

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

Contents

A quick intro to the intro to R Lesson Series	1
Data Cleaning or Data Munging or Data Wrangling	3
Intro to regular expressions	3
Classes	4
Quantifiers	4
Operators	4
Matching by position	5
Escape characters	5
Intro to string manipulation with stringr	6
A Real Messy Dataset	14
Post-Lesson Assessment	16

A quick intro to the intro to R Lesson Series

This ‘Intro to R Lesson Series’ is brought to you by the Centre for the Analysis of Genome Evolution & Function’s (CAGEF) bioinformatics training initiative. This course was developed based on feedback on the needs and interests of the Department of Cell & Systems Biology and the Department of Ecology and Evolutionary Biology.

This lesson is the fourth in a 6-part series. The idea is that at the end of the series, you will be able to import and manipulate your data, make exploratory plots, perform some basic statistical tests, and test a regression model.

How do we get there? Today we are going to be learning data cleaning and string manipulation; this is really the battleground of coding - getting your data into the format where you can analyse it. In the next lesson we will learn how to do t-tests and perform regression and modeling in R.

The structure of the class is a code-along style. It is hands on. The lecture AND code we are going through are available on GitHub for download at <https://github.com/eacton/CAGEF>, so you can spend the time coding and not taking notes. As we go along, there will be some challenge questions and multiple choice questions on Socrative. At the end of the class if you could please fill out a post-lesson survey

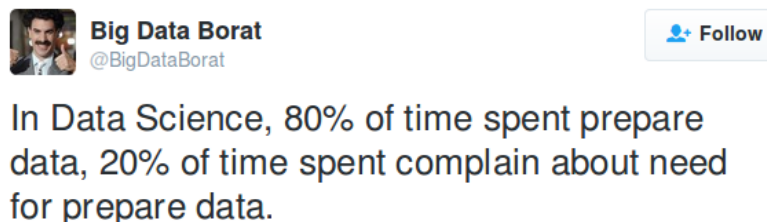


Figure 1:

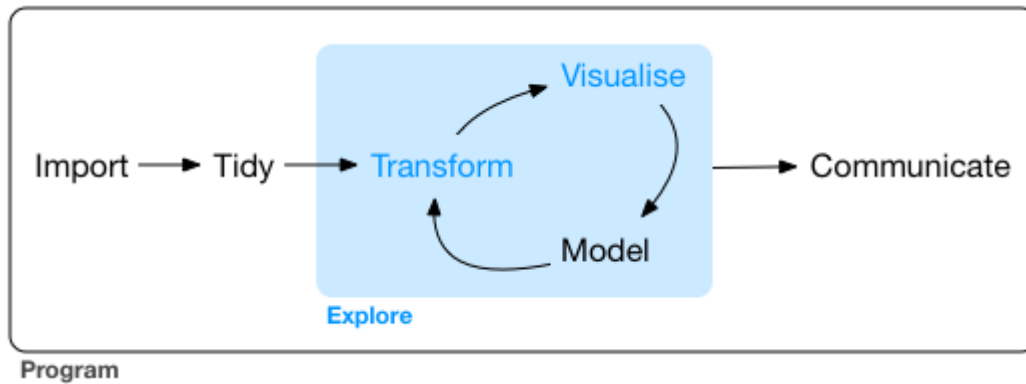


Figure 2:

(<https://www.surveymonkey.com/r/PVHDKDB>), it will help me further develop this course and would be greatly appreciated.

Packages Used in This Lesson

The following packages are used in this lesson:

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

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

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.

