

Regular Expression Feature Usage in Python

Carl Chapman
Department of Computer Science
Iowa State University
Ames, IA, USA
carl1978@iastate.edu

Kathryn T. Stolee
Departments of Computer Science and
Electrical and Computer Engineering
Iowa State University
Ames, IA, USA
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, and these decisions impact the usefulness and power of the tools. Yet, these decisions are often made with little information as there does not exist an empirical study of regular expression feature usage to inform these design decisions.

In this paper, we survey 18 professional developers about the context and frequency of their regular expression usage. Then, we explore regular expression feature usage in Python, focusing on how often features are used and the diversity of regular expressions from syntactic and semantic perspectives. We analyzed nearly 4,000 open source Python projects from GitHub and extracted nearly 14,000 unique regex patterns that were used for analysis. We also map the most common features used in regular expressions to those features supported by four common regex engines from industry and academia, brics, Hampi, Re2, and Rex. Our results indicate that developers frequently use regular expressions in their programming practices, but often those regular expressions do not persist (e.g., when used for command line or file search purposes). For regular expressions found in Python projects, the most commonly used 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. We conclude by discussing the implications for tool designers and outline several directions of future work.

1. INTRODUCTION

Regular expressions (regexes) are an abstraction of keyword search that enables the identification of text using a pattern instead of an exact keyword. Regexes are commonly used for parsing text, form validation, and text searching

within text editors (e.g., emacs), command line tools (e.g., grep, sed) and IDEs (e.g., the search feature in the Eclipse IDE). Although regexes are powerful and versatile, they can be hard to understand, maintain, and debug, resulting in tens of thousands of bug reports [18].

Due in part to their common use across programming languages and how susceptible regexes are to error, many researchers and practitioners have developed tools to support more robust creation [18] or to allow visual debugging [6]. To remove the human in the loop, other research has focused on learning regular expressions from text [4, 13]. Beyond supporting regular expression usage, the applications of regular expressions in research include test case generation [2, 9, 10, 20], solvers for string constraints [11, 21], and as queries in a data mining framework [7] or on the semantic web [12]. Regexes are also employed in critical missions like mysql injection prevention [23] and network intrusion detection [16], or in more diverse applications like DNA sequencing alignment [3]. The designers of all these tools made decisions about which regex features to support, yet no research has been done about how regexes are used in practice and what features are essential for the most common use cases.

The goal of this work is to explore 1) the context in which developers use regular expressions, and 2) the features of those regular expressions. First, we survey developers about the context of their regex usage, include how often and for what purposes regexes are composed. Second, we measure how often regex features (e.g., kleene star, character classes, and capture groups are all features) appear in regular expressions and in projects. By comparing the features to those supported by four common regex support tools, brics [15], hampi [11], Rex [22], and RE2 [17] and using a semantic analysis to cluster similar regular expressions, we explore the impact of omitting support for various 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:

- A survey of 18 professional software developers about their experience with regular expressions.
- An empirical analysis of the usage and semantic similarity of nearly 14,000 regular expressions in 3,898 open-source Python projects
- A discussion of opportunities for future work in supporting programmers in writing regular expressions.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...\$15.00.

The rest of the paper is organized as follows. Section 2 motivates this work by discussing research in supporting programmers in the use, creation, and validation of regular expressions. Section 3 presents survey design and results, and section 4 presents the research questions and study setup for exploring regular expressions in the wild. Results of these explorations are in Section 5 followed by a discussion in Section 6 and a conclusion in Section 8.

2. RELATED WORK

Regular expressions have been a focus point in a variety of research objectives. From the user perspective, tools have been developed to support more robust creation [18] or to allow visual debugging [6]. Building on the perspective that regexes are difficult to create, other research has focused on removing the human from the creation process by learning regular expressions from text [4, 13].

Regarding applications, regular expressions have been used for test case generation [2, 9, 10, 20], and solvers for string constraints [11, 21]. Regexes are also employed in critical missions like mysql injection prevention [23] and network intrusion detection [16], or in more diverse applications like DNA sequencing alignment [3] or querying RDF data [1, 12].

As a query language, lightweight regular expressions are pervasive in search. For example, some data mining frameworks use regular expressions as queries (e.g., [7]). Efforts have also been made to expedite the processing of regular expressions on large bodies of text [5].

Within standard programming languages, regular expressions libraries are very common, yet there are differences between languages in the features that they support. For example, Java supports possessive quantifiers like ‘`ab++c`’ (here the ‘`+`’ is modifying the ‘`*`’ to make it possessive) whereas Python does not.

Since regular expression languages vary somewhat in their syntax and feature set, researchers and tool designers have typically had to pick what features to include or exclude. Thus, 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.

