

MAKE PARSING GREAT AGAIN

(with string functions and regular expressions)

Robert Gebeloff @gebeloffnyt

The New York Times

Dataharvest 2020

So much of what we do in data journalism is fun. Writing queries. Making maps. Holding powerful people accountable, etc.

What's not to love?

Cleaning and parsing data, for one.

There's usually a time in every project where you're excited to dig in, only to realize that the data you want is not in easily diggable format.

As data journalists, we're trained to think in rows and columns.

Every row is a record (name of dog).

Every column is a characteristic of that record. (how many dogs have that name.)

	A	B	C
1	DogName	Count	
2	BUDDY	83	
3	MAX	81	
4	SADIE	80	
5	LUCY	74	
6	BAILEY	67	
7	DAISY	64	
8	LUNA	63	
9	CHARLIE	60	
10	MAGGIE	57	
11	MOLLY	56	
12	COOPER	54	
13	GINGER	49	
14	JACK	49	
15	WILLOW	49	
16	ABBY	17	

When we obtain data from the government or some other source, this is what we expect. Clean data organized in this fashion.

But in an increasingly cruel and difficult world, neatly formatted, ready-to-use data is harder and hard to come by.

This tutorial will introduce you to some more advanced techniques of dealing with messy data. I will walk you through two different examples of problems and solutions involving data that does not come in row-and-column format.

The first thing that's important to understand about all of this is that the skills I am going to demo are generic -- they work pretty much the same in any language you might use, so please don't get hung up if you're a Python maven and I'm not showing Python.

In fact, the first part of this tutorial, I programmed in Ruby and SQL when I did my project, but I'm demoing in R today.

The key is to learn the technique -- if you can do that, you can easily figure out how to get it to work in whatever language you use.

In the first example, I will deploy what can generically be called "String Expressions". These are commands built into whatever programming language you might use, and can, in combination, be very powerful tools for taming data.

And the second example, I will use what are called "Regular Expressions" --an ancient but powerful pattern-matching technique that can help you parse data that has some structure other than rows and columns.

So like all great stories begin, let me show you a problem I had to face and how I solved it, using string expressions.

As some of you may have heard, in the U.S we have a president that is doing things a bit differently than his predecessors.

And because one of the key presidential powers is the appointment of federal judges, we wanted to check in and see what we could learn about his appointees.

Fast-forwarding through some of the preliminary reporting, our focus became the U.S. Court of Appeals, the 2nd highest level below the U.S. Supreme Court.

A lot had been written about how President Trump was selecting ultra-conservative judges -- but we wanted to see how these judges were actually performing on the bench.

Fast-forwarding again, I found a government source that provided a standard database of court cases -- but the fields containing the names of judges handling each case were-- REDACTED!

We could have put up a big stink and insisted that the names of public judges hearing public court cases should not be redacted from a public database, but we also had a plan B -- obtaining the text of rulings issued in the cases and pulling the names right from the document.

PUBLISHED

UNITED STATES COURT OF APPEALS
FOR THE FOURTH CIRCUIT

No. 18-1886

NGAWUNG ATEMNKENG,

Petitioner,

v.

WILLIAM P. BARR, Attorney General,

Respondent.

On Petition for Review of an Order of the Board of Immigration Appeals.

Argued: October 29, 2019 Decided: January 24, 2020

NAMES OF JUDGES
Before GREGORY, Chief Judge, WYNN, and THACKER, Circuit Judges.

AUTHOR

Petition for review granted; vacated, and remanded by published opinion. Chief Judge Gregory wrote the opinion, in which Judge Wynn and Judge Thacker joined.

ARGUED: Ronald Darwin Richey. LAW OFFICE OF RONALD D. RICHEY. Rockville.

So this clearly isn't rows and columns but the data we wanted was in there -- I added blue labels for demonstration purposes but they are not in the actual text.

What I needed to do is figure out a pattern of some kind that I could use to identify where the judge names are listed. And then I needed a second pattern to determine which of the judges wrote the opinion.