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

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")
```

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!

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:

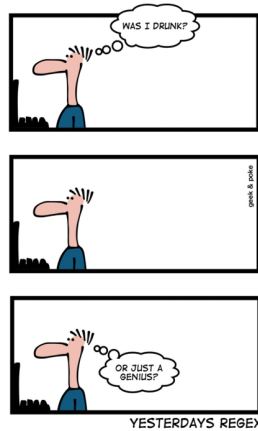


Figure 3:

<https://regex101.com/>

<https://regexpr.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.').

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

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)

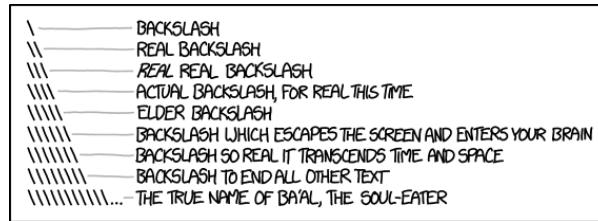


Figure 4: Joking/Not Joking (xkcd)

Expression	Meaning
()	grouping - used for backreferencing

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

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 (“\\\$”).

Expression	Meaning
\	escape for meta-characters to be used as characters (*, \$, ^, ., ?, , \, [,], {, }, (,)).
Note: the back- slash is also a meta- character.	

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 is that it is okay to google and use resources early on for regex, and that even experts still use these resources.

The Internet will do the remembering for you



Googling for the Regex

Every. Damn. Time.

ORLY?

@ThePracticalDev

Figure 5:

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 GCGTTGCTGGCGTTTTTCCATAGGCTCCGCCCCCTGACGAGCATTACGAGCCTGCGGCTTACCGGATACCTGTCCGCCTTTCTCCCTTCGGAAGCGTGGCTGCTC  
## >DinoDNA from Crichton JURASSIC PARK p. 103 nt 1-1200 GCGTTGCTGGCGTTTTTCCATAGGC  
## TCGCCCCCTGACGAGCATCACAAAATCGACGCGTGGCGAAACCCGACAGGACTATAAAGATACCAGGCGTTTCCCC  
## CTGGAAGCTCCCTCGTGTTCGACCCTGCCGCTTACCGGATACCTGTCCGCCTTTCTCCCTTCGGAAGCGTGGCTGCTC  
## ACGCTGTACCTATCTCAGTTCGGTGTAGGTCGTTTCGCTCCAAGCTGGGCTGTGTGCCGTTACGCCCAGCGCTGCGCCTT  
## ATCCGGTAACTATCGTCTTGAGTCCAACCCGTAAGTAGGACAGGTGCCGCGAGCGCTCTGGGTCATTTTCGGCGAGGA  
## CCGCTTTCGCTGGAGATCGGCCTGTCGCTTGGGTATTTCGAATCTTGCACGCCCTCGCTCAAGCCTTCGTCACTCCAAA  
## CGTTTCGGCGAGAAGCAGGCCATTATCGCCGCGATGGCGGCCGACGCGCTGGGCTGGCGTTCGCGACGCGAGGCTGGATG  
## GCCTTCCCCATTATGATTCTTCTCGCTTCCGGCGGCCCGGCTTGCAGGCCATGCTGTCCAGGCAGGTAGATGACGACCAT  
## CAGGGACAGCTTCAACGGCTCTTACCAGCCTAACTTCGATCACTGGACCGCTGATCGTCACGGCGATTTATGCCGCACAT  
## GGACGCGTTGCTGGCGTTTTTCCATAGGCTCCGCCCCCTGACGAGCATCACAAACAAGTCAGAGGTGGCGAAACCCGAC  
## AGGACTATAAAGATACCAGGCGTTTCCCCCTGGAAGCGCTCTCCTGTTCCGACCTGCCGCTTACCGGATACCTGTCCGC  
## CTTTCTCCCTTCGGGCTTTCTCAATGCTCACGCTGTAGGTATCTCAGTTCGGTGTAGGTCGTTTCGCTCCAAGCTGACGAA  
## CCCCCGTTTCAGCCGACCGCTGCGCCTTATCCGGTAACATATCGTCTTGAGTCCAACACGACTTAACGGGTGGCATGGA  
## TTGTAGGCGCCGCCCTATACCTTGCTGCCTCCCGCGGTGCATGGAGCCGGGCCACCTCGACCTGAATGGAAGCCGGCG  
## GCACCTCGTAACGGCCAAGAATTGGAGCCAATCAATTCTTGCAGGAGAACTGTGAATGCGCAACCAACCTTGGCCATC  
## GCGTCCGCATCTCCAGCAGCCGACGCGGCGCATCTCGGGCAGCGTTGGGTCCT
```

