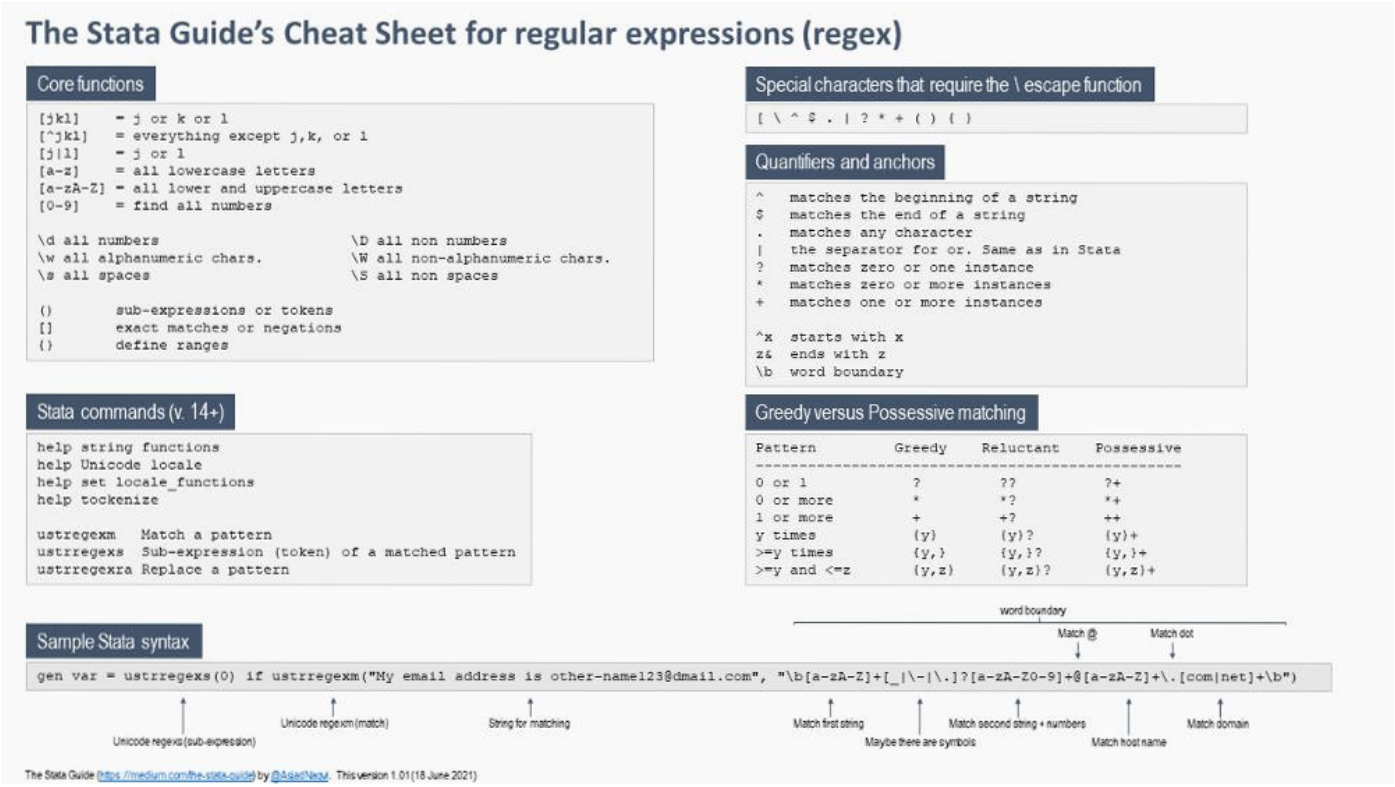
Regular expressions (regex) in Stata

Asjad Naqvi Follow

Mar 17 · 23 min read ★

This guide covers one of the most under-documented features of Stata: regular expressions, or *regex* for short. In this guide we will learn how to implement the regex features shown in the Stata cheat sheet below. This includes learning about quantifiers, building specific to generic expressions from bottom-up, word boundaries, and learning about greedy versus possessive matching:



Printable PDF version: https://github.com/asjadnaqvi/COVID19-Stata-Tutorials/blob/master/Stata_regex_cheatsheet_v1.pdf

Regex is the core algorithm that is used for searching through text using pattern matching. One comes across online regex implementations almost daily. These include auto checking whether you have filled your email address correctly, or whether your password has sufficient characters or whether it is strong enough. The regex algorithm can be layered with a lot of additional tools. These includes text mining, natural language processing (NLP), sentiment analysis, machine learning (ML), automated journalism, auto complete in your online searches, and programming web crawlers. Companies like Google probably also use some advanced version of regex to sift through your emails to find keywords for targeted advertising.

Where do I personally use regular expressions? Having dealt with survey data and public data for almost two decades now, regex helps clean up messy text entries like names, addresses, or parse strings combined in a single variable. Additionally, it is not unusual to end up with data in

PDFs. While now there are decent PDF-to-Excel or csv convertors, there is a good chance, some data columns will be messed up or combined together. For example, I extracted the data for this interactive data visualization in Flourish after parsing more than 4,500 pages from PDFs using a combination of regex and string functions. This took in all two to three days to get all the text search combinations right.

As I have mentioned in earlier guides as well, cleaning the data is most of the time spent with data. Running regressions or making tables is the easy part. So one also needs to learn to clean data efficiently. Unless one has access to a lot of resources or manpower (which is also a frequently utilized option in low-income countries), brute force cleaning or defining replace conditions one-by-one is a painfully arduous task. Plus, if one is talking about several thousands of entries, pattern matching is an important feature to learn.