Let me show you how I did it:

This is some code in the R programming language. Lines starting with # are comments, the rest is code and output:

```
# load PDF processing library or install/load it if necessary
needs(pdftools)

# declare path to the file
thefile="https://www.govinfo.gov/content/pkg/USCOURTS-ca4-18-01886/pdf/USCOURTS-ca4-18-01886-0.pdf"

# open the file and store the text into a variable
thecase=pdf_text(thefile)

#view the output
> thecase
[1] "USCA4 Appeal: 18-1886          Doc: 45          Filed:
01/24/2020      Pg: 1 of 20\n
PUBLISHED\n          UNITED STATES COURT
OF APPEALS\n          FOR THE
FOURTH CIRCUIT\n
No. 18-1886\n          NGAWUNG ATEMNKENG,\n
Petitioner,\n          v.\n          WILLIAM P. BARR,
Attorney General,\n          Respondent.\n
On Petition for Review of an Order of the Board of Immigration
Appeals.\n          Argued: October 29, 2019
Decided: January 24, 2020\n          Before GREGORY, Chief Judge,
WYNN, and THACKER, Circuit Judges.\n          Petition for review
granted; vacated, and remanded by published opinion. Chief Judge
Gregory\n          wrote the opinion,
```

Now that we have the document text stored in a variable called “thecase”, we can now figure out how to get our judge names.

This is where I pause to say that there is no one-size-fits-all approach to these problems. Every advanced parse you’re going to do will be different.

What you need to do is study your documents and look for something that they all have in common, something that, if you read the documents programmatically, your code will behave the same with each document.

In this case, what we discovered is: In every document, the names of judges are preceded by the word “Before”.

If we could just tell our program, find the word “Before” and give us what comes after it, we would be golden.

Enter the “SPLIT” function. Split is a string function that exists in every language and what it does, living up to its name, is split your text based on whatever divider you feed it.

So in our case, we can pass the word “Before” as a divider:

```
str_split(thecase,"Before")
```

Produces this output:

```

> str_split(thecase, "Before")
[[1]]
[1] "USCA4 Appeal: 18-1886      Doc: 45      Filed: 01/24/2020      Pg: 1 of 20\n
STATES COURT OF APPEALS\n
Petitioner,\n
v.\n
WILLIAM P. BARR, Attorney General,\n
an Order of the Board of Immigration Appeals.\n
Argued: October 29, 2019
[2] " GREGORY, Chief Judge, WYNN, and THACKER, Circuit Judges.\n
Petition for review granted; vacated, and remanded\n
in, in which Judge Wynn and Judge Thacker joined.\n
ARGUED: Ronald Darwin Richey, LAW OFFICE OF RONALD D. RICHEY,\n
yson, Jr., UNITED STATES DEPARTMENT\n
OF JUSTICE, Washington, D.C., for Respondent. ON BRIEF: Joseph H. Hunt, Assis\n
r, Office of Immigration Litigation,\n
Civil Division, UNITED STATES DEPARTMENT OF JUSTICE, Washington, D.C., for\n

[[2]]
[1] "USCA4 Appeal: 18-1886      Doc: 45      Filed: 01/24/2020      Pg: 2 of 20\n
GREGORY, Chief Judge:\n
y after participating in\n
anti-government meetings and protests, getting arrested and was detained without trial\n
fficers, and receiving numerous\n
death threats. An immigration judge ("IJ") initially noted some inconsistencies\n
credible and her explanations\n
plausible, and granted her asylum application. On appeal, the Board of Immigration\n
structed the IJ, in reviewing the\n
asylum application a second time, to afford Atemnkeng an opportunity to explain\n
ng has now relocated to Baltimore and the new IJ ("Baltimore\n
IJ") permitted her to submit additional documents in\n
er calendar hearing. Approximately one month prior to the hearing,\n
however, the Baltimore IJ issued a written rul\n
er reliefs. The Baltimore IJ concluded, without Atemnkeng's new\n
testimony, that she was not credible in light\n
o the BIA, the Baltimore IJ's ruling was affirmed without an opinion. Atemnkeng\n
now petitions for review of the B\n
In her petition for review, she raises several claims, most notably, that her due\n
process rights were viol\n
stify\n
on remand. Concluding that Atemnkeng's claim related to her ability to testify is\n
meritorious,\n
and remand for\n
2\n"

[[3]]
[1] "USCA4 Appeal: 18-1886      Doc: 45      Filed: 01/24/2020      Pg: 3 of 20\n
further proceedings. In light of\n
mnkeng an opportunity to testify and weigh the relevance of that testimony in\n
conjunction with the entire record,\n
determination and denials of Atemnkeng's applications for withholding of removal and\n
relief under the Convention\n
I.\n
Atemnkeng is a national and citizen of Cameroon. She and her family li\n
British Southern Cameroons Region. She is\n
"British" To the British West Cameroons such British area though the

```

So what the split function has done is divide our document into “vectors”. You don’t really need to know what a “vector” is, just that it’s an object that can be accessed with the coordinates provided, following the hierarchy provided. In this case, the PDF parser we’re using divides the document pages into vectors, and then our split function created a second set of vectors within the pages.