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 6: 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
## CCGCCCCCTGACGAGCATCACAAAAATCGACGCGGTGGCGAAACCCGACAGGACTATAAAGATACCAGGCGTTTCCCCC
## TGGAAGCTCCCTCGTGTTCGACCCCTGCCGCTTACCGGATACCTGTCCGCCTTTCTCCCTTCGGGAAGCGTGGCTGCTCA
## CGCTGTACCTATCTCAGTTCGGTGTAGGTCGTTTCGCTCCAAGCTGGGCTGTGTGCCGTTTCAGCCCGACCGCTGCGCCTTA
## TCCGGTAACTATCGTCTTGAGTCCAACCCGGTAAAGTAGGACAGGTGCCGGCAGCGCTCTGGGTCATTTTCGGCGAGGAC
## CGCTTTTCGCTGGAGATCGGCCTGTCGCTTGCGGTATTTCGGAATCTTGACGCCCTCGCTCAAGCCTTCGTCACTCCAAAC
## GTTTCGGCGAGAAGCAGGCCATTATCGCCGGCATGGCGGCCGACGCGCTGGGCTGGCGTTTCGCGACGCGAGGCTGGATGG
## CCTTCCCCATTATGATTCTTCTCGCTTCCGGCGGCCCGGTTGCAGGCCATGCTGTCCAGGCAGGTAGATGACGACCATC
## AGGGACAGCTTCAACGGCTCTTACCAGCCTAACTTCGATCACTGGACCGCTGATCGTCACGGCGATTTATGCCGCACATG
## GACGCGTTGCTGGCGTTTTTCCATAGGCTCCGCCCCCTGACGAGCATCACAAACAAGTCAGAGGTGGCGAAACCCGACA
## GGAATAAAGATACCAGGCGTTTCCCCCTGGAAGCGCTCTCCTGTTCCGACCCCTGCCGCTTACCGGATACCTGTCCGCC
## TTTCTCCCTTCGGGCTTTCTCAATGCTCAGCTGTAGGTATCTCAGTTCGGTGTAGGTGCTTCGCTCCAAGCTGACGAAC
## CCCCCGTTACGCCGACCGCTGCGCCTTATCCGGTAACTATCGTCTTGAGTCCAACACGACTTAACGGGTTGGCATGGAT
## TGTAGGCGCCGCCCTATACCTTGCTGCCTCCCCGCGGTGCATGGAGCCGGGCCACCTCGACCTGAATGGAAGCCGGCGG
## CACCTCGCTAACGGCCAAGAATTGGAGCCAATCAATTCTTGCAGGAACTGTGAATGCGCAAACCAACCCTTGGCCATCG
## CGTCCGCCATCTCCAGCAGCCGACGCGGCGCATCTCGGGCAGCGTTGGGTCCT
```

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
## CCGCCCCCTGACGAGCATCACAAAAATCGACGCGGTGGCGAAACCCGACAGGACTATAAAGATACCAGGCGTTTCCCCC
## TGGAAGCTCCCTCGTGTTCGACCCCTGCCGCTTACCGGATACCTGTCCGCCTTTCTCCCTTCGGGAAGCGTGGCTGCTCA
## CGCTGTACCTATCTCAGTTCGGTGTAGGTCGTTTCGCTCCAAGCTGGGCTGTGTGCCGTTTCAGCCCGACCGCTGCGCCTTA
## TCCGGTAACTATCGTCTTGAGTCCAACCCGGTAAAGTAGGACAGGTGCCGGCAGCGCTCTGGGTCATTTTCGGCGAGGAC
## CGCTTTTCGCTGGAGATCGGCCTGTCGCTTGCGGTATTTCGGAATCTTGACGCCCTCGCTCAAGCCTTCGTCACTCCAAAC
## GTTTCGGCGAGAAGCAGGCCATTATCGCCGGCATGGCGGCCGACGCGCTGGGCTGGCGTTTCGCGACGCGAGGCTGGATGG
## CCTTCCCCATTATGATTCTTCTCGCTTCCGGCGGCCCGGTTGCAGGCCATGCTGTCCAGGCAGGTAGATGACGACCATC
## AGGGACAGCTTCAACGGCTCTTACCAGCCTAACTTCGATCACTGGACCGCTGATCGTCACGGCGATTTATGCCGCACATG
## GACGCGTTGCTGGCGTTTTTCCATAGGCTCCGCCCCCTGACGAGCATCACAAACAAGTCAGAGGTGGCGAAACCCGACA
## GGAATAAAGATACCAGGCGTTTCCCCCTGGAAGCGCTCTCCTGTTCCGACCCCTGCCGCTTACCGGATACCTGTCCGCC
## TTTCTCCCTTCGGGCTTTCTCAATGCTCAGCTGTAGGTATCTCAGTTCGGTGTAGGTGCTTCGCTCCAAGCTGACGAAC
## CCCCCGTTACGCCGACCGCTGCGCCTTATCCGGTAACTATCGTCTTGAGTCCAACACGACTTAACGGGTTGGCATGGAT
```

```
## TGTAGGCGCCGCCCTATACCTTGTCTGCCTCCCCGGGTGCATGGAGCCGGGCCACCTCGACCTGAATGGAAGCCGGCGG
## CACCTCGCTAACGCCAAGAATTGGAGCCAATCAATTCTTGC GGAGAACTGTGAATGCGCAAACCAACCCTTGGCCATCG
## 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
## TCCCTCGTGTTCGACCTGCCGCTTACCGGATACCTGTCCGCCTTTCTCCCTTCGGGAAGCGTGGCTGCTCAGCTGTA
## CCTATCTCAGTTTCGGTGTAGGTGCTTGCCTCCAAGCTGGGCTGTGTGCCGTTAGCCCGACCGCTGCGCCTTATCCGTA
## ACTATCGTCTTGAGTCCAACCCGGTAAAGTAGGACAGGTGCCGGCAGCGCTCTGGGTCATTTTCGGCGAGGACCGCTTC
## GCTGGAGATCGGCCTGTGCTTGCCTTATCGGAATCTTGCACGCCCTCGCTCAAGCCTTCGCTACTCCAAACGTTTCGG
## CGAGAAGCAGGCCATTATCGCCGGCATGGCCGGCGACGCGCTGGGCTGGCGTTTCGCGACGCGAGGCTGGATGGCCTTCCC
## CATTATGATTCTTCTCGCTTCCGGCGGCCCGGCTTGCAGGCCATGCTGTCCAGGCAGGTAGATGACGACCATCAGGGACA
## GCTTCAACGGCTCTTACCAGCCTAACTTCGATCACTGGACCGCTGATCGTCACGGCGATTTATGCCGCACATGGACGCGT
## TGCTGGCGTTTTTCCATAGGCTCCGCCCCCTGACGAGCATCACAAACAAGTCAGAGGTGGCGAAACCCGACAGGACTAT
## AAAGATACCAGGCGTTTCCCCCTGGAAGCGCTCTCCTGTTCCGACCGTGCCGCTTACCGGATACCTGTCCGCCTTTCTCC
## CTTCGGGCTTTCTCAATGCTCAGCTGTAGGTATCTCAGTTCGGTGTAGGTGCTTCGCTCCAAGCTGACGAACCCCCCT
## TCAGCCCGACCGCTGCGCCTTATCCGGTAACTATCGTCTTGAGTCCAACACGACTTAACGGGTGGCGATGGATTGTAGGC
## GCCGCCCTATACCTTGTCTGCCTCCCCGCGGTGCATGGAGCCGGGCCACCTCGACCTGAATGGAAGCCGGCGGCACCTCG
## CTAACGCCAAGAATTGGAGCCAATCAATTCTTGC GGAGAACTGTGAATGCGCAAACCAACCCTTGGCCATCGCGTCCGC
## 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.103nt1-1200GCGTTGCTGGCGTTTTTCCATAGGCTCCGCCCCC
## TGACGAGCATCACAAAAATCGACGCGGTGGCGAAACCCGACAGGACTATAAAGATACCAGGCGTTTCCCCCTGGAAGTC
## CCTCGTGTTCGACCTGCCGCTTACCGGATACCTGTCCGCCTTTCTCCCTTCGGGAAGCGTGGCTGCTCAGCTGTACC
## TATCTCAGTTTCGGTGTAGGTGCTTGCCTCCAAGCTGGGCTGTGTGCCGTTAGCCCGACCGCTGCGCCTTATCCGTAAC
## TATCGTCTTGAGTCCAACCCGGTAAAGTAGGACAGGTGCCGGCAGCGCTCTGGGTCATTTTCGGCGAGGACCGCTTTCGC
## TGGAGATCGGCCTGTGCTTGCCTTATCGGAATCTTGCACGCCCTCGCTCAAGCCTTCGCTACTCCAAACGTTTCGGCG
## AGAAGCAGGCCATTATCGCCGGCATGGCCGGCGACGCGCTGGGCTGGCGTTTCGCGACGCGAGGCTGGATGGCCTTCCCCA
## TTATGATTCTTCTCGCTTCCGGCGGCCCGGCTTGCAGGCCATGCTGTCCAGGCAGGTAGATGACGACCATCAGGGACAGC
## TTCAACGGCTCTTACCAGCCTAACTTCGATCACTGGACCGCTGATCGTCACGGCGATTTATGCCGCACATGGACGCGTTG
## CTGGCGTTTTTCCATAGGCTCCGCCCCCTGACGAGCATCACAAACAAGTCAGAGGTGGCGAAACCCGACAGGACTATAA
## AGATACCAGGCGTTTCCCCCTGGAAGCGCTCTCCTGTTCCGACCGTGCCGCTTACCGGATACCTGTCCGCCTTTCTCCCT
## TCGGGCTTTCTCAATGCTCAGCTGTAGGTATCTCAGTTCGGTGTAGGTGCTTCGCTCCAAGCTGACGAACCCCCCTTC
## AGCCCGACCGCTGCGCCTTATCCGGTAACTATCGTCTTGAGTCCAACACGACTTAACGGGTGGCGATGGATTGTAGGCGC
## CGCCCTATACCTTGTCTGCCTCCCCGCGGTGCATGGAGCCGGGCCACCTCGACCTGAATGGAAGCCGGCGGCACCTCGCT
## AACGCCAAGAATTGGAGCCAATCAATTCTTGC GGAGAACTGTGAATGCGCAAACCAACCCTTGGCCATCGCGTCCGCCA
## 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")
```

```
## GCGTTGCTGGCGTTTTTCCATAGGCTCCGCCCCCTGACGAGCATCACAAAAATCGACGCGGTGGCGAAACCCGACAGGA
## CTATAAAGATACCAGGCGTTTCCCCCTGGAAGCTCCCTCGTGTTCGACCGCTGCGGCTTACCGGATACCTGTCCGCCTTT
## TCCCTTCGGGAAGCGTGGCTGCTCAGCTGTACCTATCTCAGTTCGGTGTAGGTGCTTCGCTCCAAGCTGGGCTGTGTG
## CCGTTCAGCCCGACCGCTGCGCCTTATCCGGTAACTATCGTCTTGAGTCCAACCCGGTAAAGTAGGACAGGTGCCGGCAG
```



```
## CGCTCTGGGTCATTTTCGGCGAGGACCGCTTTCGCTGGAGATCGGCCTGTCGCTTGCAGTATTCGGAATCTGCACGCCC
## TCGCTCAAGCCTTCGTCACCTCCAAACGTTTCGGCGAGAAGCAGGCCATTATCGCCGGCATGGCGGCCGACGCGCTGGGCT
## GGCCTTCGCGACGCGAGGCTGGATGGCCTTCCCCATTATGATTCTTCTCGCTTCCGGCGGCCCGGCTTGCAGGCCATGCT
## GTCCAGGCAGGTAGATGACGACCATCAGGGACAGCTTCAACGGCTCTTACCAGCCTAACTTCGATCACTGGACCGCTGAT
## CGTCACGGCGATTTATGCCGCACATGGACGCGTTGCTGGCGTTTTTCCATAGGCTCCGCCCCCTGACGAGCATCACAAA
## CAAGTCAGAGGTGGCGAAACCCGACAGGACTATAAAGATACCAGGCGTTTCCCCCTGGAAGCGCTCTCCTGTTCCGACCC
## TGCCGCTTACCGGATACTGTCCGCCTTCTCCCTTCGGGCTTCTCAATGCTCAGCTGTAGGTATCTCAGTTCGGTGT
## AGGTCGTTTCGCTCCAAGCTGACGAACCCCCCGTTAGCCCGACCGCTGCGCCTTATCCGTAACATCGTCTTGAGTCCA
## ACACGACTTAACGGGTTGGCATGGATTGTAGGCGCGCCCTATACCTTGTCTGCCTCCCGCGGTGCATGGAGCCGGGCC
## ACCTCGACCTGAATGGAAGCCGGCGGCACCTCGCTAACGGCCAAGAATTGGAGCCAATCAATTCTTGCGGAGAACTGTGA
## ATGCGCAAACCAACCTTGGCCATCGCGTCCGCCATCTCCAGCAGCCGACGCGGCGCATCTCGGGCAGCGTTGGGTCCT
```

