MATH550/SCC461: Programming for Data Science
Lab 5
String processing using R

Lecturer: Tom Palmer
Notes by: Debbie Costain and Stuart Sharples

Contents

Before you start

- Install the `stringr` package if it is not already available on your machine.
- Load your favourite packages at the start of a new script:

```
library(ggplot2)
library(dplyr)
library(stringr)
```

Introducing regular expressions

Regular expressions are special phrases that we construct to help find and match patterns in a large body of text or sets of strings. They exist in almost all programming languages and are a useful tool when interacting with computers in all sorts of ways:

- Finding files on a system with a particular pattern in their filename.
- Parsing a log file which contains thousands of lines, hunting down the reason for a server repeatedly crashing.
- Generating features and statistics from bodies of text e.g. how many times does the word "potato" appear in each of the works of Shakespeare. Another example; pulling out hashtags from tweets.
- Parse a web-page, storing all references to other websites.
- Automating the finding and replacing text.

You will be starting with an expression as simple as this:

```
\d
```

which is *one* way of saying "any character that matches a **d**igit from 0 to 9". And moving to something a bit more complicated, such as:

```
(\(\d{5}\)|\d{5})[ -.]?\d{5,6}
```

which matches a 10 or 11 digit, UK telephone number, with or without parentheses around the area code, and with or without a hyphen, dot (period) or space to separate the numbers.

This lab will give you an idea of the simpler ways to match patterns in text using:

- Literal text
- Digits
- Letters
- Characters of any kind

We will start by using the website RegExr to help you build and understand regular expressions:

`http://www.regexr.com/`

Open `fake_phone_numbers.txt`, a file in the data sets folder on moodle. Copy and paste its contents into the "Text" area on RegExr.

Matching literal text

The most obvious feature of regular expressions is matching strings with one or more literal characters, called *literal text* or just *literals*. The way to match literal text is with normal, literal characters. This is similar to the way you might try to find a phrase in a word document or when submitting a keyword to a search engine. When you search for a string of text, character for character, you are searching with literal text.

If you want to match the Lancaster area code, 01524, for example, which is a number sequence (string of digits), just type 01524 in the "Expression" box at the top of RegExr, and then all the matching area codes will be highlighted in the Text. If nothing is highlight, check what you typed.

Matching digits

In the Expression box, replace the literal text, 01524, with:

```
\d
```

This matches all the Arabic digits in the Text. Now in place of `\d` use a character class that matches the same thing. Enter the following range of digits as the Expression:

```
[0-9]
```

Though the syntax is different, using `\d` does the same thing as `[0-9]`. When we use square-brackets like this we are specifying a *character class*, you will learn more about character classes later. The character class `[0-9]` is a range, meaning that it will match the range of digits 0 through 9. You could also match digits 0 through 9 by listing all the digits:

```
[0123456789]
```

If you want to match only the binary digits 0 and 1, you would use this character class:

```
[01]
```

Try `[23]` in RegExr and look at the result. With a character class, you can pick the exact digits you want to match. The character shorthand for "all digits" (`\d`) is shorter and simpler, but it does not have the power or flexibility of the character class. Use character classes when you need to get very specific about what digits you need to match; otherwise, use `\d` because it is a simpler, more convenient syntax.

Matching non-digits

if you want to match characters that are not digits, use the shorthand:

```
\D
```

Try this in RegExr now. An uppercase D, rather than a lowercase, matches non-digit characters. This shorthand is the same as the following character class

```
[^0-9]
```

This is a negated class (a negated class says in essence, "do not match these" or "match all but these"), which is the same as:

```
[^\d]
```

Matching words and non-words

The fake phone numbers data obviously do not contain any word text. Go to `www.authorama.com`, which contains books that are in the public domain. Examples include, Alice in Wonderland, Flatland, and Frankenstein. Pick one and copy the first page of text into the Text area on RegExr, replacing the fake phone numbers. Now, in the Expression box, swap `\D` with:

```
\w
```

This shorthand will match all word characters. The difference between `\D` and `\w` is that `\D` matches whitespace, punctuation, quotation marks, hyphens, forward slashes, square brackets, and other similar characters, while `\w` does not, it only matches letters and numbers. In English, `\w` matches essentially the same thing as the character class:

```
[a-zA-Z0-9]
```

To match a non-word character:

```
\W
```

This shorthand matches whitespace, punctuation, and other kinds of characters that are not used in words. It is equivalent to the following character class:

```
[^a-zA-Z0-9]
```

Character classes allow you more control over what you match, but a lot of the time you do not need to type out all those characters. But sometimes you must explicitly state a character class in order to get precisely what you want. Try both:

```
[^\w]
[^\W]
```

Can you see the differences in what text they match? Table 1 shows a summary of shorthand character classes.

