

String manipulation

Reproducible data munging



Roland Krause | rworkshop | 2021-09-08

Session set-up

Learning objectives

- Perform pattern matching and string manipulation
- Print text nicely and well-formatted



Regular expressions

- Matching and substituting of strings
- `"^lecture([0-9]{1,2}).*[^_].Rmd$/\1.Rmd/g"`
- See [R for data science](#)



stringr package

- Simplifies and unifies string operations in base R
- Detection, extraction, counting, subsetting
- Gentle [stringr introduction](#)
- Different matching engines, e.g. locale-sensitive



glue package

- Join and output complex strings
- Concise [glue introduction](#)



String examples in Base R

Strings are character objects

```
# A character object = colloquially called "string"
my_string <- "cat"

my_string
```

```
[1] "cat"
```

```
my_other_string <- 'catastrophe' # single quotes
not_so_numeric <- as.character(3.1415)

not_so_numeric
```

```
[1] "3.1415"
```

```
# A character vector
my_string_vec <- c("atg", "ttg", "tga")
```

Printing complex objects

C style with placeholder

```
sprintf("Hello %s, how is day %d of this course?",
        "John Doe", 12)
```

```
[1] "Hello John Doe, how is day 12 of this course?"
```

Comparisons of Base R and stringr

```
pattern <- "r"  
my_words <- c("cat", "cart", "carrot", "catastrophe",  
             "dog", "rat", "bet")
```

Base R

```
grep(pattern, my_words)
```

```
[1] 2 3 4 6
```

```
grep(pattern, my_words, value = TRUE)
```

```
[1] "cart"      "carrot"    "catastrophe" "rat"
```

```
substr(my_words, 1, 3)
```

```
[1] "cat" "car" "car" "cat" "dog" "rat" "bet"
```

```
gsub(pattern, "R", my_words)
```

```
[1] "cat"      "caRt"      "caRRot"    "catastRophe" "do  
[6] "Rat"      "bet"
```

stringr

```
library(stringr)
```

```
str_which(my_words, pattern)
```

```
[1] 2 3 4 6
```

```
str_subset(my_words, pattern)
```

```
[1] "cart"      "carrot"    "catastrophe" "rat"
```

```
str_sub(my_words, 1, 2)
```

```
[1] "ca" "ca" "ca" "ca" "do" "ra" "be"
```

```
str_replace(my_words, pattern, "R")
```

```
[1] "cat"      "caRt"      "caRrot"    "catastRophe" "do  
[6] "Rat"      "bet"
```

Why use stringr?

Motivation

- Consistency
- Less typing and looking up things



Usage

- All functions in stringr start with `str_`
- All take a vector of strings as the first argument
 - ("data first")
 - `>%>` now works
- All functions properly vectorised

Useful additions

Viewing matches rendered in HTML

```
str_view_all(my_words, pattern)
```

cat

car^t

car^rrot

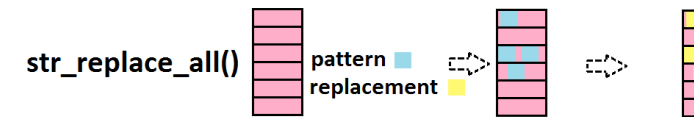
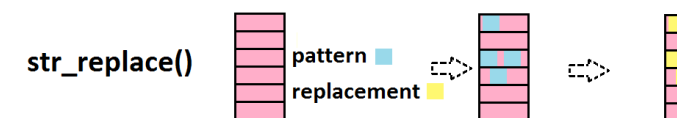
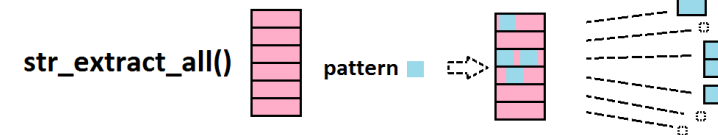
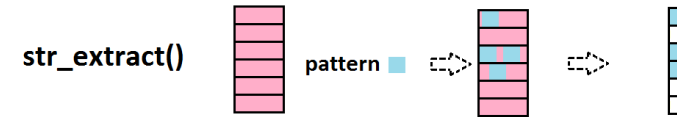
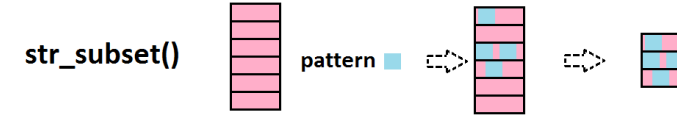
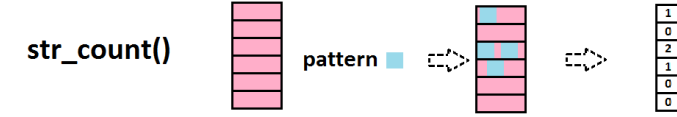
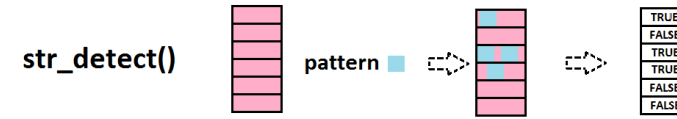
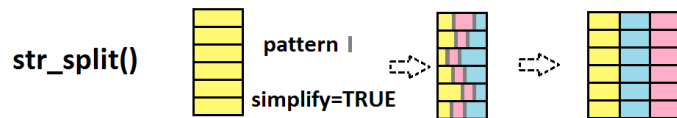
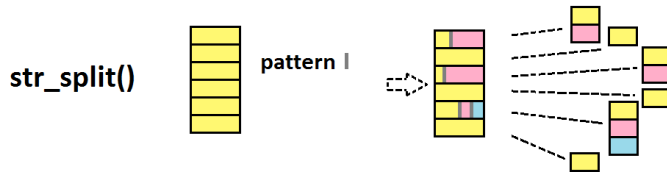
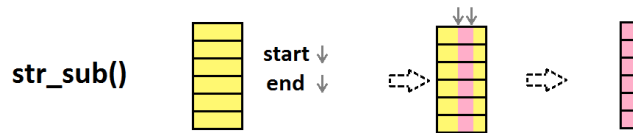
catast^rrophe