In our case, we're interested in the `[[1]][2]` vector, so:

```
judgestring<- str_split(thecase, "Before ")[[1]][2]
>
> judgestring
[1] "GREGORY, Chief Judge, WYNN, and THACKER, Circuit
Judges.\n      Petition for review granted; vacated, and remanded by
published opinion. Chief Judge Gregory\n      wrote the opinion, in
which Judge Wynn and Judge Thacker joined.\n      ARGUED:
Ronald Darwin Richey, LAW OFFICE OF RONALD D. RICHEY,
Rockville,\n      Maryland, for Petitioner. Robert Dale Tennyson, Jr.,
UNITED STATES DEPARTMENT\n      OF JUSTICE, Washington,
D.C., for Respondent. ON BRIEF: Joseph H. Hunt, Assistant\n
Attorney General, Carl McIntyre, Assistant Director, Office of
Immigration Litigation,\n      Civil Division, UNITED STATES
DEPARTMENT OF JUSTICE, Washington, D.C., for\n
Respondent.\n"
```

Now we're getting closer, but still not there. If you wanted to narrow it down more, what would our next split be?

```

> str_split(judgestring, "\n")
[[1]]
[1] "GREGORY, Chief Judge, WYNN, and THACKER, Circuit Judges."
[2] "    Petition for review granted; vacated, and remanded by published
opinion. Chief Judge Gregory"
[3] "    wrote the opinion, in which Judge Wynn and Judge Thacker joined."
[4] "    ARGUED: Ronald Darwin Richey, LAW OFFICE OF RONALD D.
RICHEY, Rockville,"
[5] "    Maryland, for Petitioner. Robert Dale Tennyson, Jr., UNITED
STATES DEPARTMENT"
[6] "    OF JUSTICE, Washington, D.C., for Respondent. ON BRIEF:
Joseph H. Hunt, Assistant"
[7] "    Attorney General, Carl McIntyre, Assistant Director, Office of
Immigration Litigation,"
[8] "    Civil Division, UNITED STATES DEPARTMENT OF JUSTICE,
Washington, D.C., for"
[9] "    Respondent."
[10] ""

>
> judgestring <- str_split(judgestring, "\n")[[1]][1]
>
> judgestring
[1] "GREGORY, Chief Judge, WYNN, and THACKER, Circuit Judges."

```

So this time we split on the line break character “\n” that surfaced when we pulled the text out of the PDF. The character has nothing to do with what we’re doing - we’re not displaying the text, we’re parsing it -- but in parsing, we use what we find and if that’s part of the document format, it’s helpful.

Now to clean up a little more, we will use another string expression called GSUB - - GSUB is shorthand for “global substitution” and in our case, we can search/replace out words that are not judge names:


```

judgestring
[1] "GREGORY, Chief Judge, WYNN, and THACKER, Circuit Judges."
>
>
> judgestring<- gsub("Chief|Judge|Circuit|Judges|and|\\.| ", "", judgestring)
>
> judgestring
[1] "GREGORY,, WYNN, THACKER,"

```

Let me explain what's going on here.

PART 1: `gsub("Chief|Judge|Circuit|Judges|and|\\.| ",`

PART 2: `""`,

PART 3: `judgestring)`

The first part of `gsub`, I'm feeding it a list of words that I want to get rid of. I'm separating it with the `|` symbol, which in R is the same as "or".

So I'm saying, get rid of any string that matches "Chief" or "Judge" etc.

Notice the `\\.`. I want to get rid of any periods, but a period is a "reserved" character that has other meanings in the programming language. So to tell R that I literally mean to get rid of the periods, I'm using the `\` symbol to "escape" my command. And because the `\` symbol is itself a reserved character, I'm using two.

(Ok, for people new to this, this is CRAZY talk. But if you do this a bit, you'll get used to it. Just remember in whatever language you're working in, you want to find out how to handle reserved characters).

Finally, notice the space between the last `|` and the close quote -- that's to get rid of any extra spaces between names.

The second part is telling R what to use as a replacement, and in this case, open quote/close quote means replace with nothing.

And the third part is telling R what string to apply this logic to.

So in sum, and in any language, there will be a version of `gsub` that works the same way, taking three options:

gsub(replacethis, withthis, fromthis).

In a full program, this string of three names can be parsed many ways. In our case, we stored the names in a database and joined them to fuller biographic info we had on the judges.

So the next question was: How do we pull out the name of the judge that wrote the opinion?

Argued: October 27, 2017

Decided: January 27, 2020

Names of Judges

Before GREGORY, Chief Judge, WYNN, and THACKER, Circuit Judges.