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 GCGTTGCTGGCGTTTTTCCATAGGCTCCGCCCCCTGACGAGCATCACAAAAATCGACGCGGTGGCGAAA
## CCCGACAGGACTATAAAGATACCAGGCGTTTCCCCCTGGAAGCTCCCTCGTGTTCGACCCCTGCGGCTTACCGGATACCT
## GTCCGCCTTCTCCCTTCGGGAAGCGTGGCTGCTCAGCTGTACCTATCTCAGTTCGGTGTAGGTGCTTCCGCTCCAAGCT
## GGGCTGTGTGCCGTTACGCCCCGACCGCTGCGCCTTATCCGGTAACTATCGTCTTGAGTCCAACCCGGTAAAGTAGGACAG
## GTGCCGGCAGCGCTCTGGGTCATTTTCGGCGAGGACCGCTTTCGCTGGAGATCGGCCTGTCGCTTGCAGTATTCGGAATC
## TTGACGCGCCTCGCTCAAGCCTTCGTCACCTCCAAACGTTTTCGGCGAGAAGCAGGCCATTATCGCCGGCATGGCGGCCGAC
## GCGCTGGGTGGCGTTTCGGCAGCGAGGCTGGATGGCCTTCCCCATTATGATTCTTCTCGCTTCCGGCGGCCCGCGTTGC
## AGGCCATGTGTCCAGGCAGGTAGATGACGACCATCAGGGACAGCTTCAACGGCTCTTACCAGCCTAACTTCGATCACTG
## GACCGCTGATCGTCACGGCGATTTATGCCGCACATGGACGCGTTGCTGGCGTTTTTCCATAGGCTCCGCCCCCTGACGA
## GCATCACAAACAAGTCAGAGGTGGCGAAACCCGACAGGACTATAAAGATACCAGGCGTTTCCCCCTGGAAGCGCTCTCCT
## GTTCCGACCTGCGGCTTACCGGATACCTGTCCGCCTTCTCCCTTCGGGCTTCTCAATGCTCAGCTGTAGGTATCTC
## AGTTCGGTGTAGGTGCTTCGCTCCAAGCTGACGAACCCCCCGTTTACGCCCCGACCGCTGCGCCTTATCCGGTAACTATCGT
## CTTGAGTCCAACACGACTTAACGGGTTGGCATGGATTGTAGGCGCGCCCTATACCTTGTCTGCCTCCCGCGGTGCATG
## GAGCCGGGCCACCTCGACCTGAATGGAAGCCGGCGGCACCTCGCTAACGGCCAAGAATTGGAGCCAATCAATTCTTGCGG
## AGAAGTGTGAATGCGCAAACCAACCTTGGCCATCGCGTCCGCCATCTCCAGCAGCCGACGCGGCGCATCTCGGGCAGC
## 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")
```

