

Search and Edit Text

*Modern Plain Text Social Science:
Week 05*

Kieran Healy

kieran.healy@duke.edu

September 2026

Getting around in the Shell

Your command history

Eventually you will accumulate a history of shell commands you have typed, many of which you may want to reuse from time to time.

Go to the previous command with the up arrow, ↑

Search your command history with control-R, ^R

^R will also work for history search at the RStudio console and in many other places.

Aside: Standard Modifier Key Symbols

Symbol	Key	Unicode	Symbol	Key	Unicode
↵	Escape	U+238B	⌫	Backspace	U+232B
→	Tab	U+21E5	⌫	Delete	U+2326
⇧	Caps Lock	U+21EA	⌵	Home	U+21F1
⇧	Shift	U+21E7	⇩	End	U+21F2
^	Control	U+2303	⇧⇩	Page Up	U+21DE
⌥	Option/Alt	U+2325	⇩⇩	Page Down	U+21DF
⌘	Command	U+2318	↵	Enter	U+2324
↵	Return	U+23CE	␣	Space	U+2423

Searching inside files in the shell

grep

`find` searches file and folder names only. To search *inside* files we use `grep`. Or rather we will use a flavor of `grep` called `egrep`.

```
grep 'Stately' files/examples/ulysses.txt
```

```
Stately, plump Buck Mulligan came from the stairhead, bearing a bowl of
```

Search more than one file:

```
## -i for case-insensitive search  
egrep -i 'jabber' files/examples/*.txt
```

```
files/examples/jabberwocky.txt:"Beware the Jabberwock, my son!  
files/examples/jabberwocky.txt:      The Jabberwock, with eyes of flame,  
files/examples/jabberwocky.txt:"And hast thou slain the Jabberwock?  
files/examples/ulysses.txt:jabber on their girdles: rogue words, tough nuggets patter in their
```

The **grep** family

grep and its derivatives are very powerful thanks to their ability to use *regular expressions*. We will learn about those momentarily. There are also more recent command-line search tools like ripgrep, or **rg**, a modern version of **grep** that is very fast, automatically searches subfolders, and has other nice additional features. For more details see the [ripgrep project page](#).

```
# On a Mac with Homebrew (https://brew.sh)  
brew install rg
```

Integrate **rg** and **fzf**

fzf is a command-line fuzzy-finder. It makes **^R** really powerful and convenient. For details see the [fzf project page](#).

```
# On a Mac  
brew install fzf
```

```
idlewild ~/D/c/mptc git:main >  
2128 R CMD INSTALL covdata_0.4.7.tar.gz  
2132 R CMD INSTALL myriad_0.0.3.9000.tar.gz  
2144 R CMD INSTALL covdata_0.5.tgz  
2201 R CMD INSTALL covmobility_0.1.tar.gz  
3647 R CMD INSTALL kjhutils_0.4.tar.gz  
3650 R CMD INSTALL myriad_0.0.2.9000.tar.gz  
3651 R CMD INSTALL sourcesans_0.0.0.9000.tar.gz  
6407 R CMD INSTALL myriad_0.0.7.9000.tar.gz  
6408 R CMD INSTALL myriad_0.0.9.9000.tar.gz  
6409 R CMD INSTALL myriad_0.0.1.9000.tar.gz  
6410 R CMD INSTALL myriad_0.1.1.9000.tar.gz  
8002 R CMD INSTALL myriad_0.2.1.9000.tar.gz  
8007 R CMD INSTALL myriad_0.2.2.9000.tar.gz  
8128 R CMD install kjhslides_0.0.11.9000.tar.gz  
8469 R CMD INSTALL myriad_0.2.3.9000.tar.gz  
8630 git add .github/workflows/R-CMD-check.yaml  
8885 R --vanilla CMD INSTALL kjhutils_0.4.tar.gz  
9411 R --vanilla CMD INSTALL flipbookr_0.1.0.tar.gz  
9447 R CMD INSTALL flipbookr_0.1.0.tar.gz  
> 9452 R CMD INSTALL flipbookr_0.2.0.tar.gz  
37/9999 +S  
> R CMD
```

