Text Processing & Regular Expressions

Lecture 12

Dr. Colin Rundel

Base R string functions

As you have likely noticed, the individual characters in a string (element of a character vector) cannot be directly accessed. Base R provides a number of helpful functions for pattern matching and manipulation of these objects:

- `paste()`, `paste0()` - concatenate strings
- `substr()`, `substring()` - extract or replace substrings
- `sprintf()`, `formatC()` - C-like string formatting
- `nchar()` - counts characters
- `strsplit()` - split a string into substrings
- `grep()`, `grepl()` - regular expression pattern matching
- `sub()`, `gsub()` - regular expression pattern replacement

+ many more - the *See Also* section of the above functions' documentation can be used to find additional functions.

Strings are not glamorous, high-profile components of R, but they do play a big role in many data cleaning and preparation tasks. The `stringr` package provides a cohesive set of functions designed to make working with strings as easy as possible. …

`stringr` is built on top of `stringi`, which uses the ICU C library to provide fast, correct implementations of common string manipulations. `stringr` focusses on the most important and commonly used string manipulation functions whereas stringi provides a comprehensive set covering almost anything you can imagine.

Fixed width strings - `str_pad()`

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

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

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

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

`formatC()` (base)

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

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

```r
formatC(
  10^(0:5), digits = 6, width = 6
) |>
  cat(sep="\n")
```

```
     1
    10
   100
  1000
 10000
100000
```

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

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

```r
formatC(
  1/10^(0:5), digits = 6, width = 6,
  format = "fg"
) |>
  cat(sep="\n")
```

```
     1
   0.1
  0.01
 0.001
0.0001
0.00001
```

Whitespace trimming - `str_trim()`, `str_squish()`

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

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

```r
str_trim(x)
```

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

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

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

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

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

```r
str_squish(x)
```

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

String shortening - `str_trunc()`

```
1  x = "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempo
```

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

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

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

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

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

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

String wrapping - `str_wrap()`

```
1 cat(x)
```

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

```
1 str_wrap(x)
```

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

```
1 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.
```

```
1 str_wrap(x, width=60) |> 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.
```

String templates - `str_glue()`

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

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

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

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

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

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

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

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

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

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

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

String capitalization

```
1  x = "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempo
```

```
1  str_to_lower(x)
```

```
[1] "lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor in
```

```
1  str_to_upper(x)
```

```
[1] "LOREM IPSUM DOLOR SIT AMET, CONSECTETUR ADIPISCING ELIT, SED DO EIUSMOD TEMPOR IN
```

```
1  str_to_title(x)
```

```
[1] "Lorem Ipsum Dolor Sit Amet, Consectetur Adipiscing Elit, Sed Do Eiusmod Tempor In
```

```
1  str_to_sentence(x)
```

```
[1] "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor in
```

Regular Expressions

Regular expressions

> A regular expression (shortened as regex or regexp), sometimes referred to as rational expression, is a sequence of characters that specifies a match pattern in text. Usually such patterns are used by string-searching algorithms for “find” or “find and replace” operations on strings, or for input validation. Regular expression techniques are developed in theoretical computer science and formal language theory.
>
> The concept of regular expressions began in the 1950s, when the American mathematician Stephen Cole Kleene formalized the concept of a regular language. They came into common use with Unix text-processing utilities. Different syntaxes for writing regular expressions have existed since the 1980s, one being the POSIX standard and another, widely used, being the Perl syntax.
>
> *Source:* Wikipedia

stringr regular expression functions

Function	Description
`str_detect`	Detect the presence or absence of a pattern in a string.
`str_subset`	Subset a vector of strings based on the presence of a pattern.
`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 string.

