String manipulation and regexes

Programming for Statistical Science

Shawn Santo

Supplementary materials

Full video lecture available in Zoom Cloud Recordings

Additional resources

- `stringr` vignette
- `stringr` cheat sheet
- regex guide

stringr

Why `stringr`?

- Part of `tidyverse`
- Fast and consistent manipulation of string data
- Readable and consistent syntax
- If you master `stringr`, you know `stringi` - http://www.gagolewski.com/software/stringi/

Usage

- All functions in `stringr` start with `str_` and take a vector of strings as the first argument.
- Most `stringr` functions work with regular expressions.
- Seven main verbs to work with strings.

Function	Description
`str_detect()`	Detect the presence or absence of a pattern in a string.
`str_count()`	Count the number of patterns.
`str_locate()`	Locate the first position of a pattern and return a matrix with start and end.
`str_extract()`	Extracts text corresponding to the first match.
`str_match()`	Extracts capture groups formed by `()` from the first match.
`str_split()`	Splits string into pieces and returns a list of character vectors.
`str_replace()`	Replaces the first matched pattern and returns a character vector.

Each have leading arguments `string` and `pattern`; all functions are vectorised over arguments `string` and `pattern`.

Regexs

Simple cases

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

```
library(tidyverse)
```

```
twister <- "thirty-three thieves thought they thrilled the throne Thursday"
```

How many occurrences of `t` exist?

```
str_count(string = twister, pattern = "t")
```

```
#> [1] 10
```

How many of `t`, `th`, and `the` exist?

```
str_count(twister, c("t", "th", "the"))
```

```
#> [1] 10  8  2
```

Do these patterns exist?

```
str_detect(twister, c("t", "th", "the"
```

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

Separate our long string at each space.

```
twister_split <- str_split(twister, " ") %>% unlist()
twister_split
```

```
#> [1] "thirty-three" "thieves"      "thought"      "they"         "thrilled"
#> [6] "the"          "throne"       "Thursday"
```

Do these patterns exist?

```
str_detect(twister_split, c("tho", "the"))
```

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

Replace certain occurrences.

```
str_replace(twister_split, c("tho", "the"), replacement = c("bro", "Wil"))
```

```
#> [1] "thirty-three" "thieves"      "brought"      "Wily"         "thrilled"
#> [6] "Wil"          "throne"       "Thursday"
```

A step up in complexity

A `.` matches any character, except a new line. It is one of a few metacharacters - special meaning and function.

```
twister <- "thirty-three thieves thought they thrilled the throne Thursday"
```

Does this pattern, `.y.` exist?

```
str_detect(twister, ".y.")
```

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

How many instances?

```
str_count(twister, ".y.")
```

```
#> [1] 2
```

View in Viewer pane.

```
str_view_all(twister, ".y.")
```

```
thirty-three thieves thought they thrilled the throne Thursday
```

How do we match an actual .?

You need to use an escape character to tell the regex you want exact matching.

Regexs use a \ as an escape character. So why doesn't this work?

```
str_view_all("show.me.the.dots...", "\.")
```

```
#> Error: '\.' is an unrecognized escape in character string starting ""\."
```