Petition for review granted; vacated, and remanded by published opinion. Chief Judge Gregory wrote the opinion, in which Judge Wynn and Judge Thacker joined.

If you look at enough of these documents, you'll notice another pattern. After listing the three judges, the NEXT time you see a judge's name, it's the one that wrote the opinion.

So what I had to figure out was how to programmatically measure which name appeared soonest in the document after all three were named.

In R, we can do this with a function called "GREGEXPR".

This function evaluations the document and tells you the starting position, in spaces, of every occurrence of the string.

I can also tell it to ignore case, which is a good idea because sometimes we expect the judge's name to be in all caps and sometimes in titlecase:

With any programming language, there are a lot of different libraries that help you get things done -- you don't have to memorize them all! You just need to know what you want to do and how to research the answer.

```
gregexpr(whattolookfor,wheretolook,ignorecase)[first]vector][first2values
```

```
> gregexpr("Gregory", thecase, ignore.case=TRUE)[[1]][1:2]  
[1] 697 846 # The position of the first 2 matches  
> gregexpr("Wynn",thecase,ignore.case=TRUE)[[1]][1:2]  
[1] 719 895 # The position of the first 2 matches  
> gregexpr("Thacker",thecase,ignore.case=TRUE)[[1]][1:2]  
[1] 729 910 # The position of the first 2 matches
```

Similar to gsub, we're feeding gregexpr a series of parameters -- the string we're looking for, which is the judge name, the name of the variable holding our text, and a flag telling it to ignore case. Then we're telling us to give us the first element - which is the list of positions for our string, and to give us the first two occurrences.

In my production app (written in a diff language so not displayed here), I looped through all of the cases, stored all of the judges names in a database as judge1, judge2, and judge3 and then looped through each case again, feeding each judge name dynamically and calculating which judge had the lowest index value for the 2nd occurrence and used that to make my preliminary estimate of the opinion author.

Here is the [resulting story](#).

In closing this first part, a few things to keep in mind.

String expressions are a fabulous tool but this type of parsing is rarely perfect. There are frequent exceptions to the patterns you find, and your scripts might produce 90 pct clean data -- and you'll have to manually fix the other 10 pct.

But this is much much better than manually cleaning 100 pct.

Also, I can't say this enough -- to be good at this, you do not have to memorize all of the string functions and all of the syntax. You can look all of this up as you need it.

The most important skill is to be able to see patterns in your unstructured data. If you can find a pattern, odds are that there is a way to parse -- you just have to find the right function and the right syntax.

Which brings us to part two, using the regular expression syntax for more intricate pattern matching.

Regular expressions, or RegEx for short, exist in numerous flavors in a wide variety of scripting environments. These different variations might carry slightly different syntax conventions, but the concepts are the same.

For this section, we're going to use a Web site that is designed to help you write Regex code and extract your results.

For dirty data, we're going to turn to none other than Donald J. Trump, and his Twitter feed.

But before we begin, let me go over some important concepts.

RegEx is all about patterns. The simplest way to think of it is to compare it to a traditional search operation.

In a text file, you can search for a specific string, like "Belgium", or in a language such as SQL, you can use a wildcard character and search for countries that begin with B, such as "where country like 'B%'".

But in RegEx, you can create very sophisticated search logic with many types of "wildcards". For example to search a document and find any four-letter word that begins with T:

(T[a-z]{3})

That expression is interpreted as "Starts with a capital 'T', and is followed by three lower-case letters and then a space." It will match "This " "Them " "That " but not this, these or those.

What's difficult for beginners is learning all of the possible RegEx tools at your disposal.

