A brief introduction to regular expressions

Data Wrangling: Session 5

Kieran Healy
Statistical Horizons, April 2022

Load the packages, as always

```r
library(here)      # manage file paths
library(socviz)    # data and some useful functions
```

```r
library(tidyverse) # your friend and mine
```

```
## ── Attaching packages ─────────────────────────────────────── tidyverse 1.3.1 ──

## ✓ ggplot2 3.3.5     ✓ purrr   0.3.4
## ✓ tibble  3.1.6     ✓ dplyr   1.0.8
## ✓ tidyr   1.2.0     ✓ stringr 1.4.0
## ✓ readr   2.1.2     ✓ forcats 0.5.1

## ── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
## x readr::edition_get()   masks testthat::edition_get()
## x dplyr::filter()        masks stats::filter()
## x purrr::is_null()       masks testthat::is_null()
## x dplyr::lag()           masks stats::lag()
## x readr::local_edition() masks testthat::local_edition()
## x dplyr::matches()       masks tidyr::matches(), testthat::matches()
```

```r
library(gapminder) # gapminder data
library(stringr)
```

Regular Expressions

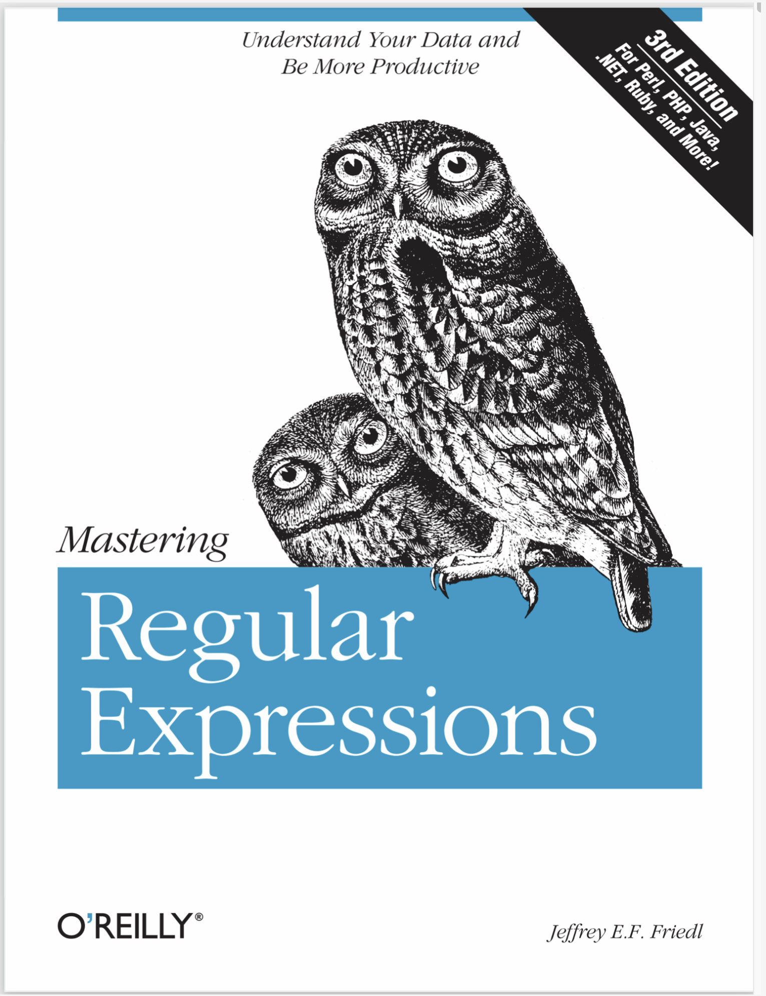
Or, waiter, there appears to be a language inside my language

stringr is your gateway to regexps

```r
library(stringr)
```

Part of the tidyverse, but not loaded by default.

regexps are their own whole world



This book is a thing of beauty.

Searching for patterns

A regular expression is a way of searching for a piece of text, or *pattern*, inside some larger body of text, called a *string*.

Searching for patterns

A regular expression is a way of searching for a piece of text, or *pattern*, inside some larger body of text, called a *string*.

The simplest sort of search is like the "Find" functionality in a Word Processor, where the pattern is a literal letter, number, punctuation mark, word or series of words and the text is a document that gets searched one line at a time. The next step up is "Find and Replace".

Searching for patterns

A regular expression is a way of searching for a piece of text, or *pattern*, inside some larger body of text, called a *string*.

The simplest sort of search is like the "Find" functionality in a Word Processor, where the pattern is a literal letter, number, punctuation mark, word or series of words and the text is a document that gets searched one line at a time. The next step up is "Find and Replace".

