Basic String Manipulation

Chapter 9

Stats 20: Introduction to Statistical Programming with R

UCLA

Contents

Acknowledgements: Miles Chen and Jake Elmstedt

Learning Objectives

After studying this chapter, you should be able to:

- Perform basic string manipulation in R
- Perform basic pattern matching with `grep()`, `grepl()`, and `gsub()`
- Interpret and use basic regular expressions
- Calculate the Flesch reading ease score

1 Introduction

Most of statistical computing involves working with numeric data. However, many modern applications have considerable amounts of data in the form of text.

There are whole areas of statistics and machine learning devoted to organizing and interpreting text-based data, such as textual data analysis, linguistic analysis, text mining, sentiment analysis, and natural language processing (NLP).

For more information and resources:

- https://cran.r-project.org/web/views/NaturalLanguageProcessing.html
- https://www.tidytextmining.com/

Text-based analyses are beyond the scope of this course. However, even in non-text-based analyses, working with data in R often requires processing of characters, such as in row/column names, dates, monetary quantities, longitude/latitude, etc.

Other common scenarios involving characters:

- Removing a given character in the names of your variables
- Changing the level(s) of a categorical variable
- Replacing a given character in a dataset
- Converting labels to upper or lower case
- Extracting a regular pattern of characters from a large text file
- Parsing input from an XML or HTML file

A basic understanding of character (or string) manipulation and regular expressions can be a valuable skill for any statistical analysis. We will discuss the most common syntax and functions for string manipulation in base R and introduce basic regular expressions in R.

For more information and resources:

Books and Articles

- Gaston Sanchez's "Handling Strings with R": https://www.gastonsanchez.com/r4strings/
- Garrett Grolemund and Hadley Wickham's "R for Data Science": http://r4ds.had.co.nz/strings.html
- https://en.wikibooks.org/wiki/R_Programming/Text_Processing
- https://cran.r-project.org/web/packages/stringr/vignettes/stringr.html

Cheat Sheets for `stringr` and Regular Expressions

- https://github.com/rstudio/cheatsheets/raw/master/strings.pdf
- https://www.cheatography.com//davechild/cheat-sheets/regular-expressions/pdf/

Sites for Testing Regular Expressions

- https://regex101.com/
- https://regexr.com/

2 Characters in R

2.1 Basic Definitions

Symbols in R that represent text or words are called **characters**. A **string** is a character variable that contains one or more characters, but we often will use “character” and “string” interchangeably.

Values that are stored as characters have base type `character` and are typically printed with quotation marks.

```
x <- "Pawnee rules"
x
```

```
[1] "Pawnee rules"
```

```
typeof(x)
```



```
[1] "character"
```

Characters can be created using single or double quotation marks, but double quotation marks are almost universally preferred.

Single quotation marks can be used within double quotation marks and vice versa, but you cannot directly insert single quotes within single quotes or double quotes within double quotes.

The double quotation inside a string is a special character, so inserting it within double quotes requires a backslash `\"` to escape the special property of the character.

Assignment Project Exam Help


```
"This is the 'R' Language"
```

```
[1] "This is the 'R' Language"
```

```
"This is the \"R\" Language"
```

https://powcoder.com

```
[1] "This is the \"R\" Language"
```

```
"This is an "error""
```

Add WeChat powcoder

```
## Error: unexpected symbol in ""This is an "error"
```

The `character()` function creates a character vector of a specified length. The default value in each element of the vector is the **empty character** `""`.

```
character(5)
```

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

Note: The empty character `""` is **not the same** as `character(0)`.

2.2 The paste() Function

The `paste()` function is one of the most important functions for creating and building strings.

The `paste()` function inputs one or more R objects, converts them to `character`, and then concatenates (pastes) them to form one or several character strings.

The basic syntax is:

```
paste(..., sep = " ", collapse = NULL)
```

- The `...` argument means the input can be any number of objects.
- The optional `sep` argument specifies the separator between characters after pasting. The default is a single whitespace `" "`.
- The optional `collapse` argument specifies characters to separate the result.

```r
paste("I ate some", pi, "and it was deloicious.")
```

```
[1] "I ate some 3.14159265358979 and it was deloicious."
```

```r
paste("Bears", "Beets", "Battlestar Galactica", sep = ", ")
```

```
[1] "Bears, Beets, Battlestar Galactica"
```

```r
paste("h", c("a", "e", "o"), sep = "")  # No collapsing
```

```
[1] "ha" "he" "ho"
```

```r
paste("h", c("a", "e", "o"), sep = "", collapse = ", and ")
```

```
[1] "ha, and he, and ho"
```


Note: Vectors of different lengths will be recycled in the usual way, but partial recycling will not throw a warning.