Table 1: Shorthand character classes

Shorthand	Description
`\d`	digit
`\D`	non-digit
`\w`	word
`\W`	non-word
`\b`	word boundary
`\B`	non-word boundary
`\s`	space character
`\S`	non-space character
`\t`	tab character
`\T`	non-tab character

Matching a literal word

Going back to matching literal text, replace the expression with ’`at`’, all instances of “at” in your text should now be highlighted. This includes instances where “at” is the actual word, and instances where “at” has occurred within a word, for example, “c**at** s**at** on the m**at**”. To match only instances of the actual word “at” change the expression so it becomes:

```
\bat\b
```

Here `\b` stands for *word-boundary*, and it matches the start or the end of a word. The first `\b` requires the ’`a`’ to occur at the very start of the word, or after a non-word character. The second `\b` requires the ’`t`’ to occur at the very end of the word, or before a non-word character.

Matching whitespace

To match whitespace, try the following in RegExr and see what is highlighted:

```
\s
```

The following character class matches the same characters:

```
[ \t\n\r]
```

which are the characters for spaces, tabs (`\t`), new lines (`\n`), and carriage returns (`\r`).

To match a non-whitespace use:

```
\S
```

which is equivalent to:

```
[^ \t\n\r]
```

Matching any character

To match any character with regular expressions use the dot, also known as a period or a full stop. The dot matches all characters but line ending characters. In RegExr, replace your current expression with:

```
.
```

An equivalent character class to dot would be combining any class and its negated class:

```
[\w\W]
```

It can be tempting (and very lazy) to use dot as our character class as we can often be more specific about the text phrases we are looking to match.

Using quantifiers

So far we have been matching individual characters or literal text, hence why when we use expressions such as `\w` all word characters are individually highlighted. To highlight all words as a whole:

```
\w+
```

or

```
[a-zA-Z0-9]+
```

The plus symbol here is used to state how many times we match the character class. '+' specifically means "1 or more" characters from the preceding character class.[1] Other quantifiers are shown in Table 2. For example to highlight all three-letter words use the expression:

Table 2: Quantifiers used after a character class.

`*`	0 or more
`+`	1 or more
`?`	0 or 1
`{3}`	exactly 3
`{3,}`	3 or more
`{3,5}`	3, 4, or 5

[1] This is why regular expressions are considered modern-day hieroglyphics for computer geeks.

```
\w{3}
```

How would you modify this to match four-letter words? Six- to seven-letter words? Also look at the differences in results between the two expressions:

```
\w{3}
.{3}
```

Matching alternate patterns

We can write regular expressions that state that there is a choice of patterns to match. For example, say you wanted to find how many occurrences of the word, "*the*", there are in the text you have selected for analysis. The problem is, the word can occur as *THE*, *The*, and *the*. You can use alternation to deal with this by writing the following expression:

```
\b(the|The|THE)\b
```

and you will see all occurrences of *the* in the text highlighted.

Being able to specify alternate patterns like this allows us to create sub-patterns in our expressions. Consider what words the following expression will match and why:

```
[tT]{1}h(eir|ere)
```

Going deeper

If you are interested in learning more about regular expressions, seek out at least one of the following books:

- Introducing Regular Expressions (2012) by M. Fitzgerald (O'Reilly Media).
- Mastering Regular Expressions, Third Edition (2006) by J. Friedl (O'Reilly Media).
- Regular Expressions Cookbook, Second Edition (2012) by J. Goyvaerts and S. Levithan (O'Reilly Media).

These cover a lot more than these notes. Each can serve as a good reference book, but obviously the "Introducing Regular Expressions" serves as the best introductory text.

Working with strings and regular expressions

Base R provides a solid set of string manipulation functions, but because they have grown organically over time, they can be inconsistent and difficult to learn. Additionally, some string processing tasks that are easy to do in languages like Ruby or Python are rather hard to do in R. The aim of the `stringr` package is overcome these problems by providing a clean, modern interface to common string operations. Some but not all allow the use of regular expressions.