```
## GCGTTGCTGGCGTTTTTCCATAGGCTCCGCCCCCTGACGAGCATCACAAAAATCGACGCGGTGGCGAAACCCGACAGGA
## CTATAAAGATACCAGGCGTTTCCCCCTGGAAGCTCCCTCGTGTTCGACCCCTGCGGCTTACCGGATACCTGTCCGCCTTT
## CTCCTTTCGGGAAGCGTGGCTGCTCAGCTGTACCTATCTCAGTTCGGTGTAGGTGCTTCCGCTCCAAGCTGGGCTGTGTG
## CCGTTCAGCCCGACCGCTGCGCCTTATCCGGTAACTATCGTCTTGAGTCCAACCCGGTAAAGTAGGACAGGTGCCGGCAG
## CGCTCTGGGTCATTTTCGGCGAGGACCGCTTTCGCTGGAGATCGGCCTGTCGCTTGCAGTATTCGGAATCTGCACGCCC
## TCGCTCAAGCCTTCGTCACCTCCAAACGTTTCGGCGAGAAGCAGGCCATTATCGCCGGCATGGCGGCCGACGCGCTGGGCT
## GGCCTTCGCGACGCGAGGCTGGATGGCCTTCCCCATTATGATTCTTCTCGCTTCCGGCGGCCCGGCTTGCAGGCCATGCT
## GTCCAGGCAGGTAGATGACGACCATCAGGGACAGCTTCAACGGCTCTTACCAGCCTAACTTCGATCACTGGACCGCTGAT
## CGTCACGGCGATTTATGCCGCACATGGACGCGTTGCTGGCGTTTTTCCATAGGCTCCGCCCCCTGACGAGCATCACAAA
## CAAGTCAGAGGTGGCGAAACCCGACAGGACTATAAAGATACCAGGCGTTTCCCCCTGGAAGCGCTCTCCTGTTCCGACCC
## TGCCGCTTACCGGATACTGTCCGCCTTCTCCCTTCGGGCTTCTCAATGCTCAGCTGTAGGTATCTCAGTTCGGTGT
## AGGTCGTTTCGCTCCAAGCTGACGAACCCCCCGTTAGCCCGACCGCTGCGCCTTATCCGTAACATCGTCTTGAGTCCA
## ACACGACTTAACGGGTTGGCATGGATTGTAGGCGCGCCCTATACCTTGTCTGCCTCCCGCGGTGCATGGAGCCGGGCC
## ACCTCGACCTGAATGGAAGCCGGCGGCACCTCGCTAACGGCCAAGAATTGGAGCCAATCAATTCTTGCGGAGAACTGTGA
## ATGCGCAAACCAACCTTGGCCATCGCGTCCGCCATCTCCAGCAGCCGACGCGGCGCATCTCGGGCAGCGTTGGGTCCT
```

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 expression).