2.3 Print Functions for Characters

There are several functions to print strings:

- `print()` is for generic printing.
- `noquote()` is for printing without quotation marks.
- `cat()` is for concatenation.
- `format()` is for (pretty) printing with special formatting.

2.3.1 The print() and noquote() Functions

The `print()` function (technically the `print.default()` method) has an optional logical `quote` argument that specifies whether to print characters with or without quotation marks.

A similar output can be produced using `noquote()`.

```r
print(x, quote = FALSE)
```



```
[1] Pawnee rules
```

```r
noquote(x)
```

```
[1] Pawnee rules
```

Side Note: While the output appears identical, the commands are not the same. The `noquote()` function outputs a `noquote` class object, which is then inputted into the `print.noquote()` method.

2.3.2 The cat() Function

The `cat()` funtion concatenates multiple character vectors into a single vector, adds a specified separator, and prints the result (without quotations).

```r
cat(x, "Eagleton drools", sep = ", ")
```

```
Pawnee rules, Eagleton drools
```

The printing is slightly different from that of `noquote()`. In particular, the printed output does not have the vector index, and the `cat()` function returns an invisible `NULL` (meaning assigning the printed output to a variable does not work).

One benefit of `cat()` is that the printed output can be saved to an external file using the `file` argument:

```r
cat(x, "Eagleton drools", sep = ", ", file = "pawnee.txt")
```

When **file** is specified, an optional logical argument **append** specifies whether the result should be appended to or overwrite an existing file.

Side Note: There are a few other optional arguments that are useful for longer text strings. Consult the R documentation for more information.

2.3.3 The format() Function

The **format()** function formats an R object for "pretty" printing.

Some useful arguments used in **format()**:

- **width** specifies the (minimum) width of strings produced.
- **trim** specifies whether there should be no padding with spaces (TRUE).
- **justify** controls how padding takes place for strings. Takes the values "left", "right", "centre", or "none".

For controling the printing of numbers:

- **digits** specifies the number of significant digits to use.
- **nsmall** specifies the minimum number of digits to the right of the decimal point to include.
- **scientific** specifies whether to use scientific notation (TRUE) or standard notation (FALSE).


```r
format(1/(1:5), digits = 2)
```

```
[1] "1.00" "0.50" "0.33" "0.25" "0.20"
```

```r
format(1/(1:5), digits = 2, scientific = TRUE)
```



```
[1] "1.0e+00" "5.0e-01" "3.3e-01" "2.5e-01" "2.0e-01"
```

```r
format(c("Pawnee", "rules", "Eagleton", "drools"), width = 10,
    justify = "left")
```



```
[1] "Pawnee    " "rules     " "Eagleton  " "drools    "
```

3 Basic String Manipulation

3.1 Functions for Basic String Manipulation

There are many functions in base R for basic string manipulation.

Function	Description
nchar()	Returns number of characters
tolower()	Converts to lower case
toupper()	Converts to upper case
casefold()	Wrapper for **tolower()** and **toupper()**
chartr()	Translates characters
abbreviate()	Abbreviates characters
substr()	Extracts substrings of a character vector
strsplit()	Splits strings into substrings

The best way to understand how these functions work is to try them on simple examples and see how the input character vector changes.

3.2 The `nchar()` Function

The **`nchar()`** function inputs a character vector and outputs the number of (human-readable) characters contained in each entry of the vector.

```
y <- c("Pawnee rules", "Eagleton drools")
nchar(y)
```

```
[1] 12 15
```

3.3 Case Folding Functions

The process of converting all characters into the same case (upper-case or lower-case) is called **case folding**. The **`tolower()`** and **`toupper()`** functions convert upper-case letters in a character vector to lower-case, or vice-versa.

```
tolower(y)
```

```
[1] "pawnee rules"    "eagleton drools"
```

```
toupper(y)
```

```
[1] "PAWNEE RULES"    "EAGLETON DROOLS"
```

The **`casefold()`** function is a wrapper for `tolower()` and `toupper()` that uses a logical argument `upper` to determine which function to use.

```
casefold(y, upper = TRUE)
```

```
[1] "PAWNEE RULES"    "EAGLETON DROOLS"
```

Question: What is the default value of the `upper` argument in `casefold()`? Equivalently, does `casefold()` use `tolower()` or `toupper()` by default?