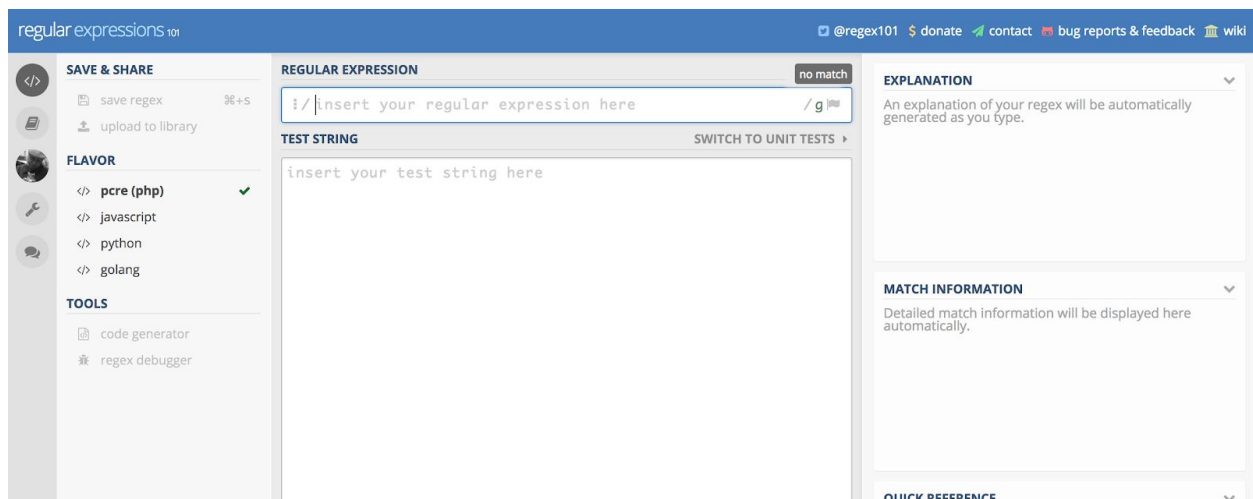
In some lessons, we'd dive in with a tour of all of the common tools, but the philosophy here is that it's better to learn while working on an actual job, and using a tool with a beautiful cheat sheet of syntax.

So hello [@realDonaldTrump](#)

In this exercise, we're going to take 499 Trump tweets from the first quarter of 2017 and parse the text into columns, turning them into a tab-delimited format that we can save into a spreadsheet.

We want to capture the dates and time of tweets, the contents of the tweets, and the software he tweeted with, which is listed at the end of each tweet. The data comes from the Trump Twitter Archive (see <http://www.trumptwitterarchive.com/about> for more info).

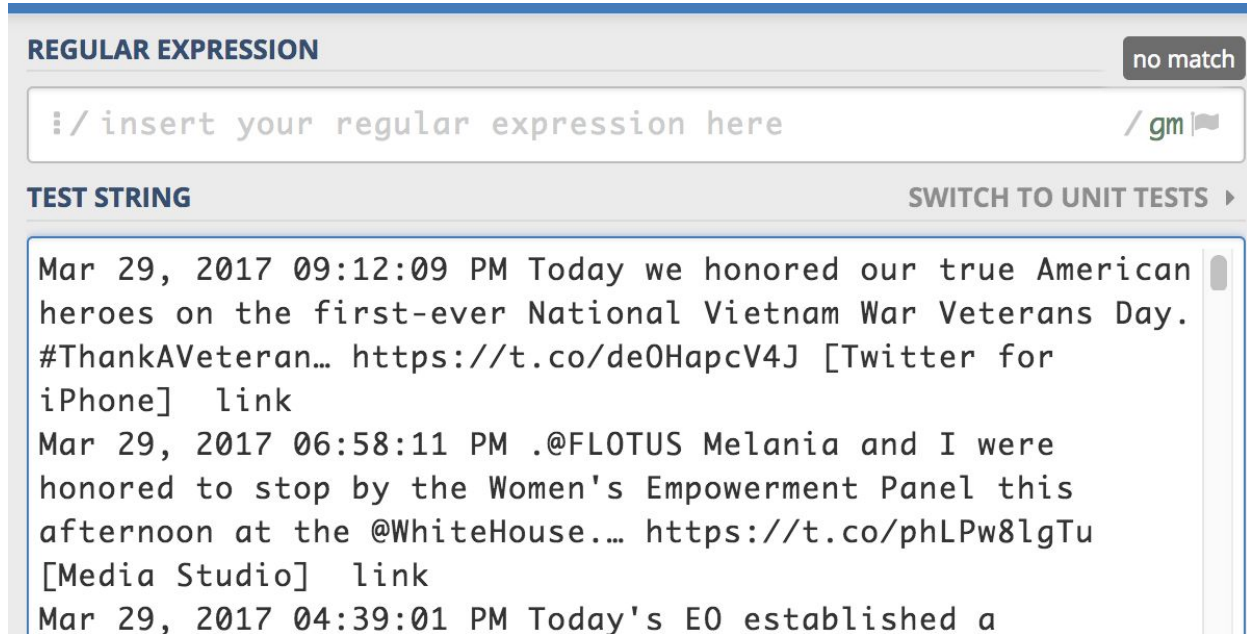
We are going to do our parsing using a great online tool, <http://regex101.com>, and I've already created the Trump tweet file here (link to come).