R escape characters

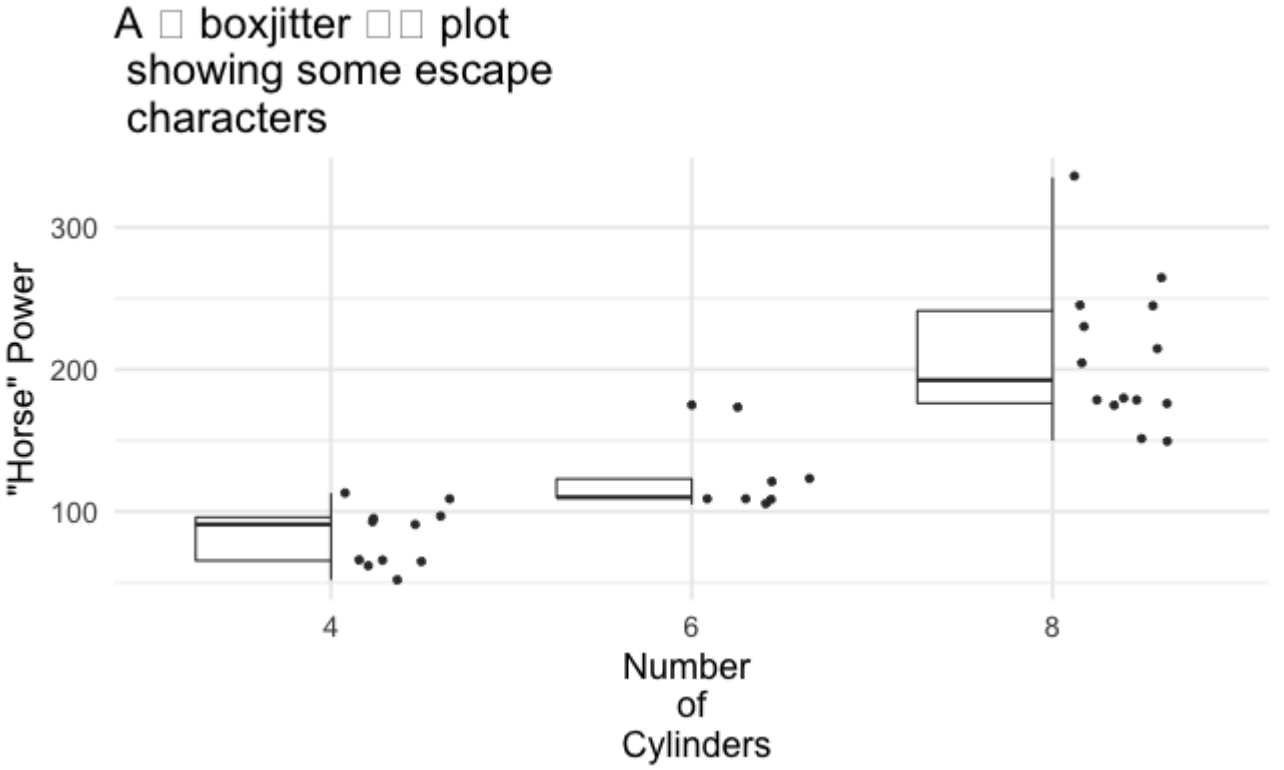
There are some special characters in R that cannot be directly coded in a string. An escape character is a character which results in an alternative interpretation of the following character(s). These vary from language to language, but **for most string implementations \ is the escape character which is modified by a single subsequent character.**

Some common examples:

Literal	Character
`\'`	single quote
`\"`	double quote
`\\`	backslash
`\n`	new line
`\r`	carriage return
`\t`	tab
`\b`	backspace
`\f`	form feed

Examples

```
mtcars %>%
  ggplot(aes(x = factor(cyl), y = hp)) + ggpol::geom_boxjitter() +
  labs(x = "Number \n of \n Cylinders", y = "\"Horse\" Power",
       title = "A \t boxjitter \t\t plot \n showing some escape \n characters") +
  theme_minimal(base_size = 18)
```



Examples

```
print("hello\world")
```

```
#> Error: '\w' is an unrecognized escape in character string starting ""hello\w"
```

```
cat("hello\world")
```

```
#> Error: '\w' is an unrecognized escape in character string starting ""hello\w"
```

```
print("hello\tworld")
```

```
#> [1] "hello\tworld"
```

```
cat("hello\tworld")
```

```
#> hello    world
```

A quote

```
print("hello\"world")
```

```
#> [1] "hello\"world"
```

```
cat("hello\"world")
```

```
#> hello"world
```

A backslash

```
print("hello\\world")
```

```
#> [1] "hello\\world"
```

```
cat("hello\\world")
```

```
#> hello\world
```

A new line

```
print("hello\nworld")
```

```
#> [1] "hello\nworld"
```

```
cat("hello\nworld")
```

```
#> hello
#> world
```

Returning to: how do we match a .?

We need to escape the backslash in our regex of \.

```
str_view_all("show.me.the.dots...", "\\.")
```

show.me.the.dots...

Regex metacharacters

```
. ^ $ * + ? { } [ ] \ | ( )
```

Allow for more advanced forms of pattern matching.

As we saw with `.`, these cannot be matched directly. Thus, if you want to match the literal `?` you will need to use `\\?`.

