
script[(is.na(one) & !is.na(two)) %>% which()]
```

```
[1] "(Benson turns around at his wits end. Meanwhile, Mordecai is attempting to set up the chair
```

One line has an unclosed parenthetical