```
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" "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" "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" "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 these entries and divide by the total number of characters in our string to get the %GC content.

```
length(str_extract_all(dna, pattern = "G|C")[[1]])/nchar(dna) *
100
```

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

Let's translate this into mRNA!

Replacement:

To replace multiple patterns at once, a character vector is supplied to `str_replace_all` of patterns and their matched replacements. This allows us to perform multiple replacements multiple times.

```
mrna <- str_replace_all(dna, c(G = "C", C = "G", A = "U", T = "A"))
```

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
```

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).

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

```
##      start end
## [1,]    21  23
```

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).

```
str_sub(mrna, 21, 1200)
# is equivalent to
mrna <- str_sub(mrna, str_locate(mrna, "AUG")[1])
```

```
## AUGGGAGGGGGGGGAGUGGUGGUAGUGUUUUAGGUGGGGGAGGGGUUUGGGGUGUGGUGAUUUUGUAUGGUGGGGAA
## AGGGGGAGGUUGGAGGGAGGAGAAGGGUGGGAGGGGGAUUGGGGUAUGGAGAGGGGGAAGAGGGAAGGGGUUGGGAGGG
## AGGAGUGGGAGAUGGAUAGAGUGAAGGGAGAUGGAGGAAGGGAGGUUGGAGGGGAGAGAGGGGAAGUGGGGUGGGGAGG
## GGGAAUAGGGGAUUGAUAGGAGAAGUGAGGUUGGGGGAUUUGAUGGUGUGGAGGGGGGUGGGGAGAGGGAGUAAAAGGGG
## GUGGUGGGGAAAGGGAGGUGUAGGGGGAGAGGGAAGGGGAUAAGGGUUAAGAAGGUGGGGGAGGGAGUUGGGAAGGAGUGA
## GGUUUGGAAAGGGGUGUUGGUGGGGUAUAGGGGGGUAAGGGGGGUGGGGGAGGGAGGGGAAGGGGUGGGGUGGGA
## GGUAGGGGAAGGGGUAUAGUAAGAAGAGGGAAGGGGGGGGGGGGAAGGUGGGGUAAGGAGAGGUGGGUGGAUGUAGUGG
## UGGUAGUGGGUGUGGAAGUUGGGGAGAAUGGUGGGAUUGAAGGUAGUGAGGUGGGGAGUAGGAGUGGGGUAUAAUAGGGG
## GUGUAGGUGGGGAAGGAGGGGAAAAAGGUAGGGAGGGGGGGGAGUGGUGGUAGUGUUUGUUGAGUGUGGAGGGGUUUG
## GGGUGUGGUGAUUUUGUAUGGUGGGGAAAGGGGAGGUUUGGGGAGAGGAGAAGGGUGGGAGGGGGAUUGGGGUAUGGAG
## AGGGGGAAGAGGGAAGGGGGAAGAGUUAGGAGUGGGAGAUUGGAUAGAGUGAAGGGAGAUGGAGGAAGGGAGGUUGGAG
## UGGUUGGGGGGAAGUGGGGUGGGGAGGGGGAUAGGGGAUUGAUAGGAGAAGUGAGGUUGUGGUGAAUUGGGGAAGGG
```

```
## UAGGUAAGAUGGGGGGGGGAUAUGGAAGAGAGGGAGGGGGGGAGGUAGGUGGGGGGGUGGAGGUGGAGUUAGGUUGG
## GGGGGGUGGAGGGAUUGGGGGUUGUUAAGGUGGGUUAAGUUAAGAAGGGGUGUUGAGAGUUAGGGGUUGGUUGGGAAGG
## GGUAGGGGAGGGGUAGAGGUGGUGGGGUGGGGGGUGAGAGGGGUGGGAAGGGAGGA
```

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

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

