Lesson 4 - Of Data Cleaning and Documentation - Conquer Regular Expressions and Challenge yourself with a 'Real' Dataset

Contents

Objective: At the end of this session you will be able to use regular expressions to 'clean' your data.

Packages Used in This Lesson

The following packages are used in this lesson:

`tidyverse` (`stringr`, `dplyr`)
`mgsub`

Please install and load these packages for the lesson. In this document I will load each package separately, but I will not be reminding you to install the package. Remember: these packages may be from CRAN OR Bioconductor.

Highlighting

`grey background` - a package, function, code or command
italics - an important term or concept

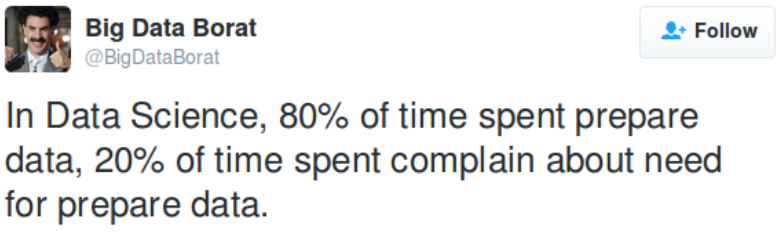

Figure 1:

bold - heading or 'grammar of graphics' term
blue text - named or unnamed hyperlink

Data Files Used in This Lesson

-University returns_for_figshare_FINAL.xlsx
-Readme_file.docx

These files can be downloaded at https://github.com/eacton/CAGEF/tree/master/Lesson_4/data. Right-click on the filename and select 'Save Link As...' to save the file locally. The files should be saved in the same folder you plan on using for your R script for this lesson.

Or click on the blue hyperlink at the start of the README.md at https://github.com/eacton/CAGEF/tree/master/Lesson_4 to download the entire folder at DownGit.

Load libraries

Since we are moving along in the world, we are now going to start loading our libraries at the start of our script. This is a 'best practice' and makes it much easier for someone to reproduce your work efficiently by knowing exactly what packages they need to run your code.

```
library("tidyverse")
library(mgsub)
```

Data Cleaning or Data Munging or Data Wrangling

Why do we need to do this?

'Raw' data is seldom (never) in a useable format. Data in tutorials or demos has already been meticulously filtered, transformed and readied to showcase that specific analysis. How many people have done a tutorial only to find they can't get their own data in the format to use the tool they have just spend an hour learning about???

Data cleaning requires us to:

- get rid of inconsistencies in our data.
- have labels that make sense.
- check for invalid character/numeric values.
- check for incomplete data.
- remove data we do not need.
- get our data in a proper format to be analyzed by the tools we are using.
- flag/remove data that does not make sense.

Some definitions might take this a bit farther and include normalizing data and removing outliers, but I consider data cleaning as getting data into a format where we can start actively doing 'the maths or the graphs' - whether it be statistical calculations, normalization or exploratory plots.

Today we are going to mostly be focusing on the **data cleaning of text**. This step is crucial to taking control of your dataset and your metadata. I have included the functions I find most useful for these tasks but I encourage you to take a look at the Strings Chapter in *R for Data Science* for an exhaustive list of functions. We have learned how to transform data into a tidy format in Lesson 2, but the prelude to transforming data is doing the grunt work of data cleaning. So let's get to it!

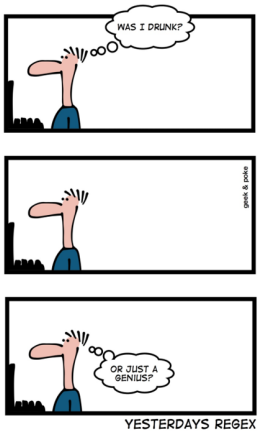

Figure 2:

Intro to regular expressions

Regular expressions

"A God-awful and powerful language for expressing patterns to match in text or for search-and-replace. Frequently described as 'write only', because regular expressions are easier to write than to read/understand. And they are not particularly easy to write." - Jenny Bryan

So why do regular expressions or 'regex' get so much flak if it is so powerful for text matching?

Scary example: how to get an email in different programming languages http://emailregex.com/.

Regex is definitely one of those times when it is important to annotate your code. There are many jokes related to people coming back to their code the next day and having no idea what their code means.

There are sites available to help you make up your regular expressions and validate them against text. These are usually not R specific, but they will get you close and the expression will only need a slight modification for R (like an extra backslash - described below).

Regex testers:

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

Today we will be practicing regex at Simon Goring's R-specific demo tester:
https://simongoring.shinyapps.io/RegularExpressionR/

What does the language look like?

The language is based on *meta-characters* which have a special meaning rather than their literal meaning. For example, '$' is used to match the end of a string, and this use supercedes its use as a character in a string (ie 'Joe paid $2.99 for chips.').

Let's start with a simple string. A string can be captured with single or double quotes.

```
x <- "The quick brown fox is quick."
# is equivalent to
x <- "The quick brown fox is quick."
```

In order to understand regular expressions we will use 2 functions:

`writeLines()` is a function that outputs lines of text.

`str_view()` is a function that will highlight the regular expression for our string.

For example, given our string:

```
writeLines(text = x)
```

```
## The quick brown fox is quick.
```

```
str_view(string = x, pattern = "quick")
```

```
## PhantomJS not found. You can install it with webshot::install_phantomjs(). If it is installed, pleas
```

```
str_view_all(string = x, pattern = "quick")
```

Note that the output of `writeLines()` does not contain the quotations. You can think of the quotations as a container for our string, rather than a part of the string itself.

To match the pattern 'quick' `str_view()` will match the first instance of a match in *each* character string. `str_view_all()` will highlight all matches. This behaviour can be seen more easily with strings:

```
y <- c("The quick brown fox is quick.", "The quick brown fox is quick.")

writeLines(text = y)
```

```
## The quick brown fox is quick.
## The quick brown fox is quick.
```

```
str_view(string = y, pattern = "quick")
```

```
str_view_all(string = y, pattern = "quick")
```

Classes

What kind of character is it?

Expression	Meaning
\w, [A-z0-9], [[:alnum:]]	word characters (letters + digits)
\d, [0-9], [[:digit:]]	digits
[A-z], [:alpha:]	alphabetical characters
\s, [[:space:]]	space
[[:punct:]]	punctuation
[[:lower:]]	lowercase
[[:upper:]]	uppercase
\W, [^A-z0-9]	not word characters
\S	not space
\D, [^0-9]	not digits

Examples:

```
a <- "Joe paid $2.99 for chips."

# word characters (letters + digits)
str_view_all(a, "[[:alnum:]]")
```

```
# digits
str_view_all(a, "\\d")
```

```
# alphabetical characters
str_view_all(a, "[A-z]")
```

```r
# space
str_view_all(a, "\\s")
```

```r
# not word characters
str_view_all(a, "[^A-z0-9]")
```

```r
# not spaces
str_view_all(a, "\\S")
```

Quantifiers

How many times will a character appear?

Expression	Meaning
?	0 or 1
	0 or more
+	1 or more
{n}	exactly n
{n,}	at least n
{,n}	at most n
{n,m}	between n and m (inclusive)

Examples:

```r
# 0 or 1
str_view_all(a, "9?")
```

```r
# 0 or more
str_view_all(a, "9*")
```

```r
# 1 or more
str_view_all(a, "9+")
```

```r
# exactly n
str_view(a, "9{1}")
```

```r
# at least
str_view(a, "9{1,}")
```

Operators

Helper actions to match your characters.

