SPEC Lab R Workshop Series: Session 5

Therese Anders

1 Plan for today

1. String operations with the `stringr` library
2. Regular expressions
3. Homework exercise

2 String operations

String operations play a large role in the data cleaning component of data management, as well as in data gathering tasks such as web scraping. All string operations work on objects of the class **character**. Base `R` has a number of string operations, but they are often not user-friendly. The `stringr` package provides a comprehensive library of string operations. As part of the "tidyverse," we can use `stringr` functions together with other funtions from the `dplyr` and `tidyr` packages within the same pipe.

2.1 Basic string operations

The simplest string operation is base `R`'s `paste()` command. It simply concatenates two strings, by default with a space. We can also specify a custom expression to concatenate the characters with the `sep =` parameter.

```
paste("My", "string")
```

```
## [1] "My string"
```

```
paste("My", "string", sep = "_")
```

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

```
paste("My", "string", sep = "pretty")
```

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

In data management, we most often use the `paste()` function when combining the values of two variables.

```
countries <- c("a", "b", "c")
years <- c(1990, 1990, 2000)
paste(countries, years, sep = "_")
```

```
## [1] "a_1990" "b_1990" "c_2000"
```

2.2 Pattern matching using `stringr`

Pattern matching is the work horse of string operations. Here, we focus on pattern matching functions that are most useful for the data management of data that is already organized in some kind of structure, such as vectors or data frames. There are many other string operation functions, such as `str_locate()` and `str_locate_all()` that are most useful for data cleaning and reorganization of unstructured data. Here, we will cover the following `stringr` functions:

- `str_sub()`: Extract substrings from a character vector by indices.

- `str_extract()`: Extract substrings from a character vector by matched pattern.
- `str_detect()`: Returns `TRUE` or `FALSE` if the pattern in matched in the input string.
- `str_replace()`: Replaces a matched pattern with a new pattern.

2.2.1 `str_sub()`

Since the `str_sub()` function operates with indices, is most useful when all elements of a vector (or some other data object of interest) have the same pattern. It extracts the information that matches the indices specified by the user. The general syntax is as follows.

```
str_sub(pattern, start = , end = )
```

```
library(stringr)
locations <- c("CA Los Angeles", "NV Las Vegas", "CA San Diego")
locations_state <- str_sub(locations, start = 1, end = 2)
locations_state
```

```
## [1] "CA" "NV" "CA"
```

We can also specify open intervals. For example, if we wanted to extract only the cities, we could extract all characters starting at the fourth position.

```
locations_city <- str_sub(locations, 4)
locations_city
```

```
## [1] "Los Angeles" "Las Vegas"   "San Diego"
```

2.2.2 `str_extract()`

Think of `str_extract()` as the smarter sibling of `str_sub()`. Rather than just extracting characters based on indices, it extracts characters based on matched patterns. Together with the regular expression operations we introduce below, this is very powerful in extracting information. Here is the syntax for `str_extract()`:

```
str_extract(string, pattern)
```

```
locations_state2 <- str_extract(locations, "CA")
locations_state2
```

```
## [1] "CA" NA   "CA"
```

2.2.3 `str_detect()`

Think of `str_detect()` as a logical operator. It returns `TRUE` if the pattern is matched and `FALSE` otherwise. The function has the following syntax:

```
str_detect(string, pattern)
```

```
str_detect(locations, "L")
```

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

```
str_detect(locations, "Los")
```

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

2.2.4 str_replace()

`str_replace()` is very helpful in data management tasks, in particular when re-coding variables. As an example, suppose we wanted to replace the state abbreviations with the full state names. The syntax of the function is as follows:

```
str_replace(string, pattern, replacement)
```

```
locations_fullname <- str_replace(locations, "CA", "California") %>%
  str_replace("NV", "Nevada")
locations_fullname
```

```
## [1] "California Los Angeles" "Nevada Las Vegas"
## [3] "California San Diego"
```

2.3 Regular Expressions

String operations are useful, but their true potential is realized when used in combination with regular expressions. Regular expressions help us to define search patterns. This obliviates the need to type out entire character strings in pattern matching. Regular expressions are like a language that is understood by more than just one program. With small changes, you would be able to use the regular expressions discussed here in other programming languages, such as Python.

Suppose, our strings weren't as well organized as before. How would you extract the state from the elements of the following character vector?

```
locs <- c("CA Los Angeles", "Las NV Vegas", "San Diego CA")
```

Even though the elements of the character vector aren't as ordered any more, there is still a pattern! State abbreviations always have two letters and they are always capitalized. Below, we will learn a number of regular expression that will help us state this pattern ("two letters, capitalized") in a language that `R` understands.