```
## [[1]]
## [1] "GGG" "AAG" "GAG" "GGG" "AAA" "AAG" "GUA" "UGG" "GAG" "GGG" "GGG"
## [12] "GGA" "GUG" "GUG" "GUA" "GUG" "UUU" "UUA" "GGU" "GGG" "GGA" "GGG"
## [23] "GUU" "UGG" "GGU" "GUG" "GUG" "AUA" "UUU" "GUA" "UGG" "UGG" "GGA"
## [34] "AAG" "GGG" "GAG" "GUU" "GGA" "GGG" "AGG" "AGA" "AGG" "GUG" "GGA"
## [45] "GGG" "GGA" "AUG" "GGG" "UAU" "GGA" "GAG" "GGG" "GAA" "AGA" "GGG"
## [56] "AAG" "GGG" "UUG" "GGA" "GGG" "AGG" "AGU" "GGG" "AGA" "UGG" "AUA"
## [67] "GAG" "UGA" "AGG" "GAG" "AUG" "GAG" "GAA" "GGG" "AGG" "UUG" "GAG"
## [78] "GGG" "AGA" "GAG" "GGG" "AAG" "UGG" "GGG" "UGG" "GGA" "GGG" "GGA"
## [89] "AUA" "GGG" "GAU" "UGA" "UAG" "GAG" "AAG" "UGA" "GGU" "UGG" "GGG"
## [100] "AUU" "UGA" "UGG" "UGU" "GGA" "GGG" "GGG" "UGG" "GGA" "GAG" "GGA"
## [111] "GUA" "AAA" "GGG" "GGU" "GGU" "GGG" "GAA" "AGG" "GAG" "GUG" "UAG"
## [122] "GGG" "GAG" "AGG" "GAA" "GGG" "GAU" "AAG" "GGU" "UAG" "AAG" "GUG"
## [133] "GGG" "GAG" "GGA" "GUU" "GGG" "AAG" "GAG" "UGA" "GGU" "UUG" "GAA"
## [144] "AGG" "GGG" "UGU" "UGG" "UGG" "GGU" "AAU" "AGG" "GGG" "GGU" "AGG"
## [155] "GGG" "GGG" "UGG" "GGG" "AGG" "GGA" "GGG" "GAA" "GGG" "GUG" "GGG"
## [166] "UGG" "GAG" "GUA" "GGG" "GAA" "GGG" "GUA" "AUA" "GUA" "AGA" "AGA"
## [177] "GGG" "AAG" "GGG" "GGG" "GGG" "GGG" "AAG" "GUG" "GGG" "UAG" "GAG"
## [188] "AGG" "UGG" "GUG" "GAU" "GUA" "GUG" "GUG" "GUA" "GUG" "GGU" "GUG"
## [199] "GAA" "GUU" "GGG" "GAG" "AAU" "GGU" "GGG" "AUU" "GAA" "GGU" "AGU"
## [210] "GAG" "GUG" "GGG" "AGU" "AGG" "AGU" "GGG" "GGU" "AAA" "UAG" "GGG"
## [221] "GUG" "UAG" "GUG" "GGG" "AAG" "GAG" "GGG" "AAA" "AAG" "GUA" "UGG"
## [232] "GAG" "GGG" "GGG" "GGA" "GUG" "GUG" "GUA" "GUG" "UUU" "GUU" "GAG"
## [243] "UGU" "GGA" "GGG" "GUU" "UGG" "GGU" "GUG" "GUG" "AUA" "UUU" "GUA"
## [254] "UGG" "UGG" "GGA" "AAG" "GGG" "GAG" "GUU" "GGG" "GAG" "AGG" "AGA"
## [265] "AGG" "GUG" "GGA" "GGG" "GGA" "AUG" "GGG" "UAU" "GGA" "GAG" "GGG"
## [276] "GAA" "AGA" "GGG" "AAG" "GGG" "GAA" "AGA" "GUU" "AGG" "AGU" "GGG"
## [287] "AGA" "UGG" "AUA" "GAG" "UGA" "AGG" "GAG" "AUG" "GAG" "GAA" "GGG"
## [298] "AGG" "UUG" "GAG" "UGG" "UUG" "GGG" "GGG" "AAG" "UGG" "GGG" "UGG"
## [309] "GGA" "GGG" "GGA" "AUA" "GGG" "GAU" "UGA" "UAG" "GAG" "AAG" "UGA"
## [320] "GGU" "UGU" "GGU" "GAA" "UUG" "GGG" "AAG" "GGU" "AGG" "UAA" "GAU"
## [331] "GGG" "GGG" "GGG" "GAU" "AUG" "GAA" "GAG" "AGG" "GAG" "GGG" "GGG"
## [342] "GAG" "GUA" "GGU" "GGG" "GGG" "GGU" "GGA" "GGU" "GGA" "GUU" "AGG"
## [353] "UUG" "GGG" "GGG" "GGU" "GGA" "GGG" "AUU" "GGG" "GGU" "UGU" "UAA"
## [364] "GGU" "GGG" "UUA" "GUU" "AAG" "AAG" "GGG" "UGU" "UGA" "GAG" "UUA"
## [375] "GGG" "GUU" "UGG" "UUG" "GGA" "AGG" "GGU" "AGG" "GGA" "GGG" "GGU"
## [386] "AGA" "GGU" "GGU" "GGG" "GGU" "GGG" "GGG" "GGU" "AGA" "GGG" "GGU"
## [397] "GGG" "AAG" "GGA" "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 TRUE 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 TRUE FALSE FALSE FALSE
## [100] FALSE TRUE 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 FALSE TRUE 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 TRUE
## [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 TRUE
## [221] FALSE TRUE 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 TRUE 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 TRUE
## [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] 68 92 93 96 101 121 130 140 186 219 222 291 315 316 319 329 363
## [18] 372
```

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