dog

^rat

bet

stringr overview



String manipulation with stringr : : CHEAT SHEET



The **stringr** package provides a set of internally consistent tools for working with character strings, i.e. sequences of characters surrounded by quotation marks.

Detect Matches



str_detect(string, **pattern**) Detect the presence of a pattern match in a string.
`str_detect(fruit, "a")`



str_which(string, **pattern**) Find the indexes of strings that contain a pattern match.
`str_which(fruit, "a")`



str_count(string, **pattern**) Count the number of matches in a string.
`str_count(fruit, "a")`



str_locate(string, **pattern**) Locate the positions of pattern matches in a string. Also **str_locate_all**.
`str_locate(fruit, "a")`

Subset Strings



str_sub(string, start = 1L, end = -1L) Extract substrings from a character vector.
`str_sub(fruit, 1, 3); str_sub(fruit, -2)`



str_subset(string, **pattern**) Return only the strings that contain a pattern match.
`str_subset(fruit, "b")`



str_extract(string, **pattern**) Return the first pattern match found in each string, as a vector. Also **str_extract_all** to return every pattern match.
`str_extract(fruit, "[aeiou]")`



str_match(string, **pattern**) Return the first pattern match found in each string, as a matrix with a column for each () group in pattern. Also **str_match_all**.
`str_match(sentences, "(a[the] ([^]+))")`

Manage Lengths



str_length(string) The width of strings (i.e. number of code points, which generally equals the number of characters). `str_length(fruit)`



str_pad(string, width, side = c("left", "right", "both"), pad = " ") Pad strings to constant width. `str_pad(fruit, 17)`



str_trunc(string, width, side = c("right", "left", "center"), ellipsis = "...") Truncate the width of strings, replacing content with ellipsis.
`str_trunc(fruit, 3)`



str_trim(string, side = c("both", "left", "right")) Trim whitespace from the start and/or end of a string. `str_trim(fruit)`

Mutate Strings



str_sub() <- value. Replace substrings by identifying the substrings with `str_sub()` and assigning into the results.
`str_sub(fruit, 1, 3) <- "str"`



str_replace(string, **pattern**, replacement) Replace the first matched pattern in each string. `str_replace(fruit, "a", ".")`



str_replace_all(string, **pattern**, replacement) Replace all matched patterns in each string. `str_replace_all(fruit, "a", ".")`



str_to_lower(string, locale = "en")¹ Convert strings to lower case.
`str_to_lower(sentences)`



str_to_upper(string, locale = "en")¹ Convert strings to upper case.
`str_to_upper(sentences)`



str_to_title(string, locale = "en")¹ Convert strings to title case. `str_to_title(sentences)`

Join and Split



str_c(..., sep = "", collapse = NULL) Join multiple strings into a single string.
`str_c(letters, LETTERS)`



str_c(..., sep = "", collapse = "") Collapse a vector of strings into a single string.
`str_c(letters, collapse = "")`



str_dup(string, times) Repeat strings times times. `str_dup(fruit, times = 2)`



str_split_fixed(string, **pattern**, n) Split a vector of strings into a matrix of substrings (splitting at occurrences of a pattern match). Also **str_split** to return a list of substrings.
`str_split_fixed(fruit, "", n=2)`



str_glue(..., .sep = "", .envir = parent.frame()) Create a string from strings and {expressions} to evaluate. `str_glue("Pi is {pi}")`



str_glue_data(.x, ..., .sep = "", .envir = parent.frame(), .na = "NA") Use a data frame, list, or environment to create a string from strings and {expressions} to evaluate.
`str_glue_data(mtcars, "{rownames(mtcars)} has {hp} hp")`