Regex101 is a fabulous resource because it allows you to create your Regex code online and then bring it back into whatever environment you're working in. For example, if you're writing a program in Python that loops through thousands of documents and extracts certain information with Regex, you can test your code here before running it on the big batch.

In our case, we're going to parse the Trump tweets and then copy and paste the results into a spreadsheet.

The first step you want to do is get your data into the tool, so copy and paste the Trump data into the big box labeled "Test String". Also at this time, let's make sure the parser knows that each Tweet in this dataset spans multiple lines by making sure the regular expression box at the top ends with the notation "gm" (global multiline).



The screenshot shows a web-based regular expression testing tool. At the top, there is a tab labeled "REGULAR EXPRESSION" and a button that says "no match". Below this is a text input field containing the placeholder text ":// insert your regular expression here". To the right of the input field is a small icon and the text "/gm". Below the input field is another tab labeled "TEST STRING" and a button labeled "SWITCH TO UNIT TESTS". The "TEST STRING" tab is active, and it contains a large text area with the following text: "Mar 29, 2017 09:12:09 PM Today we honored our true American heroes on the first-ever National Vietnam War Veterans Day. #ThankAVeteran... https://t.co/de0HapcV4J [Twitter for iPhone] link". Below this is another line of text: "Mar 29, 2017 06:58:11 PM .@FLOTUS Melania and I were honored to stop by the Women's Empowerment Panel this afternoon at the @WhiteHouse... https://t.co/phLPw8lgTu [Media Studio] link". Below that is a third line of text: "Mar 29, 2017 04:39:01 PM Today's EO established a".

To get started, we need to enter the symbol that tells Regex to start our quest at the beginning of each new line. That is done with the symbol **^**

REGULAR EXPRESSION 499 matches, 2496 steps (~13ms)

TEST STRING SWITCH TO UNIT TESTS ▶

Mar 29, 2017 09:12:09 PM Today we honored our true American heroes on the first-ever National Vietnam War Veterans Day. #ThankAVeteran... https://t.co/de0HapcV4J [Twitter for iPhone] link

Mar 29, 2017 06:58:11 PM .@FLOTUS Melania and I were honored to stop by the Women's Empowerment Panel this afternoon at the @WhiteHouse... https://t.co/phLPw8lgTu [Media Studio] link

Mar 29, 2017 04:39:01 PM Today's EO established a commission on combating drug addiction and the opioid crisis. Watch listening session... https://t.co/oof2ediqSt [Twitter for iPhone] link

Mar 29, 2017 07:21:02 AM If the people of our great country

EXPLANATION

▼ / / gm

asserts position at start of a line

▼ **Global pattern flags**

g modifier: global. All matches (don't return after first match)

m modifier: multi line. Causes and to match the begin/end of each line (not only begin/end of string)

MATCH INFORMATION

Match 1

Full match 0-0 ``

Match 2

Full match 187-187 ``

Match 3

A few things to note:

- The beginning of each line in the Test String is now marked by dots
- A box appears above our Regular Expression letting us know there are 499 matches
- An explanation appears on the right explaining to us in plain language what our expression is doing
- More information about our match appears in the “Match Information” box.
- Below that, and not pictured, is a handy Regex cheat sheet to guide your way.

Ok, so let's capture the month. If you browse the text, you'll see that Twitter uses a three-letter abbreviation for a month symbol -- this is a pattern and we can try to match it with regex syntax.

So we are going to create our first capture “group” (a future column in our database) and tell it to grab a three-letter string.

It will look like this:

(\w{3})

The parens are used to declare the group, and the expression inside literally means: Find a letter or number `\w` that repeats three times `{3}`. Notice how all the information in the regex101 updates to show you how the command is playing out. The `\` precedes many of the regex commands.

Next we need to capture the day of the month, which is always numeric but can be one or more digits -- another pattern!

So first enter a space - we need to account for the space between the month and day, and then enter `(\d+)`. As you might guess, `\d` means a digit, but in this case, the `+` sign means one or more of. How does it know when to stop? Because of the comma - it captures all numbers up to the comma, which is not a digit.

Ok, next up is the year. Can you guess how we'd capture that? Try it yourself before turning to the next page...

