Regular Expression

Admin Stuff

- assignment 2 is due today
- introduce `styler`

```
library(tidyverse)
```

Regular Expression

Introduction

A regular expression, `regex` or `regexp` is a sequence of characters that define a search pattern.

There exists any versions of regular expressions. `stringr` of tidyverse follows the ICU standard while base R follows the PCRE standard. See Comparison of regular-expression engines

We again prefer the tidyverse packages over base functions.

Basic concepts

We will use `https://regex101.com/` to illustrate the following. (Choose the Python engine to best mimic ICU standard)

- Boolean "or" - `gray|grey` can match `gray` or `grey`
- Grouping - Parentheses are used to define the scope and precedence of the operators - `gr(a|e)y` can match `gray` and `grey`.
- A quantifier after a token, character or group specifies how often that a preceding element is allowed to occur.

quantifier	
?	zero or one occurrences
	zero or more occurrences
+	one or more occurrences
{n}	exactly n occurrences
{n,}	`n` or more times
{m, n}	`m` or more times but not more than `n`

  Examples:

  - `colou?r` matches both "color" and "colour".
  - `ab*c` matches "ac", "abc", "abbc", "abbbc", and so on
  - `ab+c` matches "abc", "abbc", "abbbc", and so on, but not "ac"

- Wildcard - The wildcard `.` matches any character except a new line.

  Examples:

  - `a.c` matches “aac”, “abc” and so on.

- Anchors - `^` matches the beginning of a string and `$` matches the end of a string

  Examples:

  - `^abc` matches “abc” but not “cabc”
  - `abc$` matches “abc” but not “abcd”

- Bracket expression `[...]` matches a single character that is contained within the brackets

  Examples:

  - `[abc]` matches “a”, “b”, or “c”
  - `[abc123]` matches “a”, “b”, “c”, “1”, “2” or “3”
  - `[a-z]` specifies a range which matches any lowercase letter from “a” to “z”.
  - `[a-zA-Z0-9]` matches all alphanumerics
  - `[\[\]]` matches `[` or `]`
  - `[\]` matches \ ??

- Bracket expression `[^...]` matches a single character that is not contained within the brackets

  Examples:

  - `[^abc]` matches any character other than “a”, “b”, or “c”
  - `[^\]]` matches any `]` character which is not `]`.

- Special characters

  Honable mentions | Pattern | matches| |—-|—| |`\.`| `.`| |`\!`| `!`| |`\?`| `?`| |`\\`| \| |`\(`| `(`| |`\{`| `{`| |`\}`| `}`| |`\[`| `[`| |`\]`| `]`| |`\n`| a new line| |`\s`| a space or a tab| |`\S`| not a space nor a tab| |`\d`| a digit| |`\D`| a non digit| |`\w`| a word character (includes for example CJK chars)| |`\W`| a non word character| |`\b` | word boundaries|

Escaping characters

In R, these two characters need to be specially treated in double quotes.

- `"\\"` means a single backslash
- `"\""` means a double quote

Package `stringr`

Manage Strings

```
fruit <- c("apple", "banana", "pear", "pinapple")
str_length(fruit)
```

```
## [1] 5 6 4 8
```

```
# add leading white spaces
str_pad(fruit, 10)
```

```
## [1] "     apple" "    banana" "      pear" "  pinapple"
```

```r
# remove white spaces
str_trim(str_pad(fruit, 10))
```

```
## [1] "apple"    "banana"   "pear"     "pinapple"
```

```r
# ...
str_trunc(fruit, 5)
```

```
## [1] "apple" "ba..." "pear"  "pi..."
```

Detect Matches

```r
fruit <- c("apple", "banana", "pear", "pinapple")
# contains a?
str_detect(fruit, "a")
```

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

```r
# starts with a
str_detect(fruit, "^a")
```

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

```r
str_starts(fruit, "a")
```

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

```r
# ends with a
str_detect(fruit, "a$")
```

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

```r
str_ends(fruit, "a")
```

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

```r
# contains a, e, i, o or u
str_detect(fruit, "[aeiou]")
```

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

```r
# negate the result
str_detect(fruit, "^p", negate = TRUE)
```