Order Strings



str_order(x, decreasing = FALSE, na_last = TRUE, locale = "en", numeric = FALSE, ...) Return the vector of indexes that sorts a character vector. `x[str_order(x)]`



str_sort(x, decreasing = FALSE, na_last = TRUE, locale = "en", numeric = FALSE, ...) Sort a character vector.
`str_sort(x)`

Helpers

apple
banana
pear

str_conv(string, encoding) Override the encoding of a string. `str_conv(fruit, "ISO-8859-1")`

apple
banana
pear

str_view(string, **pattern**, match = NA) View HTML rendering of first regex match in each string. `str_view(fruit, "[aeiou]")`

str_view_all(string, **pattern**, match = NA) View HTML rendering of all regex matches.
`str_view_all(fruit, "[aeiou]")`

str_wrap(string, width = 80, indent = 0, exdent = 0) Wrap strings into nicely formatted paragraphs. `str_wrap(sentences, 20)`



¹ See bit.ly/ISO639-1 for a complete list of locales.

Need to Know

Pattern arguments in `stringr` are interpreted as regular expressions *after any special characters have been parsed*.

In R, you write regular expressions as *strings*, sequences of characters surrounded by quotes ("" or '') or single quotes ('').

Some characters cannot be represented directly in an R string. These must be represented as **special characters**, sequences of characters that have a specific meaning, e.g.

Special Character	Represents
<code>\\</code>	<code>\</code>
<code>\"</code>	"
<code>\n</code>	new line

Run `?""` to see a complete list

Because of this, whenever a `\` appears in a regular expression, you must write it as `\\` in the string that represents the regular expression.

Use `writeLines()` to see how R views your string after all special characters have been parsed.

```
writeLines("\\.")
# \.
```

```
writeLines("\\ is a backslash")
# \ is a backslash
```

INTERPRETATION

Patterns in `stringr` are interpreted as regexes. To change this default, wrap the pattern in one of:

regex(pattern, ignore_case = FALSE, multiline = FALSE, comments = FALSE, dotall = FALSE, ...) Modifies a regex to ignore cases, match end of lines as well as end of strings, allow R comments within regex's, and/or to have `.` match everything including `\n`. `str_detect("I", regex("i", TRUE))`

fixed() Matches raw bytes but will miss some characters that can be represented in multiple ways (fast). `str_detect("u0130", fixed("ı"))`

coll() Matches raw bytes and will use locale specific collation rules to recognize characters that can be represented in multiple ways (slow). `str_detect("u0130", coll("ı", TRUE, locale = "tr"))`

boundary() Matches boundaries between characters, line_breaks, sentences, or words. `str_split(sentences, boundary("word"))`

Regular Expressions - Regular expressions, or *regexps*, are a concise language for describing patterns in strings.