Expression	Meaning
|	or
.	matches any single character
[ ]	matches ANY of the characters inside the brackets
[-]	matches a RANGE of characters inside the brackets
[^]	matches any character EXCEPT those inside the bracket
( )	grouping - used for backreferencing

Examples:

```r
# or
str_view_all(a, "(o|i)")
```

```r
# any single character
str_view_all(a, "f.r")
```

```r
# any characters inside the brackets
str_view_all(a, "[aps]")
```

```r
# any characters that are not found inside the brackets
str_view_all(a, "[^a-e]")
```

```r
# grouping for a 'positive look-ahead'
str_view_all(a, "\\d(?=\\.)")
```

```r
# grouping for a 'positive look-behind'
str_view_all(a, "(?<=\\$)\\d")
```

Matching by position

Where is the character in the string?

Expression	Meaning
^	start of string
$	end of string
\b	empty string at either edge of a word
\B	empty string that is NOT at the edge of a word

Examples:

```r
# empty string at either edge of a word boundary
str_view_all(y, "\\b")
```

```r
# start of string
str_view_all(y, "^\\w+\\b")
```

```r
# end of string
str_view_all(y, "\\b\\w+[[:punct:]]$")
```

Escape characters

Sometimes a meta-character is just a character. *Escaping* allows you to use a character 'as is' rather than its special function. In R, regex gets evaluated as a string before a regular expression, and a backslash is used to escape the string - so you really need 2 backslashes to escape, say, a '$' sign (`"\\$"`).

ExpressionMeaning	
\	escape for meta-characters to be used as characters (*, $, ^, ., ?, |, \, [,], {, }, (,)).

Figure 3: Joking/Not Joking (xkcd)

Expression	Meaning
Note: the back-slash is also a meta-character.	

Examples:

```
#escaping meta characters
str_view_all(a, "\\$|\\.")

b <- "'Joe paid $2.99 for chips.'"

writeLines(b)
#escaping quotations
str_view_all(b, "\'")

#escaping escape characters
c <-  "\escape this!\'"
```

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

```
c <- "\\escape this!\\"

writeLines(c)
```

```
## \escape this!\
```

```
# remember to escape the string as well
str_view_all(c, "\\\\")
```

Trouble-shooting with escaping meta-characters means adding backslashes until something works.

While you can always refer back to this lesson for making your regular expressions, you can also use this regex cheatsheet.

What I would like to get across it that it is okay to google and use resources early on for regex, and that even experts still use these resources.

Figure 4:

Intro to string manipulation with stringr

Common uses of string manipulation are: searching, replacing or removing (making substitutions), and splitting and combining substrings.

As an example, we are going to play with a string of DNA.

```
dino <- ">DinoDNA from Crichton JURASSIC PARK  p. 103 nt 1-1200 GCGTTGCTGGCGTTTTTCCATAGGCTCCGCCCCCCTGAC
```

```
## >DinoDNA from Crichton JURASSIC PARK  p. 103 nt 1-1200 GCGTTGCTGGCGTTTTTCCATAGGC
## TCCGCCCCCCTGACGAGCATCACAAAAATCGACGCGGTGGCGAAACCCGACAGGACTATAAAGATACCAGGCGTTTCCCC
## CTGGAAGCTCCCTCGTGTTCCGACCCTGCCGCTTACCGGATACCTGTCCGCCTTTCTCCCTTCGGGAAGCGTGGCTGCTC
## ACGCTGTACCTATCTCAGTTCGGTGTAGGTCGTTCGCTCCAAGCTGGGCTGTGTGCCGTTCAGCCCGACCGCTGCGCCTT
## ATCCGGTAACTATCGTCTTGAGTCCAACCCGGTAAAGTAGGACAGGTGCCGGCAGCGCTCTGGGTCATTTTCGGCGAGGA
## CCGCTTTCGCTGGAGATCGGCCTGTCGCTTGCGGTATTCGGAATCTTGCACGCCCTCGCTCAAGCCTTCGTCACTCCAAA
## CGTTTCGGCGAGAAGCAGGCCATTATCGCCGGCATGGCGGCCGACGCGCTGGGCTGGCGTTCGCGACGCGAGGCTGGATG
## GCCTTCCCCATTATGATTCTTCTCGCTTCCGGCGGCCCGCGTTGCAGGCCATGCTGTCCAGGCAGGTAGATGACGACCAT
## CAGGGACAGCTTCAACGGCTCTTACCAGCCTAACTTCGATCACTGGACCGCTGATCGTCACGGCGATTTATGCCGCACAT
## GGACGCGTTGCTGGCGTTTTTCCATAGGCTCCGCCCCCCTGACGAGCATCACAAACAAGTCAGAGGTGGCGAAACCCGAC
## AGGACTATAAAGATACCAGGCGTTTCCCCCTGGAAGCGCTCTCCTGTTCCGACCCTGCCGCTTACCGGATACCTGTCCGC
## CTTTCTCCCTTCGGGCTTTCTCAATGCTCACGCTGTAGGTATCTCAGTTCGGTGTAGGTCGTTCGCTCCAAGCTGACGAA
## CCCCCCGTTCAGCCCGACCGCTGCGCCTTATCCGGTAACTATCGTCTTGAGTCCAACACGACTTAACGGGTTGGCATGGA
## TTGTAGGCGCCGCCCTATACCTTGTCTGCCTCCCCGCGGTGCATGGAGCCGGGCCACCTCGACCTGAATGGAAGCCGGCG
## GCACCTCGCTAACGGCCAAGAATTGGAGCCAATCAATTCTTGCGGAGAACTGTGAATGCGCAAACCAACCCTTGGCCATC
## GCGTCCGCCATCTCCAGCAGCCGCACGCGGCGCATCTCGGGCAGCGTTGGGTCCT
```

This piece of DNA is from the book Jurassic park, and was supposed to be dinosaur DNA, but is actually just a cloning vector. Bummer.

Removing:

This string is in FASTA format, but we don't need the header - we just want to deal with the DNA sequence. The header begins with '>' and ends with a number, '1200', with a space between the header and the sequence. Let's practice capturing each of these parts of a string, and then we'll make a regular expression to remove the entire header.

All `stringr` functions take in as arguments the **string** you are manipulating and the **pattern** you are

Figure 5: nope

capturing. `str_remove` replaces the matched pattern with an empty character string "". In our first search we remove '>' from our string, dino.

```
str_remove(string = dino, pattern = ">")
```

```
## DinoDNA from Crichton JURASSIC PARK  p. 103 nt 1-1200 GCGTTGCTGGCGTTTTTCCATAGGCT
## CCGCCCCCCCTGACGAGCATCACAAAAATCGACGCGGTGGCGAAACCCGACAGGACTATAAAGATACCAGGCGTTTCCCCC
## TGGAAGCTCCCTCGTGTTCCGACCCTGCCGCTTACCGGATACCTGTCCGCCTTTCTCCCTTCGGGAAGCGTGGCTGCTCA
## CGCTGTACCTATCTCAGTTCGGTGTAGGTCGTTCGCTCCAAGCTGGGCTGTGTGCCGTTCAGCCCGACCGCTGCGCCTTA
## TCCGGTAACTATCGTCTTGAGTCCAACCCGGTAAAGTAGGACAGGTGCCGGCAGCGCTCTGGGTCATTTTCGGCGAGGAC
## CGCTTTCGCTGGAGATCGGCCTGTCGCTTGCGGTATTCGGAATCTTGCACGCCCTCGCTCAAGCCTTCGTCACTCCAAAC
## GTTTCGGCGAGAAGCAGGCCATTATCGCCGGCATGGCGGCCGACGCGCTGGGCTGGCGTTCGCGACGCGAGGCTGGATGG
## CCTTCCCCATTATGATTCTTCTCGCTTCCGGCGGCCCGCGTTGCAGGCCATGCTGTCCAGGCAGGTAGATGACGACCATC
## AGGGACAGCTTCAACGGCTCTTACCAGCCTAACTTCGATCACTGGACCGCTGATCGTCACGGCGATTTATGCCGCACATG
## GACGCGTTGCTGGCGTTTTTCCATAGGCTCCGCCCCCCTGACGAGCATCACAAACAAGTCAGAGGTGGCGAAACCCGACA
## GGACTATAAAGATACCAGGCGTTTCCCCCTGGAAGCGCTCTCCTGTTCCGACCCTGCCGCTTACCGGATACCTGTCCGCC
## TTTCTCCCTTCGGGCTTTCTCAATGCTCACGCTGTAGGTATCTCAGTTCGGTGTAGGTCGTTCGCTCCAAGCTGACGAAC
## CCCCCGTTCAGCCCGACCGCTGCGCCTTATCCGGTAACTATCGTCTTGAGTCCAACACGACTTAACGGGTTGGCATGGAT
## TGTAGGCGCCGCCCTATACCTTGTCTGCCTCCCCGCGGTGCATGGAGCCGGGCCACCTCGACCTGAATGGAAGCCGGCGG
## CACCTCGCTAACGGCCAAGAATTGGAGCCAATCAATTCTTGCGGAGAACTGTGAATGCGCAAACCAACCCTTGGCCATCG
## CGTCCGCCATCTCCAGCAGCCGCACGCGGCGCATCTCGGGCAGCGTTGGGTCCT
```

