Manipulating text with regular expressions


WHENEVER I LEARN A NEW SKILL I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.
OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!
BUT TO FIND THEM WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!
IT'S HOPELESS!
EVERYBODY STAND BACK.
I KNOW REGULAR EXPRESSIONS.
PERL!
TAP TAP

Why are we covering this?

Regular expressions find patterns in strings

You're probably familiar with "Find & Replace" (e.g. Word).

`"the"`

It was a special pleasure to see things eat
blackened and changed. With **the** brass n
with this great python spitting its venom
the world, **the** blood pounded in his hea
were **the** hands of some amazing conduct
symphonies of blazing and burning to bri
tatters and charcoal ruins of history. Wit
helmet numbered 451 on his stolid head,
orange flame with **the** thought of what ca
flicked **the** igniter and **the** house jumped

FAHRENHEIT 451
Ray Bradbury

Regular expressions find patterns in strings

Regular expressions are similar, but provide a powerful language for specifying search patterns.

Regular expressions find patterns in strings

Regular expressions are similar, but provide a powerful language for specifying search patterns.

For example:

`\bt\w+`	Match any word starting with “t”
`\b[a-z]{3}\b`	Match any three-letter word

Regular expressions find patterns in strings

Regular expressions are similar, but provide a powerful language for specifying search patterns.

For example:

`\bt\w+`	Match any word starting with “t”
`\b[a-z]{3}\b`	Match any three-letter word

Regular expressions have many uses

- Check if a string variable conforms to a particular format (e.g. email address or phone number).
- Extract a substring from a longer string.

Getting started

We need two things:

1. A regular expression (‘pattern’);
2. Some text to search (‘target string’).

Getting started

We need two things:

1. A regular expression ('pattern');
2. Some text to search ('target string').

A regular expression can contain literal characters and words (e.g. "the"), but there are many characters that have special meanings.

For example, ^ to denote the start of a string, or $ to denote the end.

We'll now look at some of these in more detail...

Regular expression concept 1: Character sets

We can use square brackets to create a character set.

`[the]`	Match one of the included characters, `t`, `h`, `e`.
`[eht]`	As above.
`[e02]`	Match one of the characters `e`, `0`, `2`.

Regular expression concept 1: Character sets

We can use square brackets to create a character set.

`[the]`	Match one of the included characters, `t`, `h`, `e`.
`[eht]`	As above.
`[e02]`	Match one of the characters `e`, `0`, `2`.

We can also specify ranges of characters to match.

`[a-z]`	Match a single character in the range `a`, `b`, ..., `z`.
`[0-9]`	Match a single character in the range `0`-`9`.
`[a-d0-9]`	Match a single character in the range `a`, `b`, `c`, `d`, `0`, `1`, ..., `9`.

Shorthand replacements are available for several common sets:

Expression	Meaning	Equivalent to...
`.`	Any character	🤷
`\d`	Digit	`[0-9]`
`\D`	Not a digit	`[^\d]`
`\w`	Word character	`[A-Za-z0-9_]`
`\W`	Not a word character	
`\s`	Whitespace character	`[ \t\r\n\f]`
`\S`	Not a whitespace character	

Regular expression concept 2: Repetition

We can repeat regular expressions:

`ttt`	The character `t` repeated three times.
`\w\w\w`	Any three word characters.

Regular expression concept 2: Repetition

We can repeat regular expressions:

`ttt`	The character `t` repeated three times.
`\w\w\w`	Any three word characters.

But we can also use shorthand to express repetition:

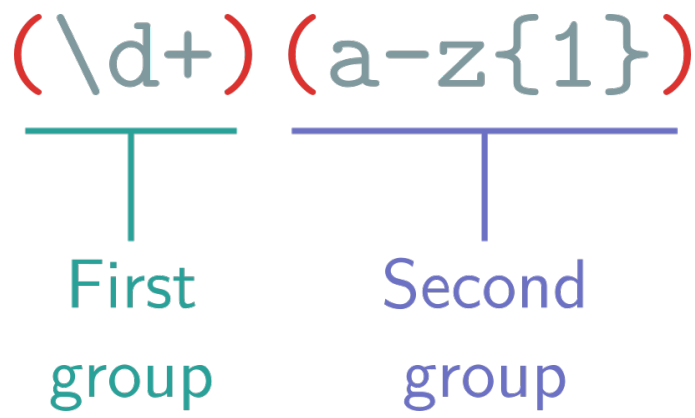
`*`	Repeat preceding expression 0 or more times
`+`	Repeat preceding expression 1 or more times
`{3}`	Repeat preceding expression exactly three times.