MATCH CHARACTERS

string (type this)	regex (to mean this)	matches (which matches this)	example
<code>\\.</code>	<code>a</code> (etc.)	<code>a</code> (etc.)	<code>see("a")</code> <code>abc ABC 123</code> <code>.!?(\\.)</code>
<code>\\.</code>	<code>.</code>	<code>.</code>	<code>see("\\.")</code> <code>abc ABC 123</code> <code>.!?(\\.)</code>
<code>\\!</code>	<code>!</code>	<code>!</code>	<code>see("\\!")</code> <code>abc ABC 123</code> <code>.!?(\\!)</code>
<code>\\?</code>	<code>?</code>	<code>?</code>	<code>see("\\?")</code> <code>abc ABC 123</code> <code>.!?(\\?)</code>
<code>\\\\\\</code>	<code>\\</code>	<code>\\</code>	<code>see("\\\\\\\\")</code> <code>abc ABC 123</code> <code>.!?(\\\\\\)</code>
<code>\\(</code>	<code>(</code>	<code>(</code>	<code>see("\\(")</code> <code>abc ABC 123</code> <code>.!?(\\(\\))</code>
<code>\\)</code>	<code>)</code>	<code>)</code>	<code>see("\\)")</code> <code>abc ABC 123</code> <code>.!?(\\))</code>
<code>\\{</code>	<code>{</code>	<code>{</code>	<code>see("\\{")</code> <code>abc ABC 123</code> <code>.!?(\\{\\})</code>
<code>\\}</code>	<code>}</code>	<code>}</code>	<code>see("\\}")</code> <code>abc ABC 123</code> <code>.!?(\\})</code>
<code>\\n</code>	<code>\n</code>	new line (return)	<code>see("\\n")</code> <code>abc ABC 123</code> <code>.!?(\\n)</code>
<code>\\t</code>	<code>\t</code>	tab	<code>see("\\t")</code> <code>abc ABC 123</code> <code>.!?(\\t)</code>
<code>\\s</code>	<code>\s</code>	any whitespace (S for non-whitespaces)	<code>see("\\s")</code> <code>abc ABC 123</code> <code>.!?(\\s)</code>
<code>\\d</code>	<code>\d</code>	any digit (D for non-digits)	<code>see("\\d")</code> <code>abc ABC 123</code> <code>.!?(\\d)</code>
<code>\\w</code>	<code>\w</code>	any word character (W for non-word chars)	<code>see("\\w")</code> <code>abc ABC 123</code> <code>.!?(\\w)</code>
<code>\\b</code>	<code>\b</code>	word boundaries	<code>see("\\b")</code> <code>abc ABC 123</code> <code>.!?(\\b)</code>
<code>[[:digit:]]</code>		digits	<code>see("[[:digit:]]")</code> <code>abc ABC 123</code> <code>.!?(\\d)</code>
<code>[[:alpha:]]</code>		letters	<code>see("[[:alpha:]]")</code> <code>abc ABC 123</code> <code>.!?(\\w)</code>
<code>[[:lower:]]</code>		lowercase letters	<code>see("[[:lower:]]")</code> <code>abc ABC 123</code> <code>.!?(\\w)</code>
<code>[[:upper:]]</code>		uppercase letters	<code>see("[[:upper:]]")</code> <code>abc ABC 123</code> <code>.!?(\\w)</code>
<code>[[:alnum:]]</code>		letters and numbers	<code>see("[[:alnum:]]")</code> <code>abc ABC 123</code> <code>.!?(\\w)</code>
<code>[[:punct:]]</code>		punctuation	<code>see("[[:punct:]]")</code> <code>abc ABC 123</code> <code>.!?(\\w)</code>
<code>[[:graph:]]</code>		letters, numbers, and punctuation	<code>see("[[:graph:]]")</code> <code>abc ABC 123</code> <code>.!?(\\w)</code>
<code>[[:space:]]</code>		space characters (i.e. <code>\s</code>)	<code>see("[[:space:]]")</code> <code>abc ABC 123</code> <code>.!?(\\w)</code>
<code>[[:blank:]]</code>		space and tab (but not new line)	<code>see("[[:blank:]]")</code> <code>abc ABC 123</code> <code>.!?(\\w)</code>
<code>.</code>		every character except a new line	<code>see(".")</code> <code>abc ABC 123</code> <code>.!?(\\w)</code>

¹ Many base R functions require classes to be wrapped in a second set of `[]`, e.g. `[[:digit:]]`

ALTERNATES

regex	matches	example
<code>ab d</code>	or	<code>alt("ab d")</code> <code>abcde</code>
<code>[abe]</code>	one of	<code>alt("[abe]")</code> <code>abcde</code>
<code>^abe</code>	anything but	<code>alt("^abe]")</code> <code>abcde</code>
<code>[a-c]</code>	range	<code>alt("[a-c]")</code> <code>abcde</code>

ANCHORS

regex	matches	example
<code>^</code>	start of string	<code>anchor("^a")</code> <code>aaa</code>
<code>\$</code>	end of string	<code>anchor("a\$")</code> <code>aaa</code>

LOOK AROUNDS

regex	matches	example
<code>(?=.)</code>	followed by	<code>look("a(?=c)")</code> <code>bacád</code>
<code>(?!.)</code>	not followed by	<code>look("a(?!c)")</code> <code>bacád</code>
<code>(?<=.)</code>	preceded by	<code>look("(?<=b)a")</code> <code>bacád</code>
<code>(?<!=.)</code>	not preceded by	<code>look("(?<!=b)a")</code> <code>bacád</code>

QUANTIFIERS

regex	matches	example
<code>a?</code>	zero or one	<code>quant("a?")</code> <code>.a.aa.aaa</code>
<code>a*</code>	zero or more	<code>quant("a*")</code> <code>.a.aa.aaa</code>
<code>a+</code>	one or more	<code>quant("a+")</code> <code>.a.aa.aaa</code>
<code>a{n}</code>	exactly n	<code>quant("a{2}")</code> <code>.a.aa.aaa</code>
<code>a{n,}</code>	n or more	<code>quant("a{2,}")</code> <code>.a.aa.aaa</code>
<code>a{n,m}</code>	between n and m	<code>quant("a{2,4}")</code> <code>.a.aa.aaa</code>