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

```r
fruit <- c("apple", "banana", "pear", "pinapple")
# count the number of matches
str_count(fruit, "p")
```

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

```r
str_count(fruit, "p{1,}")
```

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

```r
# get locations
str_locate(fruit, "a")
```

```
##      start end
## [1,]     1   1
## [2,]     2   2
## [3,]     3   3
## [4,]     4   4
```

```r
str_locate_all(fruit, "a")
```

```
## [[1]]
##      start end
## [1,]     1   1
##
## [[2]]
##      start end
## [1,]     2   2
## [2,]     4   4
## [3,]     6   6
##
## [[3]]
##      start end
## [1,]     3   3
##
## [[4]]
##      start end
## [1,]     4   4
```

```r
# The pattern variable can also be vectorized
str_locate_all(fruit, c("a", "b", "p", "p"))
```

```
## [[1]]
##      start end
## [1,]     1   1
##
## [[2]]
##      start end
## [1,]     1   1
##
```

```
## [[3]]
##      start end
## [1,]     1   1
##
## [[4]]
##      start end
## [1,]     1   1
## [2,]     5   5
## [3,]     6   6
```

Subset Strings

```
fruit <- c("apple", "banana", "pear", "pinapple")
# exact substring from start to end
str_sub(fruit, 1, 3)
```

```
## [1] "app" "ban" "pea" "pin"
```

```
str_sub(fruit, -3, -2)
```

```
## [1] "pl" "an" "ea" "pl"
```

```
# only select the elements that match
str_subset(fruit, "a")
```

```
## [1] "apple"    "banana"   "pear"     "pinapple"
```

```
# indexs that have matches
str_which(fruit, "^a")
```

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

```
shopping_list <- c("apples x4", "bag of flour", "bag of sugar", "milk x2")
# numbers
str_extract(shopping_list, "\\d")
```

```
## [1] "4" NA  NA  "2"
```

```
# lower case chars
str_extract(shopping_list, "[a-z]+")
```

```
## [1] "apples" "bag"    "bag"    "milk"
```

```
# lower case chars of length 1 to 4
str_extract(shopping_list, "[a-z]{1,4}")
```

```
## [1] "appl" "bag"  "bag"  "milk"
```

```r
# lower case chars of length 1 to 4 with word boundary
str_extract(shopping_list, "\\b[a-z]{1,4}\\b")
```

```
## [1] NA     "bag"  "bag"  "milk"
```

```r
str_extract_all(shopping_list, "[a-z]+")
```

```
## [[1]]
## [1] "apples" "x"
##
## [[2]]
## [1] "bag"   "of"    "flour"
##
## [[3]]
## [1] "bag"   "of"    "sugar"
##
## [[4]]
## [1] "milk" "x"
```

```r
str_extract_all(shopping_list, "[a-z]+", simplify = TRUE)
```

```
##      [,1]     [,2] [,3]
## [1,] "apples" "x"  ""
## [2,] "bag"    "of" "flour"
## [3,] "bag"    "of" "sugar"
## [4,] "milk"   "x"  ""
```

```r
strings <- c(
  " 219 733 8965",
  "329-293-8753 ",
  "banana",
  "239 923 8115 and 842 566 4692",
  "Work: 579-499-7527",
  "$1000",
  "Home: 543.355.3679"
)

phone <- "([2-9][0-9]{2})[- .]([0-9]{3})[- .]([0-9]{4})"

# only the matched pattern
str_extract(strings, phone)
```

```
## [1] "219 733 8965" "329-293-8753" NA             "239 923 8115" "579-499-7527"
## [6] NA             "543.355.3679"
```

```r
str_extract_all(strings, phone)
```

```
## [[1]]
## [1] "219 733 8965"
##
```

```
## [[2]]
## [1] "329-293-8753"
##
## [[3]]
## character(0)
##
## [[4]]
## [1] "239 923 8115" "842 566 4692"
##
## [[5]]
## [1] "579-499-7527"
##
## [[6]]
## character(0)
##
## [[7]]
## [1] "543.355.3679"
```

```
# with subgroups
str_match(strings, phone)
```

```
##      [,1]           [,2]  [,3]  [,4]
## [1,] "219 733 8965" "219" "733" "8965"
## [2,] "329-293-8753" "329" "293" "8753"
## [3,] NA             NA    NA    NA
## [4,] "239 923 8115" "239" "923" "8115"
## [5,] "579-499-7527" "579" "499" "7527"
## [6,] NA             NA    NA    NA
## [7,] "543.355.3679" "543" "355" "3679"
```

```
str_match_all(strings, phone)
```

```
## [[1]]
##      [,1]           [,2]  [,3]  [,4]
## [1,] "219 733 8965" "219" "733" "8965"
##
## [[2]]
##      [,1]           [,2]  [,3]  [,4]
## [1,] "329-293-8753" "329" "293" "8753"
##
## [[3]]
##      [,1] [,2] [,3] [,4]
##
## [[4]]
##      [,1]           [,2]  [,3]  [,4]
## [1,] "239 923 8115" "239" "923" "8115"
## [2,] "842 566 4692" "842" "566" "4692"
##
## [[5]]
##      [,1]           [,2]  [,3]  [,4]
## [1,] "579-499-7527" "579" "499" "7527"
##
## [[6]]
```

```
##      [,1] [,2] [,3] [,4]
##
## [[7]]
##      [,1]           [,2]  [,3]  [,4]
## [1,] "543.355.3679" "543" "355" "3679"
```

