Regular Expressions

William Totten

University of Delaware

April 14, 2015

Learning about regular expressions to help extract content from text, data from content, and information from data.

Regular Expression Examples

A regular expression is also known as an RE. It can range from the simple (any piece of text is technically a regular expression) to the complex and difficult to comprehend.

^.*$ Match a string which may, or may not have contents.

a Match a string that contains the character "a" somewhere (anywhere).

(http|https|ftp|ftps|rtmp)://[\w.]+/.* Match a string containing a URL (e.g. http://server.udel.edu/web/page.html).

^([^,]*,){3}foo,ba[rz],.*[0-9]$ Match a CSV file which contains the entry "foo" in the 4th column, either bar or baz in the 5th column, and ends with a number.

^M{0,4}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$ Match strings which are Roman numerals.

To understand what all this stuff means, why it is useful, and where you can use it we need to take a couple steps back.

The Chomsky Hierarchy of Languages

Type 0 (unrestricted grammars) Also known as Recursively Enumerable grammars. All language which can be recognized by a Turing machine (aka a computer).

Type 1 (context-sensitive) All languages which can be recognized by a bounded automaton with a reasonable section of surrounding text. Mathematically expressed $\alpha A \beta \longrightarrow \alpha \gamma \beta$ with A being a non-terminal. Think of natural languages, like English, as being the bottom boundary of context-sensitive languages. If a grammar was more complicated than English, it is most likely context sensitive.

Type 2 (context-free) Languages where text beyond terminating punctuation is irrelevant to the ability to recognize a statement. To compare, mathematically, they are expressed simply as $A \longrightarrow \gamma$ with A being a non-terminal. Computer programming languages are an excellent example of context-free languages.

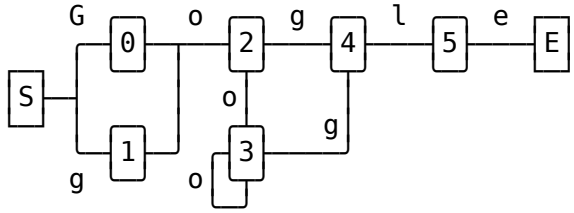
Type 3 (regular) Any language consisting of a single terminating punctuation. Regular grammars can match individual lines of a programming language, but not the whole thing. This is the simplest form of a language, but it is still quite useful. Regular languages share the property of all being recognizable by something called a finite state automaton.

Finite State Automata and Regular Expressions

A regular expression is simply a syntax for describing the Finite State Automaton(FSA) (aka Finite State Machine) you need to recognize your language. Computer software compiles your RE into an FSA for you. For example, imagine we want to match instances of the name "google", knowing they like to play with the number of "o"s in their name:

[Gg]oo*gle

This is an RE which matches 1 or more "o" characters in the middle of a name, and this is a simplified version of the FSA which is generated to match the name:



If state "E" is reached, then the match is complete, and the "text" is recognized as part of the "language". As the complexity of the RE grows, so does the FSA.

Back to practicality

So REs are limited in what they can match, and they work because the computer software converts them into a Finite State Machine and uses that to match the text. But, *how* do we write the REs? To answer that, we must first understand the different types of REs. There are almost as many RE implementations as there are languages which allow regular expressions. And, you will have to look into manuals for specifics for your language. Below are some major players which define groups of compatibility.

Type	**Description**
SQL (LIKE)	A single wild chard character "%" matching anything
Wild Cards (glob)	A limited syntax used primarily to match file names
Basic (RE)	More true to the original, with a limited feature set
Extended (ERE)	More features, and a couple incompatibilities with REs
Modern (MRE)	A super set of all of the above plus way more

Syntax and compatibility

A regular expression is a bunch of syntax used to match data in as concise a way as possible.

glob	RE	ERE	MRE	
?	.	.	.	Match any single character
[ ]	[ ]	[ ]	[ ]	Define a character class of specific characters
[^]	[^]	[^]	[^]	Define a character class of specifically not certain characters
	^	^	^	Match(anchor) against the beginning of a string
	$	$	$	Match(anchor) against the end of a string
	\< \>	\< \>	\b	Word Boundary
	\(\)	( )	( )	Grouping and backrefrence definitions
	\1	\1	\1	Backreferences
*	*	*	*	Match occurrence 0 or more times
		?	?	Match occurrence 0 or 1 times
		+	+	Match occurrence 1 or more times
		a|b	a|b	Alteration (match "a" or match "b")
		{n}	{n}	Match exactly *n* occurrences
		{n,m}	{n,m}	Match between *n* and *m* occurrences
			*?	Non-greedy 0 or more occurrence match
			+?	Non-greedy 1 or more occurrence match

Among modern implementations there are a number of small differences, but they are all consistent with respect to the above functionality. This includes perl, python, ruby, java, javascript, .NET, C++, R, and much more.

Wild Cards (File Globbing)

Lots of people are familiar with the simplest form of regular expressions, they look like ***.txt** . We call these wild cards, but the same science which underpins wild cards is the basis for regular expressions.

Wild cards are supported in a variety of tools, included UNIX shells, DOS command prompts, and more. The syntax is simple, and very consistent across implementations. Some examples are:

***.txt** All files ending in .txt.

file?.dat All files starting with file, ending with .dat, and containing one character between.

file[0-9].dat Just like the previous example, but ensure the variable character is a number.

[abcd]* All files starting with the letters 'a', 'b', 'c', or 'd'.

[^A-Z]*.?? All files which don't start with a capital letter, and have a two character extension. (eg. fileA.bk).

One important difference between the * character in Wild Cards versus Regular expressions, is that it is not a modifier in Wild Cards; it just matches everything. As we will see later with REs, what comes before a * is critical.

Using Wild Cards

Wild cards (aka file globs) are available via your shell in UNIX, and so are available for use with entering interactive commands, shell scripts, and qsub scripts.

rm ?-errors.dat You could use this command if you had a files named "α-errors.log" and "β-errors.log", and were having trouble deleting them, this command will help.

for file in *.txt; do mv $file ${file%.txt}.csv; done Rename every file ending with a ".txt" extension to have a ".csv" extension.

grep ERROR application.log.[456] Will print lines containing the word ERROR from the files "application.log.4", "application.log.5", and "application.log.6" (assuming they exist).

Regular Expression Shell Tools

These tools can be useful on the command-line, in shell scripts, and qsub scripts.

Tool	Type	Description
grep	RE	Search for lines matching an RE, and print them
egrep	ERE	Search for lines matching an ERE, and print them
pcregrep	MRE	Search for lines matching an MRE, and print them
sed	RE	A stream editor, to manipulate text pragmatically
awk	ERE	A programming utility with RE's as a central feature
perl	MRE	A programming language designed with RE's as a key feature
bash [[	ERE	The built-in extended test feature of bash supports REs

Characters and Character Classes

.

The . (dot) character matches any character in a regular expression. You will often see this character with a modifier (*, +, or ?) after it. But, it is also very useful by itself.

[class]

A character class of specific characters may be defined. It may also include ranges. So, "[abcdefg]" and "[a-g]" are equivalent. Placing a "^" (caret) at the start of the class will negate it's meaning. So, "[^a-z]" means all characters which aren't lower-case letters; it matches upper-case, numbers, punctuation, etc.

Notes on character classes

There are generally four characters you need to be mindful of when specifying a character class:

-	The "-" (dash) character is a range operator. Place it first, last, or after a backslash character to protect it from declaring a range.
^ !	The "^" (caret) and "!" (exclamation point) characters invert the meaning of a character class. Don't put them first, or put a backslash before them for protection.
****	The "\" (backslash) character is used to protect other characters, and so must also be protected.

Anchors and Boundaries

^

It is sometimes useful to ensure that an RE match occurs at the beginning of a string. The "^" (caret) character can be used to enforce this condition. An RE of "^foo" would match any string which starts with the letters foo.

$

If you need ensure a match occurs up-to the end of a string, the "$" (dollar sign) character will enforce this condition. An RE of "bar$" would match any string which ends with the letters bar.

\<, \>, and \b

Regular expression engines are usually coded with a concept of a word boundary. Depending on the implementation, these word boundaries may be recognized by using "\<" and "\>" before and after a word match or "\b" on either side. In many languages, both methods are supported. For example, "\bmatch\b" would recognize " match ", "(match)", " match.", etc. But, it would not recognize "rematch" or "matches" .

Notes on anchors and boundaries

These expressions are also part of a special class of the RE syntax called **zero-width** expressions; the part of a match associated with these types of expressions is empty.

Grouping, Backreferences, and Alteration

( )

Grouping is a very powerful feature of regular expression, especially when combine with programming languages which store the groups for use outside of the match (e.g. substitutions). Groups can be used to bind together characters to be modified or alternated together.

\1

Backreferences can be used re-use a previously matched group. Use a "\" (backslash) character, followed by a number.

|

Alteration syntax uses a "|" (pipe) character to specify multiple possible matches. These may be used inside or outside of groups.

Example matching HTML font changes

```
<(H[1-4]|FONT|TT)>.*</\1>
```

Modifiers

*****

Known as the Kleene star, this modifier will direct a regular expression to only match the previous character, character class, or group 0 or more times.

+

Known as the Kleene plus, this modifier is just like the Kleene start, but matches. The Kleene plus, is just an extension of the 0 or more star operator. As an example, ".+" is identical to "..*".

?

The optional modifier specifies that the character, character class, or group it is modifying may exist exactly 0 or 1 times, and no more.

{n} and {n,m}

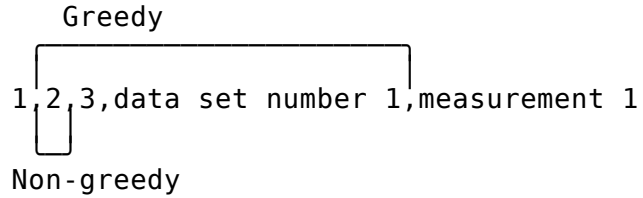
When necessary, either an exact number of matches (or a range) is allowed also. These modifiers require that a character, character class, or group must exist exactly *n* or between *n*, and *m* (inclusive) times respectively.

Non-Greedy Matching

Nearly all regular expressions are denoted as greedy. This means that the Kleene star (*) and Kleene plus (+) modifiers match as much text as possible. It is important that this behavior is well-defined, so REs can be developed with a proper understanding of how they will match. However, this is not always desired.

Many programming languages (all MRE engines) recognize a standard method to instruct matches to be non-greedy. This means, as little text is matched as possible to recognize a string. The way to enable this feature is by affixing a "?" (question mark) character after the Kleene modifier.

Imagine a CSV file and the regular expressions **,.*,** and **,.*?,** :



Pre-defined Character Classes

This table list some of the most common character and character classes

Sequence	Description
\w	Any alpha-numeric character, identical to [a-zA-Z0-9_]
\W	The opposite of \w, identical to [^a-zA-Z0-9_]
\d	Numbers/digits, identical to [0-9]
\D	The opposite of \d, identical to [^0-9]
\s	Match white-space, identical to [\t\r\n]
\S	The opposite of \s, identical to [^ \t\r\n]
\.	Use this to recognize a period, identical to [.]
\t	Matches tabs
\r	Matches carriage returns
\n	Matches new-line characters

Using Regular Expressions

REs are powerful tools which allow you find (and update) data in an efficient manner. Extracting data from and manipulating unstructured data would be much more difficult to encode into a program without REs.

Before using REs in your code, please think about the maintainability of them. Ensure that whomever comes after you is likely to be capable of supporting them. This is an important generality to programming, but applies especially with respect to regular expressions.

Two things to think about before using regular expressions.

Jamie Zawinski (1997)

> Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.

Voltaire (c. 1730) and Stan Lee (1962)

> With great power comes great responsibility

Additional Reading

- http://en.wikipedia.org/wiki/Chomsky_hierarchy
- http://www.regular-expressions.info/tutorial.html
- http://regexone.com/
- http://qntm.org/files/re/re.html
- http://linux.die.net/man/1/grep
- http://perldoc.perl.org/perlre.html
- https://docs.python.org/2/library/re.html
- https://stat.ethz.ch/R-manual/R-devel/library/base/html/regex.html
- http://www.cplusplus.com/reference/regex/
- http://fortranwiki.org/fortran/show/regex_module
- https://docs.oracle.com/javase/tutorial/essential/regex/
- http://www.mathworks.com/help/matlab/matlab_prog/regular-expressions.html
- http://www.w3schools.com/js/js_regexp.asp
- http://php.net/manual/en/refs.basic.text.php
- http://www.postgresql.org/docs/9.0/static/functions-matching.html
- https://dev.mysql.com/doc/refman/5.1/en/regexp.html
- http://docs.oracle.com/cd/B19306_01/appdev.102/b14251/adfns_regexp.htm
- https://support.google.com/docs/answer/3098244?hl=en

Questions ?

If not, let's start a tutorial

http://regexone.com/

If you want more practice

http://qntm.org/files/re/re.html