GROUPS

Use parentheses to set precedent (order of evaluation) and create groups

regex	matches	example
<code>(ab c)d</code>	sets precedence	<code>alt("(ab c)d")</code> <code>abcde</code>
string	regex (type this)	matches (which matches this)
<code>\\1</code>	<code>\\1</code> (etc.)	first () group, etc.
		example (the result is the same as <code>ref("abba")</code>)
		<code>ref("(a)(b)\\2\\1")</code> <code>abbaab</code>

Use an escaped number to refer to and duplicate parentheses groups that occur earlier in a pattern. Refer to each group by its order of appearance



`[[:space:]]`
new line

`[[:blank:]]`
space
tab

`[[:graph:]]`

`[[:punct:]]`

`.`
`,`
`:`
`;`
`?`
`!`
`|`
`|`
`/`
`/`
`=`
`=`
`+`
`+`
`-`
`-`
`^`
`^`

`[[:alnum:]]`

`[[:digit:]]`

0 1 2 3 4 5 6 7 8 9

`[[:alpha:]]`

`[[:lower:]]` `[[:upper:]]`

a b c d e f A B C D E F
g h i j k l G H I J K L
m n o p q r M N O P Q R
s t u v w x S T U V W X
z Z

Length

Length of items in character vector

```
str_length(my_words)
```

```
[1] 3 4 6 11 3 3 3
```

Warning

```
length(my_words)
```

```
[1] 7
```

Elements of strings

Substrings

```
my_words
```

```
[1] "cat"      "cart"      "carrot"     "catastrophe" "do  
[6] "rat"      "bet"
```

```
str_sub(my_words, 1, 4)
```

```
[1] "cat" "cart" "carr" "cata" "dog" "rat" "bet"
```

Replace ⇄

```
str_replace(my_words, "a", "A")
```

```
[1] "cAt"      "cArt"      "cArrot"     "cAtastrophe" "do  
[6] "rAt"      "bet"
```

Splitting strings

```
str_split(my_words, "a")
```

```
[[1]]  
[1] "c" "t"  
  
[[2]]  
[1] "c" "rt"  
  
[[3]]  
[1] "c" "rrot"  
  
[[4]]  
[1] "c" "t" "strophe"  
  
[[5]]  
[1] "dog"  
  
[[6]]  
[1] "r" "t"  
  
[[7]]  
[1] "bet"
```

Matching strings

Detect matching strings

```
str_detect(my_words, "o")
```

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

Retrieving (only) matching strings

```
str_subset(my_words, "r")
```

```
[1] "cart" "carrot" "catastrophe" "rat"
```

Useful interactively primarily. Dangerous in programming!

Retrieving matching strings

```
str_match(my_words, "a")
```

```
      [,1]  
[1,] "a"  
[2,] "a"  
[3,] "a"  
[4,] "a"  
[5,] NA  
[6,] "a"  
[7,] NA
```

Includes capture groups (see regular expressions)

Extracting matches

```
str_extract(my_words, "a")
```

```
[1] "a" "a" "a" "a" NA "a" NA
```

Better treatment of conversion

```
my_col <- c("F", "M", "female", "male", "male", "female", "female", "männlich")

convert_gender <- function(x){
  case_when(
    str_detect(x, "^[Ff]") ~ "Female",
    str_detect(x, "^[Mm]") ~ "Male",
    TRUE ~ x
  )
}

convert_gender(my_col)
```

```
[1] "Female" "Male"   "Female" "Male"   "Male"   "Female" "Female" "Male"
```

Regular expressions

Getting started with regular expressions

Higher aims

- Extract particular characters, e.g. numbers only
- Express a variety of character following or preceding patterns
- Matching any character
- Not matching a particular character

Prerequisites complex matching

- Regular expressions look like strings but are converted to a particular expression object.
- Can be done explicitly by `regex()` -- rarely necessary
- `print()` is giving the quoted strings and therefore misleading
- Use `cat()` or `writeLines()` to see strings properly escaped.
- `writeLines()` preferred for writing

Flexible matching though *metacharacters*

Metacharacters

Symbols matching a variety of characters as opposed to literal matches.