In this work, we perform a feature analysis on regular expressions used in the wild and compare that set to the features supported by four popular regular expression tools. Research tools like Hampi [11], and Rex [22], and commercial tools like brics [15] and RE2 [17], all use regular expressions for various task. Hampi was developed in academia and uses regular expressions as a specification language for a strong constraint solver. Rex was developed by Microsoft Research and generates strings for regular expressions that can be used in several applications, such as test case generation [2, 20]. Brics is an open-source package that creates automata from regular expressions for manipulation and evaluation. RE2 is an open-source tool created by Google to power Code Search with a more efficient regex engine. While there are many regular expression tools available, in this work, we focus on the features support for these four tools, which offer diversity across developers (i.e., Microsoft, Google, open source, and academia) and across applications.

Table 1: Survey results for number of regexes composed per year by technical environment

Language/Environment	0	1-5	6-10	11-20	21-50	51+
General (e.g., Java)	1	6	5	3	1	2
Scripting (e.g., Perl)	5	4	3	3	2	1
Query (e.g., SQL)	15	2	0	0	1	0
Command line (e.g., grep)	2	5	3	2	0	6
Text editor (e.g., IntelliJ)	2	5	0	5	1	5

Further, as the focus of this work is on tool designers and we wanted to perform a feature analysis, these four tools and their features are well-documented, allowing for easy comparison.

Mining properties of open source repositories is a well-studied topic, focusing, for example, on API usage patterns [14] and bug characterizations [8]. To our knowledge, this is the first work to mine and evaluate regular expression usages from existing software repositories. Related to mining work, regular expressions have been used to form queries in mining framework [7], but have not been the focus of the mining activities.

3. SURVEY

To understand the context of when and how programmers use regular expressions, we designed a survey with 41 questions about regex usage. This survey was deployed to developers at Dwolla, a company that provides software for online and mobile payment management. Participation was voluntary and participants were entered in a lottery for a \$50 gift card. The survey was completed by 18 participants that identified as software developer/maintainers who had used regular expressions in a work environment.

On average, survey participants report to compose 172 regexes per year ($\sigma = 250$) and compose regexes on average once per month, with 27% composing multiple regexes in a week and an additional 22% composing regexes once per week. Table 1 shows how frequently participants compose regexes using each of several languages and technical environments. Six (33%) of the survey participants report to compose regexes using general purpose programming languages (e.g., Java, C, C#) 1-5 times per year and 5 (28%) do this 6-10 times per year. Regexes were rarely used in query languages like SQL, but for command line usage in tools such as grep, 6 (33%) participants use regexes 51+ times per year. Overall, regexes are used frequently, but in some environments, such as command line or text editor, and sometimes query languages, the composed regular expressions do not persist.

Table 2 shows how frequently, on average, the participants use regexes for various actives. Participants answered questions using a 6-point likert scale including very frequently, frequently, occasionally, rarely, very rarely, and never. Assigning values from 1 to 6, where 6 is the most frequent, the responses were averaged across participants. Among the most common are capturing parts of a string and locating content within a file, with both occurring somewhere between occasionally and frequently.

Using a similar 7-point likert scale that includes ‘always’ as a seventh point, developers indicated that they test their code with the same frequency as they test their regexes (5.2

Table 2: Survey results for regex usage frequencies for various activities, averaged using a 6-point likert scale: Very Frequently=6, Frequently=5, Occasionally=4, Rarely=3, Very Rarely=2, and Never=1

Activity	Frequency
Locating content within a file or files	4.4
Capturing parts of strings	4.3
Parsing user input	4.0
Counting lines that match a pattern	3.2
Counting substrings that match a pattern	3.2
Parsing generated text	3.0
Filtering collections (lists, tables, etc.)	3.0
Checking for a single character	1.7

which is between frequently and very frequently). Half of the 18 developers indicate that they use tools to test their regexes, and the other half indicated that they only use tests that they write themselves. Of the 9 developers using tools, 6 of them mentioned some online composition aide similar to regex101.com where a regex and input are entered, and the input is highlighted according to what is matched.

When asked an open ended question about pain points encountered with regular expressions, 7 developers responded with an answer equivalent to ‘hard to read’, 3 responded with ‘inconsistency across implementations’ and 11 responded with ‘hard to compose’ (3 developers gave two answers).

The pattern language for Python supports default character classes like the ANY or dot character class: `.` which means ‘any character except newline’. It also supports three other default character classes: `\d`, `\w`, `\s` (and their negations). All of these default character classes can be simulated using the custom character class (CCC) feature. For example the decimal character class: `\d` is equivalent to a CCC containing all 10 digits: `\d ≡ [0123456789]`. Users have a choice when using regex between writing `\d` and `[0-9]` and whereas the first option may be shorter, the second may seem more intuitive and readable. Other default character classes such as the word character class: `\w` may not be as intuitive to encode in a CCC: `[a-zA-Z0-9_]`. Survey participants were asked if they use only CCC, use CCC more than default, use both equally, use default more than CCC or use only default. Results for this question are shown in Table 3, with 55% indicating that they use default more than CCC.