Regular expressions are without a doubt, a very powerful tool. They also exist in Stata, and were upgraded in version 14 to a more generic Unicode from the more limited POSIX.2 version. This considerably expanded Stata's ability to do more complex string matches. But extremely little documentation exists on this topic. Besides some discussions on the Stata forums, and few old articles online, the help file for regex and related commands is as rudimentary as it can get.

In this document, I will cover the basics of regex and some applications to actual data. Two points here before we start. First, a disclaimer that I am still learning the language of regular expressions which is vast and there are several ways of defining the same algorithm. So there is a chance that a more advanced and a more compact syntax can be written. Second, a lot of resources, including tutorials and guides, exists online on regex. It is important to know that regex is implemented differently in differently languages and different standards are also used. Therefore, be careful in using specific websites for learning since not all the code can be directly used in Stata. In short, if you are using online resource to

learn more, you need to find the Unicode regex version. This guide will focus on the Unicode regex implementation in Stata combined with other built-in string functions.

This document is meant to evolve over time as I find new applications or improve the code. But in its current form, this guide covers most of the basics you need to know to get started.

Before we begin, here is the highly cited regex comic from XKCD that I feel I have to add here as well:

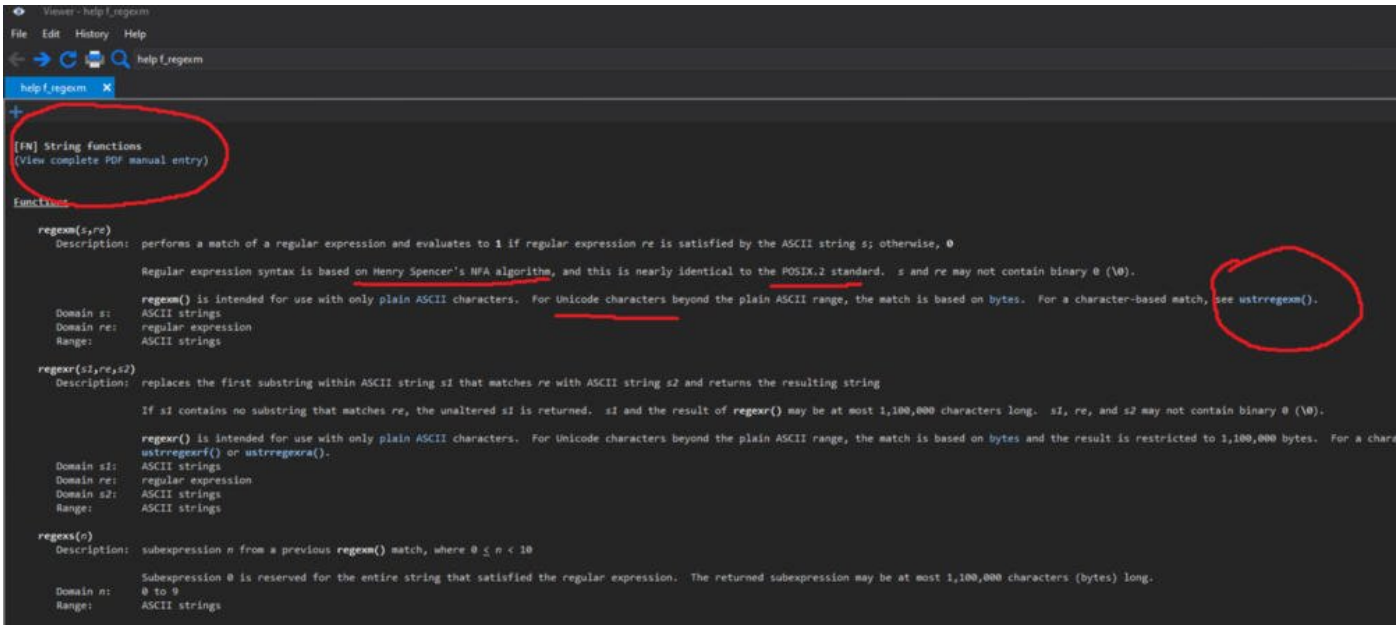
Source: XKCD (https://xkcd.com/208/)

Preamble

A basic knowledge of Stata is expected. It is preferable to have Stata 14 or higher since Stata 14+ has two regex versions. If you type:

```
help regex
```

you will end up on this page. This page drops a lot of hints but does not go in the details:



The first hint is that the original `regex` set of commands are based on the old POSIX.2 standard which is limited in its application, for example how it deals with spaces, character lengths etc. The second hint is that there is a new command `ustrregex` based on Unicode characters, which is more recent and more advanced. If you click on this, another minimally-defined help page pops up.

This guide will explicitly use the `ustrregex` version which has a different implementation from `regex` , which is what most of the existing Stata guides that one finds online use. In this guide, I will use the term regex as an abbreviation of regular expressions and not the command. I would also suggest that you learn the `ustr` version rather than the original

implementation.

For the guide below, I won't provide screenshots of the data browser window (accessed by typing `br` in the command window). But please have it open to see what is being generated. You can execute the code in a dofile and have the browser window open on the side to observe the changes in the data.

This guide is also not an easy one to write since there is no closed-form outcome like generate a graph or a table. It introduces the concept of regex, which is an open ended language so the aim of this guide is to explain how to "think" in regex language and build search functions from bottom-up.