- . (dot) represents any character
- Exception is the newline character (`\n`)

```
str_view(my_words, ".at")
```

cat

cart

carrot

catastrophe

dog

rat

bet

- . matches exactly one occurrence

```
str_subset(my_words, "c.t")
```

```
[1] "cat"          "catastrophe"
```

- + (plus) represents one or more occurrences

```
str_subset(my_words, 'c.r+')
```

```
[1] "cart"      "carrot"
```

- * (star) represents zero or more occurrences

```
str_subset(my_words, 'c.r*')
```

```
[1] "cat"          "cart"          "carrot"        "catastroph"
```

Grouping

Group terms with parentheses (and)

```
str_view(my_words, 'c(.r)+t')
```

cat

cart

carrot

catastrophe

dog

rat

bet

Capture groups with `str_match()`

```
str_match(my_words, 'c(.r)*t')
```


Alternation operator | (logical OR)

```
str_subset(my_words, '(c.t)|(c.rt)')
```

```
[1] "cat"      "cart"      "catastrophe"
```

Quantifying a number of matches

The preceding item will be matched ...

- `?` at most once.
- `*` matched zero or more times.
- `+` one or more times.
- `{n}` exactly 'n' times.
- `{n,}` 'n' or more times.
- `{n,m}` at least 'n' times, but not more than 'm' times.

```
dna <- "ATGGTAACCGGTAGGTAGTAAAGGTCCC"  
str_view_all(dna, "AA?")
```

A TGGT AA CCGGT A GGT A GT AA A GGTCCC

```
str_view_all(dna, "AA+")
```

ATGGT AA CCGGTAGGTAGT AAA GGTCCC

```
str_view_all(dna, "A{3,}")
```

ATGGTAACCGGTAGGTAGT AAA GGTCCC

Greedy and lazy matching

Matches are *greedy* by default

Match the longest possible subsequence.

```
dna <- "ATGGTAACCGGTAGGTAGTAAAGGTCCC"  
str_extract(dna, "AAG.{2,5}")
```

```
[1] "AAGGTCCC"
```

```
str_extract(dna, "ATG.+C")
```

```
[1] "ATGGTAACCGGTAGGTAGTAAAGGTCCC"
```

Lazy matching

Adding `?` to a regular expression makes it *lazy*, and returns the shortest possible match.

```
str_extract(dna, "AAG.{2,5}?")
```

```
[1] "AAGGT"
```

```
str_extract(dna, "ATG.+?C")
```

```
[1] "ATGGTAAC"
```

Anchors

^ Start of string

```
my_words
```

```
[1] "cat"      "cart"      "carrot"    "catastrophe" "do  
[6] "rat"      "bet"
```

```
str_subset(my_words , '^c')
```

```
[1] "cat"      "cart"      "carrot"    "catastrophe"
```

\$ End of string

```
str_subset(my_words, 'r.$')
```

```
[1] "cart"
```

Character classes

Special characters

Pattern	Matches	Complement	Matches
<code>\d</code>	Digit	<code>\D</code>	No digit
<code>\s</code>	Whitespace	<code>\S</code>	No whitespace
<code>\w</code>	Word chars	<code>\W</code>	No work char
<code>\b</code>	Boundaries	<code>\B</code>	Within words
<code>\p{}</code>	Property	<code>\P{}</code>	Not that property

Example for Unicode properties

Code	Description
Ll	Lowercase letter
Lu	Uppercase letter
Sc	A currency sign
Sm	Symbol of mathematical use
...	See documentation

Powerful but complex to use.

Examples

Matching digits

```
str_extract(my_words, "\\d+")
```

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

Counting word elements

```
str_count(my_words, "\\w+")
```

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

```
str_length(my_words)
```

```
[1] 3 4 6 11 3 3 3
```

```
str_count(my_words, "\\w")
```

```
[1] 3 4 6 11 3 3 3
```

Counting all matches to words.

Matching "everything that you want"

```
str_extract(my_words, "\\S+")
```

```
[1] "cat"      "cart"      "carrot"     "catastrophe" "dc"  
[6] "rat"      "bet"
```

Unicode

```
str_extract(my_words, "\\p{Lu}+")
```

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

Extended list of regular expressions

Readable short cuts

Built-in (`stringi` - `stringr`)

Requires `perl = TRUE` flag in base R.

Works out of the box in `stringr`.