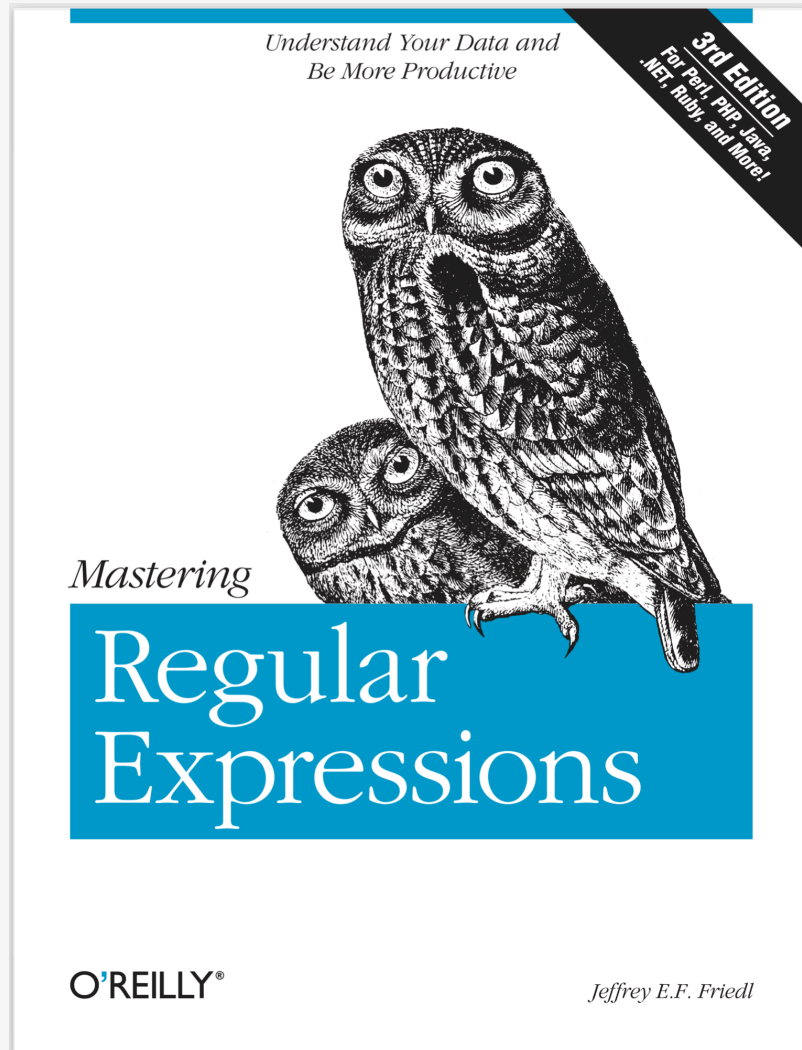
Regular Expressions

Regular Expressions

Or,

**Waiter, there
appears to be a
language inside my
language**

Regular Expressions



Regexps are their own world of
text processing

☞ This book is a thing of
beauty.

We'll focus on R

We'll look at regular expressions via R, but most of the same things apply to the shell as well.

Searching for patterns

Searching for patterns

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. The pattern is a literal letter, number, punctuation mark, word or series of words; the strings are all the sentences in a document.

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. The pattern is a literal letter, number, punctuation mark, word or series of words; the strings are all the sentences in a document.

When searching a plain-text file, our strings are the *lines* of some plain text file. We search the file for some pattern and we want to see every line in the file where there is a match.

Why we use them

To *detect* text, to *extract* it, to *replace* or *transform* it.

stringr is your gateway to regexps in R

```
library(stringr) # It's loaded by default with library(tidyverse)
```

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. The pattern is a literal letter, number, punctuation mark, word or series of words; the text is a document 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. The pattern is a literal letter, number, punctuation mark, word or series of words; the text is a document 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. The pattern is a literal letter, number, punctuation mark, word or series of words; the text is a document 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. The pattern is a literal letter, number, punctuation mark, word or series of words; the text is a document 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). The `str_view()` function just displays what your regexp will match in a given string. In practice you will use functions that work in the same way but that actually do something you want, like `str_detect()` `str_replace()`, `str_extract()` or `str_squish()`. There are a lot of useful `str_` functions.

```
x ← c("apple", "banana", "pear")
```

```
str_view(x, "an")
```

```
[2] | b<an><an>a
```

Searching for patterns

Searching for 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.