3.4 The `chartr()` Function

The **`chartr()`** function performs character translation. The `old` argument specifies the characters to be translated, and the `new` argument specifies the translations.

For example, the command below translates `P` into `p` and `E` into `e`.

```
chartr(old = "PE", new = "pe", y)
```

```
[1] "pawnee rules"    "eagleton drools"
```

We can also translate to and from non-alphabetic characters. For example, the command below translates `a` into `#`, `e` into `?`, and `o` into `!`.

```
chartr("aeo", "#?!", y)
```

```
[1] "P#wn?? rul?s"    "E#gl?t!n dr!!ls"
```

3.5 The `substr()` Function

The **`substr()`** function inputs a character vector and extracts a **substring** (i.e., a subset of the original character values) starting from the `start` position and ending with the `stop` position.

```
# Extract the 2nd to 9th characters of `x`
substr(x, start = 2, stop = 9)
```

```
[1] "awnee ru"
```


Assignment Project Exam Help
https://powcoder.com
Add WeChat powcoder

```r
# Extract the 3rd to 5th characters of each value in `y`
substr(y, start = 3, stop = 5)
```

```
[1] "wne" "gle"
```

3.6 The strsplit() Function

The **strsplit()** function splits elements of a character vector x into substrings based on the pattern specified in the split argument. The output of strsplit() is always a list object with the same length as the input vector. In particular, the ith component of the output list contains the vector of splits of x[i].

For example, the command below splits the character Pawnee rules by the letter l.

```r
strsplit(x, split = "l")
```

```
[[1]]
[1] "Pawnee ru" "es"
```


To separate a sentence into separate words, we can split by the single space character " ".

```r
z <- c("Pawnee rules and Eagleton drools.", "I love friends, waffles, and work.")
word_z <- strsplit(z, split = " ")
word_z
```



```
[[1]]
[1] "Pawnee"   "rules"    "and"      "Eagleton" "drools."

[[2]]
[1] "I"        "love"     "friends," "waffles," "and"      "work."
```


Note: The ith component in word_z contains the words in the ith sentence of z. To combine all values in separate components of a list into a single vector, we can use the **unlist()** function to remove the list structure.

```r
unlist(word_z)
```



```
 [1] "Pawnee"   "rules"    "and"      "Eagleton" "drools."  "I"
 [7] "love"     "friends," "waffles," "and"      "work."
```

4 Pattern Matching

4.1 Introduction and the %in% Operator

One main application of string manipulation is pattern matching. Finding patterns in text are useful for data validation, data scraping, text parsing, filtering search results, etc.

A first tool for pattern/value matching is the **%in%** operator. The %in% operator is a vectorized binary operator that checks each value in the vector on the left and returns TRUE if the entry matches one of the values on the right and FALSE otherwise. The output of the %in% operator is always a logical vector.

```r
1:10 %in% c(5, 7, 9)
```

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

```r
ucla <- c("u", "c", "l", "a")
letters %in% ucla
```

```
 [1]  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
[13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
[25] FALSE FALSE
```

```r
letters[letters %in% ucla]
```

```
[1] "a" "c" "l" "u"
```

Question: How would you write a function `is_vowel()` to find the vowels (`"a"`, `"e"`, `"i"`, `"o"`, `"u"`, `"y"`) in a character vector containing single letters?

4.2 The `grep()` and `grepl()` Functions

The **`grep()`** and **`grepl()`** functions searches for matches to a pattern in an input character vector. The syntax and use are largely the same, but the return output is different. The basic syntax is:

```r
grep(pattern, x)
grepl(pattern, x)
```


- The **`pattern`** argument is the character string to be matched in the character vector `x`. The `pattern` can be the literal character(s) to match or a regular expression.
- The **`x`** argument is the input character vector where matches are to be found.
- There are other optional arguments as well, including an **`ignore.case`** argument that specifies whether the pattern to match is case sensitive or not.