- `[upper:]` Upper-case letters.
- `[lower:]` Lower-case letters.
- `[alpha:]` Alphabetic characters: `[:lower:]` and `[:upper:]`.
- `[digit:]` Digits: '0 1 2 3 4 5 6 7 8 9'.
- `[punct:]` Punctuation characters: ' ! " # \$ % & ' () * + , - . / : ; < = > ? @ ' and others.
- `[space:]` Space characters: tab, newline, vertical tab, form feed, carriage return, and space.
- `[blank:]` Blank characters: space and tab.
- `[alnum:]` Alphanumeric characters: `[:alpha:]` and `[:digit:]`.
- `[graph:]` Graphical characters: `[:alnum:]` and `[:punct:]`.

```
my_words
```

```
[1] "cat"      "cart"      "carrot"     "catastrophe" "dc"  
[6] "rat"      "bet"
```

```
str_subset(my_words, "[:punct:]")
```

```
character(0)
```

```
str_extract(my_words, "[:punct:]")
```

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

Roll your own character class

Define groups


- **[a-z]** lowercase letters
- **[a-zA-Z]** any (ascii) letter
- **[0-9]** any number
- **[aeiou]** any vowel
- **[0-7ivx]** any of 0 to 7, i, v, and x

Example

```
str_subset(c("Rpl12", "Rpn12",  
            "Rps1",  
            "Pre1"), 'Rp[1s]')
```

```
[1] "Rpl12" "Rps1"
```

Rely on built-in groups where possible

Note that the set of alphabetic characters includes accents such as ß, ç or ö which are very common in some languages. Use is more general than **[A-Za-z]** which ascii characters only. 

Matching metacharacters

We saw special characters such as

- `.`
- `+`
- `*`
- or `$`

What if we want to match them?

Strings containing only a full stop

```
vec2 <- c("YKL045W-A", "12+45=57", "$1200.00", "ID2.2")  
str_subset(vec2, '.')
```

```
[1] "YKL045W-A" "12+45=57" "$1200.00" "ID2.2"
```

Not what we wanted.

Excursion

Implicit conversion

R wraps regular expressions as strings without explicit interference of the user. When converting from *string* to *regular expression* internally, single backslashes (\) are already converted.



Solution

💡 Need to escape \ with an additional one -> \\.

Double escape

```
str_subset(vec2, '\\.')
```

```
Error: '\\.' is an unrecognized escape in character string start
```

```
str_subset(vec2, '\\\\.')
```

```
[1] "$1200.00" "ID2.2"
```

No escape

To match a `\`, our pattern must represent `\\`.

How to match `c("a\\backslash", "nbackslash", "slash", "\\n")`?

Note the difference when printing meta-characters.

```
slash_vec <- c("a\\backslash", "nbackslash", "slash", "\\n")
print(slash_vec)
```

```
[1] "a\\backslash" "nbackslash"  "slash"       "\\n"
```

```
cat(slash_vec)
```

```
a\\backslash nbackslash slash
```

```
str_subset(slash_vec, '\\')
```

```
Error in stri_subset_regex(string, pattern, omit_na = TRUE, neg
```

Use more backslashes!!!

Our string must contain 4 backslashes!

```
str_subset(slash_vec, '\\\\\\')
```

```
[1] "a\\\\backslash"
```

Search and replace

str_replace()

```
mirna <-  
  c("dme-bantam",  
    "dme-let-7",  
    "dme-mir-1",  
    "dme-mir-2a-1",  
    "mmu-let-7f-2")  
str_replace(mirna, '-', '|')
```

```
[1] "dme|bantam"    "dme|let-7"     "dme|mir-1"     "dme|mir-2a-1"
```

Only the first match is replaced!

str_replace_all()

```
str_replace_all(mirna, '-', '|')
```

```
[1] "dme|bantam"    "dme|let|7"     "dme|mir|1"     "dme|mir|2a|1"
```

Backreferences

Group matches

`\1`, `\2` and so forth refer to groups matched with `()`.

Constructing new strings from regular expression matches

```
uniprot <- c("Q6QU88_CALBL", "C01A2_HUMAN", "SAMH1_HUMAN", "NPRL2_DROME")  
str_replace(uniprot, '(\S+)_\S+', "\2: \1")
```

```
[1] "CALBL: Q6QU88" "HUMAN: C01A2" "HUMAN: SAMH1" "DROME: NPRL2"
```