What do you need to match a literal `\` in regex pattern matching?

```
str_view_all("find the \\ in this string", "\\\\")
```

```
find the \ in this string
```

Regex anchors

Sometimes we want to specify that our pattern occurs at a particular location in a string, we indicate this using anchor metacharacters.

Regex	Anchor
`^` or `\A`	Start of string
`$` or `\Z`	End of string

Example: anchors

```
text <- c("Which?", "Witch", "Will", "SWitch?")
```

```
str_replace(text, "W...", "****")
```

```
#> [1] "****h?"  "****h"   "****"    "S****h?"
```

```
str_replace(text, "^W...", "****")
```

```
#> [1] "****h?"  "****h"   "****"    "SWitch?"
```

```
str_replace(text, "W...h", "****")
```

```
#> [1] "****?"  "****"   "Will"   "S****?"
```

```
str_replace(text, "W...h$", "****")
```

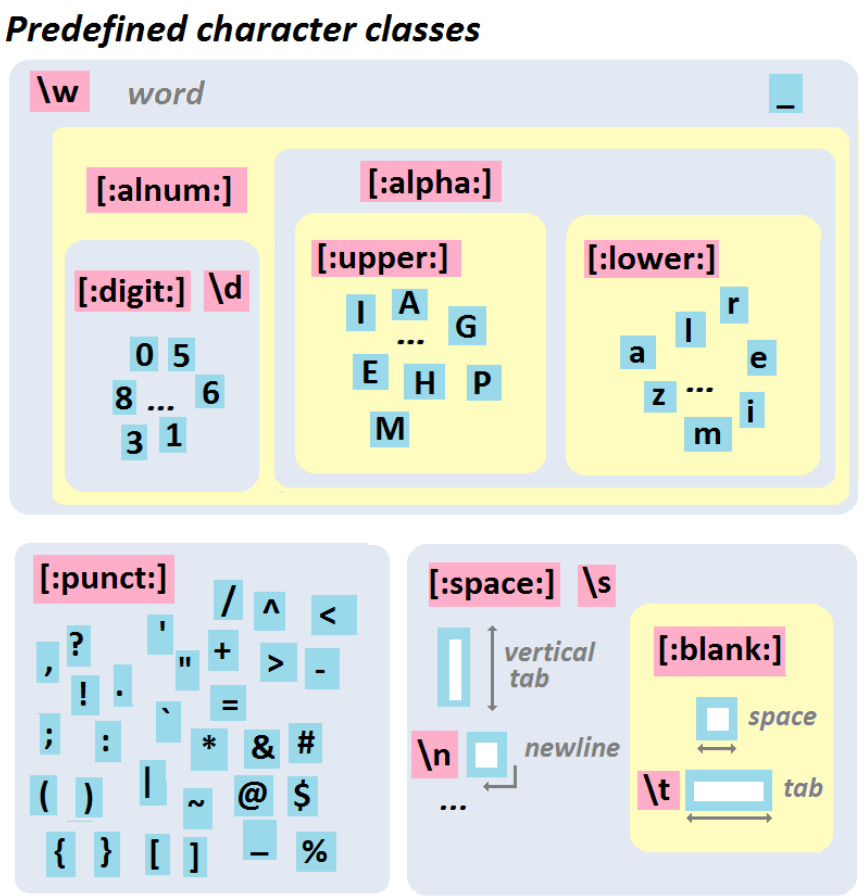
```
#> [1] "Which?"  "****"    "Will"    "SWitch?"
```

Character classes

Special patterns exist to match more than one class.

Meta Character	Class	Description
.		Any character except new line (\n)
\s	[:space:]	White space (space, tab, newline)
\S		Not white space
\d	[:digit:]	Digit (0-9)
\D		Not digit
\w		Word (A-Z, a-z, 0-9, or _)
\W		Not word

Character class overview



Ranges

We can also specify our own classes using the square bracket metacharacter.

Class	Type
`[abc]`	Class (a or b or c)
`[^abc]`	Negated class not (a or b or c)
`[a-c]`	Range lower case letter from a to c
`[A-C]`	Range upper case letter from A to C
`[0-7]`	Digit between 0 to 7

Exercises

Write a regular expression to match a

1. social security number of the form ###-##-####,
2. phone number of the form (###) ###-####,
3. license plate of the form AAA ####.

