14 Strings

14.1 Introduction

This chapter introduces you to string manipulation in R. You'll learn the basics of how strings work and how to create them by hand, but the focus of this chapter will be on regular expressions, or regexps for short. Regular expressions are useful because strings usually contain unstructured or semi-structured data, and regexps are a concise language for describing patterns in strings. When you first look at a regexp, you'll think a cat walked across your keyboard, but as your understanding improves they will soon start to make sense.

14.1.1 Prerequisites

This chapter will focus on the **stringr** package for string manipulation, which is part of the core tidyverse.

```
library(tidyverse)
```

14.2 String basics

You can create strings with either single quotes or double quotes. Unlike other languages, there is no difference in behaviour. I recommend always using `"`, unless you want to create a string that contains multiple `"`.

```
string1 <- "This is a string"
string2 <- 'If I want to include a "quote" inside a string, I use single quotes'
```

If you forget to close a quote, you'll see `+`, the continuation character:

```
> "This is a string without a closing quote
+
+
+ HELP I'M STUCK
```

If this happen to you, press Escape and try again!

To include a literal single or double quote in a string you can use `\` to “escape” it:

```
double_quote <- "\"" # or '"'
single_quote <- '\'' # or "'"
```

That means if you want to include a literal backslash, you’ll need to double it up: `"\\"`.

Beware that the printed representation of a string is not the same as string itself, because the printed representation shows the escapes. To see the raw contents of the string, use `writeLines()`:

```
x <- c("\"", "\\")
x
#> [1] "\"" "\\"
writeLines(x)
#> "
#> \
```

There are a handful of other special characters. The most common are `"\n"`, newline, and `"\t"`, tab, but you can see the complete list by requesting help on `"`: `?'"'`, or `?"'"`. You’ll also sometimes see strings like `"\u00b5"`, this is a way of writing non-English characters that works on all platforms:

```
x <- "\u00b5"
x
#> [1] "µ"
```

Multiple strings are often stored in a character vector, which you can create with `c()`:

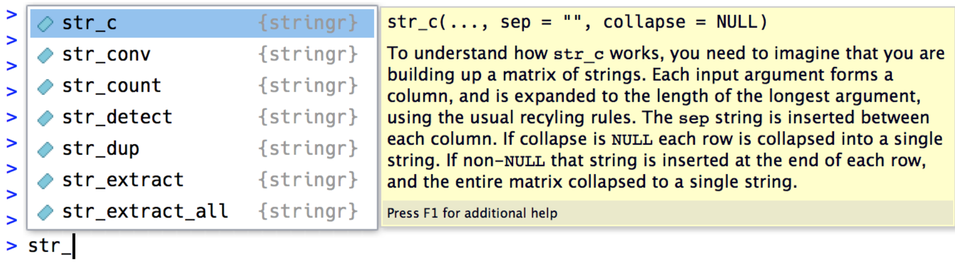
```
c("one", "two", "three")
#> [1] "one"   "two"   "three"
```

14.2.1 String length

Base R contains many functions to work with strings but we'll avoid them because they can be inconsistent, which makes them hard to remember. Instead we'll use functions from stringr. These have more intuitive names, and all start with `str_`. For example, `str_length()` tells you the number of characters in a string:

```
str_length(c("a", "R for data science", NA))
#> [1]  1 18 NA
```

The common `str_` prefix is particularly useful if you use RStudio, because typing `str_` will trigger autocomplete, allowing you to see all stringr functions:



14.2.2 Combining strings

To combine two or more strings, use `str_c()`:

```
str_c("x", "y")
#> [1] "xy"
str_c("x", "y", "z")
#> [1] "xyz"
```

Use the `sep` argument to control how they're separated:

```
str_c("x", "y", sep = ", ")
#> [1] "x, y"
```

Like most other functions in R, missing values are contagious. If you want them to print as "NA", use `str_replace_na()`:

```
x <- c("abc", NA)
str_c("|-", x, "-|")
#> [1] "|-abc-|" NA
str_c("|-", str_replace_na(x), "-|")
#> [1] "|-abc-|" "|-NA-|"
```

As shown above, `str_c()` is vectorised, and it automatically recycles shorter vectors to the same length as the longest:

```
str_c("prefix-", c("a", "b", "c"), "-suffix")
#> [1] "prefix-a-suffix" "prefix-b-suffix" "prefix-c-suffix"
```

Objects of length 0 are silently dropped. This is particularly useful in conjunction with `if`:

```
name <- "Hadley"
time_of_day <- "morning"
birthday <- FALSE

str_c(
  "Good ", time_of_day, " ", name,
  if (birthday) " and HAPPY BIRTHDAY",
  "."
)
#> [1] "Good morning Hadley."
```

To collapse a vector of strings into a single string, use `collapse`:

```
str_c(c("x", "y", "z"), collapse = ", ")
#> [1] "x, y, z"
```