Basic string operations

The following functions allow us to manipulate strings on a manual level rather than using any pattern matching:

- `str_c()` is equivalent to `paste()` which you may have already used, it concatenate vectors but it uses the empty string (`""`) as the default separator rather than a single whitespace (`" "`) and silently removes zero length arguments. Consider each of the following `str_c()` statements and each differs from the previous:

  ```
  dat <- data.frame(
    animal = c("human", "blue whale", "cat", "dog"),
    food   = c("pizza", "plankton", "human", "anything"),
    when   = c("this week", "now and then", "now", "always")
  )

  str_c(dat$animal, " wants to eat ", dat$food)
  str_c(dat$animal, dat$food, sep=" wants to eat ")
  str_c(dat$animal, dat$food, dat$when, sep=" wants to eat ")
  str_c(dat$animal, " wants to eat ", dat$food, " ", dat$when)
  ```

How strings are concatenated and collapsed can be modified by additional arguments to `str_c()`, check the help page to see how. In general, `str_c()` behaves in a more useful and consistent manner than `paste()`.

- `str_length()` calculates the length of a string (as the number of characters in the string). If passed a vector it will return a vector of lengths.

  ```
  str_length(dat$animal)
  ```

Consumer Packaged Breakfast Goods

The following sections look at how we perform various string processing tasks within R using regular expressions. To give context to when these types of tasks are useful we will be working on a data set which contains nutritional information and ingredient lists for 6,600 breakfast products. Download '`cpg_breakfast.csv`' from moodle and open it in Excel to understand the structure, then read it into R (prevent auto-conversion of strings to factors). A variable description is presented in Table 3. Only convert `manufacturer` and `brand` to factors. We will now be mainly working with the ingredient lists.

Table 3: Description of relevant variables for '`cpg_breakfast_foods.csv`'. There are other variables that are self explanatory.

`manufacturer`	The manufacturer or parent company of the product.
`brand`	The brand name commonly used by consumers.
`product_name`	The name of the product. May include variant information such as colour or flavour.
`ingredients`	List of ingredients as they appear on the products packaging (comma separated).
`calories`	Total calories per serving.
`total_carb`	Total amount of carbohydrates (grams).
`total_fat`	Total amount of fat (grams).
`size`	Size information as shown on the packaging.
`avg_price`	Average price in dollars.
`ean13`	The EAN-13 barcode used for identifying products worldwide.

Detect if a match can be found within a string

The first brick wall we run in to when using regular expressions in R is that instead of simply using one backslash for character classes (e.g. '`\w`') we have to use two (e.g. '`\\w`').

Let us try to determine how many breakfast products in our data contain oats. First we create the expression:

```
expr <- "\\b[oO]ats\\b"
```

which looks similar to the expressions we were creating earlier; we are looking to match either "oats" or "Oats" as whole words only. To then detect if either of these appear in the ingredients we do:

```
bf$has_oats <- str_detect(bf$ingredients, expr)
```

This has added the variable '`oats`' to our data frame, and it will contain a series of `TRUE`s and `FALSE`s depending on whether or not our expression was matched in the `ingredients` strings. To summarise:

```
table(bf$has_oats)
```

In terms of creating features from text-variables `str_detect()` is probably the most useful. Practice by looking for other common breakfast ingredients such as: corn and rice. Create a expression that will detect any instance of sugar, honey, or syrup.

Extract the matched text

There are many different types of syrup (e.g. agave, maple, corn). Write an expression that retrieves "syrup" and the preceding word:

```
expr <- "\\w+ [sS]yrup"
bf$syrup_type <- str_extract(bf$ingredients, expr)
table(bf$syrup_type)
```

We can see that corn syrup is a fairly common ingredient. How many of these matches are actually for "high fructose corn syrup"? In general, we can extend our expression to try to capture all proceeding words (remember the ingredients are comma separated):

```
expr <- ",[ a-zA-Z]*[sS]yrup,"
bf$syrup_type <- str_extract(bf$ingredients, expr)
table(bf$syrup_type)
```

But now our `syrup_type` variable contains commas and whitespace. We can tidy this up by writing an expression that keeps all characters between the first and the last comma. To do this we use parentheses around the part of the expression that we want to keep:

```
expr <- ", (.*),"
```

Because the text we are now working on (the text in `bf$syrup_type`) is small we can risk being a bit lazy with our expression hence why I use dot-star. When we use parentheses like this it is known as grouping, with the first set of parentheses being referenced as group one. To only keep the text that matches inside the brackets:

```
expr2 <- "\\1"
bf$syrup_type <- str_replace(bf$syrup_type, expr, expr2)
table(bf$syrup_type)
```

It is possible to have multiple sets of parentheses in our expression, these would then be accessed using `"\\2"`, `"\\3"`, etc. Also, here our second expression is very simple, we are free to make this more complicated by including literal text:

```
expr  <- "(.*)"
expr2 <- "I have found \\1!"
bf$syrup_type <- str_replace(bf$syrup_type, expr, expr2)
table(bf$syrup_type)
```

`expr2` works in a similar way to the concatenating we did earlier with `str_c()`. Try this process of extracting again, but now look for instances of vitamins being added to the product. Commonly added vitamins are A, B12, C, D, and E, ensure your expression can capture these. `str_extract()` will only retrieve the first match from a string. To retrieve all vitamins that are added to a product use `str_extract_all()`. Though you will not be able to store the results of that are returned from `str_extract_all()` simply in a variable in your data frame. Store the results in an object separate from your data frame and inspect it.