If you are already familiar with regex or are an advanced user, then please send suggestions or code improvements! The purpose of this guide is to document this very useful tool for the broader Stata community.

Part I: Basics

Let's start with a simple example. Here we generate a cell with the following entry:

```
clear
set obs 1

gen x = ""
replace x = "abcdefghiJKLMN 11123" in 1
```

It contains small letters, capital letters and numbers. Let's generate a simple string pattern search:

```
gen t1 = ustrregexm(x, "def")
```

This tells Stata to find the string `def`, and if it exists return a 1, or 0 otherwise. Here `m` in `ustrregexm` stands for match. Once Stata finds the match, it also store the information on what it has matched. This can be recovered as follows:

```
gen t2 = ustrregexs(0) if ustrregexm(x, "def")
```

where `s` stands for sub-expressions, that I will refer to as tokens. Token 0 contains all the matched patterns. If one is finding more than one occurrences of a pattern then token 0 will contain all matches, token 1 the first one, token 2 the second one, and so on. We will come back to tokens later. The code above will simply generate `def` which is exactly what we want to find.

If we type this:

```
gen t3 = ustrregexra(x, "[def]", "")
```

where `ra` stands for replace, the string `x` is searched for `[def]`, which stands for find any one of d or e or f, and replaced with a missing. In other words, we are saying to return everything except the term we specify. So the code above will return the original string minus `def`. Additionally if we specify:

```
gen t4 = ustrregexra(x, "[^def]", "")
```

The `^def` means NOT `def` and the square bracket means also NOT `def`. In other words, this double negation implies that we remove everything except `def` so what we should get in the end if `def`.

Sounds a bit twisted way to get what we want but here the pattern is fairly straightforward. For more complex patterns, for example, that remove all numbers from a string, this is a highly useful command. Will also come back to this later.

This command can also be used to replace text in the original variable rather than generate new variables. Furthermore all `ustrregex` can be padded with an extra `,1` which means that searches should be case insensitive.

For example:

```
gen t4_1 = ustrregexra("ABDEFGHIJ", "[def]", "X", 1)
```

will find `def` irrespective of the case and replace it with an `X`. I won't go further in replacement or case sensitivity applications since they depend on the search you are conducting.

If we type these two:

```
gen t5 = ustrregexs(0) if ustrregexm(x, "[a-z]+")
gen t6 = ustrregexs(0) if ustrregexm(x, "[A-Z]+")
```

then we will recover the first set of small letters and capital in the string in the two variables.

For numbers we can do similar stuff:

```
gen t7 = ustrregexs(0) if ustrregexm(x, "[0-9]+")
gen t8 = ustrregexs(0) if ustrregexm(x, "1+")
```

The first command finds the first set of numbers in the string while the second one finds all the occurrence of all the 1's.

The above search patterns help us find text or numbers that follow some simple search pattern.

Part II: String functions

Here we start with some random text we generate ourselves:

```
clear

set obs 10

gen x = ""
replace x = "The quick brown fox jumps over the lazy dog." in 1
replace x = "the sun is shining. The birds are singing." in 2
replace x = "  Pi equals 3.14159265" in 3
replace x = "TheRe arE 9 plANetS in THE solar SYstem.  eARth's
mOOn is round" in 4
replace x = "I LOVE Stata 16 . " in 5
replace x = "Always correct the regressions for clustered standard
errors." in 6
replace x = "I get an error code r(997.55). What do i do next?" in
7
replace x = "myname@coolmail.com, Tel: +43 444 5555" in 8
replace x = " othername@dmail.net, Tel: +1 800 1337. " in 9
replace x = "Firstname  Lastname  03-06-1990" in 10
```

Here we have a mix of various characters. Single sentences, double sentences. Sentences with emails and telephone numbers, or all capital letters, characters like full-stops, commas, and brackets, and badly defined spaces.

Before we get into the regex stuff, here we also need to briefly discuss Stata's built-in string functions:

```
help string functions
```

This menu brings up a lot of options, most of which the users are probably not aware of. This also include other Unicode string functions that start with the `ustr` prefix. For more details on Unicode, what it is and how it works read these two help files and their entries in the Stata manual:

```
help unicode locale
help set locale_functions
```

Without going too much into the details of all the string functions, the following ones are useful to know in Stata:

```
gen temp1 = upper(x)
gen temp2 = lower(x)
gen temp3 = proper(x)
gen temp4 = trim(x)
gen temp5 = proper(trim(x))
```

Since we are dealing with plain alphabets, the above commands are sufficient. But if your text has various characters with accents, you can

replace the above with `ustrupper`, `ustrlower`, etc.

The first three change the letter cases which is obvious from their names. `upper` makes everything uppercase, `lower` is for lowercase, and `proper` capitalizes the first letter of each word separated by a space. For string matching with messy text, converting everything to upper or lower might make some tasks easier. The command `trim` gets rid of the blank spaces at the beginning or the end of the string entries. One can also use `ftrim` or `ltrim` to clean up spaces at the beginning (`f` = first) or at the end (`l` = last) of string variables. Such spaces typically appear when importing from poorly coded Excel files. Sometimes the leading zeros disappear while important data in Stata (e.g. 00001 will show up as 1 but the length will be preserved) so the generic `trim` might not always be the best option.

The last example shows that several string functions can be combined in one variable as well. For quick fixes, knowing such functions is really useful.

Another useful function is `length`:

```
gen diff = length(x) - length(temp4)
```

Here we compare the length of characters in the original data with the trimmed data. If `diff` is greater than zero, then this implies that spaces were removed.

Another highly useful feature is `wordcount`:

```
cap drop temp*
cap drop diff
```

```
gen count = wordcount(x)
```

This command counts the words that are separated by spaces. A more commonly used feature in Stata is the `split` command, that by default splits the variables based on spaces. One can also defined other characters for string parsing.

So if we split the data by parsing on the white space:

```
split x, parse(" ") gen(test)
```

we will get the data in new variable columns with the name prefix *test*. For each row, the filled columns should equal exactly the number shown in the *count* variable. For more advanced programming Stata has another function `tokenize` that helps parsing locals as well (e.g. with stored filenames or temp variables). If the data column that needs to be cleaned has a very structured string pattern, for example, all dates are written in this format dd–mm–yyyy, then one can use `split` and parse on the `p("-")` and get the dates, months and year.

Part III: Generic text searches

Going back to regex, we can now use slightly more advanced options:

```
cap drop t*

gen t1 = ustrregexs(0) if ustrregexm(x, "\w+")  // first word
```

The command above, returns the first alphanumeric character (letters and numbers) which is defined as `\w+`. Without the `+` we will just get

the first letter. The `+` sign means that we should keep searching until we hit a non-alphanumeric character. Note that `\w+` does not work with the original `regex` command.

In regex, we can also pick non-alphanumeric characters, which is defined as:

```
gen t2 = ustrregexs(0) if ustrregexm(x, "\W+")  // first non-word
```

or simply as the capital `W`. In the text we fed above, the visible non-alphanumeric character is the @ sign and the invisible ones are spaces.

Similarly for numbers:

```
gen t3 = ustrregexs(0) if ustrregexm(x, "\d+")  // first number
```

`\d+` picks the first set of consecutive numbers. It's opposite `\D+`:

```
gen t4 = ustrregexs(0) if ustrregexm(x, "\D+")
```

picks everything that comes before the first set of numbers. This way of searching is very different from a regular search and replace in Stata (e.g. using `subinstr`)and it takes some time to get use to this.

Let's briefly summarize the core functions of regex:

```
[jkl]    = j or k or l
[^jkl]   = everything except j,k, or l
[j|l]    = j or l
```

```
[a-z]    = all lower-case letters
[a-zA-Z] = all lower and uppercase letters
[0-9]    = find all numbers
```

```
\d       all numbers            \D        all non numbers
\w       all alphanumeric       \W        all non alphanumeric
\s       all spaces             \S        all non spaces
```

Let us now do a more complex search. We want to find all the “The” in the text. If we start with the simple option:

```
cap drop t*
gen t1  = ustrregexm(x,"The")
gen t2 = ustrregexs(0) if ustrregexm(x,"The")
```

This will mark everything correctly, except the fourth line where the first word is “There”. The variable t2, will return “The” as well for the 1st, 2nd, and 4th lines.

If we type this:

```
gen t3 = ustrregexs(0) if ustrregexm(x,"[t|T]he")
```

Then we get more matches, where both the capital and small letter “the” is captured. This will pick up the “the” that occur in the middle of the sentences as well. Remember this finds the first match irrespective of where it is.

We can also specify the `^` quantifier which states that we only find the “the” that starts a sentence:

```
gen t4 = ustrregexs(0) if ustrregexm(x, "^[t|T]he")
```

Compare line 2 of the variable t4 with t1. t1 found the capital “The” which was the start of the second sentence, while t4 finds the first “the” which starts the first sentence.

Next try this:

```
gen t5 = ustrregexs(0) if ustrregexm(x, "^[t|T]he\s")
```

Here we say that find the “the” that starts the sentence and is followed by a space `\s`. This eliminates the “There” in row 4 from the search.

Try these two options:

```
gen t6 = ustrregexs(0) if ustrregexm(x, "(^[t|T]\w+)")
gen t7 = ustrregexs(0) if ustrregexm(x, "(^[t|T]\w{2}\s)")
```

The variable t6 returns all the words that start with t or T and has any number of letters in front. The variable t7 returns words that start with t or T followed by two letters and a space.

Notice how we went from a specific search of a word to a more generic pattern. While these searches are all case dependent, the examples above show how an expression can be built up from simple rules to get what we want.

Let us now focus on sentences:

```
cap drop t*
gen t1 = ustrregexs(0) if ustrregexm(x, ".*")  // returns
everything
```

Here the option `.*` returns everything since `*` is a “greedy” quantifier. This generic function, for example, can be used to find patterns between a defined starting and ending point.

The next option:

```
gen t2 = ustrregexs(0) if ustrregexm(x, ".*\.")
```

Return all rows that end with a full stop. Note that to define a full stop, we need to use the `\.` option since `.` itself is reserved for generic searches.

If we specify this option:

```
gen t3 = ustrregexs(0) if ustrregexm(x, ".*\.$")
```

Then it will return all sentences that end with a full stop without any spaces between the last letter and the full stop.

And another option:

```
gen t4 = ustrregexs(0) if ustrregexm(x, ".*\.\s")
```

This returns, all the sentences that end with a full stop followed by a space. This option can be used, for example, to find sentences in a paragraph.

Here, I would like to introduce sub-expressions or tokens as well:

```
cap drop t*

gen t5_0 = ustrregexs(0) if ustrregexm(x, "(.*\.)(.*\.)")
gen t5_1 = ustrregexs(1) if ustrregexm(x, "(.*\.)(.*\.)")
gen t5_2 = ustrregexs(2) if ustrregexm(x, "(.*\.)(.*\.)")
```

Here we define two search options, as highlighted by the expressions in the two round brackets `()()`. Here we say that the first expression is anything that ends with a full stop. The second expression is exactly the same. In other words, we are finding two sentences. The variable t5_0 returns all the matches. While t5_1 and t5_2 return the first and the second matches respectively. Another option to try would be this search pattern `(.*\.)\s?(.*\.)` which says that there might be a space between the two sentences as defined by `\s?`.

Since `ustrregexm` matches the occurrences of our string, it is not always optimal to find sets of different combinations or to eliminate characters or words from a string. In this regards, `ustrregexra` provides a more powerful tool. Try these two variables:

```
gen t6 = ustrregexra(x,"[^a-zA-Z]","")
gen t7 = ustrregexra(x,"\W","")
```

The first option basically says keep only the letters, while the second option says drop all the non-alphanumeric characters. Note the difference between the two in the data browser window.

We can also introduce word boundaries using the `\b` operator. This limits the extent of the search to return an exact match. For example, if we want to search for an email, we can limit the boundaries to something like xxx@yyy.zzz. For the text given above, this can be

operationalized as follows:

```
cap drop t*

gen t1 = ustrregexs(0) if ustrregexm(x, "\b([a-zA-Z]+[_|\-|\.]?[a-
zA-Z0-9]+@[a-zA-Z]+\.[com|net]+)\b")
```

Where we open and close with `\b` . Then we say find some text containing letters. We allow the email to maybe (`?`) contain underscores `_` dashes `-` and dots `.` . After this there is more text and number searching, followed by @ and then the domain name that can end with .com or .net or whatever else we want to specify. For the last part we can also just say we want a three letter string `\s{3}` . Again, this all depends on where we are looking and what we are looking for but it is important to understand that one can slowly build up the expression to cover the all cases and possibilities we want to cover. So one can start with a very simple expression like `\w+@\w+\.\w{3}` , which is as basic as it gets for an email address, and keep adding conditions to it.

Part IV: Generic numeric searches

Moving on to numbers:

```
cap drop t*
gen t1 = ustrregexs(0) if ustrregexm(x, "[0-9]")
gen t2 = ustrregexs(0) if ustrregexm(x, "[0-9]+")
gen t3 = ustrregexs(0) if ustrregexm(x, "[0-9][0-9][0-9]")
```

The first variable t1 returns the first number, t2 returns the first set of numbers, and t3 returns the first set of three-digit numbers. For more orderly data, these simple search functions are sufficient.

We can also be a bit parsimonious in our expression by specifying:

```
gen t4 = ustrregexs(0) if ustrregexm(x, "\d{4}")
```

where `\d{4}` says find the 4-digit number. Here we can also specify a range, for example `d{2,4}` which finds 2 to 4 digit numbers, whichever meets the criteria first.

If we specify:

```
gen t5 = ustrregexs(0) if ustrregexm(x, "\d+-\d+")
```

Then we want to find digits that are separated with a hyphen. Or if we want to specify numbers that start with a `+` sign:

```
gen t6 = ustrregexs(0) if ustrregexm(x, "\+\d+")
```

Note here that `\+` is used for the symbol since `+` is reserved for matching. We can start becoming a bit more fuzzy with our search by typing:

```
gen t7 = ustrregexs(0) if ustrregexm(x, "\+?\d+")
```

which says that the number “may” start with a plus sign. Anything that ends with a `?` gives a bit more flexibility.

This command:

```
gen t8 = ustrregexs(0) if ustrregexm(x, "(\d*\.\d+)")
```

Find all numbers that have a decimal `\.`. The `*` says that it may or may not start with number, while the number has to end with a digit since we specify `+`.

For finding the numbers in the data above, we can iteratively build an expression starting with the following:

```
cap drop t*
gen t1 = ustrregexs(0) if ustrregexm(x, "\d+")
```

here we say find all the digits. Now we want to add the plus sign as well to the numbers. Since not all the numbers start with one, we can define this as follows:

```
gen t2 = ustrregexs(0) if ustrregexm(x, "\+?\d+")
```

Next we can say that we want to add more characters to the number:

```
gen t3 = ustrregexs(0) if ustrregexm(x, "\+?\d+([\s|\.|-]?)")
```

which implies that after the number, there can be a space `\s`, a decimal `\.` or a hyphen `-`. Next step, we say that we want to add the remaining digits if they exist:

```
gen t4 = ustrregexs(0) if ustrregexm(x, "\+?\d+([\s|\.|-]?)
(\d+)?")
```

and we can add more of the same to cover all the numbers:

```
gen t5 = ustrregexs(0) if ustrregexm(x, "\+?\d+([\s|\.|-]?)(\d+)?
([\s|\.|-]?)(\d+)")
```

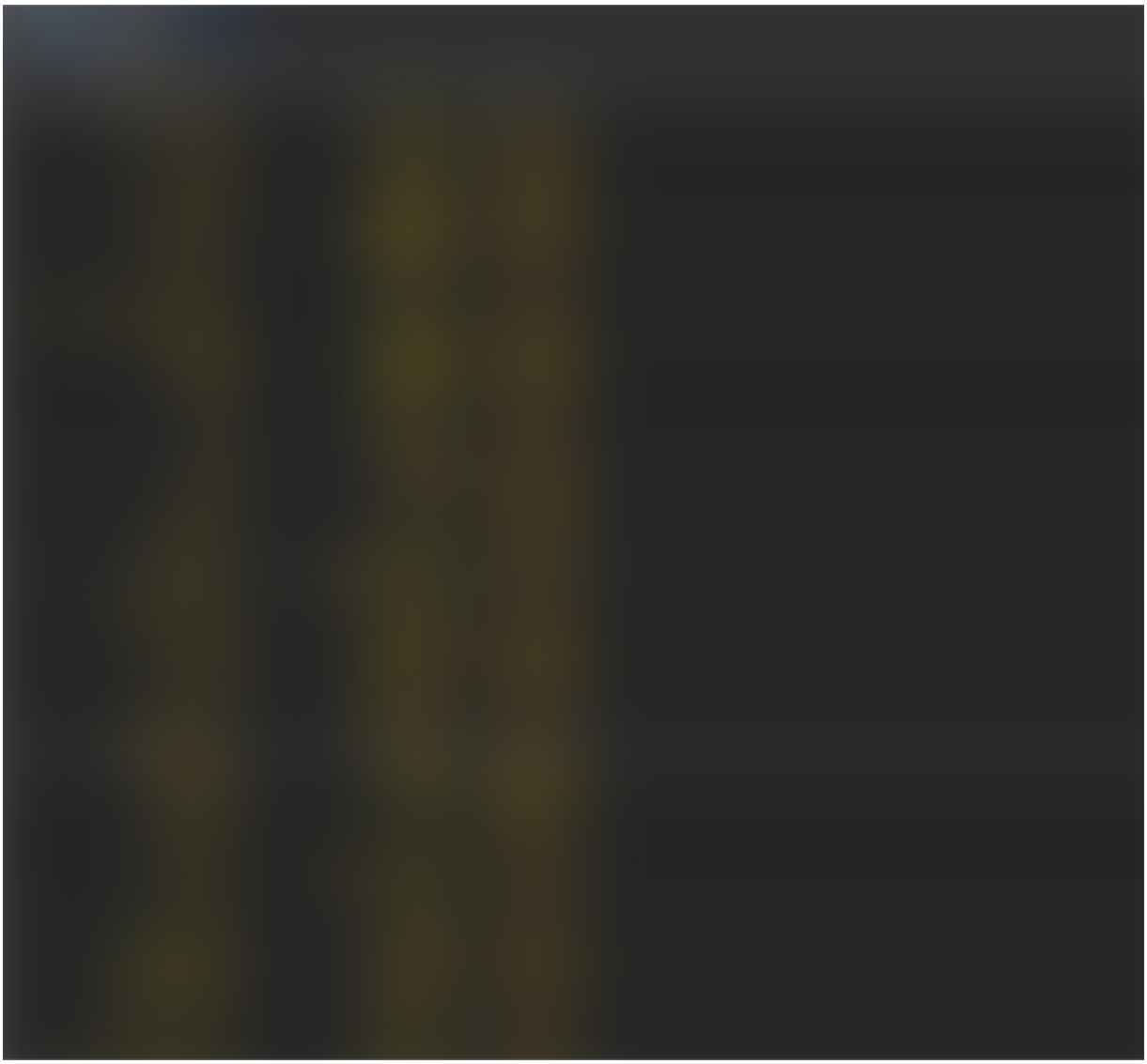
If one looks at the expression in first glance, it looks like a mess of letters and symbols but the logic is clearly defined and can be carefully interpreted. This code above is also not the most efficient since regex allows for repeating patterns by looking forward or looking backward to check for conditions. These topics are not (currently) covered in this guide but will be added later.

Actual data I: Names and date of birth

Let's load an actual data file:

```
import delimited using "https://raw.githubusercontent.com
/asjadnaqvi/COVID19-Stata-Tutorials/master/raw/file2.csv", clear
bindq(strict) varn(1)
```

The file below is a subset of standardized Grade 5 exams where roughly 1.4 million students are tested by the Punjab Education Commission (PEC) each year in the province of Punjab in Pakistan. The data is from a random exam year and has been stripped of identifiers and exam results:



Here we do some simple exercises. Split the name into first name and last name, and split the birthday into year, month, and day. Which one is the month and which one is the date is anyone's guess but it is not uncommon to run into this problem. Even csv to Excel conversions can mess up the date formats.

The names can be split as follows:

```
gen name1 = trim(proper(ustrregexs(1))) if ustrregexm(name, "([A-
Z]\w+)\s?([A-Z]\w+)?")
gen name2 = trim(proper(ustrregexs(2))) if ustrregexm(name, "([A-
Z]\w+)\s?([A-Z]\w+)?")
```

Here we use the same expression and just call it twice for the first and second tokens. Note that here we also use trim and proper in the same go. The expression says that start with a capital letter followed by any type of letters (capital or small). Since this is in a bracket, it is the first token. Next there may or may not be a space `\s.` followed by a potential second name. Since some kids don't have second names specified in the dataset, the `?` sign after the space and the second token covers this case.

For the date of birth (dob) we can get rid of the 0:00 first:

```
replace dob = trim(subinstr(dob,"0:00","",.))
```

And then we can extract the year variable as:

```
gen year = ustrregexs(0) if ustrregexm(dob, "\d+$")
```

This says pick the number that ends the variable since we have the `$` sign specified.

For the first date we can simply say:

```
gen val1 = ustrregexs(0) if ustrregexm(dob, "(\d+)")
```

and for the second date, this can be defined based on the preceding `/` or `-` characters:

```
gen val2 = ustrregexs(1) if ustrregexm(dob, "[/|-](\d+)")
```

Here we don't want to recover the full match but just the expression in the brackets so please note the use of `ustrregexs(1)` rather than `ustrregexs(0)` which would also return the `/` or the `-` sign.

Once these variables are extracted, they can be converted into numeric (`destring`), and some logic can be used to combine them in a proper date (`gen date = mdy()`)

Actual data II: School census

Here I will use a partial dataset of school names that is extracted from a random school census in Punjab, Pakistan.

```
clear

https://raw.githubusercontent.com/asjadnaqvi/COVID19-Stata-Tutorials/master/raw/file1.csv
```

I am using a sub-sample of the full data where this dataset has some 590 rows out of some 43,000 plus schools. If one browses the school name column, then there are several pieces of information stored in the string:

School names are either written in full, or abbreviated and are provided in all possible configurations: boys/girls, primary/secondary/middle /high. Additionally, the variable also contains the name of the school, the village name, and the gender information. To make this clear the following abbreviations are used: GPS = Government Primary School, GGPS = Govt. Girls Primary School. The P can be replaced with E = Elementary, M = Middle, H = High. So a Government Girls Middle School can be written as GGMS, or GOVT. GIRLS M/S. Some schools are also location in Basic Health Units (BHUs) which are also both abbreviated or written in full.

I have no idea why the education department would do this. At the point of data entry one can restrict the fields in databases. Such datasets pose an additional challenge if one tries to match names across different census rounds where in some cases unique IDs also change. Regardless, this is a very real challenge when it comes to dealing with actual data.

We can generate different school type variables based on the conditions we are specifying.

The first condition we can specify is a relatively easy one:

```
gen schooltype1 = ustrregexs(0) if ustrregexm(school, "(G?[G|B]
[E|P|M|H|S]S)")
```

Here we say that the abbreviated name can start with a G (representing government) followed by B or G (boy or girl), and then followed by E,P,S,M, or H (school level), and lastly S representing School. This already identifies 45% of the data:

```
tab schooltype1, m
```

Next is a bit more of a complex setup:

```
gen schooltype2 = ustrregexs(0) if ustrregexm(school, "(GOVT)?\.?
\s?((BOY|GIRL)S?)?\s?((E|M|P|H|S)/?\\?S(CHOOL)?)?")
```

If we read this carefully, we are saying the following:

The name may start with GOVT, which may be followed by a . and/or a

space. Next comes BOY or GIRL that can also be plural (BOYS or GIRLS). Next there can be a space followed by school type, followed by maybe a forward slash `/`, followed by S that can also be written in full. If we run this command, we identify another 45% of the schools. If you browse this variable, then you will notice that some errors still persist since not all the fields are captured correctly. These I leave as exercises to fix :)

One can also use a different logic. One can generate conditions to elicit different fields. For example gender can be one field, school level another. This is all subjective and one learns to use different methods after getting a “feel” for the data.

For the BHU data we can use this command:

```
gen schooltype3 = ustrregexs(0) if ustrregexm(school, "B(ASIC)?\.?
\s?H(EALTH)?\.?\s?U(NIT)?\.?")
```

This essentially captures most of the combinations of writing BHUs, which can be spelled out, contain dots, or spaces.

A simple check variable:

```
gen check = schooltype1!="" | schooltype2!=""  | schooltype3!=""
tab check
```

Shows that we have identified over 91% of the schools.

The gender information, that is given in brackets at the end of the name is properly homogenized and can be extracted as follows:

```
gen gender1 = ustrregexs(0) if ustrregexm(school, "\(([A-Z]+)\)")
```

```
gen gender2 = ustrregexs(1) if ustrregexm(school, "\(([A-Z]+)\)")
```

The first option returns the names with the brackets while the second only returns information within the brackets.

Assuming that we have generated everything as we want it, we can "subtract" the variables from the original column to reduce it to the remaining information. This can be done using the standard string functions in Stata as follows:

```
replace school = subinstr(school, schooltype1, "", .)
replace school = subinstr(school, schooltype2, "", .)
replace school = subinstr(school, schooltype3, "", .)
replace school = subinstr(school, gender1, "", .)

replace school = trim(school)
```

This of course can be refined further to cover all the categories and fix the errors. For example, if one scrolls down, some schools are showing up in `""`. This can already be cleaned in the first step using the `subsinstr` option to remove the extra quotes.

Cheat sheet

There are a lot of regex cheat sheets around. Here are the core ones to remember from this Guide:

- Character matching:

```
[jkl]    = j or k or l
[^jkl]   = everything except j,k, or l
[j|l]    = j or l
[a-z]    = all lower-case letters
[a-zA-Z] = all lower and uppercase letters
[0-9]    = find all numbers
```

```
\d       all numbers           \D        all non numbers
\w       all alphanumeric      \W        all non alphanumeric
\s       all spaces            \S        all non spaces
```

- Quantifiers:

```
^  matches the beginning of a string
$  matches the end of a string
.  matches any character
|  the separator for or. Same as in Stata
?  matches zero or one instance
*  matches zero or more instances
+  matches one or more instances
```

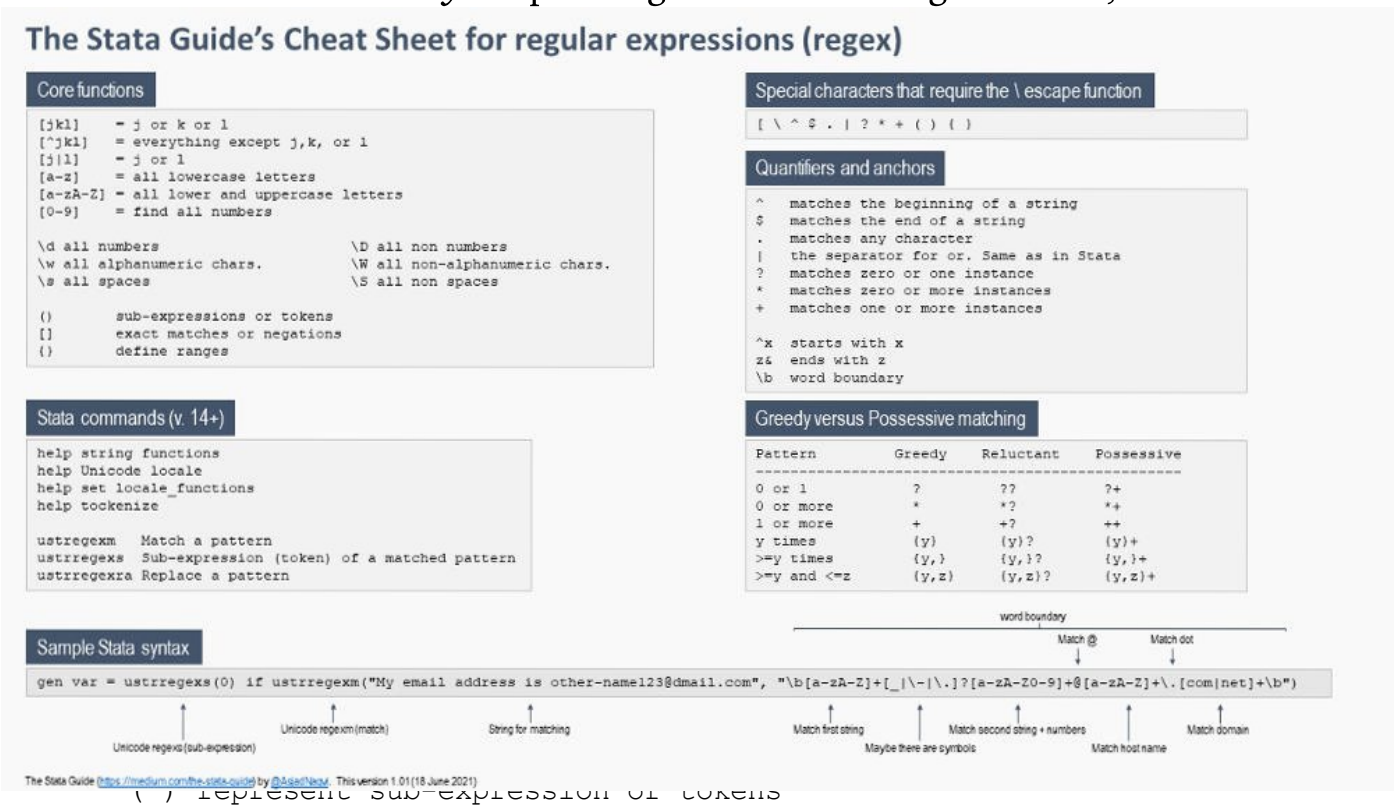
These following quantifiers are extremely important in regular expressions and can be extended to distinguish between a “greedy” versus “possessive” quantifiers:

```
Pattern             Greedy          Reluctant        Possessive
-----------------------------------------------------------------
0 or 1                ?                ??                ?+
0 or more             *                *?                *+
1 or more             +                +?                ++
y times               {y}              {y}?              {y}+
>=y times             {y,}             {y,}?             {y,}+
>=y and <=z           {y,z}            {y,z}?            {y,z}+
```

To summarize, “greedy” tries and find everything that meets the criteria while “possessive” matches aim for exact precision. These make a lot of difference on search speeds depending on the size of the dataset. A lot of natural language processing (NLP) datasets, for example archives of newspaper articles or tweets, which can be found on websites like Kaggle, and can be hundreds of gigabytes in size.

Example or a greedy match: `cool (.+) hat` will try and find everything

between a `cool` and the last occurrence of the word `hat` and hence *greedy*. `A cool (.+?) hat` will stop at the first occurrence of `hat` and hence *reluctant* or *lazy*. Depending on what is being searched, one can



The cheat sheet will be updated to including other advanced operators as well.

Example: `(\w+)\.?(\d+)` will return two tokens since there are two round brackets

Hope you like this guide! Please send suggestions, feedback, comments, requests, if you have any. Your support keeps these guides going.

`[ ]` is used for exact matches or exact negations

About the author

Example: `[abc]` matches a or b or c, `[^abc]` match anything except a or b or c, `[a-zA-Z0-9]` matches any letter or number,

I am an economist by profession and I have been using Stata since 2003. I am currently based in Vienna, Austria where I work at the Vienna University of Economics and Business (WU) and at the International Institute for Applied Systems Analysis (IIASA). You can find my research work on ResearchGate and Google Scholar, and Stata code repository on GitHub. You can follow my COVID-19 related Stata visualizations on Twitter. I am also featured on the Stata COVID-19 webpage in the visualization and graphics section.

`{ }` is used to define match ranges

Example: `A{3}` find matches where A occurs 3 times, `x{3,6}` matches where x occurs between 3 and 6 times, `\w+{4}` matches a four letter

You can connect with me via Medium, Twitter, LinkedIn or simply via email: *asjadnaqvi@gmail.com*.

My Medium blog, The Stata Guide, releases awesome new content regularly. Clap, and/or follow if you like these guides.