Next we can search for numbers. The expression '[0-9]' is looking for any number. Always make sure to check that the pattern you are using gives you the output you expect.

```
str_remove(string = dino, pattern = "[0-9]")
```

```
## >DinoDNA from Crichton JURASSIC PARK  p. 03 nt 1-1200 GCGTTGCTGGCGTTTTTCCATAGGCT
## CCGCCCCCCCTGACGAGCATCACAAAAATCGACGCGGTGGCGAAACCCGACAGGACTATAAAGATACCAGGCGTTTCCCCC
## TGGAAGCTCCCTCGTGTTCCGACCCTGCCGCTTACCGGATACCTGTCCGCCTTTCTCCCTTCGGGAAGCGTGGCTGCTCA
## CGCTGTACCTATCTCAGTTCGGTGTAGGTCGTTCGCTCCAAGCTGGGCTGTGTGCCGTTCAGCCCGACCGCTGCGCCTTA
## TCCGGTAACTATCGTCTTGAGTCCAACCCGGTAAAGTAGGACAGGTGCCGGCAGCGCTCTGGGTCATTTTCGGCGAGGAC
## CGCTTTCGCTGGAGATCGGCCTGTCGCTTGCGGTATTCGGAATCTTGCACGCCCTCGCTCAAGCCTTCGTCACTCCAAAC
## GTTTCGGCGAGAAGCAGGCCATTATCGCCGGCATGGCGGCCGACGCGCTGGGCTGGCGTTCGCGACGCGAGGCTGGATGG
## CCTTCCCCATTATGATTCTTCTCGCTTCCGGCGGCCCGCGTTGCAGGCCATGCTGTCCAGGCAGGTAGATGACGACCATC
## AGGGACAGCTTCAACGGCTCTTACCAGCCTAACTTCGATCACTGGACCGCTGATCGTCACGGCGATTTATGCCGCACATG
## GACGCGTTGCTGGCGTTTTTCCATAGGCTCCGCCCCCCTGACGAGCATCACAAACAAGTCAGAGGTGGCGAAACCCGACA
## GGACTATAAAGATACCAGGCGTTTCCCCCTGGAAGCGCTCTCCTGTTCCGACCCTGCCGCTTACCGGATACCTGTCCGCC
## TTTCTCCCTTCGGGCTTTCTCAATGCTCACGCTGTAGGTATCTCAGTTCGGTGTAGGTCGTTCGCTCCAAGCTGACGAAC
## CCCCCGTTCAGCCCGACCGCTGCGCCTTATCCGGTAACTATCGTCTTGAGTCCAACACGACTTAACGGGTTGGCATGGAT
```

```
## TGTAGGCGCCGCCCTATACCTTGTCTGCCTCCCCGCGGTGCATGGAGCCGGGCCACCTCGACCTGAATGGAAGCCGGCGG
## CACCTCGCTAACGGCCAAGAATTGGAGCCAATCAATTCTTGCGGAGAACTGTGAATGCGCAAACCAACCCTTGGCCATCG
## CGTCCGCCATCTCCAGCAGCCGCACGCGGCGCATCTCGGGCAGCGTTGGGTCCT
```

Why aren't all of the numbers replaced? `str_remove` only replaces the first match in a character string. Switching to `str_remove_all` replaces all instances of numbers in the character string.

```
str_remove_all(string = dino, pattern = "[0-9]")
```

```
## >DinoDNA from Crichton JURASSIC PARK  p.  nt - GCGTTGCTGGCGTTTTTCCATAGGCTCCGCCCC
## CCTGACGAGCATCACAAAAATCGACGCGGTGGCGAAACCCGACAGGACTATAAAGATACCAGGCGTTTCCCCCTGGAAGC
## TCCCTCGTGTTCCGACCCTGCCGCTTACCGGATACCTGTCCGCCTTTCTCCCTTCGGGAAGCGTGGCTGCTCACGCTGTA
## CCTATCTCAGTTCGGTGTAGGTCGTTCGCTCCAAGCTGGGCTGTGTGCCGTTCAGCCCGACCGCTGCGCCTTATCCGGTA
## ACTATCGTCTTGAGTCCAACCCGGTAAAGTAGGACAGGTGCCGGCAGCGCTCTGGGTCATTTTCGGCGAGGACCGCTTTC
## GCTGGAGATCGGCCTGTCGCTTGCGGTATTCGGAATCTTGCACGCCCTCGCTCAAGCCTTCGTCACTCCAAACGTTTCGG
## CGAGAAGCAGGCCATTATCGCCGGCATGGCGGCCGACGCGCTGGGCTGGCGTTCGCGACGCGAGGCTGGATGGCCTTCCC
## CATTATGATTCTTCTCGCTTCCGGCGGCCCGCGTTGCAGGCCATGCTGTCCAGGCAGGTAGATGACGACCATCAGGGACA
## GCTTCAACGGCTCTTACCAGCCTAACTTCGATCACTGGACCGCTGATCGTCACGGCGATTTATGCCGCACATGGACGCGT
## TGCTGGCGTTTTTCCATAGGCTCCGCCCCCCTGACGAGCATCACAAACAAGTCAGAGGTGGCGAAACCCGACAGGACTAT
## AAAGATACCAGGCGTTTCCCCCTGGAAGCGCTCTCCTGTTCCGACCCTGCCGCTTACCGGATACCTGTCCGCCTTTCTCC
## CTTCGGGCTTTCTCAATGCTCACGCTGTAGGTATCTCAGTTCGGTGTAGGTCGTTCGCTCCAAGCTGACGAACCCCCCGT
## TCAGCCCGACCGCTGCGCCTTATCCGGTAACTATCGTCTTGAGTCCAACACGACTTAACGGGTTGGCATGGATTGTAGGC
## GCCGCCCTATACCTTGTCTGCCTCCCCGCGGTGCATGGAGCCGGGCCACCTCGACCTGAATGGAAGCCGGCGGCACCTCG
## CTAACGGCCAAGAATTGGAGCCAATCAATTCTTGCGGAGAACTGTGAATGCGCAAACCAACCCTTGGCCATCGCGTCCGC
## CATCTCCAGCAGCCGCACGCGGCGCATCTCGGGCAGCGTTGGGTCCT
```