14.2.3 Subsetting strings

You can extract parts of a string using `str_sub()`. As well as the string, `str_sub()` takes `start` and `end` arguments which give the (inclusive) position of the substring:

```
x <- c("Apple", "Banana", "Pear")
str_sub(x, 1, 3)
#> [1] "App" "Ban" "Pea"
# negative numbers count backwards from end
str_sub(x, -3, -1)
#> [1] "ple" "ana" "ear"
```

Note that `str_sub()` won't fail if the string is too short: it will just return as much as possible:

```
str_sub("a", 1, 5)
#> [1] "a"
```

You can also use the assignment form of `str_sub()` to modify strings:

```
str_sub(x, 1, 1) <- str_to_lower(str_sub(x, 1, 1))
x
#> [1] "apple"  "banana" "pear"
```

14.2.4 Locales

Above I used `str_to_lower()` to change the text to lower case. You can also use `str_to_upper()` or `str_to_title()`. However, changing case is more complicated than it might at first appear because different languages have different rules for changing case. You can pick which set of rules to use by specifying a locale:

```
# Turkish has two i's: with and without a dot, and it
# has a different rule for capitalising them:
str_to_upper(c("i", "ı"))
#> [1] "I" "I"
str_to_upper(c("i", "ı"), locale = "tr")
#> [1] "İ" "I"
```

The locale is specified as a ISO 639 language code, which is a two or three letter abbreviation. If you don't already know the code for your language, Wikipedia has a good list. If you leave the locale blank, it will use the current locale, as provided by your operating system.

Another important operation that's affected by the locale is sorting. The base R `order()` and `sort()` functions sort strings using the current locale. If you want robust behaviour across different computers, you may want to use `str_sort()` and `str_order()` which take an additional `locale` argument:

```
x <- c("apple", "eggplant", "banana")

str_sort(x, locale = "en")  # English
#> [1] "apple"    "banana"   "eggplant"

str_sort(x, locale = "haw") # Hawaiian
#> [1] "apple"    "eggplant" "banana"
```

14.2.5 Exercises

1. In code that doesn't use stringr, you'll often see `paste()` and `paste0()` . What's the difference between the two functions? What stringr function are they equivalent to? How do the functions differ in their handling of `NA` ?

2. In your own words, describe the difference between the `sep` and `collapse` arguments to `str_c()` .

3. Use `str_length()` and `str_sub()` to extract the middle character from a string. What will you do if the string has an even number of characters?

4. What does `str_wrap()` do? When might you want to use it?

5. What does `str_trim()` do? What's the opposite of `str_trim()` ?

6. Write a function that turns (e.g.) a vector `c("a", "b", "c")` into the string `a, b, and c` . Think carefully about what it should do if given a vector of length 0, 1, or 2.

14.3 Matching patterns with regular expressions

Regexps are a very terse language that allow you to describe patterns in strings. They take a little while to get your head around, but once you understand them, you'll find them extremely useful.

To learn regular expressions, we'll use `str_view()` and `str_view_all()`. These functions take a character vector and a regular expression, and show you how they match. We'll start with very simple regular expressions and then gradually get more and more complicated. Once you've mastered pattern matching, you'll learn how to apply those ideas with various stringr functions.

14.3.1 Basic matches

The simplest patterns match exact strings:

```
x <- c("apple", "banana", "pear")
str_view(x, "an")
```

```
apple
banana
pear
```

The next step up in complexity is `.`, which matches any character (except a newline):

```
str_view(x, ".a.")
```

```
apple
banana
pear
```

But if "`.`" matches any character, how do you match the character "`.`"? You need to use an "escape" to tell the regular expression you want to match it exactly, not use its special behaviour. Like strings, regexps use the backslash, `\`, to escape special behaviour. So to match an `.`, you need the regexp `\.`. Unfortunately this creates a problem. We use strings to represent regular expressions, and `\` is also used as an escape symbol in strings. So to create the regular expression `\.` we need the string `"\\."`.

```
# To create the regular expression, we need \\
dot <- "\\."

# But the expression itself only contains one:
writeLines(dot)
#> \.

# And this tells R to look for an explicit .
str_view(c("abc", "a.c", "bef"), "a\\.c")
```

```
abc
a.c
bef
```

If `\` is used as an escape character in regular expressions, how do you match a literal `\` ? Well you need to escape it, creating the regular expression `\\` . To create that regular expression, you need to use a string, which also needs to escape `\` . That means to match a literal `\` you need to write `"\\\\"` — you need four backslashes to match one!

```
x <- "a\\b"
writeLines(x)
#> a\b

str_view(x, "\\\\")
```

```
a\b
```

In this book, I'll write regular expression as `\.` and strings that represent the regular expression as `"\\."` .

14.3.1.1 Exercises

1. Explain why each of these strings don't match a `\` : `"\"` , `"\\"` , `"\\\"` .
2. How would you match the sequence `"'\` ?
3. What patterns will the regular expression `\..\..\..` match? How would you represent it as a string?