Note that unless otherwise specified, regular expressions are case sensitive. The following patterns are examples for specifying the substantive content of the pattern.

- `^`: String **starts with** following pattern..
- `$`: String **ends with** preceding pattern.
- `[ ]`: or.
- `[a-z]`: Any letter.
- `[0-9]`: Any digit.

There is also a number of regular expressions that are used to specify the quantity of matches.

- `?`: Preceding pattern is optional (matched 0 or 1 times).
- `*`: Preceding pattern is matched 0 or more times.
- `+`: Preceding pattern is matched at least once.
- `{n}`: Preceding pattern is matched exactly`n` times.
- `{n, m}`: Preceding pattern is matched at least `n` times and up to `m` times.
- `{n,}`: Preceding pattern is matched at least `n` times.

So in our example above, we could use the following regular expression to extract the state names:

```
str_extract(locs, "[A-Z]{2}")
```

```
## [1] "CA" "NV" "CA"
```

2.3.1 Examples for regular expressions

```
fruits <- c("apple", "banana", "pineapple", "cherries4")
```

Exercise 1 What is the output of the following command?

```
str_detect(fruits, "a")
```

Exercise 2 What is the output of the following command?

```
str_detect(fruits, "^a")
```

Exercise 3 How would you ask `R` to print out `TRUE` for all elements of the `fruits` vector that end with the letter `e`?

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

Exercise 4 What is the output of the following command?

```
str_detect(fruits, "[ie]")
```

Exercise 5 How would you extract the number from the last element of the `fruits` vector?

```
## [1] "4"
```

We can use regular expressions to create more complex queries. Suppose, we wanted to know which strings start with an `a` and end with an `e`.

```
str_detect(fruits, "^a[a-z]*e$")
```

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

Exercise 6 What do you think is the output of the following command?

```
str_detect(fruits, "a[a-z]*e$")
```

2.3.2 Escape character

Regular expressions can also be used to match special characters. Note, however, that since some special characters have a special meaning in `R`, we sometimes have to use a so-called **escape character**, i.e. a backslash, to specify these patterns within regular expressions. For example, since the open square bracket `[` is part of the regular expression for `OR`, in order to match a `[` in the text, you need to type `\\[` to get the desired output.

```
test <- "a []"
#str_detect(test, "[") # This throws and error.
str_detect(test, "\\[") # This correctly recognized the pattern.
```

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

2.3.3 Application: Phone Numbers

Suppose, we have data on contact information and wanted to extract only the values that match phone numbers. Unfortunately, phone numbers come in different formats, but there are a number of common patterns.

What are common patterns for phone numbers you guys can think of?

```
phone <- c("123 456 7890",
           "(123) 456 7890",
           "123-456-7890")
```

We cannot use simple indices any longer to extract the phone numbers from this vector. The lengths of the elements differ!

```
nchar(phone[1])
```

```
## [1] 12
```

```
nchar(phone[2])
```

```
## [1] 14
```

Lets write a regular expression that outputs `TRUE` for all of these phone numbers.

```
str_detect(phone, "[(]?[0-9]{3}[)]?[ -]{1}[0-9]{3}[ -]{1}[0-9]{3}")
```

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

Exercise 7 How would you alter this if I gave you the following number format and asked you to match it as well? `123.456.7890`

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

2.3.4 Application: Creating a data frame

Why do we need this? Suppose I gave you a long string of data and asked you to clean the data and give be a data set of phone numbers and ZIP codes. You could make your life **a lot** easier if you know regular expressions.