How do we capture spaces? The pattern '\s' replaces a space. However, for the backslash to not be used as an escape character (its special function), we need to add another backslash, making our pattern '\\s'.

```
str_remove_all(string = dino, pattern = "\\s")
```

```
## >DinoDNAfromCrichtonJURASSICPARKp.103nt1-1200GCGTTGCTGGCGTTTTTCCATAGGCTCCGCCCCCC
## TGACGAGCATCACAAAAATCGACGCGGTGGCGAAACCCGACAGGACTATAAAGATACCAGGCGTTTCCCCCTGGAAGCTC
## CCTCGTGTTCCGACCCTGCCGCTTACCGGATACCTGTCCGCCTTTCTCCCTTCGGGAAGCGTGGCTGCTCACGCTGTACC
## TATCTCAGTTCGGTGTAGGTCGTTCGCTCCAAGCTGGGCTGTGTGCCGTTCAGCCCGACCGCTGCGCCTTATCCGGTAAC
## TATCGTCTTGAGTCCAACCCGGTAAAGTAGGACAGGTGCCGGCAGCGCTCTGGGTCATTTTCGGCGAGGACCGCTTTCGC
## TGGAGATCGGCCTGTCGCTTGCGGTATTCGGAATCTTGCACGCCCTCGCTCAAGCCTTCGTCACTCCAAACGTTTCGGCG
## AGAAGCAGGCCATTATCGCCGGCATGGCGGCCGACGCGCTGGGCTGGCGTTCGCGACGCGAGGCTGGATGGCCTTCCCCA
## TTATGATTCTTCTCGCTTCCGGCGGCCCGCGTTGCAGGCCATGCTGTCCAGGCAGGTAGATGACGACCATCAGGGACAGC
## TTCAACGGCTCTTACCAGCCTAACTTCGATCACTGGACCGCTGATCGTCACGGCGATTTATGCCGCACATGGACGCGTTG
## CTGGCGTTTTTCCATAGGCTCCGCCCCCCCTGACGAGCATCACAAACAAGTCAGAGGTGGCGAAACCCGACAGGACTATAA
## AGATACCAGGCGTTTCCCCCTGGAAGCGCTCTCCTGTTCCGACCCTGCCGCTTACCGGATACCTGTCCGCCTTTCTCCCT
## TCGGGCTTTCTCAATGCTCACGCTGTAGGTATCTCAGTTCGGTGTAGGTCGTTCGCTCCAAGCTGACGAACCCCCCGTTC
## AGCCCGACCGCTGCGCCTTATCCGGTAACTATCGTCTTGAGTCCAACACGACTTAACGGGTTGGCATGGATTGTAGGCGC
## CGCCCTATACCTTGTCTGCCTCCCCGCGGTGCATGGAGCCGGGCCACCTCGACCTGAATGGAAGCCGGCGGCACCTCGCT
## AACGGCCAAGAATTGGAGCCAATCAATTCTTGCGGAGAACTGTGAATGCGCAAACCAACCCTTGGCCATCGCGTCCGCCA
## TCTCCAGCAGCCGCACGCGGCGCATCTCGGGCAGCGTTGGGTCCT
```

To remove the entire header, we need to combine these patterns. The header is everything in between '>' and the number '1200' followed by a space. The operator `.` captures any single character and the quantifier `*` is any number of times (including zero).

```
str_remove(string = dino, pattern = ">.*[0-9]\\s")
```

```
## GCGTTGCTGGCGTTTTTCCATAGGCTCCGCCCCCCTGACGAGCATCACAAAAATCGACGCGGTGGCGAAACCCGACAGGA
## CTATAAAGATACCAGGCGTTTCCCCCTGGAAGCTCCCTCGTGTTCCGACCCTGCCGCTTACCGGATACCTGTCCGCCTTT
## CTCCCTTCGGGAAGCGTGGCTGCTCACGCTGTACCTATCTCAGTTCGGTGTAGGTCGTTCGCTCCAAGCTGGGCTGTGTG
## CCGTTCAGCCCGACCGCTGCGCCTTATCCGGTAACTATCGTCTTGAGTCCAACCCGGTAAAGTAGGACAGGTGCCGGCAG
```

```
## CGCTCTGGGTCATTTTCGGCGAGGACCGCTTTCGCTGGAGATCGGCCTGTCGCTTGCGGTATTCGGAATCTTGCACGCCC
## TCGCTCAAGCCTTCGTCACTCCAAACGTTTCGGCGAGAAGCAGGCCATTATCGCCGGCATGGCGGCCGACGCGCTGGGCT
## GGCGTTCGCGACGCGAGGCTGGATGGCCTTCCCCATTATGATTCTTCTCGCTTCCGGCGGCCCGCGTTGCAGGCCATGCT
## GTCCAGGCAGGTAGATGACGACCATCAGGGACAGCTTCAACGGCTCTTACCAGCCTAACTTCGATCACTGGACCGCTGAT
## CGTCACGGCGATTTATGCCGCACATGGACGCGTTGCTGGCGTTTTTCCATAGGCTCCGCCCCCCTGACGAGCATCACAAA
## CAAGTCAGAGGTGGCGAAACCCGACAGGACTATAAAGATACCAGGCGTTTCCCCCTGGAAGCGCTCTCCTGTTCCGACCC
## TGCCGCTTACCGGATACCTGTCCGCCTTTCTCCCTTCGGGCTTTCTCAATGCTCACGCTGTAGGTATCTCAGTTCGGTGT
## AGGTCGTTCGCTCCAAGCTGACGAACCCCCCGTTCAGCCCGACCGCTGCGCCTTATCCGGTAACTATCGTCTTGAGTCCA
## ACACGACTTAACGGGTTGGCATGGATTGTAGGCGCCGCCCTATACCTTGTCTGCCTCCCCGCGGTGCATGGAGCCGGGCC
## ACCTCGACCTGAATGGAAGCCGGCGGCACCTCGCTAACGGCCAAGAATTGGAGCCAATCAATTCTTGCGGAGAACTGTGA
## ATGCGCAAACCAACCCTTGGCCATCGCGTCCGCCATCTCCAGCAGCCGCACGCGGCGCATCTCGGGCAGCGTTGGGTCCT
```

You may have noticed that we also have a number followed by a space earlier in the header, '103'. Why didn't the replacement end at that first match? The first instance is an example of *greedy* matching - it will take the longest possible string. To curtail this behaviour and use *lazy* matching - the shortest possible string - you can add the `?` quantifier.

```
str_remove(string = dino, pattern = ">.*?[0-9]\\s")
```

```
## nt 1-1200 GCGTTGCTGGCGTTTTTCCATAGGCTCCGCCCCCCTGACGAGCATCACAAAAATCGACGCGGTGGCGAAA
## CCCGACAGGACTATAAAGATACCAGGCGTTTCCCCCTGGAAGCTCCCTCGTGTTCCGACCCTGCCGCTTACCGGATACCT
## GTCCGCCTTTCTCCCTTCGGGAAGCGTGGCTGCTCACGCTGTACCTATCTCAGTTCGGTGTAGGTCGTTCGCTCCAAGCT
## GGGCTGTGTGCCGTTCAGCCCGACCGCTGCGCCTTATCCGGTAACTATCGTCTTGAGTCCAACCCGGTAAAGTAGGACAG
## GTGCCGGCAGCGCTCTGGGTCATTTTCGGCGAGGACCGCTTTCGCTGGAGATCGGCCTGTCGCTTGCGGTATTCGGAATC
## TTGCACGCCCTCGCTCAAGCCTTCGTCACTCCAAACGTTTCGGCGAGAAGCAGGCCATTATCGCCGGCATGGCGGCCGAC
## GCGCTGGGCTGGCGTTCGCGACGCGAGGCTGGATGGCCTTCCCCATTATGATTCTTCTCGCTTCCGGCGGCCCGCGTTGC
## AGGCCATGCTGTCCAGGCAGGTAGATGACGACCATCAGGGACAGCTTCAACGGCTCTTACCAGCCTAACTTCGATCACTG
## GACCGCTGATCGTCACGGCGATTTATGCCGCACATGGACGCGTTGCTGGCGTTTTTCCATAGGCTCCGCCCCCCTGACGA
## GCATCACAAACAAGTCAGAGGTGGCGAAACCCGACAGGACTATAAAGATACCAGGCGTTTCCCCCTGGAAGCGCTCTCCT
## GTTCCGACCCTGCCGCTTACCGGATACCTGTCCGCCTTTCTCCCTTCGGGCTTTCTCAATGCTCACGCTGTAGGTATCTC
## AGTTCGGTGTAGGTCGTTCGCTCCAAGCTGACGAACCCCCCGTTCAGCCCGACCGCTGCGCCTTATCCGGTAACTATCGT
## CTTGAGTCCAACACGACTTAACGGGTTGGCATGGATTGTAGGCGCCGCCCTATACCTTGTCTGCCTCCCCGCGGTGCATG
## GAGCCGGGCCACCTCGACCTGAATGGAAGCCGGCGGCACCTCGCTAACGGCCAAGAATTGGAGCCAATCAATTCTTGCGG
## AGAACTGTGAATGCGCAAACCAACCCTTGGCCATCGCGTCCGCCATCTCCAGCAGCCGCACGCGGCGCATCTCGGGCAGC
## GTTGGGTCCT
```