Mutate Strings

```
fruit <- c("apple", "banana", "pear", "pinapple")
str_sub(fruit, 1, 5) <- "APPLE"
fruit
```

```
## [1] "APPLE"    "APPLEa"   "APPLE"    "APPLEple"
```

```
fruits <- c("one apple", "two pears", "three bananas")
# change the first a, e, i, o and u to -
str_replace(fruits, "[aeiou]", "-")
```

```
## [1] "-ne apple"     "tw- pears"     "thr-e bananas"
```

```
# change all a, e, i, o and u to -
str_replace_all(fruits, "[aeiou]", "-")
```

```
## [1] "-n- -ppl-"     "tw- p--rs"     "thr-- b-n-n-s"
```

```
# apply a function to the matches
str_replace_all(fruits, "[aeiou]", toupper)
```

```
## [1] "OnE ApplE"     "twO pEArs"     "thrEE bAnAnAs"
```

```
# remove all a, e, i, o and u
str_replace_all(fruits, "[aeiou]", "")
```

```
## [1] "n ppl"    "tw prs"   "thr bnns"
```

```
str_remove_all(fruits, "[aeiou]")
```

```
## [1] "n ppl"    "tw prs"   "thr bnns"
```

References of the form \1, \2, etc will be replaced with the contents of the respective matched group

```
fruits <- c("one apple", "two pears", "three bananas")
str_match_all(fruits, "([aeiou])")
```

```
## [[1]]
##      [,1] [,2]
## [1,] "o"  "o"
## [2,] "e"  "e"
## [3,] "a"  "a"
## [4,] "e"  "e"
##
## [[2]]
##      [,1] [,2]
## [1,] "o"  "o"
## [2,] "e"  "e"
## [3,] "a"  "a"
##
## [[3]]
##      [,1] [,2]
## [1,] "e"  "e"
## [2,] "e"  "e"
## [3,] "a"  "a"
## [4,] "a"  "a"
## [5,] "a"  "a"
```

```
str_replace_all(fruits, "([aeiou])", "[\\1]")
```

```
## [1] "[o]n[e] [a]ppl[e]"       "tw[o] p[e][a]rs"
## [3] "thr[e][e] b[a]n[a]n[a]s"
```

```
strings <- c(
  "Work: 219 733 8965",
  "Mobile: 579-499-7527",
  "Home: 543.355.3679"
)

phone <- "([2-9][0-9]{2})[- .]([0-9]{3})[- .]([0-9]{4})"

str_match_all(strings, phone)
```

```
## [[1]]
##      [,1]           [,2]  [,3]  [,4]
## [1,] "219 733 8965" "219" "733" "8965"
##
## [[2]]
##      [,1]           [,2]  [,3]  [,4]
## [1,] "579-499-7527" "579" "499" "7527"
##
## [[3]]
##      [,1]           [,2]  [,3]  [,4]
## [1,] "543.355.3679" "543" "355" "3679"
```

```
str_replace_all(strings, phone, "(\\1)-\\2-\\3")
```

```
## [1] "Work: (219)-733-8965"   "Mobile: (579)-499-7527" "Home: (543)-355-3679"
```

```r
# apply replacement mutliple times
str_replace_all("foobar", c("foo" = "hello", "bar" = "world"))
```

```
## [1] "helloworld"
```