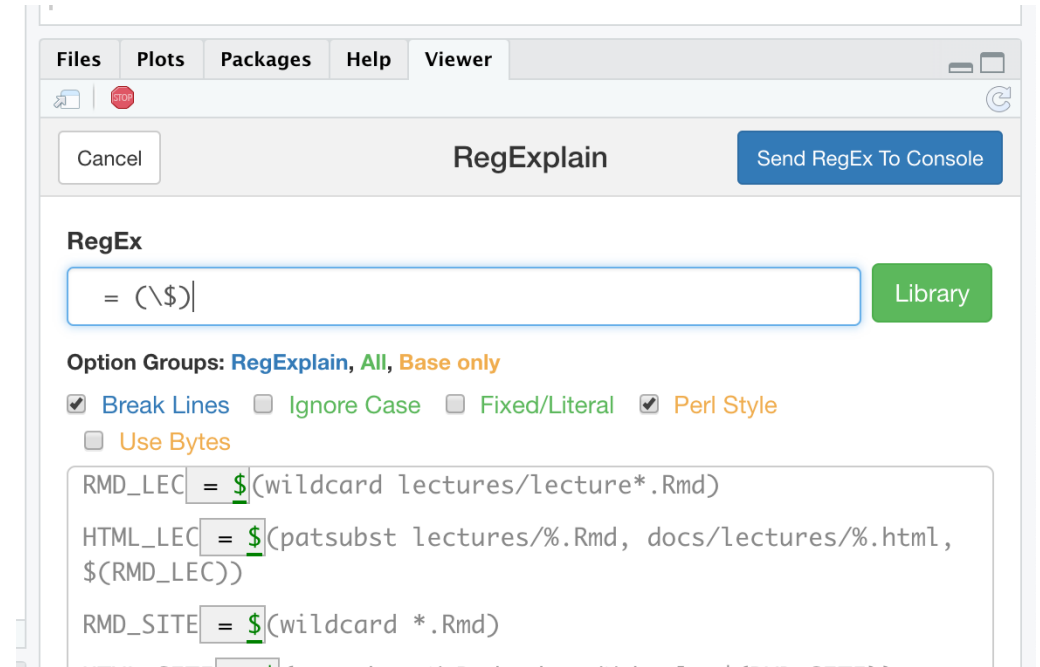
Helpers

regexplain

Simple [addin for RStudio](#) by [Garrick Aden-Buie](#)

- Test regular expressions on the fly
- Reference library
- Cheatsheet
- test it [live](#)

```
devtools::install_github("gadenbuie/regexplain")  
regexplain::regexplain_gadget()
```



Constructing strings with glue and stringr

glue is not load with library(tidyverse)

```
library(glue)
```

Attaching package: 'glue'

The following object is masked from 'package:dplyr':

collapse

Joining strings

Base R and **glue**

Several ways to join strings even in the tidyverse.

This is the **stringr** way, options are used in other packages too, e.g. **paste()**.

Concatenation

```
str_c(my_words, collapse = "|")
```

```
[1] "cat|cart|carrot|catastrophe|dog|rat|bet"
```

Vectorization of concatenation

```
str_c(my_words, my_words, sep = ": ")
```

```
[1] "cat: cat"           "cart: cart"
[3] "carrot: carrot"     "catastrophe: catastrophe"
[5] "dog: dog"           "rat: rat"
[7] "bet: bet"
```

Padding

```
str_pad(my_words, 12)
```

```
[1] "      cat" "      cart" "      carrot" "catastrophe"
[6] "      rat" "      bet"  "
```

Trimming

```
str_trunc(c("anachronism", "antebellum", "antithesis"), 6)
```

```
[1] "ana..." "ant..." "ant..."
```


Examples

Functions of glue

- Format complex character objects with an easy syntax
- Concatenate strings with useful white space treatment
- Made to work with `%>%`

```
year_pub <- 1881
book <- "The Formation of Vegetable Mould through the Action of Worms"
author <- "Charles Darwin"

glue("The author {author}",
     'also wrote "{book}"', # note the use of single quote to escape double quotes
     "in {year_pub}.")
```

The author Charles Darwin also wrote "The Formation of Vegetable Mould through the Action of Worms" in 1881.

Collapsing

```
glue::glue_collapse(letters, '&')
```

```
a&b&c&d&e&f&g&h&i&j&k&l&m&n&o&p&q&r&s&t&u&v&w&x&y&z
```

Vectorised operation utilizing pipes, glue_data()

```
head(patient)
```

```
# A tibble: 6 × 2
  subject_id gender_age
  <int> <chr>
1      1001 m-34
2      1002 f-24
3      1003 m-53
4      1004 f-44
5      1005 m-24
6      1006 f-30
```

```
# with pipes and non-standard evaluation
sample_n( patient, 5) %>%
  separate(gender_age, into=c("gender", "age")) %>%
  glue_data("This case is {age} years old and reports gender as '{gender}'. ")
```

```
This case is 30 years old and reports gender as 'f'.
This case is 53 years old and reports gender as 'm'.
This case is 34 years old and reports gender as 'm'.
This case is 44 years old and reports gender as 'f'.
```

Before we stop

Resources

- [stringi](#) -- General implementation of regular expressions
- [stringr](#) -- Wrapper for vectorisation and convenience functions
- [glue](#) -- formatting complex strings

Acknowledgments ☐ ☐

- Charlotte Wickham
- Hadley Wickham
- Marek Gagolewski (Author of [stringi](#) implementation)

Further reading 📖

- [Strings in R for Data Science](#)

Thank you for your attention!

Try it all yourself!

- Select the practical "String manipulation" for this lecture.