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 literal character `.`?

Escaping

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")
```

```
[2] | <a.c>
```

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

```
x ← "a\\b"  
writeLines(x)
```

```
a\b
```

```
#> a\b
```

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

```
[1] | a<\>b
```

But ... 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**

Matching start and end

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

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

```
[1] | <a>pple
```

Matching start and end

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

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

```
[1] | <a>pple
```

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

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

```
[2] | banan<a>
```

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")
```

```
[1] | <apple> pie  
[2] | <apple>  
[3] | <apple> cake
```

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

```
[2] | <apple>
```

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")
```

```
[2] | <a.c>
```

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

```
[3] | <a*c>
```

This works for most (but not all) regex metacharacters: `$.|?*+()[]`.

Unfortunately, a few characters have special meaning even inside a character class and must be handled with backslash escapes. These are `]`, `\`, `^` and `-`.

Alternation

Use parentheses to make the precedence of the 'or' operator **|** clear:

```
str_view(c("groy", "grey", "griy", "gray"), "gr(ela)y")
```

```
[2] | <grey>
```

```
[4] | <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?")
```

```
[1] | 1888 is the longest year in Roman numerals: MD<CC><C>LXXXVIII
```

Repeated patterns

? is 0 or 1

+ is 1 or more

* is 0 or more

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

```
[1] | 1888 is the longest year in Roman numerals: MD<CCC>LXXXVIII
```

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]+')
```

```
[1] | 1888 is the longest year in Roman numerals: MDCC<CLXXX>VIII
```

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}")
```

```
[1] | 1888 is the longest year in Roman numerals: MD<CC>CLXXXVIII
```

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,}")
```

```
[1] | 1888 is the longest year in Roman numerals: MD<CCC>LXXXVIII
```

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{2,3}")
```

```
[1] | 1888 is the longest year in Roman numerals: MD<CCC>LXXXVIII
```

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 regexps use *greedy* matches. You can make them match the *shortest* string possible by putting a `?` after them. **This is often very useful!**

```
str_view(x, 'C{2,3}?')
```

```
[1] | 1888 is the longest year in Roman numerals: MD<CC>CLXXXVIII
```

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]+?')
```

```
[1] | 1888 is the longest year in Roman numerals: MDCC<CL>XXXVIII
```

And **finally** ... backreferences

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"

Grouping and backreferences

Find all fruits that have a repeated pair of letters:

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

```
[4] | b<anan>a  
[20] | <coco>nut  
[22] | <cucu>mber  
[41] | <juju>be  
[56] | <papa>ya  
[73] | s<alal> berry
```

Grouping and backreferences

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

OK that was a lot



Application

Example: Politics and Placenames

```
library(tidyverse)
```

Example: Politics and Placenames

```
library(tidyverse)
library(ukelection2019)
```

Example: Politics and Placenames

```
library(tidyverse)
library(ukelection2019)

ukvote2019
```

```
# A tibble: 3,320 × 13
  cid constituency electorate party_name candidate
votes vote_share_percent
  <chr>      <chr>          <int> <chr>      <chr>
<int>          <dbl>
1 W07000... Aberavon          50747 Labour      Stephen ...
17008          53.8
2 W07000... Aberavon          50747 Conservat... Charlott...
6518          20.6
3 W07000... Aberavon          50747 The Brexi... Glenda D...
3108          9.8
4 W07000... Aberavon          50747 Plaid Cym... Nigel Hu...
2711          8.6
5 W07000... Aberavon          50747 Liberal D... Sheila K...
1072          3.4
6 W07000... Aberavon          50747 Independe... Captain ...
731          2.3
7 W07000... Aberavon          50747 Green        Giorgia ...
450          1.4
8 W07000... Aberconwy          44699 Conservat... Robin Mi...
14687         46.1
9 W07000... Aberconwy          44699 Labour      Emily Ow...
12653         39.7
10 W07000... Aberconwy          44699 Plaid Cym... Lisa Goo...
2704          8.5
```

