Regular Expressions (Regex)

William Hancock, Zachary D'Urso

2024-11-5

Regular Expressions

This is a method of specifying patterns to search for in text. It is used in R, Python, SQL and many other places. It works pretty much the same in all of the places it appears.

Learning to write regex search patterns takes a while, it's kind of a puzzle solving problem.

Tinkering with it a bit, it looks like ChatGPT does pretty well at figuring out regex patterns. But they don't always seem to work, so one needs to be careful. They did show some interesting ideas or methods in the ChatGPT output though.

See chapter 15 of Wickham et al

```
library(tidyverse)
```

```
## ── Attaching core tidyverse packages ──────────────────────── tidyverse 2.0.0 ──
## ✓ dplyr     1.1.4     ✓ readr     2.1.5
## ✓ forcats   1.0.0     ✓ stringr   1.5.1
## ✓ ggplot2   3.5.1     ✓ tibble    3.2.1
## ✓ lubridate 1.9.3     ✓ tidyr     1.3.1
## ✓ purrr     1.0.2
## ── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
## ✖ dplyr::filter() masks stats::filter()
## ✖ dplyr::lag()    masks stats::lag()
## ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to becom
e errors
```

```
library(babynames)
library(stringr)
```

str_view

Str_view seems to be a tool largely for figuring out how a regex expression is working

It often takes a bit of tinkering to figure out hwo to write a good search target using regex, sort of testing as you go. str_view() seems to help with that

Prints out the section of each string found with the target in it

fruit is a list of names of 80 fruits

```
str_view(fruit,"berry")
```

```
##  [6] │ bil<berry>
##  [7] │ black<berry>
## [10] │ blue<berry>
## [11] │ boysen<berry>
## [19] │ cloud<berry>
## [21] │ cran<berry>
## [29] │ elder<berry>
## [32] │ goji <berry>
## [33] │ goose<berry>
## [38] │ huckle<berry>
## [50] │ mul<berry>
## [70] │ rasp<berry>
## [73] │ salal <berry>
## [76] │ straw<berry>
```

Question/Action

try this with apple and star

```
str_view(fruit,"apple")
```

```
##  [1] │ <apple>
## [62] │ pine<apple>
```

```
str_view(fruit,"star")
```

```
## [75] │ <star> fruit
```

combinations and wild cards

a. means a followed by any number of characters or white space

the period is a “wildcard” for any character

```
str_view(c("a","ab","a:b","ed"," a b","eab"),"a.")
```

```
## [2] │ <ab>
## [3] │ <a:>b
## [5] │  <a >b
## [6] │ e<ab>
```

All fruits with an “a”, then any 3 letters, then an “e”

```
str_view(fruit,"a...e")
```

```
##  [1] │ <apple>
##  [7] │ bl<ackbe>rry
## [48] │ mand<arine>
## [51] │ nect<arine>
## [62] │ pine<apple>
## [64] │ pomegr<anate>
## [70] │ r<aspbe>rry
## [73] │ sal<al be>rry
```

to find a period “.” we use an escapte sequence “\.”

```
str_view("Go, no Stop.Don't Stop","\\.")
```

```
## [1] │ Go, no Stop<.>Don't Stop
```

Quantifiers-

How many times does the pattern have to appear

? - makes a pattern option, so “ab?” is an an “a” followed by or or 1 b, ie “a”, “ab”

- means “one or more” so “ab+” matches “ab”, “abb”, “abbb” etc

{}- indicates a range “ab{2,4}” means a plus two to four “b”s, abb, abbb,abbbb

[[:digit:]]- means a digit, so [[:digit:]]{1,3} means 1 tp 3 digits in a row

```
str_view(fruit,"ba?")
```

```
##  [4] │ <ba>nana
##  [5] │ <b>ell pepper
##  [6] │ <b>il<b>erry
##  [7] │ <b>lack<b>erry
##  [8] │ <b>lackcurrant
##  [9] │ <b>lood orange
## [10] │ <b>lue<b>erry
## [11] │ <b>oysen<b>erry
## [12] │ <b>readfruit
## [19] │ cloud<b>erry
## [21] │ cran<b>erry
## [22] │ cucum<b>er
## [29] │ elder<b>erry
## [32] │ goji <b>erry
## [33] │ goose<b>erry
## [38] │ huckle<b>erry
## [40] │ jam<b>ul
## [41] │ juju<b>e
## [50] │ mul<b>erry
## [69] │ ram<b>utan
## ... and 3 more
```