The command `grep(pattern, x)` returns a numeric vector of the indices of the entries of `x` that contain a match to `pattern`. The command `grepl(pattern, x)` returns a logical vector of whether each entry of `x` contains a match (`TRUE`) or not (`FALSE`).

Assignment Project Exam Help

```r
test <- c("April", "and", "Andy", "love", "Champion", "and",
    "Lil'", "Sebastian")
grep(pattern = "a", test)
```

```
[1] 2 5 6 8
```

https://powcoder.com

```r
grepl(pattern = "a", test)
```

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

Add WeChat powcoder

4.3 The `gsub()` Function

The **`gsub()`** function finds and replaces patterns in an input character vector. The basic syntax is:

```r
gsub(pattern, replacement, x)
```

The `pattern`, `x`, and optional arguments of `gsub()` are identical to those found in `grep()` and `grepl()`. While `grep()` and `grepl()` identify pattern matches, `gsub()` replaces the `pattern` match by the **`replacement`** argument.

```r
gsub(pattern = "A", replacement = "a", test)
```

```
[1] "april"     "and"       "andy"      "love"      "Champion"  "and"
[7] "Lil'"      "Sebastian"
```

```r
gsub(pattern = "a", replacement = "x", test)
```

```
[1] "April"     "xnd"       "Andy"      "love"      "Chxmpion"  "xnd"
[7] "Lil'"      "Sebxstixn"
```

4.4 Regular Expressions

For more complicated patterns, we need more tools to efficiently specify the pattern to match.

A **regular expression** (or **regex**) is a set of symbols that describes a text pattern. More formally, a regular expression is a pattern that describes a set of strings.

Regular expressions are a formal language in their own right in the sense that the symbols have a defined set of rules to specify the desired patterns. Most programming languages, including R, can use and implement regular expressions. The best way to learn the syntax and become fluent with regular expressions is to practice.

Some common applications of regular expressions:

- Test if a phone number has the correct number of digits
- Test if a date follows a specifc format (e.g. mm/dd/yy)
- Test if an email address is in a valid format
- Test if a password has numbers and special characters
- Search a document for gray spelled either as "gray" or "grey"
- Search a document and replace all occurrences of "Will", "Bill", or "W." with "William"
- Count the number of times in a document that the word "analysis" is immediately preceded by the words "data", "computer", or "statistical"
- Convert a comma-delimited file into a tab-delimited file
- Find duplicate words in a text

We will not cover a full treatment of regular expressions in R (it is typically covered in detail in Stats 102A). For more information, refer to the `?regex` help documentation or one of the references at the beginning of this chapter.

We will introduce a few basic regular expressions in the next section to allow us to compute a readability measure of the difficulty level of a passage in English is to understand.

5 Application: The Flesch Reading Ease Score

5.1 Introduction

The **Flesch reading ease score** is a numeric measure of English readability, i.e., the ease with which a reader can understand text.

The formula to compute the Flesch reading ease (RE) score is

$$\begin{aligned} \text{RE} &= 206.835 - \left(1.015 \times \frac{\text{total words}}{\text{total sentences}}\right) - \left(84.6 \times \frac{\text{total syllables}}{\text{total words}}\right) \\ &= 206.835 - (1.015 \times \text{ASL}) - (84.6 \times \text{ASW}) \end{aligned}$$

where ASL is the **average sentence length** and ASW is the **average number of syllables per word**.

The reading ease score RE is usually a number between 0 and 100, though there are some exceptions for non-standard words/sentences. Higher values of RE indicate text that is easier to read, and lower values indicate text that is more difficult to read.

More information on the Flesch reading ease score:

- http://www.readabilityformulas.com/flesch-reading-ease-readability-formula.php
- https://en.wikipedia.org/wiki/Flesch-Kincaid_readability_tests#Flesch_reading_ease

From the Readability Formulas site:

> "Though simple it might seem, the Flesch Reading Ease Formula has certain ambiguities. For instance, periods, explanation [sic] points, colons and semicolons serve as sentence delimiters; each group of continuous non-blank characters with beginning and ending punctuation removed counts as a word; each vowel in a word is considered one syllable subject to: (a) -es, -ed and -e (except -le) endings are ignored; (b) words of three letters or shorter count as single syllables; and (c) consecutive vowels count as one syllable."

We want to write a function in R that inputs an English passage and outputs the reading ease score of the passage.