14.3.2 Anchors

By default, regular expressions will match any part of a string. It's often useful to *anchor* the regular expression so that it matches from the start or end of the string. You can use:

- `^` to match the start of the string.
- `$` to match the end of the string.

```
x <- c("apple", "banana", "pear")
str_view(x, "^a")
```

```
apple
banana
pear
```

```
str_view(x, "a$")
```

```
apple
banana
pear
```

To remember which is which, try this mnemonic which I learned from Evan Misshula: if you begin with power (`^`), you end up with money (`$`).

To force a regular expression to only match a complete string, anchor it with both `^` and `$` :

```
x <- c("apple pie", "apple", "apple cake")
str_view(x, "apple")
```

```
apple pie
apple
apple cake
```

```
str_view(x, "^apple$")
```

```
apple pie
```

```
apple

apple cake
```

You can also match the boundary between words with `\b`. I don't often use this in R, but I will sometimes use it when I'm doing a search in RStudio when I want to find the name of a function that's a component of other functions. For example, I'll search for `\bsum\b` to avoid matching `summarise`, `summary`, `rowsum` and so on.

14.3.2.1 Exercises

1. How would you match the literal string `"$^$"`?

2. Given the corpus of common words in `stringr::words`, create regular expressions that find all words that:

   1. Start with "y".
   2. End with "x"
   3. Are exactly three letters long. (Don't cheat by using `str_length()`!)
   4. Have seven letters or more.

   Since this list is long, you might want to use the `match` argument to `str_view()` to show only the matching or non-matching words.

14.3.3 Character classes and alternatives

There are a number of special patterns that match more than one character. You've already seen `.`, which matches any character apart from a newline. There are four other useful tools:

- `\d`: matches any digit.
- `\s`: matches any whitespace (e.g. space, tab, newline).
- `[abc]`: matches a, b, or c.
- `[^abc]`: matches anything except a, b, or c.

Remember, to create a regular expression containing `\d` or `\s`, you'll need to escape the `\` for the string, so you'll type `"\\d"` or `"\\s"`.

A character class containing a single character is a nice alternative to backslash escapes when you want to include a single metacharacter in a regex. Many people find this more readable.

```
# Look for a literal character that normally has special meaning in a regex
str_view(c("abc", "a.c", "a*c", "a c"), "a[.]c")
```

abc

a.c

a*c

a c

```
str_view(c("abc", "a.c", "a*c", "a c"), ".[*]c")
```

abc

a.c

a*c

a c

```
str_view(c("abc", "a.c", "a*c", "a c"), "a[ ]")
```

abc

a.c

a*c

a c

This works for most (but not all) regex metacharacters: `$` `.` `|` `?` `*` `+` `(` `)` `[` `{`. Unfortunately, a few characters have special meaning even inside a character class and must be handled with backslash escapes: `]` `\` `^` and `-`.

You can use *alternation* to pick between one or more alternative patterns. For example, `abc|d..f` will match either '"abc"', or `"deaf"`. Note that the precedence for `|` is low, so that `abc|xyz` matches `abc` or `xyz` not `abcyz` or `abxyz`. Like with mathematical expressions, if precedence ever gets confusing, use parentheses to make it clear what you want:

```
str_view(c("grey", "gray"), "gr(e|a)y")
```

grey

```
gray
```

14.3.3.1 Exercises

1. Create regular expressions to find all words that:
    1. Start with a vowel.
    2. That only contain consonants. (Hint: thinking about matching “not”-vowels.)
    3. End with `ed`, but not with `eed`.
    4. End with `ing` or `ise`.
2. Empirically verify the rule “i before e except after c”.
3. Is “q” always followed by a “u”?
4. Write a regular expression that matches a word if it’s probably written in British English, not American English.
5. Create a regular expression that will match telephone numbers as commonly written in your country.

14.3.4 Repetition

The next step up in power involves controlling how many times a pattern matches:

- `?`: 0 or 1
- `+`: 1 or more
- `*`: 0 or more

```
x <- "1888 is the longest year in Roman numerals: MDCCCLXXXVIII"
str_view(x, "CC?")
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

```
str_view(x, "CC+")
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

```
str_view(x, 'C[LX]+')
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

Note that the precedence of these operators is high, so you can write: `colou?r` to match either American or British spellings. That means most uses will need parentheses, like `bana(na)+`.

You can also specify the number of matches precisely:

- `{n}` : exactly n
- `{n,}` : n or more
- `{,m}` : at most m
- `{n,m}` : between n and m

```
str_view(x, "C{2}")
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

```
str_view(x, "C{2,}")
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

```
str_view(x, "C{2,3}")
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

By default these matches are "greedy": they will match the longest string possible. You can make them "lazy", matching the shortest string possible by putting a `?` after them. This is an advanced feature of regular expressions, but it's useful to know that it exists:

```
str_view(x, 'C{2,3}?')
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

```
str_view(x, 'C[LX]+?')
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

14.3.4.1 Exercises

1. Describe the equivalents of `?`, `+`, `*` in `{m,n}` form.

2. Describe in words what these regular expressions match: (read carefully to see if I'm using a regular expression or a string that defines a regular expression.)