In this case, we want the greedy matching to replace the entire header. Let's save the dna into its own object.

```
dna <- str_remove(string = dino, pattern = ">.*[0-9]\\s")
```

```
## GCGTTGCTGGCGTTTTTCCATAGGCTCCGCCCCCCTGACGAGCATCACAAAAATCGACGCGGTGGCGAAACCCGACAGGA
## CTATAAAGATACCAGGCGTTTCCCCCTGGAAGCTCCCTCGTGTTCCGACCCTGCCGCTTACCGGATACCTGTCCGCCTTT
## CTCCCTTCGGGAAGCGTGGCTGCTCACGCTGTACCTATCTCAGTTCGGTGTAGGTCGTTCGCTCCAAGCTGGGCTGTGTG
## CCGTTCAGCCCGACCGCTGCGCCTTATCCGGTAACTATCGTCTTGAGTCCAACCCGGTAAAGTAGGACAGGTGCCGGCAG
## CGCTCTGGGTCATTTTCGGCGAGGACCGCTTTCGCTGGAGATCGGCCTGTCGCTTGCGGTATTCGGAATCTTGCACGCCC
## TCGCTCAAGCCTTCGTCACTCCAAACGTTTCGGCGAGAAGCAGGCCATTATCGCCGGCATGGCGGCCGACGCGCTGGGCT
## GGCGTTCGCGACGCGAGGCTGGATGGCCTTCCCCATTATGATTCTTCTCGCTTCCGGCGGCCCGCGTTGCAGGCCATGCT
## GTCCAGGCAGGTAGATGACGACCATCAGGGACAGCTTCAACGGCTCTTACCAGCCTAACTTCGATCACTGGACCGCTGAT
## CGTCACGGCGATTTATGCCGCACATGGACGCGTTGCTGGCGTTTTTCCATAGGCTCCGCCCCCCTGACGAGCATCACAAA
## CAAGTCAGAGGTGGCGAAACCCGACAGGACTATAAAGATACCAGGCGTTTCCCCCTGGAAGCGCTCTCCTGTTCCGACCC
## TGCCGCTTACCGGATACCTGTCCGCCTTTCTCCCTTCGGGCTTTCTCAATGCTCACGCTGTAGGTATCTCAGTTCGGTGT
## AGGTCGTTCGCTCCAAGCTGACGAACCCCCCGTTCAGCCCGACCGCTGCGCCTTATCCGGTAACTATCGTCTTGAGTCCA
## ACACGACTTAACGGGTTGGCATGGATTGTAGGCGCCGCCCTATACCTTGTCTGCCTCCCCGCGGTGCATGGAGCCGGGCC
## ACCTCGACCTGAATGGAAGCCGGCGGCACCTCGCTAACGGCCAAGAATTGGAGCCAATCAATTCTTGCGGAGAACTGTGA
## ATGCGCAAACCAACCCTTGGCCATCGCGTCCGCCATCTCCAGCAGCCGCACGCGGCGCATCTCGGGCAGCGTTGGGTCCT
```

Extracting:

We may also want to retain our header in a separate string. `str_extract` will retain the string that matches our pattern instead of removing it. We can save this in an object called header (I removed the final space from our expresssion).

```
header <- str_extract(string = dino, pattern = ">.*[0-9]")
header
```

```
## [1] ">DinoDNA from Crichton JURASSIC PARK  p. 103 nt 1-1200"
```

Searching:

Now we can look for patterns in our (dino) DNA!

Does this DNA have balanced GC content? We can use `str_extract_all` to capture every character that is either a G or a C.

```
str_extract_all(dino, pattern = "G|C")
```

```
## [[1]]
##   [1] "C" "C" "G" "C" "G" "G" "C" "G" "G" "C" "G" "C" "C" "G" "G" "C" "C"
##  [18] "C" "G" "C" "C" "C" "C" "C" "C" "G" "C" "G" "G" "C" "C" "C" "C" "G"
##  [35] "C" "G" "C" "G" "G" "G" "G" "C" "G" "C" "C" "C" "G" "C" "G" "G" "C"
##  [52] "G" "C" "C" "G" "G" "C" "G" "C" "C" "C" "C" "C" "G" "G" "G" "C" "C"
##  [69] "C" "C" "C" "G" "G" "C" "C" "G" "C" "C" "C" "G" "C" "C" "G" "C" "C"
##  [86] "C" "G" "G" "C" "C" "G" "C" "C" "G" "C" "C" "C" "C" "C" "C" "C" "G"
## [103] "G" "G" "G" "C" "G" "G" "G" "C" "G" "C" "C" "C" "G" "C" "G" "C" "C"
## [120] "C" "C" "G" "C" "G" "G" "G" "G" "G" "C" "G" "C" "G" "C" "C" "C" "G"
## [137] "C" "G" "G" "G" "C" "G" "G" "G" "C" "C" "G" "C" "G" "C" "C" "C" "G"
## [154] "C" "C" "G" "C" "G" "C" "G" "C" "C" "C" "C" "G" "G" "C" "C" "G" "C"
## [171] "G" "G" "C" "C" "C" "C" "C" "G" "G" "G" "G" "G" "C" "G" "G" "G" "C"
## [188] "C" "G" "G" "C" "G" "C" "G" "C" "C" "G" "G" "G" "C" "C" "G" "G" "C"
## [205] "G" "G" "G" "C" "C" "G" "C" "C" "G" "C" "G" "G" "G" "C" "G" "G" "C"
## [222] "C" "G" "C" "G" "C" "G" "C" "G" "G" "C" "G" "G" "C" "G" "C" "C" "G"
## [239] "C" "C" "C" "C" "G" "C" "C" "G" "C" "C" "C" "G" "C" "C" "C" "C" "C"
## [256] "G" "C" "G" "G" "C" "G" "G" "G" "C" "G" "G" "C" "C" "C" "G" "C" "C"
## [273] "G" "G" "C" "G" "G" "C" "G" "G" "C" "C" "G" "C" "G" "C" "G" "C" "G"
## [290] "G" "G" "C" "G" "G" "C" "G" "C" "G" "C" "G" "C" "G" "C" "G" "G" "G"
## [307] "C" "G" "G" "G" "G" "C" "C" "C" "C" "C" "C" "G" "C" "C" "C" "G" "C"
## [324] "C" "C" "G" "G" "C" "G" "G" "C" "C" "C" "G" "C" "G" "G" "C" "G" "G"
## [341] "C" "C" "G" "C" "G" "C" "C" "G" "G" "C" "G" "G" "G" "G" "C" "G" "C"
## [358] "C" "C" "G" "G" "G" "C" "G" "C" "C" "C" "G" "G" "C" "C" "C" "C" "G"
## [375] "C" "C" "C" "C" "G" "C" "C" "G" "G" "C" "C" "G" "C" "G" "C" "G" "C"
## [392] "C" "G" "G" "C" "G" "G" "C" "C" "G" "C" "C" "G" "G" "C" "G" "C" "G"
## [409] "G" "C" "G" "G" "C" "G" "C" "C" "G" "G" "C" "C" "C" "G" "C" "C" "C"
## [426] "C" "C" "C" "G" "C" "G" "G" "C" "C" "C" "C" "G" "C" "G" "G" "G" "G"
## [443] "G" "C" "G" "C" "C" "C" "G" "C" "G" "G" "C" "G" "C" "C" "G" "G" "C"
## [460] "G" "C" "C" "C" "C" "C" "G" "G" "G" "C" "G" "C" "C" "C" "C" "G" "C"
## [477] "C" "G" "C" "C" "C" "G" "C" "C" "G" "C" "C" "C" "G" "G" "C" "C" "G"
## [494] "C" "C" "G" "C" "C" "C" "C" "C" "C" "C" "C" "G" "G" "G" "C" "C" "C" "G"
## [511] "C" "C" "C" "G" "C" "G" "G" "G" "C" "C" "G" "C" "G" "G" "G" "G" "G"
## [528] "C" "G" "C" "G" "C" "C" "C" "G" "C" "G" "C" "G" "C" "C" "C" "C" "C"
## [545] "C" "G" "C" "G" "C" "C" "C" "G" "C" "C" "G" "C" "G" "C" "G" "C" "C"
## [562] "C" "C" "G" "G" "C" "C" "G" "C" "G" "G" "C" "C" "C" "C" "G" "C" "C"
## [579] "G" "G" "G" "G" "G" "C" "G" "G" "G" "G" "G" "C" "G" "C" "C" "G" "C"
## [596] "C" "C" "C" "C" "G" "C" "G" "C" "C" "C" "C" "C" "C" "C" "G" "C" "G" "G"
## [613] "G" "C" "G" "G" "G" "C" "C" "G" "G" "G" "C" "C" "C" "C" "C" "G" "C"
## [630] "C" "G" "G" "G" "G" "C" "C" "G" "G" "C" "G" "G" "C" "C" "C" "C" "G"
```

```
## [647] "C" "C" "G" "G" "C" "C" "G" "G" "G" "G" "C" "C" "C" "C" "G" "C" "G"
## [664] "G" "G" "C" "G" "G" "G" "C" "G" "C" "C" "C" "C" "C" "C" "G" "G" "C"
## [681] "C" "C" "G" "C" "G" "C" "C" "G" "C" "C" "C" "C" "C" "G" "C" "G" "C"
## [698] "C" "G" "C" "C" "G" "C" "G" "G" "C" "G" "C" "C" "C" "G" "G" "G" "C"
## [715] "G" "C" "G" "G" "G" "G" "C" "C"
```

