Text Processing & Regular Expressions

Statistical Programming

Fall 2021

Dr. Colin Rundel


stringr
www.rstudio.com

str_pad()

```
str_pad(10^(0:5), width = 8, side = "left") %>%
  cat(sep="\n")
```

```
##        1
##       10
##      100
##     1000
##    10000
##    1e+05
```

```
str_pad(10^(0:5), width = 8, side = "right") %>%
  cat(sep="\n")
```

```
## 1
## 10
## 100
## 1000
## 10000
## 1e+05
```

```
cat(10^(0:5), sep="\n")
```

```
## 1
## 10
## 100
## 1000
## 10000
## 1e+05
```

```
cat(1/10^(0:5), sep="\n")
```

```
## 1
## 0.1
## 0.01
## 0.001
## 1e-04
## 1e-05
```

```
formatC(10^(0:5), digits = 6, width = 6) %>% cat(
```

```
##      1
```

```
formatC(1/10^(0:5), digits = 6, width = 6, format
```

```
##      1
```

str_trim() + str_squish()

```
x = c("   abc" , "ABC   " , " Hello.  World ")
```

```
str_trim(x)
```

```
## [1] "abc"           "ABC"           "Hello.  World"
```

```
str_trim(x, side="left")
```

```
## [1] "abc"            "ABC   "         "Hello.  World "
```

```
str_trim(x, side="right")
```

```
## [1] "   abc"         "ABC"            " Hello.  World"
```

```
str_squish(x)
```

```
## [1] "abc"          "ABC"          "Hello. World"
```

str_trunc()

```
x = "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore
```

```
str_trunc(x, width=60)
```

```
## [1] "Lorem ipsum dolor sit amet, consectetur adipiscing elit, ..."
```

```
str_trunc(x, width=60, side = "left")
```

```
## [1] "...in culpa qui officia deserunt mollit anim id est laborum."
```

```
str_trunc(x, width=60, side = "center")
```

```
## [1] "Lorem ipsum dolor sit amet, c... mollit anim id est laborum."
```

str_wrap()

```
cat(x)
```

```
## Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolor
```

```
str_wrap(x)
```

```
## [1] "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor\nincididunt ut labore et
```

```
str_wrap(x) %>% cat()
```

```
## Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
## incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis
## nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
## Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu
## fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
## culpa qui officia deserunt mollit anim id est laborum.
```

```
str_wrap(x, width=60) %>% cat()
```

```
## Lorem ipsum dolor sit amet, consectetur adipiscing elit,
```

str_glue()

This is a simplified wrapper around `glue::glue()`, use the original for additional control.

```
paste("The value of pi is" , pi)
```

```
## [1] "The value of pi is 3.14159265358979"
```

```
str_glue("The value of pi is {pi}")
```

```
## The value of pi is 3.14159265358979
```

```
paste("The value of tau is" , 2*pi)
```

```
## [1] "The value of tau is 6.28318530717959"
```

```
str_glue("The value of tau is {2*pi}")
```

```
## The value of tau is 6.28318530717959
```

```
str_glue_data(
  iris %>% count(Species),
  "{Species} has {n} observations"
)
```

```
## setosa has 50 observations
## versicolor has 50 observations
## virginica has 50 observations
```

Regular Expressions

stringr - regular expression functions

Function	Description
`str_detect`	Detect the presence or absence of a pattern in a string.
`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.
`str_remove`	Removes the first matched pattern and returns a character vector.
`str_view`	Show the matches made by a pattern.

Many of these functions have variants with an `_all` suffix (e.g. `str_replace_all`) which will match more than one occurrence of the pattern in a given string.

Simple Pattern Detection

```
text = c("The quick brown" , "fox jumps over" , "the lazy dog")
```

```
str_detect(text, "quick")
```

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

```
str_detect(text, "o")
```

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

```
str_detect(text, "row")
```

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

```
str_detect(text, "the")
```

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

```
str_detect(text, regex("the" , ignore_case = TRUE))
```

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

Aside - Escape Characters

An escape character is a character which results in an alternative interpretation of the subsequent character(s). These vary from language to language but for most string implementations `\` is the escape character which is modified by a single following 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

```
print("a\"b")
```

```
## [1] "a\"b"
```

```
print("a\tb")
```

```
## [1] "a\tb"
```

```
print("a\nb")
```

```
## [1] "a\nb"
```

```
print("a\\b")
```

```
## [1] "a\\b"
```

```
cat("a\"b")
```

```
## a"b
```

```
cat("a\tb")
```

```
## a    b
```

```
cat("a\nb")
```

```
## a
## b
```

```
cat("a\\b")
```

```
## a\b
```