Test your regexs on some examples with `str_detect()` or `str_view()`.

Repetition with quantifiers

Attached to literals or character classes, these allow a match to repeat some number of times.

Quantifier	Description
`*`	Match 0 or more
`+`	Match 1 or more
`?`	Match 0 or 1
`{3}`	Match Exactly 3
`{3,}`	Match 3 or more
`{3,5}`	Match 3, 4 or 5

Examples: quantifiers

```
text <- c("My", "cell: ", "(610)-867-5309")
```

```
str_detect(text, "\\(\\d{3}\\)-\\d{3}-\\d{4}")
```

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

```
str_extract(text, "\\(\\d{3}\\)-\\d{3}-\\d{4}")
```

```
#> [1] NA               NA               "(610)-867-5309"
```

```
text <- "2 too two 4 for four 8 ate eight"
```

```
str_extract(text, "\\d.*\\d")
```

```
#> [1] "2 too two 4 for four 8"
```

Greedy matches

By default matches are greedy. This is why we get

```
#> [1] "2 too two 4 for four 8"
```

instead of

```
#> [1] "2 too two 4"
```

when we run code

```
str_extract(text, "\\d.*\\d")
```

To make matching lazy, include `?` after so you return the shortest substring possible.

```
str_extract(text, "\\d.*?\\d")
```

```
#> [1] "2 too two 4"
```

What will this result be?

```
str_extract_all(c("fruit flies", "fly faster"), "[aeiou]{1,2}[a-z]+")
```

Groups

Groups allow you to connect pieces of a regular expression for modification or capture.

```
str_extract(c("grey", "gray", "gravitas", "great"), "gre|ay")
```

```
#> [1] "gre" "ay"  NA    "gre"
```

```
str_extract(c("grey", "gray", "gravitas", "great"), "grey|gray")
```

```
#> [1] "grey" "gray" NA     NA
```

```
str_extract(c("grey", "gray", "gravitas", "great"), "gr(e|a)y")
```

```
#> [1] "grey" "gray" NA     NA
```

Their use can improve readability and allow for backreferencing.

Capture groups

Suppose we have the following files, where want to capture their name and not the file type.

```
files <- c("dog.png", "cat44.png", "file_0292.png", "notes-v2.png")
```

```
str_match(files, "(\\w+[[:punct:]]?\\w+)\\.png")
```

```
#>      [,1]             [,2]
#> [1,] "dog.png"        "dog"
#> [2,] "cat44.png"      "cat44"
#> [3,] "file_0292.png" "file_0292"
#> [4,] "notes-v2.png"   "notes-v2"
```

Without the parentheses we get

```
str_match(files, "\\w+[[:punct:]]?\\w+\\.png")
```

```
#>      [,1]
#> [1,] "dog.png"
#> [2,] "cat44.png"
#> [3,] "file_0292.png"
#> [4,] "notes-v2.png"
```

Backreferences

Backreferencing allows us to reference groups with `\1`, `\2`, etc.

```
text <- "Some numbers include 00, 11, 3434, 41, 1010, 23, and 1"
```

```
str_match_all(text, "(\\d)\\1")
```

```
#> [[1]]
#>      [,1] [,2]
#> [1,] "00" "0"
#> [2,] "11" "1"
```

```
str_match_all(text, "(\\d{2})\\1")
```

```
#> [[1]]
#>      [,1]   [,2]
#> [1,] "3434" "34"
#> [2,] "1010" "10"
```

Exercises

```
text <- c(
  "apple",
  "219 733 8965",
  "329-293-8753",
  "Work: (579) 499-7527; Home: (543) 355 3679"
)
```

1. Write a regular expression that will extract all phone numbers contained in the vector above.

2. Once that works, use groups to extracts the area code separately from the rest of the phone number.

References

1. Grolemund, G., & Wickham, H. (2020). R for Data Science. https://r4ds.had.co.nz/

2. Regular expressions. (2020). Stringr.tidyverse.org. https://stringr.tidyverse.org/articles/regular-expressions.html

3. Regular-Expression.info