The output is a list object in which is stored an entry for each G or C extracted. We count the number of occurences of G and C using `str_count` and divide by the total number of characters in our string to get the %GC content.

```
str_count(dna, pattern = "G|C")/str_length(dna) * 100
```

```
## [1] 60
```

Let's translate this into mRNA!

Replacement:

To replace one pattern or multiple patterns with one replacement, `str_replace_all` can be used.

However, to replace multiple patterns at once with multiple replacements, we need another package. `str_replace_all` works iteratively, which means a replacement can be replaced. `mgsub` from the `mgsub` package averts this behaviour by performing simultaneous replacement.

```
# all G replaced with C - one pattern, one replacement
str_replace_all(dna, pattern = "G", replacement = "C")
```

```
## [1] "CCCTTCCTCCCCTTTTTCCATACCCTCCCCCCCCTCACCACCATCACAAAAATCCACCCCCTCCCCAAACCCCACACCACTATAAACATACCACC
```

```
# all AGT replaced with C - multiple patterns, one
# replacement
str_replace_all(dna, pattern = "[AGT]", replacement = "C")
```

```
## [1] "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
```

```
# multiple patterns with multiple replacement works
# iteratively, G is replaced with C, but then ALL C get
# replaced with G, as such, our replacement was replaced
str_replace_all(dna, c(G = "C", C = "G", A = "U", T = "A"))
```

```
## [1] "GGGAAGGAGGGGAAAAAGGUAUGGGAGGGGGGGGAGUGGUGGUAGUGUUUUUAGGUGGGGGAGGGGUUUGGGGUGUGGUGAUAUUUGUAUGGUGG
```

```
# multiple patterns with multiple replacements working
# simultaneously - safe!
mrna <- mgsub::mgsub(dna, pattern = c("G", "C", "A", "T"), replacement = c("C",
    "G", "U", "A"))

mrna
```

```
## [1] "CGCAACGACCGCAAAAAGGUAUCCGAGGCGGGGGACUGCUCGUAGUGUUUUUAGCUGCGCCACCGCUUUGGGCUGUCCUGAUAUUUCUAUGGUCC
```

More Searching:

Is there even a start codon in this thing? `str_detect` can be used to get a local (TRUE or FALSE) answer to whether or not a match is found.

```
str_detect(mrna, "AUG")
```

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

It might be more useful to know exactly how many possible start codons we have. `str_count` will count the number of matches in the string for our pattern.

```r
str_count(mrna, "AUG")
```

```
## [1] 9
```

To get the position of a possible start codon we can use `str_locate` which will return the indices of where the start and end of our substring occurs (`str_locate_all` can be used to find the all possible locations).

```r
str_locate(mrna, "AUG")
```

```
##      start end
## [1,]    90  92
```

Splitting:

Let's split this string into substrings of codons, starting at the position of our start codon. We have the position of our start codon from `str_locate`. We can use `str_sub` to subset the string by position (we will just go to the end of the string for now).

```r
str_sub(mrna, 90, 1200)
# is equivalent to
mrna <- str_sub(mrna, str_locate(mrna, "AUG")[1])
```

```
## AUGGUCCGCAAAGGGGGACCUUCGAGGGAGCACAAGGCUGGGACGGCGAAUGGCCUAUGGACAGGCGGAAAGAGGGAAGC
## CCUUCGCACCGACGAGUGCGACAUGGAUAGAGUCAAGCCACAUCCAGCAAGCGAGGUUCGACCCGACACACGGCAAGUCG
## GGCUGGCGACGCGGAAUAGGCCAUUGAUAGCAGAACUCAGGUUGGGCCAUUUCAUCCUGUCCACGGCCGUCGCGAGACCC
## AGUAAAAGCCGCUCCUGGCGAAAGCGACCUCUAGCCGGACAGCGAACGCCAUAAGCCUUAGAACGUGCGGGAGCGAGUUC
## GGAAGCAGUGAGGUUUGCAAAGCCGCUCUUCGUCCGGUAAUAGCGGCCGUACCGCCGGCUGCGCGACCCGACCGCAAGCG
## CUGCGCUCCGACCUACCGGAAGGGGUAAUACUAAGAAGAGCGAAGGCCGCCGGGCGCAACGUCCGGUACGACAGGUCCGU
## CCAUCUACUGCUGGUAGUCCCUGUCGAAGUUGCCGAGAAUGGUCGGAUUGAAGCUAGUGACCUGGCGACUAGCAGUGCCG
## CUAAAUACGGCGUGUACCUGCGCAACGACCGCAAAAAGGUAUCCGAGGCGGGGGGACUGCUCGUAGUGUUUGUUCAGUCU
## CCACCGCUUUGGGCUGUCCUGAUAUUUCUAUGGUCCGCAAAGGGGGACCUUCGCGAGAGGACAAGGCUGGGACGGCGAAU
## GGCCUAUGGACAGGCGGAAAGAGGGAAGCCCGAAAGAGUUACGAGUGCGACAUCCAUAGAGUCAAGCCACAUCCAGCAAG
## CGAGGUUCGACUGCUUGGGGGGCAAGUCGGGCUGGCGACGCGGAAUAGGCCAUUGAUAGCAGAACUCAGGUUGUGCUGAA
## UUGCCCAACCGUACCUAACAUCCGCGGCGGGAUAUGGAACAGACGGAGGGGCGCCACGUACCUCGGCCCGGUGGAGCUGG
## ACUUACCUUCGGCCGCCGUGGAGCGAUUGCCGGUUCUUAACCUCGGUUAGUUAAGAACGCCUCUUGACACUUACGCGUUU
## GGUUGGGAACCGGUAGCGCAGGCGGUAGAGGUCGUCGGCGUGCGCCGCGUAGAGCCCGUCGCAACCCAGGA
```