Replace matches

Say, we know someone who is an angry vegan, this someone also happens to be fairly good with regular expressions, let's call this person Rose Myris. If Rose Myris wanted to replace all instances of meat-based ingredients with the phrase "dead animal" then she could do this by using `str_replace()`. Firstly, let's find some breakfast foods that do actually contain meat:

```
expr <- "(Duck|Beef|Pork|Bacon|Chicken|Poultry|Turkey)"
bf$has_meat <- str_detect(bf$ingredients, expr)
bf_meat <- filter(bf, has_meat == TRUE)
```

Inspect `bf_meat` to see the kinds of products we are dealing with, in particular look at row 161 (which also includes corn syrup). Using `str_replace()`, we can reuse the same expression in order to swap out each type of meat for a different phrase:

```
bf_meat$ingredients <- str_replace(bf_meat$ingredients,
                                   expr, "dead animal")
```

Now look at what changes have been made to the ingredients in `bf_meat`. In each string only the first match has been replaced. Obviously, Rose Myris, would prefer to have all matches replaced, the is where `str_replace_all()` comes in to action:

```
bf_meat$ingredients <- str_replace_all(bf_meat$ingredients,
                                       expr, "dead animal")
```

At last, Rose Myris' anger can now reach new heights! To really get her blood boiling let's count how many replaces were made:

```
dead_hits <- str_match_all(bf_meat$ingredients, "dead animal")

dead_count <- c()
for (i in 1:length(dead_hits)) {
    dead_count[i] <- length(dead_hits[[i]])
}
sum(dead_count)
bf_meat[which.max(dead_count), ]
```

The last command allows us to look at the worst offender.

Building large expressions

If we wanted to find products that contained fruit, we simply have to create one long alternating pattern of literal fruit names (e.g. `(apple|orange|pear|raisin)`, as there is no smart compact expression we can write down that will match only fruit names.

Go find a list of fruit on the web and copy this list into an empty text file in your working directory, format the contents such that it looks like a list of fruit, with the first few rows looking something like this:

```
Apple
Apricot
Avocado
Banana
```

Now read this file into R:

```
fruit <- read.table(file="list_of_fruit.txt", header=FALSE,
                    stringsAsFactors=FALSE)
```

We are not interested in keeping this data in a data frame, so we can go ahead and overwrite the `fruit` object with the `V1` variable:

```
fruit <- fruit$V1
```

Now `fruit` is just a vector of fruit names. Check the length of `fruit` to see how many we actually have. Next, we can start to build our expression by first collapsing all the different fruit names together, and then adding parenthesis either side:

```
fruit_expr <- str_c(fruit, collapse="|")
fruit_expr <- str_c("(", fruit_expr, ")")
str_length(fruit_expr)
# [1] 769
```

Our fruit expression has 769 characters which means it's very long. Let's put this into action:

```
bf$has_fruit <- str_detect(bf$ingredients, fruit_expr)
table(bf$has_fruit)
#
# FALSE  TRUE
#  2278  2074
```

We can use this technique of concatenating a list to build several features, such as whether or not a product:

- Contains gluten.
- Is suitable for those who are lactose-intolerant.
- Is suitable for vegetarians or vegans.

For each of these we would need to create specific lists which we would then collapse to form a single, but very long, expression. In some cases, it may be easier to try to match ingredients that they can not or will not eat rather than match those that they can.

Exercises

1. Create separate boxplots comparing the distribution of total carbohydrates in fruit-based and non-fruit-based breakfast food.
2. Repeat but for total fat instead of total carbs.
3. The are a wide variety of products that class as breakfast food:

    - Cereal-bars.
    - Cereal (e.g. corn flakes, porridge, granola, puffed rice).
    - Waffles, pancakes.
    - Delicious baked goods (e.g. croissants, pan au chocolat).
    - Meat products (e.g. sausage, bacon).
    - Other.

    Construct regular expressions which attempt to classify products into these groups. You will have to decide which variable in the data set is the most appropriate to be working with. You may need to break this task down into several smaller sub-tasks.

    Once complete, construct similar boxplots to the first two tasks, comparing distributions of total fat and carbs for these groups.