Simple Pattern Detection

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

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

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

```
str_subset(text, "quick")
```

```
[1] "The quick brown"
```

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

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

```
str_subset(text, "o")
```

```
[1] "The quick brown" "fox jumps over"  "t
```

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

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

```
str_subset(text, "row")
```

```
[1] "The quick brown"
```

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

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

```
str_subset(text, "the")
```

```
[1] "the lazy dog"
```

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

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

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

```
[1] "The quick brown" "the lazy dog"
```

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

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

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

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

```
a"b
```

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

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

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

```
a	b
```

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

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

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

```
a
b
```

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

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

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

```
a\b
```

Raw character constants

As of v4.0, R has the ability to define raw character sequences which avoids the need for most escape characters, they can be constructed using the `r"(...)"` syntax, where `...` is the raw string.

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

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

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

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

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

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

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

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

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 (may) need to escape it first (precede it by `\`).

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

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

Example

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

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

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

```
[1] TRUE
```

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

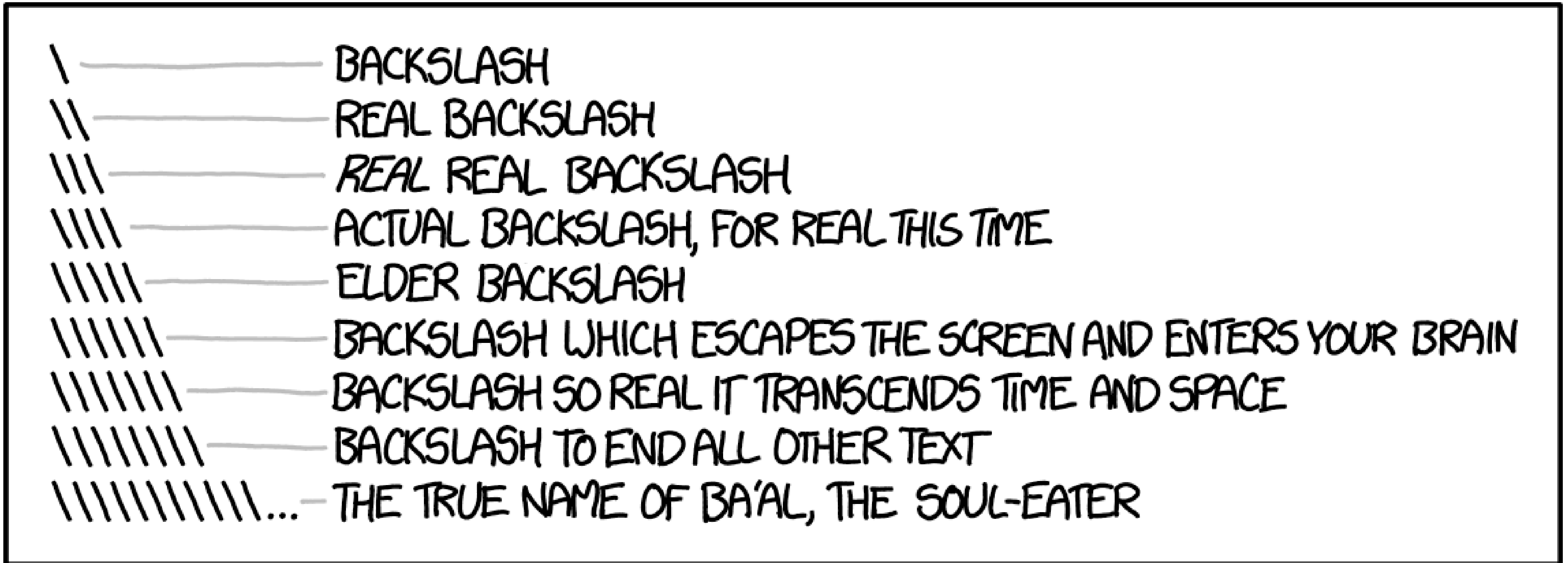
```
1  cat("abc\\def\n")
```

```
abc\def
```

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

```
[1] TRUE
```

XKCD's take



Anchors

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

Regex	Anchor
`^` or `\A`	Start of string
`$` or `\Z`	End of string
`\b`	Word boundary
`\B`	Not word boundary