Every pattern-searching function in `stringr` has the same basic form:

```
str_view(<STRING>, <PATTERN>, [...]) # where [...] means "maybe some options"
```

Searching for patterns

A regular expression is a way of searching for a piece of text, or *pattern*, inside some larger body of text, called a *string*.

The simplest sort of search is like the "Find" functionality in a Word Processor, where the pattern is a literal letter, number, punctuation mark, word or series of words and the text is a document that gets searched one line at a time. The next step up is "Find and Replace".

Every pattern-searching function in `stringr` has the same basic form:

```
str_view(<STRING>, <PATTERN>, [...]) # where [...] means "maybe some options"
```

Functions that *replace* as well as *detect* strings all have this form:

```
str_replace(<STRING>, <PATTERN>, <REPLACEMENT>)
```

Searching for patterns

A regular expression is a way of searching for a piece of text, or *pattern*, inside some larger body of text, called a *string*.

The simplest sort of search is like the "Find" functionality in a Word Processor, where the pattern is a literal letter, number, punctuation mark, word or series of words and the text is a document that gets searched one line at a time. The next step up is "Find and Replace".

Every pattern-searching function in `stringr` has the same basic form:

```
str_view(<STRING>, <PATTERN>, [...]) # where [...] means "maybe some options"
```

Functions that *replace* as well as *detect* strings all have this form:

```
str_replace(<STRING>, <PATTERN>, <REPLACEMENT>)
```

(If you think about it, `<STRING>`, `<PATTERN>` and `<REPLACEMENT>` above are all kinds of pattern: they are meant to "stand for" all kinds of text, not be taken literally.)

Searching for patterns

Here I'll follow the exposition in Wickham & Grolemund (2017).

```
x <- c("apple", "banana", "pear")

str_view(x, "an")
```

```
apple
banana
pear
```

Searching for patterns

Regular expressions get their real power from *wildcards*, i.e. tokens that match more than just literal strings, but also more general and more complex patterns.

Searching for patterns

Regular expressions get their real power from *wildcards*, i.e. tokens that match more than just literal strings, but also more general and more complex patterns.

The most general pattern-matching token is, "Match everything!" This is represented by the period, or `.`]

Searching for patterns

Regular expressions get their real power from *wildcards*, i.e. tokens that match more than just literal strings, but also more general and more complex patterns.

The most general pattern-matching token is, "Match everything!" This is represented by the period, or `.`]

But ... if “`.`” matches any character, how do you specifically match the character “`.`”?

Escaping

You have to "escape" the period to tell the regex you want to match it exactly, rather than interpret it as meaning "match anything".

Escaping

You have to "escape" the period to tell the regex you want to match it exactly, rather than interpret it as meaning "match anything".

regexs use the backslash, ****, to signal "escape the next character".

Escaping

You have to "escape" the period to tell the regex you want to match it exactly, rather than interpret it as meaning "match anything".

regexs use the backslash, `\`, to signal "escape the next character".

To match a ".", you need the regex "`\.`"

Hang on, I see a further problem

We use strings to represent regular expressions. `\` is also used as an escape symbol in strings. So to create the regular expression `\.` we need the string `"\\."`

```
# To create the regular expression, we need \\
dot <- "\\."

# But the expression itself only contains one:
writeLines(dot)
```

```
## \.
```

```
# And this tells R to look for an explicit .
str_view(c("abc", "a.c", "bef"), "a\\.c")
```

abc

a.c

bef

But ... then how do you match a literal \ ?

```
x <- "a\\b"
writeLines(x)
```

```
## a\b
```

```
#> a\b

str_view(x, "\\\\") # you need four!
```

a\b

But … then how do you match a literal \?

This is the price we pay for having to express searches for patterns using a language containing these same characters, which we may also want to search for.

I *promise* this will pay off

Use `^` to match the start of a string.

Use `$` to match the end of a string.

I *promise* this will pay off

Use ^ to match the start of a string.

Use $ to match the end of a string.

```
x <- c("apple", "banana", "pear")
str_view(x, "^a")
```

```
apple
banana
pear
```

I *promise* this will pay off

Use **^** to match the start of a string.

Use **$** to match the end of a string.

```
x <- c("apple", "banana", "pear")
str_view(x, "^a")
```

```
apple
banana
pear
```

```
str_view(x, "a$")
```

```
apple
banana
pear
```

Matching start and end

To force a regular expression to only match a complete string, anchor it with both `^` and `$`

Matching start and end

To force a regular expression to only match a complete string, anchor it with both **^** and **$**

```
x <- c("apple pie", "apple", "apple cake")
str_view(x, "apple")
```

```
apple pie
apple
apple cake
```

Matching start and end

To force a regular expression to only match a complete string, anchor it with both ^ and $

```r
x <- c("apple pie", "apple", "apple cake")
str_view(x, "apple")
```

```
apple pie
apple
apple cake
```

```r
str_view(x, "^apple$")
```

```
apple pie
apple
apple cake
```

Matching character classes

`\d` matches any digit.