The above examples could thus be expressed more succinctly as `t{3}` and `\w{3}`.

Regular expression concept 3: Match groups

- Often, we want to extract the search pattern from a longer string.
- We can use match groups to help with this.

A match group is denoted with parentheses:



Having specified our groups, we can refer to them in later expressions at `\1`, `\2`, ..., `\N`.

You don't have to remember all these expressions.

1. Learn the core expressions

```
\d  \s
\w  .  [a-z]  *  +  {3}
```

2. For simple regular expressions, try writing from memory, make mistakes, get help, repeat…
3. For anything complicated, use a helper tool.

`https://regex101.com`

Practical

Regular expressions in R

We're using `stringr`, part of the `tidyverse`:

Function	What it does	Result
`str_detect`	Does the string contain this match?	`TRUE` / `FALSE`
`str_count`	How many times does this match occur?	Count
`str_locate`	Where is this match located (i.e. position)?	Position [start, end]; numeric
`str_extract`	Extract the first match.	The match.
`str_match`	Extract all match groups.	The matched groups, if they occur.
`str_replace`	Replace the first match.	The original string, with the match replaced.
`str_split`	Split the string at match.	Two strings.

`str_detect`

`str_detect` checks whether a string contains a regular expression, a returns a logical value (i.e `TRUE` or `FALSE`).

str_detect

`str_detect` checks whether a string contains a regular expression, a returns a logical value (i.e `TRUE` or `FALSE`).

```
> target <- "A long time ago in a galaxy far..."
> str_detect(target, "time")
[1] TRUE
> str_detect(target, "away")
[1] FALSE
> str_detect(target, "\\bl")
[1] TRUE
> str_detect(target, "\\bG")
[1] FALSE
> str_detect("2018-10-30", "^\\d{4}-\\d{2}\\d{2}")
[1] TRUE
```

→ If you get stuck, try it on regex101.com.

`str_match`

`str_match` extracts matched groups from a string.

str_match

`str_match` extracts matched groups from a string.

```
> names <- c("Luke Skywalker", "R2-D2",
>            "Darth Vader")
> str_match(names, "(^\\w+).*")
```

str_match

`str_match` extracts matched groups from a string.

```
> names <- c("Luke Skywalker", "R2-D2",
>            "Darth Vader")
> str_match(names, "(^\\w+).*")
[1,] "Luke Skywalker" "Luke"
[2,] "R2-D2"          "R2"
[3,] "Darth Vader"    "Darth"
```

str_match

`str_match` extracts matched groups from a string.

```
> names <- c("Luke Skywalker", "R2-D2",
>            "Darth Vader")
> str_match(names, "(^\\w+).*")
[1,] "Luke Skywalker" "Luke"
[2,] "R2-D2"          "R2"
[3,] "Darth Vader"    "Darth"
> str_match(names, "(^[\\w-]+).*")
[1,] "Luke Skywalker" "Luke"
[2,] "R2-D2"          "R2-D2"
[3,] "Darth Vader"    "Darth"
```

```
> str_match(names, "(^[\\w-]+) *([\\w-]*)")
     [,1]                [,2]    [,3]
[1,] "Luke Skywalker" "Luke"  "Skywalker"
[2,] "R2-D2"          "R2-D2" ""
[3,] "Darth Vader"    "Darth" "Vader"
```

str_replace

`str_replace` replaces the pattern with another string.

```
> str_replace(names, "D", "G")
[1] "Luke Skywalker" "R2-G2"    "Garth Vader"

> str_replace(names, "e", "!")
[1] "Luk! Skywalker" "R2-D2"    "Darth Vad!r"

> str_replace_all(names, "e", "!")
[1] "Luk! Skywalk!r" "R2-D2"    "Darth Vad!r"
```

Where to go next...

Next steps

1. Practice, practice, practice
    - Apply what you've learnt to a specific project.
    - Recognise that things will take longer at first.
2. Learn your editor (e.g., RStudio, Positron, VS Code)
3. Familiarise yourself with the command line.

    `https://jeroenjanssens.com/dsatcl/`
4. Learn version control with Git.
5. Learn package management with renv.

    `https://rstudio.github.io/renv/`
6. Build pipelines with `Make` or targets.

    `https://books.ropensci.org/targets/`
7. Get involved in the community

    `https://rladies.org`
    `https://posit.co/conference/`
    `https://nhsrcommunity.com/conference24.html`