Participants were also asked to explain their preferences. Responses varied, but the participants who favored CCC mostly said something equivalent to ‘it is more explicit’, whereas the participants who favored default character classes said something like ‘it is less verbose’ and ‘I like using built-in code’.

Participants were asked five questions (on a 6-point likert scale) about how frequently they used specific related groups of features. These five feature groups are:

- endpoint anchors: `^` and `$`
- capture groups: (capture me)
- word boundaries: `\b`
- (negative) look-ahead/behinds: `a(=bc)`, `(?<x)yz!`, `(?<=a)`, `a(?yz)!`

Table 3: Survey results for preferences between CCC and default character classes with options being: use only CCC, use CCC more than default, use both equally, use default more than CCC or use only default.

Preference	Frequency
use only CCC	1
use CCC more than default	5
use both equally	2
use default more than CCC	10
use only default	2

Table 4: Survey results for regex usage frequencies for five feature groups, averaged using a 6-point likert scale: Very Frequently=6, Frequently=5, Occasionally=4, Rarely=3, Very Rarely=2, and Never=1

Group	Frequency
endpoint anchors	4.4
capture groups	4.2
word boundaries	3.5
lazy repetition	2.9
(negative) look-ahead/behinds	2.5

- lazy repetition: `ab+?`, `xy{2,3}?`

The survey validates the assumption that regex are widely used by professional software developers. Future research into regex can focus on activities that prove to be most important to developers, namely capturing parts of strings and searching for specific content. Although research into regex use in general purpose and scripting languages is important, usage within command line tools and text editors should also be considered. The fact that all the surveyed developers compose regexes, and half of the developers use tools to test their regexes indicates the importance of tool development for regex. Developers complain about regex being hard to read and hard to compose, and most of the tools that they indicate using are focused on composition, indicating a need for tools that help make existing regexes more readable.

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

Utilization: A *utilization* occurs whenever a developer uses a regex in a project. We detect utilizations by recording all calls to the `re` module in Python. Within a particular file in a project, a utilization is composed of a function, a pattern, and 0 or more flags. Figure 1 presents an example of one regex utilization, with key components labeled. The function call is `re.compile`, `(0|-?[1-9][0-9]*)$` is the regex string, or pattern, and `re.MULTILINE` is an (optional) flag. This utilization will compile a regex object in the variable `r1` from the pattern `(0|-?[1-9][0-9]*)$`, with

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

Figure 1: Example of one regex utilization

the `$` token matching at the end of each line because of the `re.MULTILINE` flag. Thought of another way, a regular expression utilization is one single invocation of the `re` library.

Pattern: A *pattern* is extracted from a utilization, as shown in Figure 1. In essence, it is a string, but more formally it is an ordered series of regular expression language feature tokens. The pattern in Figure 1 will match if it finds a zero at the end of a line, or a (possibly negative) integer at the end of a line (i.e., due to the `-?` sequence denoting zero or one instance of the `-`).

Note that because the vast majority of regex features are shared across most general programming languages (e.g., Java, C, C#, or Ruby), a Python pattern will (almost always) behave the same when used in other languages, whereas a utilization is not universal in the same way (i.e., it may not compile in other languages, even with small modifications to function and flag names). As an example, the `re.MULTILINE` flag, or similar, is present in Python, Java, and C#, but the Python `re.DOTALL` flag is not present in C# though it has an equivalent flag in Java.

In this work, we primarily focus on patterns since they are cross-cutting across languages and are the primary way of specifying the matching behavior for every utilization. Next, we describe the research questions and how the data set was collected and analyzed.

4.1 Research Questions

To understand how regular expressions and regular expression features are used in Python projects, we aim to answer the following research questions:

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

We measure how often calls are made to the `re` module per file and per project in Python projects. To provide context as to the overlap among regular expression strings used in Python, we explore the most common regex patterns across all utilizations.

RQ2: Which regular expression 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, for example, the `+` in `ab+`. All studied features are listed and described in Section 4.2 with examples.

RQ3: What is the impact of *not* supporting various regular expression features on tool users and designers?

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

Next, we describe how the corpus of regex patterns was built, how features were analyzed, and how the clustering was performed.