Example: Politics and Placenames

```
library(tidyverse)
library(ukelection2019)

ukvote2019 >
  group_by(constituency)
```

```
# A tibble: 3,320 × 13
# Groups:   constituency [650]
   cid      constituency electorate party_name candidate
votes vote_share_percent
  <int>    <chr>          <int> <chr>      <chr>
<int>    <dbl>
1 W07000... Aberavon      50747 Labour    Stephen ...
17008      53.8
2 W07000... Aberavon      50747 Conservat... Charlott...
6518      20.6
3 W07000... Aberavon      50747 The Brexi... Glenda D...
3108      9.8
4 W07000... Aberavon      50747 Plaid Cym... Nigel Hu...
2711      8.6
5 W07000... Aberavon      50747 Liberal D... Sheila K...
1072      3.4
6 W07000... Aberavon      50747 Independe... Captain ...
731      2.3
7 W07000... Aberavon      50747 Green       Giorgia ...
450      1.4
8 W07000... Aberconwy     44699 Conservat... Robin Mi...
14687     46.1
9 W07000... Aberconwy     44699 Labour      Emily Ow...
12653     39.7
10 W07000... Aberconwy     44699 Plaid Cym... Lisa Goo...
```

Example: Politics and Placenames

```
library(tidyverse)
library(ukelection2019)

ukvote2019 >
  group_by(constituency) >
  slice_max(votes)
```

```
# A tibble: 650 × 13
# Groups:   constituency [650]
   cid constituency electorate party_name candidate
  <chr>      <chr>      <int> <chr>      <chr>
<int>          <dbl>
1 W07000... Aberavon      50747 Labour      Stephen ...
17008              53.8
2 W07000... Aberconwy     44699 Conservat... Robin Mi...
14687              46.1
3 S14000... Aberdeen No... 62489 Scottish ... Kirsty B...
20205              54
4 S14000... Aberdeen So... 65719 Scottish ... Stephen ...
20388              44.7
5 S14000... Aberdeenshi... 72640 Conservat... Andrew B...
22752              42.7
6 S14000... Airdrie & S... 64008 Scottish ... Neil Gray
17929              45.1
7 E14000... Aldershot     72617 Conservat... Leo Doch...
27980              58.4
8 E14000... Aldridge-Br... 60138 Conservat... Wendy Mo...
27850              70.8
9 E14000... Altrincham ... 73096 Conservat... Graham B...
26311              48
10 W07000... Alyn & Dees... 62783 Labour      Mark Tami
```

Example: Politics and Placenames

```
library(tidyverse)
library(ukelection2019)
```

```
ukvote2019 >
  group_by(constituency) >
  slice_max(votes) >
  ungroup()
```

```
# A tibble: 650 × 13
  cid      constituency electorate party_name candidate
votes vote_share_percent
  <chr>    <chr>          <int> <chr>      <chr>
<int>          <dbl>
1 W07000... Aberavon          50747 Labour      Stephen ...
17008          53.8
2 W07000... Aberconwy          44699 Conservat... Robin Mi...
14687          46.1
3 S14000... Aberdeen No...          62489 Scottish ... Kirsty B...
20205          54
4 S14000... Aberdeen So...          65719 Scottish ... Stephen ...
20388          44.7
5 S14000... Aberdeenshi...          72640 Conservat... Andrew B...
22752          42.7
6 S14000... Airdrie & S...          64008 Scottish ... Neil Gray
17929          45.1
7 E14000... Aldershot          72617 Conservat... Leo Doch...
27980          58.4
8 E14000... Aldridge-Br...          60138 Conservat... Wendy Mo...
27850          70.8
9 E14000... Altrincham ...          73096 Conservat... Graham B...
26311          48
10 W07000... Alyn & Dees...          62783 Labour      Mark Tami
18271          42.5
```

Example: Politics and Placenames

```
library(tidyverse)
library(ukelection2019)

ukvote2019 >
  group_by(constituency) >
  slice_max(votes) >
  ungroup() >
  select(constituency, party_name)
```

```
# A tibble: 650 × 2
  constituency party_name
  <chr>         <chr>
1 Aberavon     Labour
2 Aberconwy    Conservative
3 Aberdeen North Scottish National
Party
4 Aberdeen South Scottish National
Party
5 Aberdeenshire West & Kincardine Conservative
6 Airdrie & Shotts Scottish National
Party
7 Aldershot    Conservative
8 Aldridge-Brownhills Conservative
9 Altrincham & Sale West Conservative
10 Alyn & Deeside Labour
# i 640 more rows
```