Changes cases

```r
str_to_lower(c("one Apple", "tWo BANANAs", "THREE orangeS"))
```

```
## [1] "one apple"     "two bananas"   "three oranges"
```

```r
str_to_upper(c("one Apple", "tWo BANANAs", "THREE orangeS"))
```

```
## [1] "ONE APPLE"     "TWO BANANAS"   "THREE ORANGES"
```

```r
str_to_title(c("one Apple", "tWo BANANAs", "THREE orangeS"))
```

```
## [1] "One Apple"     "Two Bananas"   "Three Oranges"
```

Join and split

```r
str_c("apple", "pie")
```

```
## [1] "applepie"
```

```r
str_c(letters, LETTERS)
```

```
##  [1] "aA" "bB" "cC" "dD" "eE" "fF" "gG" "hH" "iI" "jJ" "kK" "lL" "mM" "nN" "oO"
## [16] "pP" "qQ" "rR" "sS" "tT" "uU" "vV" "wW" "xX" "yY" "zZ"
```

```r
str_c(letters, collapse = "")
```

```
## [1] "abcdefghijklmnopqrstuvwxyz"
```

```r
str_c(letters, LETTERS, collapse = "")
```

```
## [1] "aAbBcCdDeEfFgGhHiIjJkKlLmMnNoOpPqQrRsStTuUvVwWxXyYzZ"
```

```r
str_flatten(letters) # faster than str_c(letters, collapse = "") marginally
```

```
## [1] "abcdefghijklmnopqrstuvwxyz"
```

```r
fruits <- c(
  "apples and oranges and pears and bananas",
  "pineapples and mangos and guavas"
)

str_split(fruits, " and ")
```

```
## [[1]]
## [1] "apples"  "oranges" "pears"   "bananas"
##
## [[2]]
## [1] "pineapples" "mangos"     "guavas"
```

```r
str_split(fruits, " and ", simplify = TRUE)
```

```
##      [,1]         [,2]      [,3]     [,4]
## [1,] "apples"     "oranges" "pears"  "bananas"
## [2,] "pineapples" "mangos"  "guavas" ""
```

```r
str_split(fruits, " and ", n = 2)
```

```
## [[1]]
## [1] "apples"                        "oranges and pears and bananas"
##
## [[2]]
## [1] "pineapples"        "mangos and guavas"
```

```r
str_split(fruits, " and ", n = 3, simplify = TRUE)
```

```
##      [,1]         [,2]      [,3]
## [1,] "apples"     "oranges" "pears and bananas"
## [2,] "pineapples" "mangos"  "guavas"
```

```r
# a shorthand for str_split(..., n, simplify = TRUE)
str_split_fixed(fruits, " and ", n = 3)
```

```
##      [,1]         [,2]      [,3]
## [1,] "apples"     "oranges" "pears and bananas"
## [2,] "pineapples" "mangos"  "guavas"
```

Glue String

```r
name <- c("John", "Peter")
age <- c(23, 17)
# get variables from globals
str_glue("{name} is {age}")
```

```
## John is 23
## Peter is 17
```

```r
# get variables from arguments
str_glue("{name} is {age}", name = "Anna", age = 43)
```

```
## Anna is 43
```

```r
mtcars %>%
  group_by(cyl) %>%
  summarize(m = mean(mpg)) %>%
  str_glue_data("A {cyl}-cylinder car has an average mpg of {round(m, 2)}.")
```

```
## A 4-cylinder car has an average mpg of 26.66.
## A 6-cylinder car has an average mpg of 19.74.
## A 8-cylinder car has an average mpg of 15.1.
```