    1. `^.*$`
    2. `"\\{.+\\}"`
    3. `\d{4}-\d{2}-\d{2}`
    4. `"\\\\{4}"`

3. Create regular expressions to find all words that:

    1. Start with three consonants.
    2. Have three or more vowels in a row.
    3. Have two or more vowel-consonant pairs in a row.

4. Solve the beginner regexp crosswords at https://regexcrossword.com/challenges/beginner.

14.3.5 Grouping and backreferences

Earlier, you learned about parentheses as a way to disambiguate complex expressions. Parentheses also create a *numbered* capturing group (number 1, 2 etc.). A capturing group stores *the part of the string* matched by the part of the regular expression inside the parentheses. You can refer to the same text as previously matched by a capturing group with *backreferences*, like `\1`, `\2` etc. For example, the following regular expression finds all fruits that have a repeated pair of letters.

```
str_view(fruit, "(..)\\1", match = TRUE)
```

```
banana
coconut
cucumber
jujube
papaya
salal berry
```

(Shortly, you'll also see how they're useful in conjunction with `str_match()`.)

14.3.5.1 Exercises

1. Describe, in words, what these expressions will match:

    1. `(.)\1\1`
    2. `"(.)(.)\\2\\1"`
    3. `(..)\1`
    4. `"(.).\\1.\\1"`
    5. `"(.)(.)(.).*\\3\\2\\1"`
2. Construct regular expressions to match words that:

    1. Start and end with the same character.

    2. Contain a repeated pair of letters (e.g. “church” contains “ch” repeated twice.)

    3. Contain one letter repeated in at least three places (e.g. “eleven” contains three “e”s.)

14.4 Tools

Now that you’ve learned the basics of regular expressions, it’s time to learn how to apply them to real problems. In this section you’ll learn a wide array of stringr functions that let you:

- Determine which strings match a pattern.
- Find the positions of matches.
- Extract the content of matches.
- Replace matches with new values.
- Split a string based on a match.

A word of caution before we continue: because regular expressions are so powerful, it’s easy to try and solve every problem with a single regular expression. In the words of Jamie Zawinski:

> Some people, when confronted with a problem, think “I know, I’ll use regular expressions.” Now they have two problems.

As a cautionary tale, check out this regular expression that checks if a email address is valid:

```
(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t]
)+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:
\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(
?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[
\t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\0
31]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\
](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+
(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:
(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z
|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)
?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\
r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[
 \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)
?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t]
)*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[
 \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*
)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t]
)+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*)
*:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+
|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r
\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:
\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t
]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031
]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](
?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?
:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?
:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*)|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?
:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?
[ \t]))*"(?:(?:\r\n)?[ \t])*)*:(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\]
\000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|
\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>
@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"
(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t]
)*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\
".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?
:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[
\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-
\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(
```

```
?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;
:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([
^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\"
.\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\
]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\
[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\
r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\]
 \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]
|\\.)*\](?:(?:\r\n)?[ \t])*))*)*:(?:(?:\r\n)?[ \t])*)?(?:[^()<>@,;:\\".\[\] \0
00-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\
.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,
;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?
:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*
(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".
\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[
^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]
]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(?:\r\n)?[ \t])*)(?:,\s*(
?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\
".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(
?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[
\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t
])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t
])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?
:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|
\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*|(?:
[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\
]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)*\<(?:(?:\r\n)
?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["
()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)
?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>
@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*(?:,@(?:(?:\r\n)?[
 \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,
;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t]
)*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\
".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*)*:(?:(?:\r\n)?[ \t])*)?
(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".
\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:
\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\[
```

```
"()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])
*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])
+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\
.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z
|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(
?:\r\n)?[ \t])*))*)?;\s*)
```

This is a somewhat pathological example (because email addresses are actually surprisingly complex), but is used in real code. See the stackoverflow discussion at http://stackoverflow.com/a/201378 for more details.

Don't forget that you're in a programming language and you have other tools at your disposal. Instead of creating one complex regular expression, it's often easier to write a series of simpler regexps. If you get stuck trying to create a single regexp that solves your problem, take a step back and think if you could break the problem down into smaller pieces, solving each challenge before moving onto the next one.

14.4.1 Detect matches

To determine if a character vector matches a pattern, use `str_detect()`. It returns a logical vector the same length as the input:

```
x <- c("apple", "banana", "pear")
str_detect(x, "e")
#> [1]  TRUE FALSE  TRUE
```

Remember that when you use a logical vector in a numeric context, `FALSE` becomes 0 and `TRUE` becomes 1. That makes `sum()` and `mean()` useful if you want to answer questions about matches across a larger vector:

```
# How many common words start with t?
sum(str_detect(words, "^t"))
#> [1] 65
# What proportion of common words end with a vowel?
mean(str_detect(words, "[aeiou]$"))
#> [1] 0.277
```

When you have complex logical conditions (e.g. match a or b but not c unless d) it's often easier to combine multiple `str_detect()` calls with logical operators, rather than trying to create a single regular expression. For example, here are two ways to find all words that don't contain any vowels:

```
# Find all words containing at least one vowel, and negate
no_vowels_1 <- !str_detect(words, "[aeiou]")
# Find all words consisting only of consonants (non-vowels)
no_vowels_2 <- str_detect(words, "^[^aeiou]+$")
identical(no_vowels_1, no_vowels_2)
#> [1] TRUE
```

The results are identical, but I think the first approach is significantly easier to understand. If your regular expression gets overly complicated, try breaking it up into smaller pieces, giving each piece a name, and then combining the pieces with logical operations.