Example: Politics and Placenames

```
library(tidyverse)
library(ukelection2019)

ukvote2019 >
  group_by(constituency) >
  slice_max(votes) >
  ungroup() >
  select(constituency, party_name) >
  mutate(shire = str_detect(constituency, "shire"),
         field = str_detect(constituency, "field"),
         dale = str_detect(constituency, "dale"),
         pool = str_detect(constituency, "pool"),
         ton = str_detect(constituency, "(ton$)|(ton )"),
         wood = str_detect(constituency, "(wood$)|(wood )"),
         saint = str_detect(constituency, "(St )|(Saint)"),
         port = str_detect(constituency, "(Port)|(port)"),
         ford = str_detect(constituency, "(ford$)|(ford )"),
         by = str_detect(constituency, "(by$)|(by )"),
         boro = str_detect(constituency, "(boro$)|(boro )|(borough$)|(borough )"),
         ley = str_detect(constituency, "(ley$)|(ley )|(leigh$)|(leigh )"))
```

```
# A tibble: 650 × 14
  constituency party_name shire field dale pool ton
  <chr>          <chr>    <lgl> <lgl> <lgl> <lgl> <lgl>
1 Aberavon      Labour    FALSE FALSE FALSE FALSE FALSE
2 Aberconwy     Conservat... FALSE FALSE FALSE FALSE FALSE
3 Aberdeen No... Scottish ... FALSE FALSE FALSE FALSE FALSE
4 Aberdeen So... Scottish ... FALSE FALSE FALSE FALSE FALSE
5 Aberdeenshi... Conservat... TRUE  FALSE FALSE FALSE FALSE
6 Airdrie & S... Scottish ... FALSE FALSE FALSE FALSE FALSE
7 Aldershot     Conservat... FALSE FALSE FALSE FALSE FALSE
8 Aldridge-Br... Conservat... FALSE FALSE FALSE FALSE FALSE
9 Altrincham ... Conservat... FALSE FALSE FALSE FALSE FALSE
10 Alyn & Dees... Labour     FALSE FALSE FALSE FALSE FALSE
```

Example: Politics and Placenames

```
library(tidyverse)
library(ukelection2019)

ukvote2019 >
  group_by(constituency) >
  slice_max(votes) >
  ungroup() >
  select(constituency, party_name) >
  mutate(shire = str_detect(constituency, "shire"),
         field = str_detect(constituency, "field"),
         dale = str_detect(constituency, "dale"),
         pool = str_detect(constituency, "pool"),
         ton = str_detect(constituency, "(ton$)|(ton )"),
         wood = str_detect(constituency, "(wood$)|(wood )"),
         saint = str_detect(constituency, "(St )|(Saint)"),
         port = str_detect(constituency, "(Port)|(port)"),
         ford = str_detect(constituency, "(ford$)|(ford )"),
         by = str_detect(constituency, "(by$)|(by )"),
         boro = str_detect(constituency, "(boro$)|(boro )|(borough$)|(borough )"),
         ley = str_detect(constituency, "(ley$)|(ley )|(leigh$)|(leigh )")) >
  pivot_longer(shire:ley, names_to = "toponym")
```

```
# A tibble: 7,800 × 4
  constituency party_name toponym value
  <chr>         <chr>      <chr> <lgl>
1 Aberavon     Labour     shire FALSE
2 Aberavon     Labour     field FALSE
3 Aberavon     Labour     dale  FALSE
4 Aberavon     Labour     pool  FALSE
5 Aberavon     Labour     ton   FALSE
6 Aberavon     Labour     wood  FALSE
7 Aberavon     Labour     saint FALSE
8 Aberavon     Labour     port  FALSE
9 Aberavon     Labour     ford  FALSE
10 Aberavon    Labour     by    FALSE
# i 7,790 more rows
```