Load the file `data.RData` and implicitly print out at its content, a vector called `vec`.

```
load("data.RData")
vec
```

```
## [1] "4: 12345"           "6: 123 456 7890"    "77: 90089"
## [4] "77: (234) 567 1234" "6: 90090"           "3: 90090"
## [7] "1: 90090"           "1: 789.345.6712"    "1000: 45123"
```

```
# First, lets get the phone numbers (since we already have a parser)
phone <- str_extract(vec, "[()]?[0-9]{3}[)]?[ -.]{1}[0-9]{3}[ -.]{1}[0-9]{3}")
phone
```

```
## [1] NA              "123 456 789"   NA              "(234) 567 123"
## [5] NA              NA              NA              "789.345.671"
## [9] NA
```

```
# Second, get ZIP codes
zip <- str_extract(vec, "[0-9]{5}")
zip
```

```
## [1] "12345" NA      "90089" NA      "90090" "90090" "90090" NA      "45123"
```

```
# Lastly, lets get the IDs
id <- str_extract(vec, "[0-9]+:")
id
```

```
## [1] "4:"    "6:"    "77:"   "77:"   "6:"    "3:"    "1:"    "1:"    "1000:"
```

```r
id <- str_replace(id, ":", "")
id
```

```
## [1] "4"    "6"    "77"   "77"   "6"    "3"    "1"    "1"    "1000"
```

```r
# Putting it all into one data frame
df <- data.frame(id, phone, zip)
print(df)
```

```
##     id         phone   zip
## 1    4          <NA> 12345
## 2    6   123 456 789  <NA>
## 3   77          <NA> 90089
## 4   77 (234) 567 123  <NA>
## 5    6          <NA> 90090
## 6    3          <NA> 90090
## 7    1          <NA> 90090
## 8    1   789.345.671  <NA>
## 9 1000          <NA> 45123
```

```r
# Now cleaning it
# Assumptions:
# a) no one has two zip codes or phone numbers,
# b) No id is missing.
library(dplyr)
library(tidyr)
df_cleaned <- df %>%
  gather(indicator, value, phone:zip) %>%
  na.omit() %>%
  spread(indicator, value)
df_cleaned
```

```
##     id         phone   zip
## 1    1   789.345.671 90090
## 2 1000          <NA> 45123
## 3    3          <NA> 90090
## 4    4          <NA> 12345
## 5    6   123 456 789 90090
## 6   77 (234) 567 123 90089
```

3 Homework

In this homework, you will be working with an sample data that I scraped from Wikipedia, using the `rvest` package. In this workshop, we do not cover webscraping, but if you are interested in how to do it, you can find the code I used below. Note that since Wikipedia constantly changes, it is possible that the `xpath` specified below might not be accurate at a later point in time.

```
library(rvest) #Webscraping package
library(stringr) #String methods, aka regular expressions etc.
url <- "https://en.wikipedia.org/wiki/List_of_U.S._states_by_population_density"
tbl <- url %>%
  read_html() %>% #This is from the rvest package
  html_nodes(xpath = '//*[@id="mw-content-text"]/div/table[1]') %>% #xpath from Wikipedia
  html_table() #Put it all into a table.
tbl <- tbl[[1]] #rvest saves it as a table inside list, we want only the table.
write.csv(tbl, "hw5_data.csv", row.names = F)
```

Tasks

1. Load the data called `hw5_data.csv`.
2. Use string operations to remove all `.` from the variable names. Pass the new names to the data frame (i.e. rename the variables).
3. Variables should not start with a letter. This is why `R` automatically attached a leading `X` to the variable `2015.population`. Use string operations to remove both the `X` and the `2015` from this variable name. Then, use the `paste()` function to attach `_2015` to the variable name.
4. Convert all variable names to lower case. Use the `tolower()` function.
5. Use the `str()` function to investigate the structure of the data set. Convert any factor variables to characters before using string operations on them! You can do this manually or google how to convert multiple columns at once.
6. Remove any of the "—" in the population density rank variable and replace it with a missing value.
7. Try converting the variables `population_2015`, `landareami2`, and `landareakm2` to numeric. Why do you think this fails? How can you fix this? Fix it and convert all variables, except the `state` variable to numeric!

Hints:

- Read up on the difference between `str_replace()` and `str_replace_all()` (for 2.).
- Note that `.` is a special character. It is a wild card. This means that it can be used as a place holder for anything, a number, letter, special character, or white space. Use escape characters to remove the `.` in 2.

Bonus If you get tired of manually converting variables from factor to character or from character to numeric, google how to make your life easier! There are multiple ways to do this, but none of them is straighforward. Don't worry if you cannot figure it out. We will go over a solution that uses loops in the next session.

You should get the following output when running `str(data)`:

```
str(data)
```

```
## 'data.frame':    56 obs. of  10 variables:
##  $ state              : chr  "District of Columbia" "New Jersey" "Puerto Rico" "Rhode Island" ...
##  $ popdensrank        : num  1 2 3 4 5 6 7 8 9 10 ...
##  $ popdensrank50states: num  NA 1 NA 2 3 NA NA 4 NA 5 ...
##  $ densitypopmi2      : num  11011 1218 1046 1021 871 ...
##  $ densitypopkm2      : num  4251 470 404 394 336 ...
##  $ poprank            : num  50 11 29 44 15 53 54 30 55 19 ...
##  $ population_2015    : num  672228 8958013 3680058 1056298 6794422 ...
##  $ landrank           : num  56 46 49 51 45 52 54 48 55 42 ...
```

```
##  $ landareami2        : num  61 7354 3515 1034 7800 ...
##  $ landareakm2        : num  158 19047 9104 2678 20202 ...
```