A common use of `str_detect()` is to select the elements that match a pattern. You can do this with logical subsetting, or the convenient `str_subset()` wrapper:

```
words[str_detect(words, "x$")]
#> [1] "box" "sex" "six" "tax"
str_subset(words, "x$")
#> [1] "box" "sex" "six" "tax"
```

Typically, however, your strings will be one column of a data frame, and you'll want to use filter instead:

```r
df <- tibble(
  word = words,
  i = seq_along(word)
)
df %>%
  filter(str_detect(word, "x$"))
#> # A tibble: 4 x 2
#>   word      i
#>   <chr> <int>
#> 1 box     108
#> 2 sex     747
#> 3 six     772
#> 4 tax     841
```

A variation on `str_detect()` is `str_count()` : rather than a simple yes or no, it tells you how many matches there are in a string:

```r
x <- c("apple", "banana", "pear")
str_count(x, "a")
#> [1] 1 3 1

# On average, how many vowels per word?
mean(str_count(words, "[aeiou]"))
#> [1] 1.99
```

It's natural to use `str_count()` with `mutate()` :

```
df %>%
  mutate(
    vowels = str_count(word, "[aeiou]"),
    consonants = str_count(word, "[^aeiou]")
  )
#> # A tibble: 980 x 4
#>   word         i vowels consonants
#>   <chr>    <int>  <int>      <int>
#> 1 a            1      1          0
#> 2 able         2      2          2
#> 3 about        3      3          2
#> 4 absolute     4      4          4
#> 5 accept       5      2          4
#> 6 account      6      3          4
#> # … with 974 more rows
```

Note that matches never overlap. For example, in `"abababa"` , how many times will the pattern `"aba"` match? Regular expressions say two, not three:

```
str_count("abababa", "aba")
#> [1] 2
str_view_all("abababa", "aba")
```

abababa

Note the use of `str_view_all()` . As you'll shortly learn, many stringr functions come in pairs: one function works with a single match, and the other works with all matches. The second function will have the suffix `_all` .

14.4.1.1 Exercises

1. For each of the following challenges, try solving it by using both a single regular expression, and a combination of multiple `str_detect()` calls.

    1. Find all words that start or end with `x` .

    2. Find all words that start with a vowel and end with a consonant.

    3. Are there any words that contain at least one of each different vowel?

2. What word has the highest number of vowels? What word has the highest proportion of vowels? (Hint: what is the denominator?)

14.4.2 Extract matches

To extract the actual text of a match, use `str_extract()` . To show that off, we're going to need a more complicated example. I'm going to use the Harvard sentences, which were designed to test VOIP systems, but are also useful for practicing regexps. These are provided in `stringr::sentences` :

```
length(sentences)
#> [1] 720
head(sentences)
#> [1] "The birch canoe slid on the smooth planks."
#> [2] "Glue the sheet to the dark blue background."
#> [3] "It's easy to tell the depth of a well."
#> [4] "These days a chicken leg is a rare dish."
#> [5] "Rice is often served in round bowls."
#> [6] "The juice of lemons makes fine punch."
```

Imagine we want to find all sentences that contain a colour. We first create a vector of colour names, and then turn it into a single regular expression:

```
colours <- c("red", "orange", "yellow", "green", "blue", "purple")
colour_match <- str_c(colours, collapse = "|")
colour_match
#> [1] "red|orange|yellow|green|blue|purple"
```

Now we can select the sentences that contain a colour, and then extract the colour to figure out which one it is:

```
has_colour <- str_subset(sentences, colour_match)
matches <- str_extract(has_colour, colour_match)
head(matches)
#> [1] "blue" "blue" "red"  "red"  "red"  "blue"
```

Note that `str_extract()` only extracts the first match. We can see that most easily by first selecting all the sentences that have more than 1 match:

```
more <- sentences[str_count(sentences, colour_match) > 1]
str_view_all(more, colour_match)
```

```
It is hard to erase blue or red ink.
The green light in the brown box flickered.
The sky in the west is tinged with orange red.
```

```
str_extract(more, colour_match)
#> [1] "blue"   "green"  "orange"
```

This is a common pattern for stringr functions, because working with a single match allows you to use much simpler data structures. To get all matches, use `str_extract_all()`. It returns a list:

```
str_extract_all(more, colour_match)
#> [[1]]
#> [1] "blue" "red"
#>
#> [[2]]
#> [1] "green" "red"
#>
#> [[3]]
#> [1] "orange" "red"
```

You'll learn more about lists in lists and iteration.