As an example, we will compute the Flesch reading ease score of the following passage:

> "We need to remember what's important in life: friends, waffles, work. Or waffles, friends, work. Doesn't matter, but work is third." – Leslie Knope (Parks and Recreation)

For convenience, the command to create a `waffles` object containing this passage is in the `waffles.R` file on CCLE.

```r
source("waffles.R")
waffles
```

```
[1] "We need to remember what's important in life: friends, waffles, work. Or
waffles, friends, work. Doesn't matter, but work is third."
```

The primary components of the Flesch reading score formula are: sentences, words, and syllables. The main steps to compute the reading score are:

1. Separate the text passage into individual sentences, and count the number of sentences.
2. Separate each sentence into individual words, and count the number of words for each sentence.
3. Separate each individual word into individual syllables, and count the number of syllables.

5.2 Splitting Text Into Sentences

The `waffles` object is a single character value (i.e., a character vector of length 1) that contains the entire passage. We want to separate the text passage into individual sentences.

To find the individual sentences, we need to split the text string based on "end of sentence" punctuation. The **sentence delimiters**, i.e., what symbols represent the end of a sentence, we want to consider are periods (`.`), exclamation points (`!`), question marks (`?`), colons (`:`), and semicolons (`;`).

A regular expression that represents the pattern of "any sentence delimiter" would be `[.!?:;]`. In the context of regular expressions, the square brackets define a **character set**, which means any single character that is contained within the brackets will match the pattern. The regular expression for "any vowel" would be `[aeiouy]`.

We will use this regular expression to split the sentences in the `waffles` object into separate characters.

```
strsplit(waffles, split = "[.!?:;]")
```

```
[[1]]
[1] "We need to remember what's important in life"
[2] " friends, waffles, work"
[3] " Or waffles, friends, work"
[4] " Doesn't matter, but work is third"
```


Remember that the output of `strsplit()` is always a list. Since the `waffles` object was a single character value, then the output of `strsplit(waffles, split = "[.!?:;]")` has a single component. To continue processing the text, we will extract the character vector inside.


```
waffles_sentences <- strsplit(waffles, split = "[.!?:;]")[[1]]
waffles_sentences
```



```
[1] "We need to remember what's important in life"
[2] " friends, waffles, work"
[3] " Or waffles, friends, work"
[4] " Doesn't matter, but work is third"
```


Within each sentence, the capitalization and punctuation are not important to the Flesch reading ease formula (since it only counts syllables, words, and sentences, not capital letters, commas, or apostrophes). We can thus prepare our sentences for splitting into words by converting all letters to lower case and removing all remaining punctuation.

We can apply the `tolower()` function to perform the case folding.

```
waffles_sentences <- tolower(waffles_sentences)
waffles_sentences
```

```
[1] "we need to remember what's important in life"
[2] " friends, waffles, work"
[3] " or waffles, friends, work"
[4] " doesn't matter, but work is third"
```

To remove the remaining punctuation, we can use the regular expression `[[:punct:]]` that represents the pattern of *any* single punctuation symbol. We will replace the punctuation by the empty character `""`.

```
waffles_sentences <- gsub(pattern = "[[:punct:]]", replacement = "",
    waffles_sentences)
waffles_sentences
```

```
[1] "we need to remember whats important in life"
[2] " friends waffles work"
[3] " or waffles friends work"
[4] " doesnt matter but work is third"
```

Question: Why did we not use `[[:punct:]]` to remove the sentence delimiters? Why is it safe to match all punctuation at this step, including sentence delimiters?

5.3 Splitting Sentences Into Words

We now have a vector of sentences that have been processed to remove cases and punctuation. We are ready to move on to counting words!

At this stage in the notes, each entry in the `waffles_sentences` vector is a sentence that we want to split into words. Since we have removed all punctuation already, the only character that separates words is the single whitespace character `" "`. We thus can use `strsplit()` again, splitting based on `" "`.

```
waffles_words <- strsplit(waffles_sentences, split = " ")
waffles_words
```

```
[[1]]
[1] "we"         "need"       "to"         "remember"   "whats"      "important"
[7] "in"         "life"

[[2]]
[1] ""        "friends" "waffles" "work"

[[3]]
[1] ""        "or"      "waffles" "friends" "work"

[[4]]
[1] ""       "doesnt" "matter" "but"    "work"   "is"     "third"
```


Caution: Notice the output! By splitting based on the whitespace `" "`, we have leading empty characters in all components of `waffles_words` except for the first component. This is due to the space after the end of a sentence. We do not want to count the empty character `""` as a word in our reading ease formula, so we need to remove them.