Anchor Examples

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

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

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

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

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

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

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

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

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

```
1 str_replace_all(text, "the" , "---")
```

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

Anchor Examples - word boundaries

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

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

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

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

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

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

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

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

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

More complex patterns

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

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

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

```r
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"
```

```r
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"
```

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

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

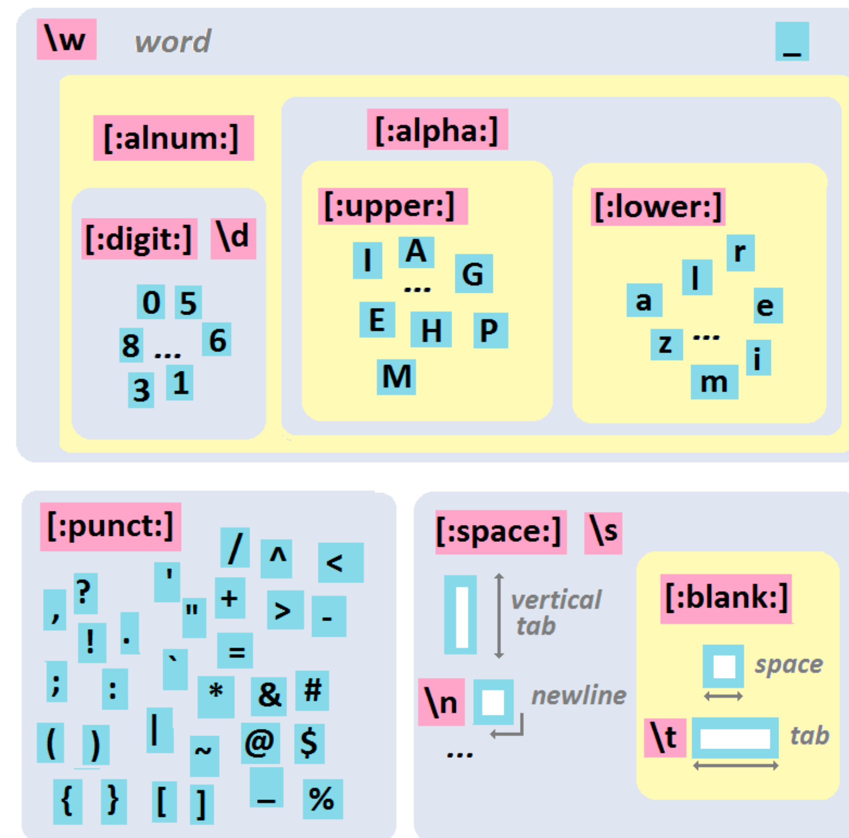
Character Classes

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:]`	Punctuation

A hierarchical view

Predefined character classes



Example

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

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

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

```
## Error: '\d' is an unrecognized escape in character string sta
```

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

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

```
1 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, to simplify these classes ranges can be used for contiguous characters or numbers.

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

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

```r
str_replace_all(text, "[aeiou]" , "*")
```

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

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

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

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

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

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

```
[1] "a***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 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

Example

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

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

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

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

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

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

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

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

Greedy vs non-greedy matching

What went wrong here?

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

```
1  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).

```
1  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:]+) [:alp
```

```
[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:]+) ([:al
```

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

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

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

Backreferences

Backreferences allow you to refer to previously captured groups within a regex or replacement string using `\1`, `\2`, etc.

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

```
1 str_replace(text, "^([:alpha:]+) ([:alpha:]+)", "\\2, \\1")
```

```
[1] "Smith, Bob"   "Smith, Alice" "Apple"
```

Backreferences can also be used for matching repeated patterns,

```
1 str_detect(c("abab" , "cdcd" , "abcd"), "^(..)\\1$")
```

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

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-\x0
@(?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a
(?:(?: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\x0
```

Exercise 2

```r
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.