If you use `simplify = TRUE`, `str_extract_all()` will return a matrix with short matches expanded to the same length as the longest:

```
str_extract_all(more, colour_match, simplify = TRUE)
#>      [,1]     [,2] 
#> [1,] "blue"   "red"
#> [2,] "green"  "red"
#> [3,] "orange" "red"

x <- c("a", "a b", "a b c")
str_extract_all(x, "[a-z]", simplify = TRUE)
#>      [,1] [,2] [,3]
#> [1,] "a"  ""   ""  
#> [2,] "a"  "b"  ""  
#> [3,] "a"  "b"  "c" 
```

14.4.2.1 Exercises

1. In the previous example, you might have noticed that the regular expression matched "flickered", which is not a colour. Modify the regex to fix the problem.

2. From the Harvard sentences data, extract:

    1. The first word from each sentence.
    2. All words ending in `ing`.
    3. All plurals.

14.4.3 Grouped matches

Earlier in this chapter we talked about the use of parentheses for clarifying precedence and for backreferences when matching. You can also use parentheses to extract parts of a complex match. For example, imagine we want to extract nouns from the sentences. As a heuristic, we'll look for any word that comes after "a" or "the". Defining a "word" in a regular expression is a little tricky, so here I use a simple approximation: a sequence of at least one character that isn't a space.

```
noun <- "(a|the) ([^ ]+)"

has_noun <- sentences %>%
  str_subset(noun) %>%
  head(10)
has_noun %>%
  str_extract(noun)
#>  [1] "the smooth" "the sheet"  "the depth"  "a chicken"  "the parked"
#>  [6] "the sun"    "the huge"   "the ball"   "the woman"  "a helps"
```

`str_extract()` gives us the complete match; `str_match()` gives each individual component. Instead of a character vector, it returns a matrix, with one column for the complete match followed by one column for each group:

```
has_noun %>%
  str_match(noun)
#>       [,1]         [,2]  [,3]
#>  [1,] "the smooth" "the" "smooth"
#>  [2,] "the sheet"  "the" "sheet"
#>  [3,] "the depth"  "the" "depth"
#>  [4,] "a chicken"  "a"   "chicken"
#>  [5,] "the parked" "the" "parked"
#>  [6,] "the sun"    "the" "sun"
#>  [7,] "the huge"   "the" "huge"
#>  [8,] "the ball"   "the" "ball"
#>  [9,] "the woman"  "the" "woman"
#> [10,] "a helps"    "a"   "helps"
```

(Unsurprisingly, our heuristic for detecting nouns is poor, and also picks up adjectives like smooth and parked.)

If your data is in a tibble, it's often easier to use `tidyr::extract()`. It works like `str_match()` but requires you to name the matches, which are then placed in new columns:

```
tibble(sentence = sentences) %>%
  tidyr::extract(
    sentence, c("article", "noun"), "(a|the) ([^ ]+)",
    remove = FALSE
  )
#> # A tibble: 720 x 3
#>   sentence                                    article noun
#>   <chr>                                       <chr>   <chr>
#> 1 The birch canoe slid on the smooth planks.  the     smooth
#> 2 Glue the sheet to the dark blue background. the     sheet
#> 3 It's easy to tell the depth of a well.      the     depth
#> 4 These days a chicken leg is a rare dish.    a       chicken
#> 5 Rice is often served in round bowls.        <NA>    <NA>
#> 6 The juice of lemons makes fine punch.       <NA>    <NA>
#> # … with 714 more rows
```

Like `str_extract()` , if you want all matches for each string, you'll need `str_match_all()` .

14.4.3.1 Exercises

1. Find all words that come after a "number" like "one", "two", "three" etc. Pull out both the number and the word.

2. Find all contractions. Separate out the pieces before and after the apostrophe.

14.4.4 Replacing matches

`str_replace()` and `str_replace_all()` allow you to replace matches with new strings. The simplest use is to replace a pattern with a fixed string:

```
x <- c("apple", "pear", "banana")
str_replace(x, "[aeiou]", "-")
#> [1] "-pple"  "p-ar"   "b-nana"
str_replace_all(x, "[aeiou]", "-")
#> [1] "-ppl-"  "p--r"   "b-n-n-"
```

With `str_replace_all()` you can perform multiple replacements by supplying a named vector:

```
x <- c("1 house", "2 cars", "3 people")
str_replace_all(x, c("1" = "one", "2" = "two", "3" = "three"))
#> [1] "one house"    "two cars"     "three people"
```

Instead of replacing with a fixed string you can use backreferences to insert components of the match. In the following code, I flip the order of the second and third words.

```
sentences %>%
  str_replace("([^ ]+) ([^ ]+) ([^ ]+)", "\\1 \\3 \\2") %>%
  head(5)
#> [1] "The canoe birch slid on the smooth planks."
#> [2] "Glue sheet the to the dark blue background."
#> [3] "It's to easy tell the depth of a well."
#> [4] "These a days chicken leg is a rare dish."
#> [5] "Rice often is served in round bowls."
```

14.4.4.1 Exercises

1. Replace all forward slashes in a string with backslashes.
2. Implement a simple version of `str_to_lower()` using `replace_all()`.
3. Switch the first and last letters in `words`. Which of those strings are still words?