Order Strings

```r
names <- c("John", "Albert", "Peter", "Charles")
str_order(names)
```

```
## [1] 2 4 1 3
```

```r
str_sort(names)
```

```
## [1] "Albert"  "Charles" "John"    "Peter"
```

```r
str_sort(names, decreasing = TRUE)
```

```
## [1] "Peter"   "John"    "Charles" "Albert"
```

```r
files <- c("file10", "file2", "file5", "file1")
str_sort(files)
```

```
## [1] "file1"  "file10" "file2"  "file5"
```

```r
# more natrual order
str_sort(files, numeric = TRUE)
```

```
## [1] "file1"  "file2"  "file5"  "file10"
```

```r
str_order(files, numeric = TRUE)
```

```
## [1] 4 2 3 1
```

Pattern interpretation

Patterns in stringr functions are interpreted as regex in default, you could use `fixed` or `regex` to change the default behavior.

```
strings <- c("abb", "a.b")

str_detect(strings, "a.b")
```

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

```
str_detect(strings, fixed("a.b"))
```

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

```
str_detect(strings, fixed("A.B", ignore_case = TRUE))
```

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

```
str_match_all("abaa\na", "^a")
```

```
## [[1]]
##      [,1]
## [1,] "a"
```

```
str_match_all("abaa\na", regex("^a", multiline = TRUE))
```

```
## [[1]]
##      [,1]
## [1,] "a"
## [2,] "a"
```

```
str_match_all("abaa\na", regex("^A", ignore_case = TRUE, multiline = TRUE))
```

```
## [[1]]
##      [,1]
## [1,] "a"
## [2,] "a"
```

An exercise

```
# we need some string
calculus_url <- "https://en.wikipedia.org/wiki/Calculus"
calculus <- read_lines(calculus_url) %>%
  str_c(collapse = "\n")
```

```
calculus %>%
  str_extract_all("<p>(.|\\n)*?</p>") %>%  # . doesn't match new lines
  unlist() %>%
  str_remove_all("</?\\w+[^>]*>") %>%
  str_extract_all("[a-zA-Z]+") %>%
  unlist() %>%
```

```
  str_to_lower() %>%
  tibble(word = .) %>%
  count(word) %>%
  arrange(desc(n))
```

```
## # A tibble: 1,140 x 2
##    word         n
##    <chr>    <int>
##  1 the        424
##  2 of         290
##  3 a          162
##  4 and        156
##  5 is         133
##  6 to         126
##  7 in         121
##  8 calculus    87
##  9 function    64
## 10 as          52
## # ... with 1,130 more rows
```

More advanced topics of regex

I assume that you are now comfortable with the basic regex. Let's talk about some more advanced topics.

- non-capturing groups

  a capturing group could be created by using a pair of parentheses `(<regex>)` and a non-captureing group can be created by using `(?:<regex>)`.

```
# with captureing groups
str_match("12mb", "([0-9]+)([a-z]+)")
```

```
##      [,1]   [,2] [,3]
## [1,] "12mb" "12" "mb"
```

```
# with non-capturing groups
str_match("12mb", "(?:[0-9]+)(?:[a-z]+)")
```

```
##      [,1]
## [1,] "12mb"
```

- atomic groups

  an atomic group prevents the regex engine from backtracking back into the group

```
strings <- c("abc", "abcc")
str_extract(strings, "a(?>bc|b)c")
```

```
## [1] NA     "abcc"
```

- lazy quantifier

quantifier	
??	zero or one occurrence, lazy
?	zero or more occurrences, lazy
+?	one or more occurrences, lazy
{n,}	`n` or more times, lazy
{m, n}	`m` or more times but not more than `n`, lazy

```
strings <- c("abc", "acb")
# it is greedy
str_extract(strings, "ab?c?")
```

```
## [1] "abc" "ac"
```

```
# it is lazy
str_extract(strings, "ab??c?")
```

```
## [1] "a"  "ac"
```

```
strings <- "acbcbbc"
# it is greedy, longest match
str_extract(strings, "a.*c")
```

```
## [1] "acbcbbc"
```

```
# it is lazy, shortest match
str_extract(strings, "a.*?c")
```

```
## [1] "ac"
```

```
strings <- "acbcbbc"
# it is greedy, longest match
str_extract(strings, "a.+c")
```

```
## [1] "acbcbbc"
```

```
# it is lazy, shortest match
str_extract(strings, "a.+?c")
```

```
## [1] "acbc"
```

```
strings <- "acbcbbc"
str_extract(strings, "a.{2,}c")
```

```
## [1] "acbcbbc"
```

```r
str_extract(strings, "a.{2,}?c")
```

```
## [1] "acbc"
```