Within each component of the `waffles_words` list, we want to only keep the character values that have a nonzero number of characters. This can be done in one line, but we will create a helper function for clarity.

```
keep_words <- function(words) {
    words[nchar(words) > 0]
}
```


The `keep_words()` function inputs a vector of words and returns only the words that have a positive number of characters. We can now use `lapply()` to apply the `keep_words()` function to each component of the `waffles_words` list.

```
waffles_words <- lapply(waffles_words, keep_words)
waffles_words
```

```
[[1]]
[1] "we"         "need"       "to"         "remember"   "whats"      "important"
[7] "in"         "life"

[[2]]
[1] "friends" "waffles" "work"

[[3]]
[1] "or"      "waffles" "friends" "work"

[[4]]
[1] "doesnt" "matter" "but"    "work"   "is"     "third"
```

Note: At this point in the notes, you are able to compute the ASL (average sentence length) value in the reading ease formula.

Question: How would you compute the ASL for the `waffles` text using the `waffles_words` object?

5.4 Splitting Words Into Syllables

The last piece of the reading ease formula we need to compute is the number of syllables in a word.

Formally, each vowel sound, with or without consonants, defines a syllable within a word. For the purposes of the reading ease formula, we will count the syllables by the number of vowels in a word, subject to three rules:

(a) Words of three letters or shorter count as single syllables
(b) -es, -ed and -e (except -le) endings are ignored
(c) Consecutive vowels count as one syllable

Side Note: These rules will inevitably yield different numbers of syllables than you may expect due to a myriad of exceptions in English spelling and pronunciation. We will follow these rules to be consistent with the way the Flesch reading ease score is calculated, even if it may not be 100% accurate in all cases.

Before tackling all of the words in the entire text, we first want to know how to count the syllables in a single word.

As a first step, we need to separate each word into its component letters. To split a word into letters, we can again use `strsplit()`, now splitting by the empty character `""`.


```
tom_letters <- unlist(strsplit("tom", split = ""))
tom_letters
```

```
[1] "t" "o" "m"
```


Assignment Project Exam Help

```
horses_letters <- unlist(strsplit("horses", split = ""))
horses_letters
```

```
[1] "h" "o" "r" "s" "e" "s"
```

https://powcoder.com

```
eagleton_letters <- unlist(strsplit("eagleton", split = ""))
eagleton_letters
```

```
[1] "e" "a" "g" "l" "e" "t" "o" "n"
```

Add WeChat powcoder

Note: Remember that the output of `strsplit()` is always a list.

5.4.1 Accounting For Short Words

The first rule: Words of three letters or shorter count as single syllables.

Regardless of how many vowels are in a short word (at most 3 letters long), the Flesch reading ease formula counts the entire word as one syllable. Thus, before counting vowels, we simply need to check whether there are at most 3 letters in the word.

The `tom_letters` vector contains the individual letters of the word `"tom"`. The number of letters in the word is the length of the `tom_letters` vector.

```
length(tom_letters)
```

```
[1] 3
```

Since `length(tom_letters)` is 3, then `tom` is one syllable.

For longer words, such as `"horses"` and `"eagleton"`, we will need to consider the other rules and count the vowels.

5.4.2 Accounting for Special Word Endings

The second rule: -es, -ed and -e (except -le) endings are ignored.

Before we count the syllables in words longer than 3 letters, we need to first ignore the special word endings: -es, -ed, and -e, *unless* the ending is -le.