14.4.5 Splitting

Use `str_split()` to split a string up into pieces. For example, we could split sentences into words:

```
sentences %>%
  head(5) %>%
  str_split(" ")
#> [[1]]
#> [1] "The"     "birch"   "canoe"   "slid"    "on"      "the"     "smooth"
#> [8] "planks."
#>
#> [[2]]
#> [1] "Glue"        "the"         "sheet"       "to"          "the"
#> [6] "dark"        "blue"        "background."
#>
#> [[3]]
#> [1] "It's"  "easy"  "to"    "tell"  "the"   "depth" "of"    "a"     "well."
#>
#> [[4]]
#> [1] "These"   "days"    "a"       "chicken" "leg"     "is"      "a"
#> [8] "rare"    "dish."
#>
#> [[5]]
#> [1] "Rice"   "is"     "often"  "served" "in"     "round"  "bowls."
```

Because each component might contain a different number of pieces, this returns a list. If you're working with a length-1 vector, the easiest thing is to just extract the first element of the list:

```
"a|b|c|d" %>%
  str_split("\\|") %>%
  .[[1]]
#> [1] "a" "b" "c" "d"
```

Otherwise, like the other stringr functions that return a list, you can use `simplify = TRUE` to return a matrix:

```
sentences %>%
  head(5) %>%
  str_split(" ", simplify = TRUE)
#>      [,1]    [,2]    [,3]    [,4]      [,5]  [,6]    [,7]     [,8]
#> [1,] "The"   "birch" "canoe" "slid"    "on"  "the"   "smooth" "planks."
#> [2,] "Glue"  "the"   "sheet" "to"      "the" "dark"  "blue"   "background."
#> [3,] "It's"  "easy"  "to"    "tell"    "the" "depth" "of"     "a"
#> [4,] "These" "days"  "a"     "chicken" "leg" "is"    "a"      "rare"
#> [5,] "Rice"  "is"    "often" "served"  "in"  "round" "bowls." ""
#>      [,9]
#> [1,] ""
#> [2,] ""
#> [3,] "well."
#> [4,] "dish."
#> [5,] ""
```

You can also request a maximum number of pieces:

```
fields <- c("Name: Hadley", "Country: NZ", "Age: 35")
fields %>% str_split(": ", n = 2, simplify = TRUE)
#>      [,1]      [,2]
#> [1,] "Name"    "Hadley"
#> [2,] "Country" "NZ"
#> [3,] "Age"     "35"
```

Instead of splitting up strings by patterns, you can also split up by character, line, sentence and word `boundary()` s:

```
x <- "This is a sentence.  This is another sentence."
str_view_all(x, boundary("word"))
```

This is a sentence. This is another sentence.

```
str_split(x, " ")[[1]]
#> [1] "This"      "is"        "a"         "sentence." ""          "This"
#> [7] "is"        "another"   "sentence."
str_split(x, boundary("word"))[[1]]
#> [1] "This"     "is"       "a"        "sentence" "This"     "is"       "anothe
#> [8] "sentence"
```

14.4.5.1 Exercises

1. Split up a string like `"apples, pears, and bananas"` into individual components.
2. Why is it better to split up by `boundary("word")` than `" "` ?
3. What does splitting with an empty string (`""`) do? Experiment, and then read the documentation.

14.4.6 Find matches

`str_locate()` and `str_locate_all()` give you the starting and ending positions of each match. These are particularly useful when none of the other functions does exactly what you want. You can use `str_locate()` to find the matching pattern, `str_sub()` to extract and/or modify them.

14.5 Other types of pattern

When you use a pattern that's a string, it's automatically wrapped into a call to `regex()` :

```
# The regular call:
str_view(fruit, "nana")
# Is shorthand for
str_view(fruit, regex("nana"))
```

You can use the other arguments of `regex()` to control details of the match:

- `ignore_case = TRUE` allows characters to match either their uppercase or lowercase forms. This always uses the current locale.

```
bananas <- c("banana", "Banana", "BANANA")
str_view(bananas, "banana")
```

banana

Banana

BANANA

```
str_view(bananas, regex("banana", ignore_case = TRUE))
```

banana

Banana

BANANA

- `multiline = TRUE` allows `^` and `$` to match the start and end of each line rather than the start and end of the complete string.

```
x <- "Line 1\nLine 2\nLine 3"
str_extract_all(x, "^Line")[[1]]
#> [1] "Line"
str_extract_all(x, regex("^Line", multiline = TRUE))[[1]]
#> [1] "Line" "Line" "Line"
```

- `comments = TRUE` allows you to use comments and white space to make complex regular expressions more understandable. Spaces are ignored, as is everything after `#`. To match a literal space, you'll need to escape it: `"\\ "`.

```
phone <- regex("
  \\(?     # optional opening parens
  (\\d{3}) # area code
  [) -]?   # optional closing parens, space, or dash
  (\\d{3}) # another three numbers
  [ -]?    # optional space or dash
  (\\d{3}) # three more numbers
  ", comments = TRUE)

str_match("514-791-8141", phone)
#>      [,1]          [,2]  [,3]  [,4] 
#> [1,] "514-791-814" "514" "791" "814"
```

- `dotall = TRUE` allows `.` to match everything, including `\n`.