- [skip] Possessive quantifier (does not backtrack)

quantifier	
?+	zero or one occurrence, posessive
+	zero or more occurrences, posessive
++	one or more occurrences, posessive

```r
strings <- c("apple", "pineapple", "pinepineapple")
str_extract(strings, "(pine)?pineapple")
```

```
## [1] NA              "pineapple"     "pinepineapple"
```

```r
str_extract(strings, "(pine)?+pineapple")
```

```
## [1] NA              NA              "pinepineapple"
```

```r
str_extract(strings, "(pine)*pineapple")
```

```
## [1] NA              "pineapple"     "pinepineapple"
```

```r
str_extract(strings, "(pine)*+pineapple")
```

```
## [1] NA NA NA
```

- back reference

  back reference is used to reference a previous matched group

```r
str_match(":abc:", "(:)[a-z]+\\1")
```

```
##      [,1]    [,2]
## [1,] ":abc:" ":"
```

- look ahead
  - positive look ahead

```r
# it matches only the second `t` because the second `t` is followed by `s`
str_locate_all("streets", "t(?=s)")
```

```
## [[1]]
##      start end
## [1,]     6   6
```

- negative look ahead

```r
# it matches only the first `t` because the second `t` is followed by `s`
str_locate_all("streets", "t(?!s)")
```

```
## [[1]]
##      start end
## [1,]     2   2
```

- look behind
  - positive look behind

```r
# it matches only the first `t` because the second `t` follows `s`
str_locate_all("streets", "(?<=s)t")
```

```
## [[1]]
##      start end
## [1,]     2   2
```

- negative look behind

```r
# it matches only the second `t` because the first `t` follows `s`
str_locate_all("streets", "(?<!s)t")
```

```
## [[1]]
##      start end
## [1,]     6   6
```

```r
# it also works if the look behind pattern is a (bounded) regex
# (base R functions do not support regex in look behind pattern)
str_locate_all("twisty streets", "(?<![se]{1,100})t")
```

```
## [[1]]
##      start end
## [1,]     1   1
```

- [skip] recursion

We try to find the highest level of paranthesse in `((((x))))`, `((x)((y))(z)) (x) (y)`

```r
# what doesn't work
strings <- c("((((x))))", "((x) ((y)) (z))", "(x) (y)")
str_extract_all(strings, "\\([^(]*\\)")
```

```
## [[1]]
## [1] "(x))))"
##
## [[2]]
## [1] "(x)"  "(y))" "(z))"
##
## [[3]]
## [1] "(x)" "(y)"
```

```
str_extract_all(strings, "\\([^()]*?\\)")
```

```
## [[1]]
## [1] "(x)"
##
## [[2]]
## [1] "(x)" "(y)" "(z)"
##
## [[3]]
## [1] "(x)" "(y)"
```

```
# how about back reference the entire group? no, it doesnt work
str_extract_all(strings, "(\\((?:\\1*|[^()]*?)+\\))")
```

```
## [[1]]
## [1] "((((x)"
##
## [[2]]
## [1] "((x)" "((y)" "(z)"
##
## [[3]]
## [1] "(x)" "(y)"
```

`stringr`'s functions don't support recusions, we will need to use base R functions (which are difficult to use and slower). Luckily, there is `rematch2` which provides nice wrapping functions to base R functions (still, it is slower).

```
library(rematch2)
re_match_all(strings, "\\((?:(?0)*|[^()]*?)+\\)") %>%
  pull(.match)
```

```
## [[1]]
## [1] "((((x))))"
##
## [[2]]
## [1] "((x) ((y)) (z))"
##
## [[3]]
## [1] "(x)" "(y)"
```

Three games to learn regex

- http://play.inginf.units.it
- https://alf.nu/RegexGolf
- https://regexcrossword.com/

Reference

- Online regex tester https://regex101.com/
- R for Data Science https://r4ds.had.co.nz/strings.html