Raw character constants

As of R 4.0, R has the ability to define raw character sequences which avoids the need for most escape characters using the `r"(...)"` syntax, where `...` is the raw string.

```
print("\\int_0^\\infty 1/e^x")
```

```
## [1] "\\int_0^\\infty 1/e^x"
```

```
print(r"(\int_0^\infty 1/e^x)")
```

```
## [1] "\\int_0^\\infty 1/e^x"
```

```
cat("\\int_0^\\infty 1/e^x")
```

```
## \int_0^\infty 1/e^x
```

```
cat(r"(\int_0^\infty 1/e^x)")
```

```
## \int_0^\infty 1/e^x
```

```
print("\"test\" and 'test'")
```

```
## [1] "\"test\" and 'test'"
```

```
print('"test" and \'test\'')
```

```
## [1] "\"test\" and 'test'"
```

```
print(r"("test" and 'test')")
```

`[]` and `{}` can be used instead of `()` - see `?Quotes` for details

RegEx Metacharacters

The power of regular expressions comes from their ability to use special metacharacters to modify how pattern matching is performed.

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

Because of their special properties they cannot be matched directly, if you need to match one you need to escape it first (precede it by `\`).

The problem is that regex escapes live on top of character escapes, so there needs to use *two* levels of escapes.

To match	Regex	Literal	Raw
.	\.	"\\."	r"(\.)"
?	\?	"\\?"	r"(\?)"
!	\!	"\\!"	r"(\!)"

Example

```
str_detect("abc[def" ,"\[")
```

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

```
str_detect("abc[def" ,"\\[")
```

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

How do we detect if a string contains a `\` character?

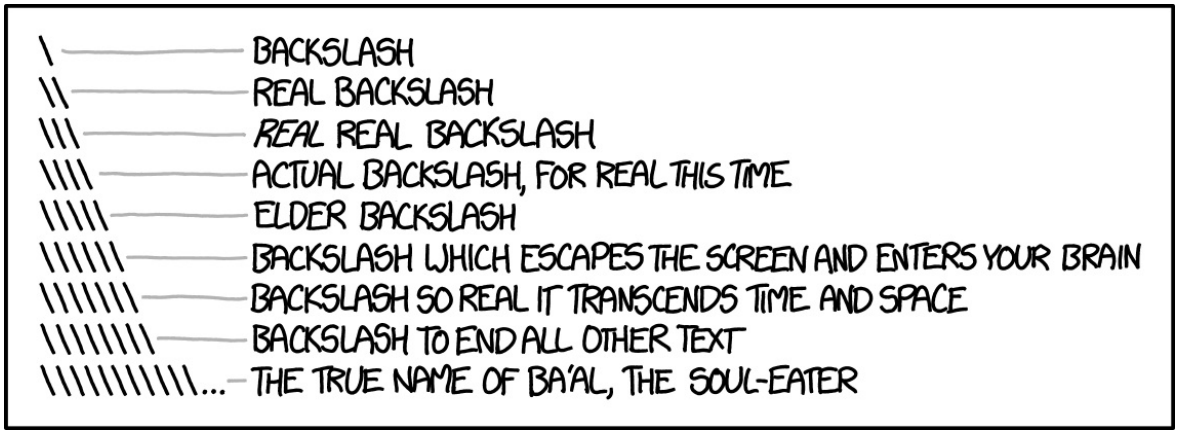
```
cat("abc\\def\n")
```

```
## abc\def
```

```
str_detect("abc\\def" ,"\\\\")
```

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

XKCD's take


\ BACKSLASH
\\ REAL BACKSLASH
\\\ *REAL* REAL BACKSLASH
\\\\ ACTUAL BACKSLASH, FOR REAL THIS TIME
\\\\\ ELDER BACKSLASH
\\\\\\ BACKSLASH WHICH ESCAPES THE SCREEN AND ENTERS YOUR BRAIN
\\\\\\\ BACKSLASH SO REAL IT TRANSCENDS TIME AND SPACE
\\\\\\\\ BACKSLASH TO END ALL OTHER TEXT
\\\\\\\\\\... THE TRUE NAME OF BA'AL, THE SOUL-EATER