The `horses_letters` vector contains the individual letters of the word `"horses"`. We can use the `tail()` function to extract the last two letters of the word.

```
horses_tail <- tail(horses_letters, n = 2)
```

We can write a helper function `is_special_ending()` that inputs a vector of two letters (that represent the last two letters of a word) and returns `TRUE` if the word ends in a special ending (-es, -ed, or -e except -le) and `FALSE` otherwise.


```
is_special_ending <- function(ending) {
    is_es <- all(ending == c("e", "s"))
    is_ed <- all(ending == c("e", "d"))
    is_e_not_le <- ending[2] == "e" & ending[1] != "l"
    is_es | is_ed | is_e_not_le
}
```



```
is_special_ending(horses_tail)
```

```
[1] TRUE
```


Since the word ends in -es, we will remove the word ending and count the syllables in the remaining "word" `"hors"`.


```
rm_special_endings <- function(word_letters) {
    word_tail <- tail(word_letters, n = 2)
    if (is_special_ending(word_tail)) {
        if (word_tail[2] == "e") {
            word_letters[-length(word_letters)]
        } else {
            head(word_letters, n = -2)
        }
    } else {
        word_letters
    }
}
```



```
rm_special_endings(horses_letters)
```

```
[1] "h" "o" "r" "s"
```

```
rm_special_endings(eagleton_letters)
```

```
[1] "e" "a" "g" "l" "e" "t" "o" "n"
```

Note: Notice that `rm_special_endings()` will remove two letters from words ending in -es or -ed but only one letter from words ending in -e other than -le. If there is no special ending, the function will simply leave the input letters unchanged.

5.4.3 Accounting For Consecutive Vowels

The third rule: Consecutive vowels count as one syllable.

Once the first two rules are accounted for, we next need to be able to identify which letters are vowels. As an example, we will use the character `"eagleton"`.

We can write a helper function `is_vowel()` that inputs a vector of letters and returns `TRUE` for each vowel and `FALSE` otherwise.

```
is_vowel <- function(letter) {
    letter %in% c("a", "e", "i", "o", "u", "y")
}
```

```
eagleton_vowels <- is_vowel(eagleton_letters)
eagleton_vowels
```



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

If every vowel counted as one syllable, then `sum(is_vowel(eagleton_letters))` would be the number of syllables in the word. However, we need to count consecutive vowels as a single syllable. For example, the word `"eagleton"` has two consecutive vowels `e` and `a` that count as one syllable instead of two. How do we identify when two vowels are next to each other?

There are many ways to account for consecutive vowels. One way is to consider the numeric indices of the vowels: Notice that letters that are consecutive vowels will appear as consecutive `TRUE` values in the output of `is_vowel()`. This translates into consecutive indices of the `TRUE` values.


```
which(eagleton_vowels)
```

```
[1] 1 2 5 7
```


Based on the numeric indices alone, can you tell which vowels are consecutive?

Consecutive vowels will have consecutive indices! One trick to find consecutive indices is to find the consecutive differences with the `diff()` function.


```
diff(which(eagleton_vowels))
```

```
[1] 1 3 2
```

Each consecutive difference of 1 indicates consecutive vowels. So the total number of syllables in the word is

$$(\text{number of syllables}) = (\text{number of vowels}) - (\text{number of consecutive differences of 1 in the vowel indices}).$$

5.4.4 Computing the Number of Syllables In A Word

We now have all the components to compute the syllables of a word.

```
count_syllables <- function(word) {
    word_letters <- unlist(strsplit(word, split = ""))
    if (length(word_letters) <= 3) {
        1
    } else {
        word_letters <- rm_special_endings(word_letters)
        word_vowels <- is_vowel(word_letters)
        sum(word_vowels) - sum(diff(which(word_vowels)) == 1)
    }
}
```

```r
count_syllables("tom")
```

```
[1] 1
```

```r
count_syllables("horses")
```

```
[1] 1
```

```r
count_syllables("eagleton")
```

```
[1] 3
```

```r
count_syllables("pneumonoultramicroscopicsilicovolcanoconiosis")
```

```
[1] 17
```


Note: At this point in the notes, you are able to compute the ASW (average number of syllables per word) value in the reading ease formula.

Question: How would you compute the ASW for the `waffles` text using the `waffles_words` object?

5.5 Combining Everything Together

Success! We have found a way to compute each component of the Flesch reading score formula.

Your task is to use the workflow and helper functions explained and provided in this section to write a `reading_ease()` function that will compute the Flesch reading ease score for an input text passage.

```r
waffles
```

```
[1] "We need to remember what's important in life: friends, waffles, work. Or
waffles, friends, work. Doesn't matter, but work is third."
```



```r
reading_ease(waffles)
```

```
[1] 96.76339
```


For the `waffles` text, the Flesch reading score is 96.7633929. Use this value to verify that you have implemented your function correctly.