```
translation <- codons[1:121]
```

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 `str_replace_all` using multiple patterns and replacements as before.

```
unique(translation)

## [1] "GGG" "AAG" "GAG" "AAA" "GUA" "UGG" "GGA" "GUG" "UUU" "UUA" "GGU"
## [12] "GUU" "AUA" "AGG" "AGA" "AUG" "UAU" "GAA" "UUG" "AGU" "UGA"

translation <- str_replace_all(translation, c(AUG = "M", GGA = "G",
  GGG = "G", AGU = "S", GGU = "G", GUU = "V", UUU = "F", AGG = "R",
  UGG = "W", UUG = "L", UGU = "C", GAU = "D", AUU = "I", GUG = "V",
  AAA = "K", GAG = "E", AAG = "K", AAU = "N", GUA = "V", AGA = "R",
  GAA = "E", AUA = "M", UAA = ""))

translation

## [1] "G" "K" "E" "G" "K" "K" "V" "W" "E" "G" "G"
## [12] "G" "V" "V" "V" "V" "F" "UUA" "G" "G" "G" "G"
## [23] "V" "W" "G" "V" "V" "M" "F" "V" "W" "W" "G"
## [34] "K" "G" "E" "V" "G" "G" "R" "R" "R" "V" "G"
## [45] "G" "G" "M" "G" "UAU" "G" "E" "G" "E" "R" "G"
## [56] "K" "G" "L" "G" "G" "R" "S" "G" "R" "W" "M"
## [67] "E" "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] "GKEGKKVWEGGGVVVFUAGGGGVWGVVMFVWVGKGEVGGRRRVGGGMGUAUGEGERGKGLGGRSGRWMEUGA"
```

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 GKEGKKVWEGGGVVVFUAGGGGV
## WGVVMFVWVGKGEVGGRRRVGGGMGUAUGEGERGKGLGGRSGRWMEUGA
```

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

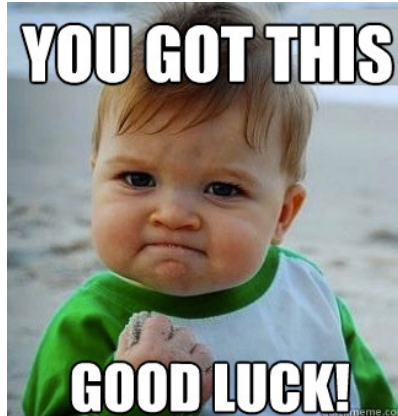


Figure 7:

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 mean cost of publishing for the top 3 most popular publishers?
3. What is the number of publications by PLOS One in dataset?
4. Convert sterling to CAD. What is the median cost of publishing with Elsevier in CAD?

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

- 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.
 - convert all entries to lowercase
 - correct typos
 - correct multiple names for a publisher to one name
 - remove newline characters and trailing whitespace
- Answer Questions #2-4

There is a README file to go with this spreadsheet if you have questions about the data fields.

The blogger's opinion of cleaning this dataset:

'I now have no hair left; I've torn it all out. My teeth are just stumps from excessive gnashing. My faith in humanity has been destroyed!'

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

We can talk about how this went at the beginnning of next week's lesson.

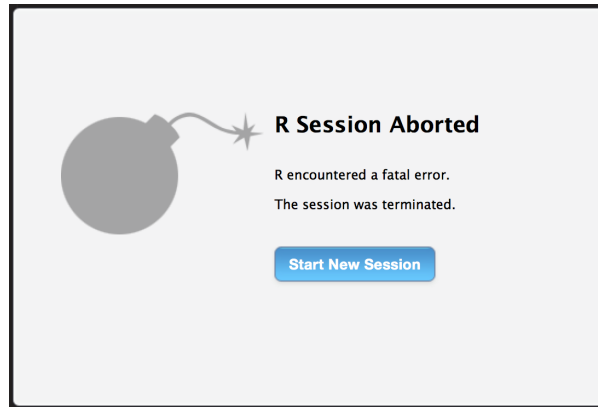


Figure 8:

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/>

Post-Lesson Assessment

Your feedback is essential to help the next cohort of trainees. Please take a minute to complete the following short survey: <https://www.surveymonkey.com/r/PVHDKDB>

Thanks for coming!!!