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
`\b`	Word boundary
`\B`	Not word boundary

Anchor Examples

```
text = "the quick brown fox jumps over the lazy dog"
```

```
str_replace(text, "^the" , "---")
```

```
## [1] "--- quick brown fox jumps over the lazy dog"
```

```
str_replace(text, "^dog" , "---")
```

```
## [1] "the quick brown fox jumps over the lazy dog"
```

```
str_replace(text, "the$" , "---")
```

```
## [1] "the quick brown fox jumps over the lazy dog"
```

```
str_replace(text, "dog$" , "---")
```

```
## [1] "the quick brown fox jumps over the lazy ---"
```

Anchor Examples - word boundaries

```
text = "the quick brown fox jumps over the lazy dog"
```

```
str_replace_all(text, "\\Brow\\B" , "---")
```

```
## [1] "the quick b---n fox jumps over the lazy dog"
```

```
str_replace_all(text, "\\brow\\b" , "---")
```

```
## [1] "the quick brown fox jumps over the lazy dog"
```

```
str_replace_all(text, "\\bthe" , "---")
```

```
## [1] "--- quick brown fox jumps over --- lazy dog"
```

```
str_replace_all(text, "the\\b" , "---")
```

```
## [1] "--- quick brown fox jumps over --- lazy dog"
```

More complex patterns

If there are more than one pattern we would like to match we can use the or (|) metacharacter.

```
str_replace_all(text,"the|dog" ,"---")
```

```
## [1] "--- quick brown fox jumps over --- lazy ---"
```

```
str_replace_all(text,"a|e|i|o|u" ,"-")
```

```
## [1] "th- q--ck br-wn f-x j-mps -v-r th- l-zy d-g"
```

```
str_replace_all(text,"\\ba|e|i|o|u" ,"-")
```

```
## [1] "th- q--ck br-wn f-x j-mps -v-r th- lazy d-g"
```

```
str_replace_all(text,"\\b(a|e|i|o|u)" ,"-")
```

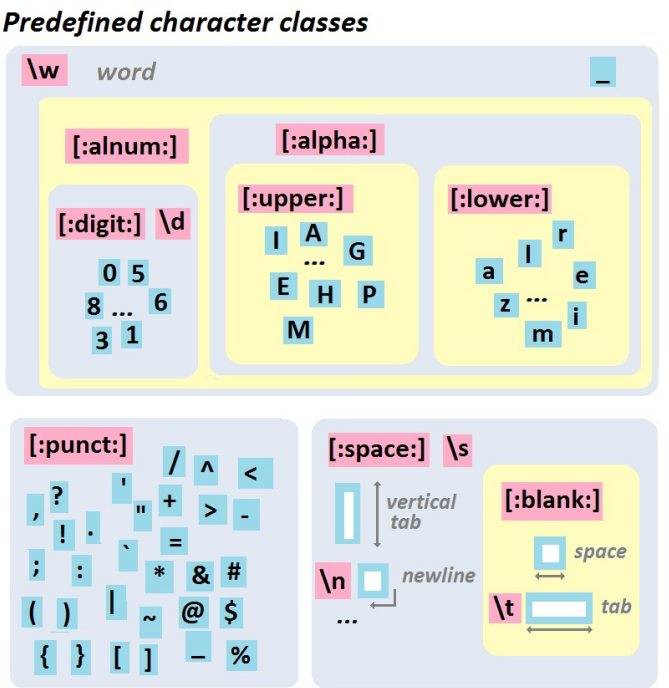
```
## [1] "the quick brown fox jumps -ver the lazy dog"
```

Character Classes

When we want to match whole classes of characters at a time there are a number of convenience patterns built in,

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

A hierarchical view



From http://perso.ens-lyon.fr/lise.vaudor/strings-et-expressions-regulieres/

Example

How would we write a regular expression to match a telephone number with the form `(###) ###-####`?

```
text = c("apple" , "(219) 733-8965" , "(329) 293-8753")
```

```
str_detect(text, "(\d\d\d) \d\d\d-\d\d\d\d")
```

```
## Error: '\d' is an unrecognized escape in character string starting ""(\d"
```

```
str_detect(text, "(\\d\\d\\d) \\d\\d\\d-\\d\\d\\d\\d")
```

```
## [1] FALSE FALSE FALSE
```

```
str_detect(text, "\\(\\d\\d\\d\\) \\d\\d\\d-\\d\\d\\d\\d")
```