Example: Politics and Placenames

```
place_tab ← ukvote2019 ▷
  group_by(constituency) ▷
  slice_max(votes) ▷
  ungroup() ▷
  select(constituency, party_name) ▷
  # We could write these more efficiently but we don't care about that rn
  mutate(shire = str_detect(constituency, "shire"),
         field = str_detect(constituency, "field"),
         dale = str_detect(constituency, "dale"),
         pool = str_detect(constituency, "pool"),
         ton = str_detect(constituency, "(ton$)|(ton )"),
         wood = str_detect(constituency, "(wood$)|(wood )"),
         saint = str_detect(constituency, "(St )|(Saint)"),
         port = str_detect(constituency, "(Port)|(port)"),
         ford = str_detect(constituency, "(ford$)|(ford )"),
         by = str_detect(constituency, "(by$)|(by )"),
         boro = str_detect(constituency, "(boro$)|(boro )|(borough$)|(borough )"),
         ley = str_detect(constituency, "(ley$)|(ley )|(leigh$)|(leigh )")) ▷
  pivot_longer(shire:ley, names_to = "toponym")
```

Example: Politics and Placenames

```
place_tab
```

```
# A tibble: 7,800 × 4
  constituency party_name toponym value
  <chr>         <chr>      <chr>  <lgl>
1 Aberavon     Labour     shire  FALSE
2 Aberavon     Labour     field  FALSE
3 Aberavon     Labour     dale   FALSE
4 Aberavon     Labour     pool   FALSE
5 Aberavon     Labour     ton    FALSE
6 Aberavon     Labour     wood   FALSE
7 Aberavon     Labour     saint  FALSE
8 Aberavon     Labour     port   FALSE
9 Aberavon     Labour     ford   FALSE
10 Aberavon    Labour     by     FALSE
# i 7,790 more rows
```

Example: Politics and Placenames

```
place_tab ►  
  group_by(party_name, toponym)
```

```
# A tibble: 7,800 × 4  
# Groups:   party_name, toponym [120]  
  constituency party_name toponym value  
  <chr>        <chr>    <chr>  <lgl>  
1 Aberavon     Labour    shire  FALSE  
2 Aberavon     Labour    field  FALSE  
3 Aberavon     Labour    dale   FALSE  
4 Aberavon     Labour    pool   FALSE  
5 Aberavon     Labour    ton    FALSE  
6 Aberavon     Labour    wood   FALSE  
7 Aberavon     Labour    saint  FALSE  
8 Aberavon     Labour    port   FALSE  
9 Aberavon     Labour    ford   FALSE  
10 Aberavon    Labour    by     FALSE  
# i 7,790 more rows
```

Example: Politics and Placenames

```
place_tab ►  
  group_by(part_name, toponym) ►  
  filter(part_name %in% c("Conservative", "Labour"))
```

```
# A tibble: 6,816 × 4  
# Groups:   part_name, toponym [24]  
  constituency part_name toponym value  
  <chr>         <chr>    <chr>  <lgl>  
1 Aberavon     Labour    shire   FALSE  
2 Aberavon     Labour    field   FALSE  
3 Aberavon     Labour    dale    FALSE  
4 Aberavon     Labour    pool    FALSE  
5 Aberavon     Labour    ton      FALSE  
6 Aberavon     Labour    wood     FALSE  
7 Aberavon     Labour    saint    FALSE  
8 Aberavon     Labour    port     FALSE  
9 Aberavon     Labour    ford     FALSE  
10 Aberavon    Labour    by        FALSE  
# i 6,806 more rows
```

Example: Politics and Placenames

```
place_tab ►  
  group_by(part_name, toponym) ►  
  filter(part_name %in% c("Conservative", "Labour")) ►  
  group_by(toponym, part_name)
```

```
# A tibble: 6,816 × 4  
# Groups:   toponym, part_name [24]  
  constituency part_name toponym value  
  <chr>         <chr>    <chr>  <lgl>  
1 Aberavon     Labour    shire   FALSE  
2 Aberavon     Labour    field   FALSE  
3 Aberavon     Labour    dale    FALSE  
4 Aberavon     Labour    pool    FALSE  
5 Aberavon     Labour    ton      FALSE  
6 Aberavon     Labour    wood     FALSE  
7 Aberavon     Labour    saint    FALSE  
8 Aberavon     Labour    port     FALSE  
9 Aberavon     Labour    ford     FALSE  
10 Aberavon    Labour    by        FALSE  
# i 6,806 more rows
```