REGULAR EXPRESSION v1 499 matches, 7485 steps (~15ms)

TEST STRING SWITCH TO UNIT TESTS

Mar 29, 2017 09:12:09 PM Today we honored our true American heroes on the first-ever National Vietnam War Veterans Day. #ThankAVeteran... https://t.co/deOHapcV4J [Twitter for iPhone] link

Mar 29, 2017 06:58:11 PM .@FLOTUS Melania and I were honored to stop by the Women's Empowerment Panel this afternoon at the @WhiteHouse... https://t.co/phLPw8lgTu [Media Studio] link

Mar 29, 2017 04:39:01 PM Today's E0 established a commission on combating drug addiction and the opioid crisis. Watch listening session... https://t.co/ooF2ediqSt [Twitter for iPhone] link

Mar 29, 2017 07:21:02 AM If the people of our great country could only see how viciously and inaccurately my administration is covered by certain media! [Twitter for iPhone] link

EXPLANATION

^(\w{3}) (\d+), (\d{4}) / gm

- asserts position at start of a line
- 1st Capturing Group (\w{3})
 - \w{3} matches any word character (equal to [a-zA-Z0-9_])
 - {3} Quantifier — Matches exactly 3 times
 - matches the character literally (case sensitive)
- 2nd Capturing Group (\d+)
 - \d+ matches a digit (equal to [0-9])

MATCH INFORMATION

Match 1

Full match	0-12	`Mar 29, 2017`
Group 1.	0-3	`Mar`
Group 2.	4-6	`29`
Group 3.	8-12	`2017`

Match 2

Full match	187-199	`Mar 29, 2017`
------------	---------	----------------

Yes, combining the techniques from group 1 and group 2, we tell Regex to expect a comma and a space, and to then create a new group with four-digit string.

Could we have used `d+`? Yes, that would have also worked, but in this case, we know the year has to be 4 digits, so declaring a four-digit number is more precise.

Note that in data work, dates and times formats can be messy. A valid date in Excel might not be a valid format in MySQL. Therefore, it's often wise to initially work with dates as components -- month, day, year -- as opposed to trying to capture the entire date in the source data's format.

For the time, we're going to accept the source format, but keep the AM/PM flag as a separate column.

```
^(\\w{3}) (\\d+), (\\d{4}) (\\d\\d:\\d\\d:\\d\\d) (\\w{2})
```

The screenshot shows a regular expression testing interface. The regular expression being tested is `^(\\w{3}) (\\d+), (\\d{4}) (\\d\\d:\\d\\d:\\d\\d) (\\w{2})`. The test string contains three tweets. The first tweet is highlighted, showing a match for the timestamp `Mar 29, 2017 09:12:09 PM`. The match information table shows the following details:

Group	Match
Full match	0-25 `Mar 29, 2017 09:12:09 P`
Group 1.	0-3 `Mar`
Group 2.	4-6 `29`
Group 3.	8-12 `2017`
Group 4.	13-21 `09:12:09`
Group 5.	22-24 `PM`

Ok, so now we're up to the Tweet itself. In this case, the pattern we want to match is a little different. Each Tweet begins after the AM/PM label and goes on until the meta info about what software was used, which begins with a "[" character. Because "[" is a reserved character (remember those?) , we're going to have to escape it.

This sounds complicated but in regex, only requires a few characters of code:

`(.+)[`

This literally means, after a space, create a group that starts with any character and goes on indefinitely. But the group should end when it hits the first “[” sign, which in this case is preceded by a `\` to signify that we mean the literal “[”, not the regex [syntax (Also note, if you copy and paste this code, the “[” symbol might be interpreted as a different symbol in your Web browser - if the code doesn't work, delete the [you pasted and retype it in the regex101 browser window.)

The screenshot shows the regex101.com interface. The regular expression `^(?!\s)(.+)[` is entered in the top bar. The test string contains several lines of text, including dates, times, and social media links. The regex engine has found 499 matches in 43952 steps. The explanation pane on the right details the components of the regex: `^` asserts position at start of a line, `(?! \s)` is a negative lookahead assertion that the next character is not a space, `(.+)` is the 1st capturing group that matches any character one or more times, and `[` matches the character '[' literally. The match information pane shows the following matches:

Group	Match
Group 4.	13-21 `09:12:09`
Group 5.	22-24 `PM`
Group 6.	25-160 `Today we honored our true American heroes on the first-ever National Vietnam War Veterans Day. #ThankAVeteran... https://t.co/de0HapcV4J`

The last part -- “Twitter for iPhone”, “Media Studio” -- we can capture with similar syntax.

Give it a try before turning the page to see the results....