There are three other functions you can use instead of `regex()`:

- `fixed()`: matches exactly the specified sequence of bytes. It ignores all special regular expressions and operates at a very low level. This allows you to avoid complex escaping and can be much faster than regular expressions. The following microbenchmark shows that it's about 3x faster for a simple example.

```
microbenchmark::microbenchmark(
  fixed = str_detect(sentences, fixed("the")),
  regex = str_detect(sentences, "the"),
  times = 20
)
#> Unit: microseconds
#>   expr   min    lq  mean median    uq max neval
#>  fixed  50.3  53.3  93.6   64.5  83.2 596    20
#>  regex 249.5 253.7 268.8  256.9 260.2 451    20
```

Beware using `fixed()` with non-English data. It is problematic because there are often multiple ways of representing the same character. For example, there are two ways to define "á": either as a single character or as an "a" plus an accent:

```
a1 <- "\u00e1"
a2 <- "a\u0301"
c(a1, a2)
#> [1] "á" "á"
a1 == a2
#> [1] FALSE
```

They render identically, but because they're defined differently, `fixed()` doesn't find a match. Instead, you can use `coll()`, defined next, to respect human character comparison rules:

```
str_detect(a1, fixed(a2))
#> [1] FALSE
str_detect(a1, coll(a2))
#> [1] TRUE
```

- `coll()`: compare strings using standard **coll**ation rules. This is useful for doing case insensitive matching. Note that `coll()` takes a `locale` parameter that controls which rules are used for comparing characters. Unfortunately different parts of the world use different rules!

```
# That means you also need to be aware of the difference
# when doing case insensitive matches:
i <- c("I", "İ", "i", "ı")
i
#> [1] "I" "İ" "i" "ı"

str_subset(i, coll("i", ignore_case = TRUE))
#> [1] "I" "i"
str_subset(i, coll("i", ignore_case = TRUE, locale = "tr"))
#> [1] "İ" "i"
```

Both `fixed()` and `regex()` have `ignore_case` arguments, but they do not allow you to pick the locale: they always use the default locale. You can see what that is with the following code; more on stringi later.

```
stringi::stri_locale_info()
#> $Language
#> [1] "en"
#>
#> $Country
#> [1] "US"
#>
#> $Variant
#> [1] ""
#>
#> $Name
#> [1] "en_US"
```

The downside of `coll()` is speed; because the rules for recognising which characters are the same are complicated, `coll()` is relatively slow compared to `regex()` and `fixed()`.

- As you saw with `str_split()` you can use `boundary()` to match boundaries. You can also use it with the other functions:

```
x <- "This is a sentence."
str_view_all(x, boundary("word"))
```

This is a sentence.

```
str_extract_all(x, boundary("word"))
#> [[1]]
#> [1] "This"     "is"       "a"        "sentence"
```

14.5.1 Exercises

1. How would you find all strings containing `\` with `regex()` vs. with `fixed()`?
2. What are the five most common words in `sentences`?

14.6 Other uses of regular expressions

There are two useful function in base R that also use regular expressions:

- `apropos()` searches all objects available from the global environment. This is useful if you can't quite remember the name of the function.

```
apropos("replace")
#> [1] "%+replace%"       "replace"          "replace_na"       "setReplaceMe
#> [5] "str_replace"      "str_replace_all"  "str_replace_na"   "theme_replac
```

- `dir()` lists all the files in a directory. The `pattern` argument takes a regular expression and only returns file names that match the pattern. For example, you can find all the R Markdown files in the current directory with:

```
head(dir(pattern = "\\.Rmd$"))
#> [1] "communicate-plots.Rmd" "communicate.Rmd"       "datetimes.Rmd"
#> [4] "EDA.Rmd"               "explore.Rmd"           "factors.Rmd"
```

(If you're more comfortable with "globs" like `*.Rmd`, you can convert them to regular expressions with `glob2rx()`):

14.7 stringi

stringr is built on top of the **stringi** package. stringr is useful when you're learning because it exposes a minimal set of functions, which have been carefully picked to handle the most common string manipulation functions. stringi, on the other hand, is designed to be comprehensive. It contains almost every function you might ever need: stringi has 244 functions to stringr's 49.

If you find yourself struggling to do something in stringr, it's worth taking a look at stringi. The packages work very similarly, so you should be able to translate your stringr knowledge in a natural way. The main difference is the prefix: `str_` vs. `stri_`.

14.7.1 Exercises

1. Find the stringi functions that:

    1. Count the number of words.
    2. Find duplicated strings.

    3. Generate random text.

2. How do you control the language that `stri_sort()` uses for sorting?