Example: Politics and Placenames

```
place_tab >
  group_by(party_name, toponym) >
  filter(party_name %in% c("Conservative", "Labour")) >
  group_by(toponym, party_name) >
  summarize(freq = sum(value))
```

```
# A tibble: 24 × 3
# Groups:   toponym [12]
  toponym party_name freq
  <chr>    <chr>    <int>
1 boro    Conservative  7
2 boro    Labour       1
3 by      Conservative  6
4 by      Labour       2
5 dale    Conservative  3
6 dale    Labour       1
7 field   Conservative 10
8 field   Labour       10
9 ford    Conservative 17
10 ford   Labour       12
# i 14 more rows
```

Example: Politics and Placenames

```
place_tab >
  group_by(party_name, toponym) >
  filter(party_name %in% c("Conservative", "Labour")) >
  group_by(toponym, party_name) >
  summarize(freq = sum(value)) >
  mutate(pct = freq/sum(freq))
```

```
# A tibble: 24 × 4
# Groups:   toponym [12]
  toponym party_name    freq  pct
  <chr>    <chr>      <int> <dbl>
1 boro    Conservative     7 0.875
2 boro    Labour           1 0.125
3 by      Conservative     6 0.75
4 by      Labour           2 0.25
5 dale    Conservative     3 0.75
6 dale    Labour           1 0.25
7 field   Conservative    10 0.5
8 field   Labour          10 0.5
9 ford    Conservative    17 0.586
10 ford    Labour          12 0.414
# i 14 more rows
```

Example: Politics and Placenames

```
place_tab >
  group_by(party_name, toponym) >
  filter(party_name %in% c("Conservative", "Labour")) >
  group_by(toponym, party_name) >
  summarize(freq = sum(value)) >
  mutate(pct = freq/sum(freq)) >
  filter(party_name == "Conservative")
```

```
# A tibble: 12 × 4
# Groups:   toponym [12]
  toponym party_name    freq  pct
  <chr>    <chr>      <int> <dbl>
1 boro     Conservative     7 0.875
2 by       Conservative     6 0.75
3 dale     Conservative     3 0.75
4 field    Conservative    10 0.5
5 ford     Conservative    17 0.586
6 ley      Conservative    26 0.722
7 pool     Conservative     2 0.286
8 port     Conservative     3 0.333
9 saint    Conservative     3 0.5
10 shire    Conservative    37 0.974
11 ton      Conservative    37 0.507
12 wood     Conservative     7 0.636
```

Example: Politics and Placenames

```
place_tab >
  group_by(party_name, toponym) >
  filter(party_name %in% c("Conservative", "Labour")) >
  group_by(toponym, party_name) >
  summarize(freq = sum(value)) >
  mutate(pct = freq/sum(freq)) >
  filter(party_name == "Conservative") >
  arrange(desc(pct))
```

```
# A tibble: 12 × 4
# Groups:   toponym [12]
  toponym party_name    freq  pct
  <chr>    <chr>      <int> <dbl>
1 shire    Conservative    37 0.974
2 boro     Conservative     7 0.875
3 by       Conservative     6 0.75
4 dale     Conservative     3 0.75
5 ley      Conservative    26 0.722
6 wood     Conservative     7 0.636
7 ford     Conservative    17 0.586
8 ton      Conservative    37 0.507
9 field    Conservative    10 0.5
10 saint    Conservative     3 0.5
11 port     Conservative     3 0.333
12 pool     Conservative     2 0.286
```

Example: Politics and Placenames

```
place_tab >
  group_by(party_name, toponym) >
  filter(party_name %in% c("Conservative", "Labour")) >
  group_by(toponym, party_name) >
  summarize(freq = sum(value)) >
  mutate(pct = freq/sum(freq)) >
  filter(party_name == "Conservative") >
  arrange(desc(pct))
```

```
# A tibble: 12 × 4
# Groups:   toponym [12]
  toponym party_name    freq  pct
  <chr>    <chr>      <int> <dbl>
1 shire    Conservative    37 0.974
2 boro     Conservative     7 0.875
3 by       Conservative     6 0.75
4 dale     Conservative     3 0.75
5 ley      Conservative    26 0.722
6 wood     Conservative     7 0.636
7 ford     Conservative    17 0.586
8 ton      Conservative    37 0.507
9 field    Conservative    10 0.5
10 saint    Conservative     3 0.5
11 port     Conservative     3 0.333
12 pool     Conservative     2 0.286
```