The screenshot shows a regex testing interface with the following components:

- REGULAR EXPRESSION:** `^(?w{3}) (?d+), (?d{4}) (?d\d:~d\d:~d\d) (?w{2}) (?+)[?+]`
- TEST STRING:**

```

Mar 29, 2017 09:12:09 PM Today we honored our true American
heroes on the first-ever National Vietnam War Veterans Day.
#ThankAVeteran... https://t.co/de0HapcV4J [Twitter for
iPhone] link
Mar 29, 2017 06:58:11 PM .@FLOTUS Melania and I were
honored to stop by the Women's Empowerment Panel this
afternoon at the @WhiteHouse... https://t.co/phLPw8lgTu
[Media Studio] link
Mar 29, 2017 04:39:01 PM Today's EO established a
commission on combating drug addiction and the opioid
crisis. Watch listening session... https://t.co/ooF2ediqSt
[Twitter for iPhone] link

```
- EXPLANATION:**
 - `^` asserts position at start of a line
 - 1st Capturing Group** `(?w{3})`
 - `(?w{3})` matches any word character (equal to `[a-zA-Z0-9_]`)
 - `{3}` Quantifier — Matches exactly 3 times
- MATCH INFORMATION:**
 - Match 1**
 - Full match** 0-180: `Mar 29, 2017 09:12:09 PM Today we honored our true American heroes on the first-ever National Vietnam War Veterans Day. #ThankAVeteran... https://t.co/de0HapcV4J [Twitter for iPhone] link`

That’s right. We immediately create a new group after the “[” and tell it to capture all text until reaching the “]”, which we also have to escape. Also, for the first time in this parse, we’re not adding a space -- if you do, the code will not work.

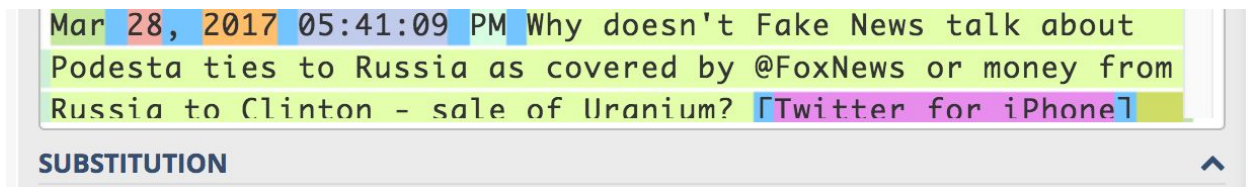
The last thing we need to do is account for the dummy text at the end of every tweet. The original source had a link back to the original tweet, which

is meaningless in this case, but we want to capture it in a group so we can dispose of it. `(.+)`

So our final expression string looks like this:

```
^(\w{3}) (\d+), (\d{4}) (\d\d:\d\d:\d\d) (\w{2}) (.+)\[.(+)\](.+)
```

Ok, so how do we get our parsed data into a more useful format? On the left side of the page, click on a bar that says “Substitution”.



A box opens up, and in here, we’re going to tell Regex how we want our results formatted. Let’s create a comma-delimited format with quote marks separating the columns:

```
"1","2","\3","\4","\5","\6","\7"
```

```
SUBSTITUTION
"\1", "\2", "\3", "\4", "\5", "\6", "\7"|

"Mar", "29", "2017", "09:12:09", "PM", "Today we honored our
true American heroes on the first-ever National Vietnam War
Veterans Day. #ThankAVeteran... https://t.co/de0HapcV4J
", "Twitter for iPhone"
"Mar", "29", "2017", "06:58:11", "PM", ".@FLOTUS Melania and I
were honored to stop by the Women's Empowerment Panel this
afternoon at the @WhiteHouse.... https://t.co/phLPw8lgTu
", "Media Studio"
"Mar", "29", "2017", "04:39:01", "PM", "Today's EO established
a commission on combating drug addiction and the opioid
crisis. Watch listening session... https://t.co/ooF2ediaSt
```

Literally speaking, we're telling it to print a “, then group 1, then a quote, comma, quote, then group 2, etc.

You can copy and paste this content into your favorite text editor and save it as a csv file, or paste it directly into a spreadsheet program.

This entire exercise is saved at <https://regex101.com/r/XxpV5G/3>

On your own, clear out the Regex and start again, thinking of other things you might want to capture: Hashtags? Links? Etc.

You'll also find more help using Regex online. While we covered only a small portion of what you can do with Regex, I hope this exercise gives you a base from which you can learn more.

And remember -- even the best coder/journalists in the world often struggle with Regex -- be patient and keep trying. Almost anything is possible!



Matt Waite ✓ @mattwaite · Aug 27

I've been writing code professionally for almost 14 years now and I still fist pump and then sigh in relief every time I make a regular expression work.

Also note that this lesson was inspired by a [tutorial presented at NICAR](#) a few years back by Christian McDonald, then of The Dallas Morning News and now at the University of Texas.