We can get codons by extracting groups of (any) 3 nucleotides/characters in our reading frame.

```r
str_extract_all(mrna, "...")
```

```
## [[1]]
##   [1] "CGC" "AAC" "GAC" "CGC" "AAA" "AAG" "GUA" "UCC" "GAG" "GCG" "GGG"
##  [12] "GGA" "CUG" "CUC" "GUA" "GUG" "UUU" "UUA" "GCU" "GCG" "CCA" "CCG"
##  [23] "CUU" "UGG" "GCU" "GUC" "CUG" "AUA" "UUU" "CUA" "UGG" "UCC" "GCA"
##  [34] "AAG" "GGG" "GAC" "CUU" "CGA" "GGG" "AGC" "ACA" "AGG" "CUG" "GGA"
##  [45] "CGG" "CGA" "AUG" "GCC" "UAU" "GGA" "CAG" "GCG" "GAA" "AGA" "GGG"
##  [56] "AAG" "CCC" "UUC" "GCA" "CCG" "ACG" "AGU" "GCG" "ACA" "UGG" "AUA"
##  [67] "GAG" "UCA" "AGC" "CAC" "AUC" "CAG" "CAA" "GCG" "AGG" "UUC" "GAC"
##  [78] "CCG" "ACA" "CAC" "GGC" "AAG" "UCG" "GGC" "UGG" "CGA" "CGC" "GGA"
##  [89] "AUA" "GGC" "CAU" "UGA" "UAG" "CAG" "AAC" "UCA" "GGU" "UGG" "GCC"
## [100] "AUU" "UCA" "UCC" "UGU" "CCA" "CGG" "CCG" "UCG" "CGA" "GAC" "CCA"
## [111] "GUA" "AAA" "GCC" "GCU" "CCU" "GGC" "GAA" "AGC" "GAC" "CUC" "UAG"
## [122] "CCG" "GAC" "AGC" "GAA" "CGC" "CAU" "AAG" "CCU" "UAG" "AAC" "GUG"
## [133] "CGG" "GAG" "CGA" "GUU" "CGG" "AAG" "CAG" "UGA" "GGU" "UUG" "CAA"
## [144] "AGC" "CGC" "UCU" "UCG" "UCC" "GGU" "AAU" "AGC" "GGC" "CGU" "ACC"
## [155] "GCC" "GGC" "UGC" "GCG" "ACC" "CGA" "CCG" "CAA" "GCG" "CUG" "CGC"
## [166] "UCC" "GAC" "CUA" "CCG" "GAA" "GGG" "GUA" "AUA" "CUA" "AGA" "AGA"
```

```
## [177] "GCG" "AAG" "GCC" "GCC" "GGG" "CGC" "AAC" "GUC" "CGG" "UAC" "GAC"
## [188] "AGG" "UCC" "GUC" "CAU" "CUA" "CUG" "CUG" "GUA" "GUC" "CCU" "GUC"
## [199] "GAA" "GUU" "GCC" "GAG" "AAU" "GGU" "CGG" "AUU" "GAA" "GCU" "AGU"
## [210] "GAC" "CUG" "GCG" "ACU" "AGC" "AGU" "GCC" "GCU" "AAA" "UAC" "GGC"
## [221] "GUG" "UAC" "CUG" "CGC" "AAC" "GAC" "CGC" "AAA" "AAG" "GUA" "UCC"
## [232] "GAG" "GCG" "GGG" "GGA" "CUG" "CUC" "GUA" "GUG" "UUU" "GUU" "CAG"
## [243] "UCU" "CCA" "CCG" "CUU" "UGG" "GCU" "GUC" "CUG" "AUA" "UUU" "CUA"
## [254] "UGG" "UCC" "GCA" "AAG" "GGG" "GAC" "CUU" "CGC" "GAG" "AGG" "ACA"
## [265] "AGG" "CUG" "GGA" "CGG" "CGA" "AUG" "GCC" "UAU" "GGA" "CAG" "GCG"
## [276] "GAA" "AGA" "GGG" "AAG" "CCC" "GAA" "AGA" "GUU" "ACG" "AGU" "GCG"
## [287] "ACA" "UCC" "AUA" "GAG" "UCA" "AGC" "CAC" "AUC" "CAG" "CAA" "GCG"
## [298] "AGG" "UUC" "GAC" "UGC" "UUG" "GGG" "GGC" "AAG" "UCG" "GGC" "UGG"
## [309] "CGA" "CGC" "GGA" "AUA" "GGC" "CAU" "UGA" "UAG" "CAG" "AAC" "UCA"
## [320] "GGU" "UGU" "GCU" "GAA" "UUG" "CCC" "AAC" "CGU" "ACC" "UAA" "CAU"
## [331] "CCG" "CGG" "CGG" "GAU" "AUG" "GAA" "CAG" "ACG" "GAG" "GGG" "CGC"
## [342] "CAC" "GUA" "CCU" "CGG" "CCC" "GGU" "GGA" "GCU" "GGA" "CUU" "ACC"
## [353] "UUC" "GGC" "CGC" "CGU" "GGA" "GCG" "AUU" "GCC" "GGU" "UCU" "UAA"
## [364] "CCU" "CGG" "UUA" "GUU" "AAG" "AAC" "GCC" "UCU" "UGA" "CAC" "UUA"
## [375] "CGC" "GUU" "UGG" "UUG" "GGA" "ACC" "GGU" "AGC" "GCA" "GGC" "GGU"
## [386] "AGA" "GGU" "CGU" "CGG" "CGU" "GCG" "CCG" "CGU" "AGA" "GCC" "CGU"
## [397] "CGC" "AAC" "CCA" "GGA"
```

The codons are extracted into a list, but we can get our character substrings using `unlist`.

```
codons <- unlist(str_extract_all(mrna, "..."))
```

We now have a vector with 400 codons.

Do we have a stop codon in our reading frame? Let's check with `str_detect`. We can use round brackets `( )` to separately group the different stop codons.

```
str_detect(codons, "(UAG)|(UGA)|(UAA)")
```

```
##   [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [23] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [34] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [45] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [56] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [67] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [78] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [89] FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
## [100] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [111] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
## [122] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
## [133] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
## [144] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [155] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [166] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [177] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [188] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [199] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [210] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [221] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [232] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [243] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
## [254] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [265] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [276] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [287] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [298] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [309] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE
## [320] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
## [331] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [342] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [353] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
## [364] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
## [375] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [386] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [397] FALSE FALSE FALSE FALSE
```

Looks like we have many matches. We can subset the codons using `str_detect` (instances where the presence of a stop codon is equal to TRUE) to see which stop codons are represented. We can use the `which` function to find the which indices the stop codons are positioned at.

```
which(str_detect(codons, "(UAG)|(UGA)|(UAA)") == TRUE)
```

```
##  [1]  92  93 121 130 140 315 316 329 363 372
```

Let's subset our codon strings to end at the first stop codon.

```
translation <- codons[1:98]

# equivalent to
translation <- codons[1:which(str_detect(codons, "(UAG)|(UGA)|(UAA)") ==
    TRUE)[1]]
```

More Replacing:

After finding our unique codons, we can translate codons into their respective proteins by using `mgsub` using multiple patterns and replacements as before.

```
unique(translation)
```

```
##  [1] "CGC" "AAC" "GAC" "AAA" "AAG" "GUA" "UCC" "GAG" "GCG" "GGG" "GGA"
## [12] "CUG" "CUC" "GUG" "UUU" "UUA" "GCU" "CCA" "CCG" "CUU" "UGG" "GUC"
## [23] "AUA" "CUA" "GCA" "CGA" "AGC" "ACA" "AGG" "CGG" "AUG" "GCC" "UAU"
## [34] "CAG" "GAA" "AGA" "CCC" "UUC" "ACG" "AGU" "UCA" "CAC" "AUC" "CAA"
## [45] "GGC" "UCG" "CAU" "UGA"
```

```
translation <- mgsub::mgsub(translation, pattern = c("AUG", "GUC",
    "CGC", "AAA", "GGG", "GGA", "CCU", "UCG", "AGG", "GAG", "CAC",
    "AAG", "GCU", "ACG", "GCG", "AAU", "GGC", "CUA", "UGG", "ACA",
    "GAA", "GCC", "CUU", "ACC", "GAC", "UGC", "GAU", "AGA", "CCA",
    "CAU", "GCA", "AGC", "GUU", "CGA", "CCC", "CGG", "CAA", "CUG",
    "UUG", "AUA", "CUC", "UUC", "AUC", "UCC", "AGU", "CAG", "UAA"),
    replacement = c("M", "V", "R", "K", "G", "G", "P", "S", "R",
        "E", "H", "K", "A", "T", "A", "N", "G", "L", "W", "T",
        "E", "A", "L", "T", "D", "C", "D", "R", "P", "H", "A",
        "S", "V", "R", "P", "R", "Q", "L", "L", "I", "L", "F",
        "I", "S", "S", "Q", ""))

translation
```

```
##  [1] "R"   "AAC" "D"   "R"   "K"   "K"   "GUA" "S"   "E"   "A"   "G"
```

```
## [12] "G"   "L"   "L"   "GUA" "GUG" "UUU" "UUA" "A"   "A"   "P"   "CCG"
## [23] "L"   "W"   "A"   "V"   "L"   "I"   "UUU" "L"   "W"   "S"   "A"
## [34] "K"   "G"   "D"   "L"   "R"   "G"   "S"   "T"   "R"   "L"   "G"
## [45] "R"   "R"   "M"   "A"   "UAU" "G"   "Q"   "A"   "E"   "R"   "G"
## [56] "K"   "P"   "F"   "A"   "CCG" "T"   "S"   "A"   "T"   "W"   "I"
## [67] "E"   "UCA" "S"   "H"   "I"   "Q"   "Q"   "A"   "R"   "F"   "D"
## [78] "CCG" "T"   "H"   "G"   "K"   "S"   "G"   "W"   "R"   "R"   "G"
## [89] "I"   "G"   "H"   "UGA"
```

Combining:

What is our final protein string? `str_flatten` allows us to collapse our individual protein strings into one long string.

```
translation <- str_flatten(translation)
translation
```

```
## [1] "RAACDRKKGUASEAGGLLGUAGUGUUUUUAAAPCCGLWAVLIUUULWSAKGDLRGSTRLGRRMAUAUGQAERGKPFACCGTSATWIEUCASHIQQ
```

We can add our header back using `str_c`, which allows us to combine strings. We can use a space to separate our original strings.

```
str_c(header, translation, sep = " ")
```

```
## >DinoDNA from Crichton JURASSIC PARK  p. 103 nt 1-1200 RAACDRKKGUASEAGGLLGUAGUGU
## UUUUAAAPCCGLWAVLIUUULWSAKGDLRGSTRLGRRMAUAUGQAERGKPFACCGTSATWIEUCASHIQQARFDCCGTHG
## KSGWRRGIGHUGA
```

A Real Messy Dataset

I looked for a messy dataset for data cleaning and found it in a blog titled:
"Biologists: this is why bioinformaticians hate you. . . "

Challenge:

This is Wellcome Trust APC dataset on the costs of open access publishing by providing article processing charge (APC) data.

https://figshare.com/articles/Wellcome_Trust_APC_spend_2012_13_data_file/963054

The main and common issue with this dataset is that when data entry was done there was no *structured vocabulary*; people could type whatever they wanted into free text answer boxes instead of using dropdown menus with limited options, giving an error if something is formatted incorrectly, or stipulating some rules (ie. must be all lowercase, uppercase, no numbers, spacing, etc).

I must admit I have been guilty of messing with people who have made databases without rules. For example, giving an emergency contact, there was a line to input 'Relationship', which could easily have been a dropdown menu: 'parent, partner, friend, other'. Instead I was allowed to write in a free text box 'lifelong kindred spirit, soulmate and doggy-daddy'. I don't think anyone here was trying to be a nuisance, this messy data is just a consequence of poor data collection.

What I want to know is:

1. List 3 problems with this dataset that require data cleaning.
2. What is the number of publications by PLOS One in dataset?

3. What is the median cost of publishing with Elsevier?

The route I suggest to take in answering these question is:

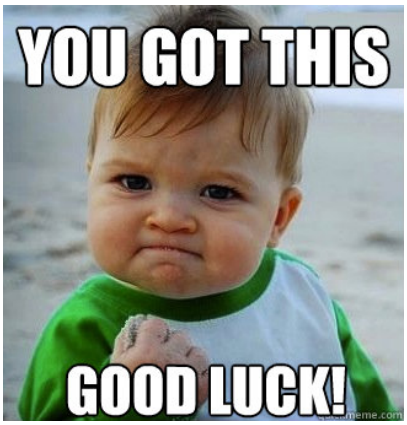

Figure 6:

- Inspect your dataset. Are the data types what you expect?
- Identify any immediate problems. (Answer Question #1)
- Clean up column names.
- Data clean the publisher column for publications from Elsevier and PLOS One by:
  - converting all entries to lowercase
  - correcting typos
  - correcting multiple names for a publisher to one name
  - removing newline characters and trailing whitespace
- Answer Questions #2-3

The dataset doesn't need to be perfect. No datasets are 100% clean. Just do what you gotta do to answer these questions.

Resources:
http://stat545.com/block022_regular-expression.html
http://stat545.com/block027_regular-expressions.html
http://stat545.com/block028_character-data.html
http://r4ds.had.co.nz/strings.html http://www.gastonsanchez.com/Handling_and_Processing_Strings_in_R.pdf
http://www.opiniomics.org/biologists-this-is-why-bioinformaticians-hate-you/
https://figshare.com/articles/Wellcome_Trust_APC_spend_2012_13_data_file/963054 >
http://emailregex.com/
https://regex101.com/
https://regexr.com/
https://www.regular-expressions.info/backref.html
https://www.regular-expressions.info/unicode.html
https://www.rstudio.com/wp-content/uploads/2016/09/RegExCheatsheet.pdf
https://simongoring.shinyapps.io/RegularExpressionR/
https://github.com/bmewing/mgsub

Thanks for coming!!!

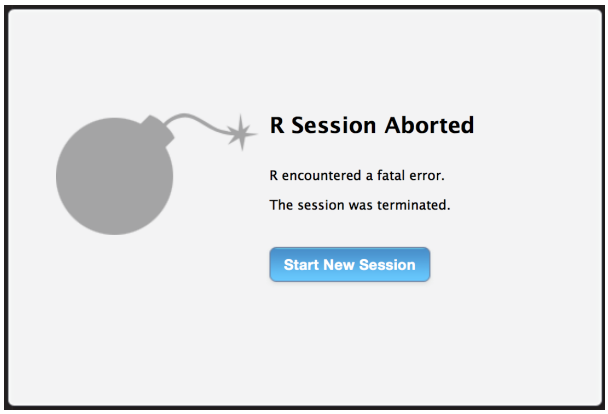
R Session Aborted
R encountered a fatal error.
The session was terminated.
Start New Session

Figure 7: