

Exploring Regular Expression Feature Usage in Practice and the Impact on Tool Design

Carl Chapman and Kathryn T. Stolee
Department of Computer Science
Iowa State University
{carl1978, kstolee}@iastate.edu

Abstract—Regular expressions are used frequently in programming languages for form validation, ad-hoc file searches, and simple parsing. Due to the popularity and pervasive use of regular expressions, researchers have created tools to support their creation, validation, and use. Each tool has made design decisions about which regular expression features to support, yet, there does not exist an empirical study of regular expression feature usage to inform these design decisions.

In this paper, we explore regular expression feature usage, focusing on how often features are used and the diversity of regular expressions from syntactic and semantic perspectives. To do this, we analyzed 3,898 open source Python projects from GitHub. Our results indicate that the most commonly used regular expression features are also supported by popular research tools and that programmers frequently reinvent the wheel by writing identical or nearly identical regular expressions in different ways.

I. INTRODUCTION

In essence, regular expressions are search patterns for strings. Regular expressions are used extensively in many programming languages, for example, to search text files [1], in form validation [], and for XYZ []. Due in part to their pervasive use across programming languages, many researchers and practitioners have developed tools to support the creation [], validation [], and testing [] of regular expressions.

In writing tools to support regular expressions, tool designers make decisions about which features to support and which not to support. These decisions are often made casually and are dependent on the regular expressions the designers happen to have experience with, the designers have seen in the wild, or their complexity. The goal of this work is to bring more context and information about regular expression feature usage so these decisions can be better informed.

To motivate the study of regular expressions in general, we scrape GitHub for Python projects that use the `re` module, which is the regular expression library for Python. We measure how frequently regular expressions appear in projects, and after doing a feature analysis (e.g., kleene star, literals, and capture groups are all features), we further measure how often such features appear in regular expressions and in projects. Then, we compare the features to those supported by four common regex support tools, `brice` [], `hampi` [], `Rex` [], and `RE2` []. We then explore the features not supported by common tools and explore the impact of omitting those features. Our results indicate that these tools support all of the top six most common features and that some of the omitted features, such as the lazy

quantifier, are used in over 35% of projects containing regular expressions.

The contributions of this work are:

- An empirical analysis of the usage of regular expressions in XYZ open-source Python projects
- An analysis of which features are omitted from common regular expression tools and the impact of ignoring those features
- A discussion on the semantic similarity of regular expressions in practice and identification of opportunities for future work in supporting programmers in writing regular expressions.

The rest of the paper is organized as follows. Section II motivates this work by discussing research in supporting programmers in the use, creation, and validation of regular expressions. Section III presents the research questions and study setup for exploring regular expressions in the wild. Results are in Section IV followed by a discussion in Section V and a conclusion in Section VI.

II. MOTIVATION AND RELATED WORK

With regexes, there is a common saying: ‘now you have two problems’. A skilled programmer can quickly solve many problems using regular expressions, but these regular expressions can be hard to understand and maintain, resulting in tens of thousands of bug reports [2]. Regular expression languages enable an irreplaceable search technique used within all kinds of text editors, command line tools and system tools. Regexes are also employed in critical missions like mysql injection prevention, malicious packet filtering and web form validation.

Tools like `Hampi`, `Kudzu`, `brics`¹, Microsoft’s `Rex`, `Automata` and the `z3`-based `QF_VRE` projects, all attempt to support modeling some subset of regular expression language features, empowering users to do reasoning and validation on regexes.

To improve test coverage for code using regular expressions, and to generate strings from regular expressions for whatever other reasons, projects like `Reggae`, `Rex`, `JST`,

¹<http://www.brics.dk/automaton/>

regex-tester², regldg³, uttool⁴, xeger⁵, Genex⁶, Hoa/Regex⁷, Genex⁸, Randexp⁹, txt2re¹⁰, Pex¹¹ (and countless others) have been developed.

Going the other direction, a few projects have attempted to take a set of strings and generate a good regular expression that matches them like RegexGenerator++¹² and Regex-PreSuf¹³ (a problem that suffers from overmatching).

One common misconception is that all regular expression languages are *regular languages* which can be represented using deterministic finite automata (DFA), and so they are easy to model, easy to describe formally and execute in O(n) time. In fact, most regular expression matching engines run in exponential time¹⁴ in order to support useful features such as lazy quantifiers, capturing groups, look-aheads and back-references¹⁵. In the RE2¹⁶ project, Russ Cox aimed to use DFAs as much as possible (maximizing speed) while supporting as many useful features as possible.

Thousands of research papers have focused on various other regular expression-related investigations. Because regular expression languages vary somewhat in their syntax and feature set, these papers have typically had to describe a particular language to reason about and have had to pick what features to include or exclude.

In all of these regex-related projects, researchers and tool designers face a difficult design decision: supporting advanced features is always more expensive, taking more time and potentially making the tool or research project too complex and cumbersome to execute well. A selection of only the simplest of regex features is common in research papers and automata libraries, but this limits the applicability/relevance of that work in the real world.

The authors of this paper have their own Regex-related tool that they want to implement, and when faced with the inevitable question of what features are okay to exclude, they searched for some empirical research into how regular expressions are used in practice. Finding no such research that could inform this choice, they decided to do that research themselves and hopefully empower other researchers with that information.

TODO: incorporate references

²<https://github.com/nickawatts/regex-tester>

³<http://regldg.com/>

⁴<http://uttool.com/text/regexstr/default.aspx>

⁵<https://code.google.com/p/xeger/>

⁶<https://github.com/mifmif/Genex>

⁷<https://github.com/hoaproject/Regex/>

⁸<http://search.cpan.org/~bowmanbs/Regexp-Genex-0.07/lib/Regexp/Genex.pm>

⁹<https://www.ruby-toolbox.com/projects/randexp>

¹⁰<http://txt2re.com/>

¹¹<http://research.microsoft.com/en-us/projects/pex/>

¹²<http://regex.inginf.units.it/>

¹³<http://search.cpan.org/~jhi/Regex-PreSuf-1.17/PreSuf.pm>

¹⁴<https://swtch.com/~rsc/regexp/regexp1.html>

¹⁵<https://msdn.microsoft.com/en-us/library/0yyc2yb0.aspx>

¹⁶<https://github.com/google/re2>

III. STUDY

To understand how programmers use regular expressions in Python projects, we scraped 3,898 Python projects from GitHub, and recorded regex usages for analysis as described in Section III-B.

Using this data, we intend to answer the following research questions:

RQ1: How often is the `re` module used in Python projects?

To address this research question, we measure how often any calls are made to the `re` module per file and per project in Python projects.

Furthermore, we measure the frequency of usage for calls to the 8 functions of the `re` module (`re.compile`, `re.search`, `re.match`, `re.split`, `re.findall`, `re.finditer`, `re.sub` and `re.subn`) in Python projects scraped from GitHub.

We also measure usage of the 8 flags (`re.DEFAULT`, `re.IGNORECASE`, `re.LOCALE`, `re.MULTILINE`, `re.DOTALL`, `re.UNICODE`, `re.VERBOSE` and `re.DEBUG`) of the `re` module.

RQ2: Which regex language features are most commonly used in python?

We consider regex language features to be tokens that specify the matching behavior of a regex pattern, like the `+` in `ab+`. All studied features are coded and described in III-B with examples.

To measure feature usage, we parse Python regular expression patterns using Bart Kiers' PCRE parser, as described in Section III-B. We then count the number of usages of each feature per project, per file and as a percent of all distinct regex patterns.

RQ3: What are Features being used for, and so what is the impact of *not* supporting various regex features on tool designers and users?

We use semantic analysis to illustrate the impact of missing features on a tool's applicability, identifying what each feature (or group of features) is commonly used for.

Semantic analysis is accomplished by first establishing a similarity matrix between regexes using a set of strings that match each regex, generated by Rex. Then clusters of regexes with similar behavior are discovered using Markov Clustering¹⁷. These clusters are used to interpret what a feature is used for.

Since our semantic analysis is based on Rex, this semantic analysis cannot be applied to all features studied. For these unsupported features, we use 6 string similarity metrics (Jaro-Winkler, Levenshtein, Longest Common Substring, Sift3, Jaccard and Cosine) to build similarity matrices. As before, these matrices are used to find clusters of regexes, which are used to interpret what a feature is used for.

We chose Rex to build matching strings because it supports the most features of any String-generation tool. We chose the

¹⁷<http://micans.org/mcl/>

mcl clustering tool because it offers a fast and tunable way to cluster items by similarity (without knowing the number of clusters in advance).

A. DEFINITIONS

Throughout the rest of this paper, we will employ the following terminology:

Utilization: A *utilization* occurs whenever a developer uses a regex engine in a project. Within a particular file in a project, a *utilization* is composed of a function, a pattern and 0 or more flags. Here is an example of one regex *utilization*, with key components labeled:

function	pattern	flags
<code>r1 = re.compile('</code>	<code>(0 -?[1-9][0-9]*)\$'</code>	<code>, re.MULTILINE)</code>

Fig. 1. example of one regex utilization

Pattern: A *pattern* is an ordered series of regular expression language feature tokens that can be used to find match start and end indices within an input string. Notice that because the vast majority of regex features are shared across most all-purpose languages, a Python *pattern* will (almost always) behave the same when used in Java, C#, Javascript, Ruby, etc, whereas a *utilization* is not universal in the same way (would not compile in other languages).

B. Building the Corpus

Github is a popular project hosting site containing over 100,000 Python projects. We used the GitHub api to page through all repositories, cloning projects that contain Python code.

For each project, we used Astroid¹⁸ to build the AST of each Python file and find *utilizations* of Python's `re` module.

Using git, each project was scanned at 20 evenly-spaced commits (or all commits if there were less than 20) in its history. Within one project, we define a duplicate *utilization* as a *utilization* having the same function, pattern and flags within the same file (same relative path). We ignored duplicate *utilizations* to protect against over-counting the same *utilization* as we rewind the project through its history. We observed and recorded 53,894 non-duplicate regex *utilizations* in 3,898 projects.

C. Selecting Patterns

Because the focus of this study is regex features, analysis focuses on the patterns found, so we ignored the 12.7% of *utilizations* using flags that can alter regex behavior. An additional 6.5% of *utilizations* contained patterns that could not be compiled because the pattern was non-static (used some runtime variable), or because of other unknown parsing failures.

The remaining 80.8% (43,525) *utilizations* were collapsed into 14,113 distinct pattern strings. The resulting set of pattern strings were parsed using an antlr-based, open source PCRE

```

for each row i:
    obtain set of Rex-generated strings Ri from
    pattern at index i
    sRi = size of Ri
    for each col j:
        Nij = number of strings in Ri matched by
        pattern at index j
        M[i][j] = Nij/sRi
    G = empty graph
    for each row i:
        for each col j:
            SIMij = (M[i][j]+M[j][i])/2
            if SIMij > 0.75:
                add edge (i,j)=SIMij to G

```

Fig. 2. Constructing Similarity Graph

parser released by Bart Kiers¹⁹. This parser was unable to support 0.5% (76) of the patterns due to unsupported unicode characters. Another 0.2% (27) of the patterns used regex features that we have chosen to exclude in this study²⁰. The 13,912 distinct pattern strings that remain were each assigned a weight value equal to the number of distinct projects the pattern appeared in. We will refer to this set of weighted, distinct pattern strings as the *corpus*.

D. Analyzing Features

TODO: revise this subsection After picking four large regex research projects, the big table with the features was created in order to decide which unsupported features are used most often. Our semantic analysis is dependent on the use of Rex to generate strings so we can identify semantically related clusters. For three common features unsupported by Rex, we rely on syntactic analysis to determine similarity among regular expressions containing those features. For those features supported by Rex, we cluster the regular expressions based on semantic diversity.

1) *Syntactic Diversity:* For the negative perspective, we picked three features: LZY, NCG, WNW that are unsupported by Rex and other projects. For each of these features, we created a subset of the *collection* where all the patterns contain that feature. Then we used syntactic analysis...to create a similarity matrix. We then used markov clustering [X] (MCL) to find clusters in the subset. We used these clusters to assist our manual search for some common use cases for the unsupported feature.

2) *Semantic Diversity:* For the positive perspective, we created another subset of patterns (XYZ patterns) where Rex was able to generate strings that the pattern matched. We then created a similarity graph with weighted, undirected edges as shown in Figure 2.

Again we used MCL to find clusters that aided a manual search for use cases strongly associated with particular features.

¹⁸<https://bitbucket.org/logilab/astroid>

¹⁹<https://github.com/bkiers/pcpre-parser>

²⁰www.details.#thisopic

source	Q1	Avg	Med	Q3	Max
files per project	2	53	6	21	5,963
files with regex per project	1	11	2	6	541
regex utilizations per file	1	2	1	3	207

TABLE I. HOW SATURATED ARE PROJECTS WITH *utilizations*? (RQ1)

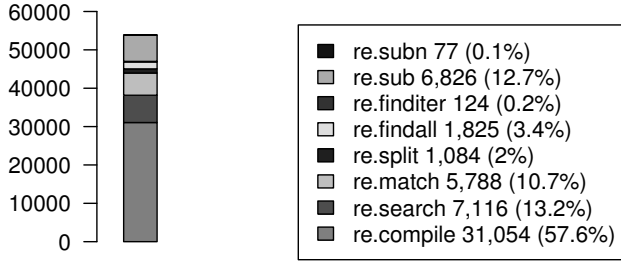


Fig. 3. How often are the 8 re functions used? (RQ1)

IV. RESULTS

A. Frequency of Regex Utilizations in Python Projects

1) *Saturation of Projects with Regex Utilizations*: Out of the 3,898 projects scanned, 42.2% (1,645) contained at least one *utilization*. For context about how saturated these projects were with regex *utilizations*, we consider how many files the average project scanned contained, how many of those files contained *utilizations*, and how many *utilizations* occurred per file in table I.

2) *Usage Frequency of re Module Functions and Flags*: As seen in Figure 3 The ‘compile’ function encompasses 57.6% of all *utilizations*, presumably because each usage of those functions could accept the compiled regex as an argument.

When considering flag use, we excluded non-behavioral flags (default and debug), which account for 87.3% of all *utilizations*.

As shown in figure 4, of all behavioral flags used, ignore-case (43.8%) and multiline (25.8%) were the most frequently used. It is also worth noting that although multiple flags can be combined using a bitwise or, this was never observed.

TODO: topNW table?

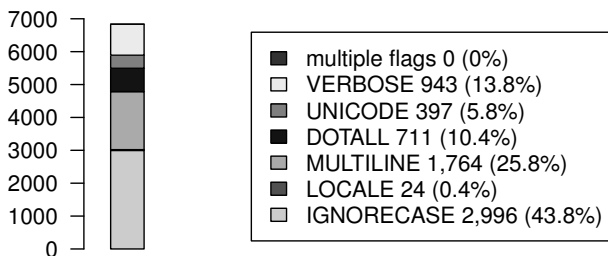


Fig. 4. Which behavioral flags are used? (RQ1)

B. Frequency of Feature Usage

Literal tokens were found in 97.5% of patterns, and accounted for 70.6% of all tokens. We consider literal tokens to be ubiquitous in all *utilizations*, and necessary for any regex related tool, and so exclude them from the rest of the feature analysis. In table II, we display a large body of information about feature usage and relate it to four major regex related projects.

V. DISCUSSION

A. Frequency of re Module Usage

...only 11.2% of the files observed had at least one regex usage. This indicates that regex usage may usually be concentrated in just a few files.

From table I, we see that on average each project had 2 files containing any regex usage, out of an average of 6 files. Each of the files that did have a regex usage had an average of 1 regex usages. Because we scanned 3,898 projects, we would expect to have seen 23,388 regex usages, which is lower than the actual 53,894 usages observed.

1) *Saturation*: Although 42.2% of the projects observed had at least one regex usage, only 11.2% of the files observed had at least one regex usage.

Fun fact: while creating similarity matrix, row 5464 took 2 hours, or almost 1 second per cell avg, only suffering 18 timeouts (1.2 secs). What is this pesky pattern?

We do not assume that Python projects represent a perfect sample of regular expression usage in all environments, but to make the work of collecting data for the paper reasonable, we had to choose one language to focus on (we hope to compare results across languages in future work). Python is an attractive choice because the culture of Python programming makes it seem likely that someone would write the pattern directly in the function, not trying to over-complicate things with some extra Classes or functions. Other attractive choices are Perl (which probably has the most active regex community), javascript and ruby (which may emphasize web tasks like form validation), sql or a general purpose language like java or C#.

Features discussion:

Capture groups are arguably the most powerful feature, since they allow us to get the useful information that we were searching for. They are probably used simply to group some sequence even when there is no intention of getting the content of the group.

For some reason I want to lump the OR in with CG, or try it.

We can logically group all repetition into the DBB type, since QST=tt {0,1}, SNG=tt {N,N}, and KLE=tt {0,MAX}, ADD=tt {1,MAX}, and LWB=tt {1,MAX} for some MAX like Integer.MAX_VALUE.

But I want to look at ADD and KLE on their own - or look at them as super convenience shorthand. SO convenience needs to enter the discussion, not just equivalence.

We can group STR, END and ENDZ together as endpoint anchors.

rank	code	description	example	brics	hampi	Rex	RE2	nPatterns	% patterns	nFiles	%files	nProjects	% projects
1	ADD	one-or-more repetition	z+	●	●	●	●	6,122	44	9,330	50.3	1,209	73.5
2	CG	a capture group	(caught)	●	●	●	●	7,248	52.1	9,759	52.6	1,197	72.8
3	KLE	zero-or-more repetition	. *	●	●	●	●	6,104	43.9	8,323	44.9	1,100	66.9
4	CCC	custom character class	[aeiou]	●	●	●	●	4,581	32.9	7,808	42.1	1,027	62.4
5	ANY	any non-newline char	.	●	●	●	●	4,708	33.8	6,394	34.5	1,006	61.2
6	RNG	chars within a range	[a-z]	●	●	●	●	2,698	19.4	5,196	28	849	51.6
7	STR	start-of-line	^	○	●	●	●	3,660	26.3	5,622	30.3	847	51.5
8	END	end-of-line	\$	○	●	●	●	3,258	23.4	5,549	29.9	828	50.3
9	NCCC	negated CCC	[^qwxf]	●	●	●	●	1,970	14.2	4,027	21.7	777	47.2
10	WSP	\t \n \r \b \f or space	\s	○	●	●	●	2,908	20.9	4,812	25.9	764	46.4
11	OR	logical or	a b	●	●	●	●	2,161	15.5	4,039	21.8	711	43.2
12	DEC	any of: 0123456789	\d	○	●	●	●	2,385	17.1	4,366	23.5	694	42.2
13	WRD	[a-zA-Z0-9_]	\w	○	●	●	●	1,457	10.5	3,004	16.2	652	39.6
14	QST	zero-or-one repetition	z?	●	●	●	●	1,922	13.8	3,821	20.6	647	39.3
15	LZY	as few reps as possible	z+?	○	●	○	●	1,318	9.5	2,291	12.4	606	36.8
16	NCG	group without capturing	a(?:b)c	○	●	○	●	813	5.8	1,748	9.4	404	24.6
17	PNG	named capture group	(?P<name>x)	○	●	○	●	934	6.7	1,517	8.2	354	21.5
18	SNG	exactly n repetition	z{8}	●	●	●	●	623	4.5	1,359	7.3	340	20.7
19	NWSP	any non-whitespace	\S	○	●	●	●	490	3.5	788	4.2	271	16.5
20	DBB	$n \leq x \leq m$ repetition	z{3,8}	●	●	●	●	384	2.8	692	3.7	242	14.7
21	NLKA	sequence doesn't follow	a(?:yz)	○	○	●	○	137	1	503	2.7	184	11.2
22	NWRD	non-word chars	\W	○	●	●	●	97	0.7	315	1.7	169	10.3
23	LWB	at least n repetition	z{15,}	●	●	●	●	97	0.7	337	1.8	167	10.2
24	WNW	word/non-word boundary	\b	○	○	○	●	248	1.8	438	2.4	166	10.1
25	LKA	matching sequence follows	a(?:=bc)	○	○	○	○	114	0.8	360	1.9	159	9.7
26	OPT	options wrapper	(?i) CasE	○	●	○	●	232	1.7	378	2	154	9.4
27	NLKB	sequence doesn't precede	(?<!x)yz	○	○	○	○	102	0.7	321	1.7	139	8.4
28	LKB	matching sequence precedes	(?<=a)bc	○	○	○	○	82	0.6	262	1.4	120	7.3
29	ENDZ	absolute end of string	\Z	○	○	○	●	91	0.7	154	0.8	94	5.7
30	BKR	match the i^{th} CG	\1	○	○	○	○	60	0.4	129	0.7	84	5.1
31	NDEC	any non-decimal	\D	○	●	●	●	36	0.3	92	0.5	58	3.5
32	BKRN	references NCG	\g<name>	○	●	○	○	17	0.1	44	0.2	28	1.7
33	VWSP	matches U+000B	\v	○	○	●	●	13	0.1	16	0.1	15	0.9
34	NWNW	negated WNW	\B	○	○	○	●	4	0	11	0.1	11	0.7

TABLE II. HOW FREQUENTLY DO FEATURES APPEAR IN PROJECTS, AND WHICH FEATURES ARE SUPPORTED BY FOUR MAJOR REGEX PROJECTS? (RQ2)

We can group PNG, BKR and BKRN together as all back-referencing.

Really, CCC, NCCC and RNG are all very fundamental, and supported by all. It is worth looking at what they do.

Meanwhile you could say that WSP, DEC, WRD and their opposites are reducible to CCC, but it would be interesting to see how often they are used.

ANY could also be reduced, but I think it should be treated separately from these 6.

I think that looking ahead and looking behind can be separated into two things, though the NCG and LKA - I want to know if they are not equivalent?

WNW boundary and the opposite is special enough to do on its own.

LZY gets its own show and is special.

OPT - I have no idea, look into it.

VWSP - super rare, maybe ignore it.

VI. CONCLUSION

ACKNOWLEDGMENT

This work is supported in part by NSF SHF-1218265, NSF SHF-EAGER-1446932, and the Harpole-Pentair endowment at Iowa State University.

REFERENCES

- [1] C. L. A. Clarke and G. V. Cormack. On the use of regular expressions for searching text. *ACM Trans. Program. Lang. Syst.*, 19(3):413–426, May 1997.
- [2] E. Spishak, W. Dietl, and M. D. Ernst. A type system for regular expressions. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, FTfJP '12, pages 20–26, New York, NY, USA, 2012. ACM.