`\s` matches any whitespace (e.g. space, tab, newline).

`[abc]` matches a, b, or c.

`[^abc]` matches anything except a, b, or c.

Matching the *special* characters

Look for a literal character that normally has special meaning in a regex

```
str_view(c("abc", "a.c", "a*c", "a c"), "a[.]c")
```

```
abc
a.c
a*c
a c
```

Matching the *special* characters

Look for a literal character that normally has special meaning in a regex

```
str_view(c("abc", "a.c", "a*c", "a c"), "a[.]c")
```

```
abc
a.c
a*c
a c
```

```
str_view(c("abc", "a.c", "a*c", "a c"), ".[*]c")
```

```
abc
a.c
a*c
a c
```

Alternation

Use parentheses to make the precedence of | clear:

```
str_view(c("groy", "grey", "griy", "gray"), "gr(e|a)y")
```

```
groy
grey
griy
gray
```

Repeated patterns

? is 0 or 1

+ is 1 or more

***** is 0 or more

```
x <- "1888 is the longest year in Roman numerals: MDCCCLXXXVIII"
str_view(x, "CC?")
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

Repeated patterns

? is 0 or 1

+ is 1 or more

***** is 0 or more

```
str_view(x, "CC+")
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

Repeated patterns

? is 0 or 1

+ is 1 or more

***** is 0 or more

```
x <- "1888 is the longest year in Roman numerals: MDCCCLXXXVIII"
str_view(x, 'C[LX]+')
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

Exact numbers of repetitions

`{n}` is exactly n

`{n,}` is n or more

`{,m}` is at most m

`{n,m}` is between n and m

```
str_view(x, "C{2}")
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

Exact numbers of repetitions

`{n}` is exactly n

`{n,}` is n or more

`{,m}` is at most m

`{n,m}` is between n and m

```
str_view(x, "C{2,}")
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

Exact numbers of repetitions

`{n}` is exactly n

`{n,}` is n or more

`{,m}` is at most m

`{n,m}` is between n and m

By default these are *greedy* matches. You can make them “lazy”, matching the shortest string possible by putting a **?** after them. **This is often very useful!**

```
str_view(x, 'C[LX]+?')
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

And finally ... backreferences

```r
fruit # built into stringr
```

```
##  [1] "apple"             "apricot"           "avocado"
##  [4] "banana"            "bell pepper"       "bilberry"
##  [7] "blackberry"        "blackcurrant"      "blood orange"
## [10] "blueberry"         "boysenberry"       "breadfruit"
## [13] "canary melon"      "cantaloupe"        "cherimoya"
## [16] "cherry"            "chili pepper"      "clementine"
## [19] "cloudberry"        "coconut"           "cranberry"
## [22] "cucumber"          "currant"           "damson"
## [25] "date"              "dragonfruit"       "durian"
## [28] "eggplant"          "elderberry"        "feijoa"
## [31] "fig"               "goji berry"        "gooseberry"
## [34] "grape"             "grapefruit"        "guava"
## [37] "honeydew"          "huckleberry"       "jackfruit"
## [40] "jambul"            "jujube"            "kiwi fruit"
## [43] "kumquat"           "lemon"             "lime"
## [46] "loquat"            "lychee"            "mandarine"
## [49] "mango"             "mulberry"          "nectarine"
## [52] "nut"               "olive"             "orange"
## [55] "pamelo"            "papaya"            "passionfruit"
## [58] "peach"             "pear"              "persimmon"
## [61] "physalis"          "pineapple"         "plum"
## [64] "pomegranate"       "pomelo"            "purple mangosteen"
## [67] "quince"            "raisin"            "rambutan"
## [70] "raspberry"         "redcurrant"        "rock melon"
## [73] "salal berry"       "satsuma"           "star fruit"
## [76] "strawberry"        "tamarillo"         "tangerine"
## [79] "ugli fruit"        "watermelon"
```

Grouping and backreferences

Find all fruits that have a repeated pair of letters:

```
str_view(fruit, "(..)\\1", match = TRUE)
```

```
banana
coconut
cucumber
jujube
papaya
salal berry
```

Grouping and backreferences

Backreferences and grouping will be very useful for string *replacements*.

OK that was a lot

Learning and testing regexps

Practice with a tester like https://regexr.com

Or an app like Patterns

The regex engine or "flavor" used by `stringr` is Perl- or PCRE-like.

What was the point of that?

We use basic or slightly fancy regexps *very often* when importing and cleaning data.

What was the point of that?

We use basic or slightly fancy regexps *very often* when importing and cleaning data.

As we'll soon see! It's time to read in a bunch of data.