Character classes

We can define a set of characters to use in a match [abcde]

```
str_view(fruit,"[abcde]{2,4}")
```

```
##  [3] │ avo<cad>o
##  [4] │ <ba>nana
##  [5] │ <be>ll pepper
##  [6] │ bil<be>rry
##  [7] │ bl<ac>k<be>rry
##  [8] │ bl<ac>kcurrant
## [10] │ blu<ebe>rry
## [11] │ boysen<be>rry
## [12] │ br<ead>fruit
## [13] │ <ca>nary melon
## [14] │ <ca>ntaloupe
## [19] │ clou<dbe>rry
## [21] │ cran<be>rry
## [22] │ cucum<be>r
## [24] │ <da>mson
## [25] │ <da>te
## [29] │ el<de>r<be>rry
## [32] │ goji <be>rry
## [33] │ goos<ebe>rry
## [37] │ honey<de>w
## ... and 16 more
```

OR operation

“(a|b)”- a or b

“(aa|ee|ii|oo|uu)”- doubled consonant

```
str_view(fruit,"(aa|ee|ii|oo|uu)")
```

```
##  [9] │ bl<oo>d orange
## [33] │ g<oo>seberry
## [47] │ lych<ee>
## [66] │ purple mangost<ee>n
```

Major functions

Find a target in a large number of strings.

The function to do this is

str_detect

Say fruits that start with “a”

I got the search target from ChatGPT

```
    "\b[aA]\W*"
```

which means a blank followed by an upper or lower case a, then any word type item of any length

The original version did not work, I had to tweak it a bit. Expect to have to tweak chatGPT outpus

```
str_detect(fruit,"\\b[aA]\\W*")
```

```
##  [1]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [25] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [37] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [49] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [61] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [73] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

we can print this out

```
fruit[str_detect(fruit,"\\b[aA]\\W*")]
```

```
## [1] "apple"   "apricot" "avocado"
```

I probably would have used

“^(a|A)” - which is “a string that starts with a or A”

```
fruit[str_detect(fruit,"^(a|A)")]
```

```
## [1] "apple"   "apricot" "avocado"
```

Replace values

str_replace() and str_replace_all

```
x <- c("apple", "pear", "banana")
str_replace_all(x, "[aeiou]", "*")
```

```
## [1] "*ppl*"  "p**r"   "b*n*n*"
```

Parsing a column in a data frame

One problem you will see a lot is breaking up a column that has multiple items or values in a single column of text

Here is an example

```
df <- tribble(
  ~str,
  "<Sheryl>-F_34",
  "<Kisha>-F_45",
  "<Brandon>-N_33",
  "<Sharon>-F_38",
  "<Penny>-F_58",
  "<Justin>-M_41",
  "<Patricia>-F_84",
)
```

```
df
```

```
## # A tibble: 7 × 1
##   str
##   <chr>
## 1 <Sheryl>-F_34
## 2 <Kisha>-F_45
## 3 <Brandon>-N_33
## 4 <Sharon>-F_38
## 5 <Penny>-F_58
## 6 <Justin>-M_41
## 7 <Patricia>-F_84
```

We want the name, gender and age from this single column of text

There are delimiters, which we want to remove

< - first delimite [[A-Za-z]]+ - text, no spaces, any number of text >- -second delimiter, two pieces (N|n|F|F|M|m) - gender, we could use . here as well _ -delimiter between our gender and age [0-9]{1,3} -age, 1 to 3 digits

Since this is a common task, there is a tidyverse tool that will do this in one step

sepearate_wider_regex(string, patterns)

patterns is a vector c(), and all named patterns are stored in the extracted data

patterns with no names are dropped

```
separate_wider_regex(df,
        str,
        patterns=c(
                    "<",
                    name = "[[:alpha:]]+",
                    ">-",
                    gender = ".",
                     "_",
                    age = "[[:digit:]]+"
          )
)
```

```
## # A tibble: 7 × 3
##   name     gender age
##   <chr>    <chr>  <chr>
## 1 Sheryl   F      34
## 2 Kisha    F      45
## 3 Brandon  N      33
## 4 Sharon   F      38
## 5 Penny    F      58
## 6 Justin   M      41
## 7 Patricia F      84
```

separate_wider_delim

There is also a delimiter based splitter

it allows only one possible delimiter between values, so it is not as flexible as separate_wider_regex

```
df2=data.frame(str=c(
  "Smith,Bob",
  "Jones, Sar",
  "Kim, Amanda")
  )


separate_wider_delim(df2,
                     str,
                     delim=",",
                     names=c("last","first")
                     )
```

```
## # A tibble: 3 × 2
##   last  first
##   <chr> <chr>
## 1 Smith "Bob"
## 2 Jones " Sar"
## 3 Kim   " Amanda"
```