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

Classes and Ranges

We can also specify our own classes using square brackets

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

Example

```
text = c("apple" , "(219) 733-8965" , "(329) 293-8753")
```

```
str_replace_all(text, "[aeiou]" , "&")
```

```
## [1] "&ppl&"          "(219) 733-8965" "(329) 293-8753"
```

```
str_replace_all(text, "[13579]" , "*")
```

```
## [1] "apple"          "(2**) ***-8*6*" "(*2*) 2**-8***"
```

```
str_replace_all(text, "[1-5a-ep]" , "*")
```

```
## [1] "***l*"          "(**9) 7**-896*" "(**9) *9*-87**"
```

```
str_replace_all(text, "[^1-5a-ep]" , "*")
```

```
## [1] "app*e"          "*21****33****5" "*32***2*3***53"
```

Exercises 1

For the following vector of randomly generated names, write a regular expression that,

- detects if the person's first name starts with a vowel (a,e,i,o,u)
- detects if the person's last name starts with a vowel
- detects if either the person's first or last name start with a vowel
- detects if neither the person's first nor last name start with a vowel

```
c("Jeremy Cruz", "Nathaniel Le", "Jasmine Chu", "Bradley Calderon Raygoza",
"Quinten Weller", "Katelien Kanamu-Hauanio", "Zuhriyaa al-Amen",
"Travale York", "Alexis Ahmed", "David Alcocer", "Jairo Martinez",
"Dwone Gallegos", "Amanda Sherwood", "Hadiyya el-Eid", "Shaimaaa al-Can",
"Sarah Love", "Shelby Villano", "Sundus al-Hashmi", "Dyani Loving",
"Shanelle Douglas")
```

Quantifiers

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

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

Example

How would we improve our previous regular expression for matching a telephone number with the form `(###) ###-####`?

```
text = c("apple" , "(219) 733-8965" , "(329) 293-8753")
```

```
str_detect(text, "\\(\\d\\d\\d\\) \\d\\d\\d-\\d\\d\\d\\d")
```

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

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

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

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

```
## [1] NA               "(219) 733-8965" "(329) 293-8753"
```

Greedy vs ungreedy matching

What went wrong here?

```
text = "<div class='main'> <div> <a href='here.pdf'>Here!</a> </div> </div>"
```

```
str_extract(text, "<div>.*</div>")
```

```
## [1] "<div> <a href='here.pdf'>Here!</a> </div> </div>"
```

If you add `?` after a quantifier, the matching will be *non-greedy* (find the shortest possible match, not the longest).

```
str_extract(text, "<div>.*?</div>")
```

```
## [1] "<div> <a href='here.pdf'>Here!</a> </div>"
```

Groups

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

Group	Description
(a | b)	match literal "a" or "b" , group either
`a(bc)?`	match "a" or "abc" , group bc or nothing
`(abc)def(hig)`	match "abcdefhig" , group abc and hig
`(?:abc)`	match "abc" , non-capturing group

Example

```
text = c("Bob Smith" , "Alice Smith" , "Apple")
```

```
str_extract(text, "^[:alpha:]+")
```

```
## [1] "Bob"   "Alice" "Apple"
```

```
str_match(text, "^[:alpha:]+")
```

```
##      [,1]
## [1,] "Bob"
## [2,] "Alice"
## [3,] "Apple"
```

```
str_extract(text, "^([:alpha:]+) [:alpha:]+")
```

```
## [1] "Bob Smith"   "Alice Smith" NA
```

```
str_match(text, "^([:alpha:]+) [:alpha:]+")
```

```
##      [,1]          [,2]
## [1,] "Bob Smith"   "Bob"
## [2,] "Alice Smith" "Alice"
## [3,] NA            NA
```

```
str_extract(text, "^([:alpha:]+) ([:alpha:]+)")
```

```
## [1] "Bob Smith"   "Alice Smith" NA
```

```
str_match(text, "^([:alpha:]+) ([:alpha:]+)")
```

```
##      [,1]          [,2]    [,3]
```

How not to use a RegEx

Validating an email address:

```
(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*|"
(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])*")
@(?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?|\\[
(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}
(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?|[a-z0-9-]*[a-z0-9]:
(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])+)\])
```

Exercise 2

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

- Write a regular expression that will extract *all* phone numbers contained in the vector above.
- Once that works use groups to extracts the area code separately from the rest of the phone number.