4.2 Regex Corpus

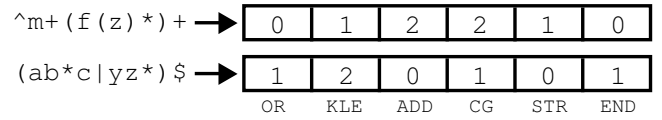


Figure 2: Two patterns parsed into feature vectors

Using the GitHub API, we scraped 3,898 Python projects. Each project’s commit history was scanned at 20 evenly-spaced commits. If the project had fewer than 20 commits, then all commits were scanned. The most recent commit was always included, and the spacing between all other chosen commits was determined by dividing the remaining number of commits by 19 (rounding as needed). All regex utilizations were obtained, sans duplicates. Within a project, a duplicate utilization was marked when two versions of the same file have the same function, pattern and flags. In the end, we observed and recorded 53,894 non-duplicate regex utilizations in 3,898 projects.

In collecting the set of distinct patterns for analysis, we ignore the 12.7% of utilizations using flags, which can alter regex behavior. An additional 6.5% of utilizations contained patterns that could not be compiled because the pattern was non-static (e.g., used some runtime variable), or because of other unknown parsing failures.

The remaining 80.8% (43,525) utilizations were collapsed into 13,749 distinct pattern strings using sql. Each of the pattern strings was pre-processed by removing Python quotes (`'\w'` becomes `\w`), unescaping escaped characters (`\w` becomes `\w`) and parsing the resulting string using an ANTLR-based, open source PCRE parser¹.

This parser was unable to support 0.1% (19) of the patterns due to unsupported unicode characters. Another 0.6% (78) of the patterns used regex features that we have chosen to exclude in this study because they did not appear often enough (e.g., Reference Conditions). The 13,379 distinct pattern strings that remain were each assigned a weight value equal to the number of distinct projects the pattern appeared in. We refer to this set of weighted, distinct pattern strings as the *corpus*.

4.3 Analyzing Features

For each escaped pattern, the PCRE-parser produces a tree of feature tokens, which is converted to a vector by counting the number of each token present in the tree. For a simple example, consider the patterns in Figure 2. The pattern `^m+(f(z)*)+` contains four different types of tokens. It contains the kleene star (KLE), which is specified using the asterisk `*` character, additional repetition (ADD), which is specified using the plus `+` character, capture groups (CG), which are specified using pairs of parenthesis `(...)` characters, and the start anchor (STR), which is specified using the caret `^` character at the beginning of a pattern.

Once all patterns were transformed into vectors, we examined each feature independently for all patterns, tracking the number of patterns it was in, and the size of the sets of projects and files that the patterns containing the features appeared in at least once.

4.4 Clustering and Semantic Analysis

¹<https://github.com/bkiers/pcre-parser>

Pattern A matches 100/100 of A's strings
 Pattern B matches 90/100 of A's strings
 Pattern A matches 50/100 of B's strings
 Pattern B matches 100/100 of B's strings

	A	B
A	1.0	0.9
B	0.5	1.0

Figure 3: A similarity matrix created by counting strings matched


Our semantic analysis clusters regular expressions by their behavioral similarity. Consider two unspecified patterns A and B, a set \mathbf{mA} of 100 strings that pattern A matches, and a set \mathbf{mB} of 100 strings that pattern B matches. If pattern B matches 90 of the 100 strings in the set \mathbf{mA} , then B is 90% similar to A. If pattern A only matches 50 of the strings in \mathbf{mB} , then A is 50% similar to B. We use similarity scores to create a similarity matrix as shown in Figure 3. In row A, column B we see that B is 90% similar to A. In row B, column A, we see that A is 50% similar to B. Each pattern is always 100% similar to itself, by definition.

In the implementation, strings are generated for each pattern using Rex [22]. Rex generates matching strings by representing the regular expression as an automation, and taking a random walk within that automation. By avoiding any repeat walks of the automation, Rex never produces duplicate strings. If asked to produce more strings than the automation can provide, Rex will instead produce a list of all possible strings. Our goal is to generate 384 strings for each pattern to balance the runtime of the similarity analysis with the precision of the similarity calculations. Since Rex does not support all the features present in the corpus, we could only generate sets of matching strings for 9,727 (70%) of the 13,379 patterns in the corpus. The impact is that 270 projects were excluded from the data set for the similarity analysis. Omitted features are indicated in Table 7, as described in Section 5.3. The generated strings for each pattern are used to measure the pairwise similarity for all patterns and construct the similarity matrix.

Once the similarity matrix is built, the values of cells reflected across the diagonal of the matrix were averaged to create a half-matrix of undirected similarity edges, as illustrated in Figure 4. This facilitated clustering by means of the Markov Clustering (MCL) algorithm². We chose the mcl clustering tool because it offers a fast and tunable way to cluster items by similarity and it is particularly useful when the number of clusters is not known *a priori*. We note that Markov clustering can be tuned using many parameters, including inflation and filtering out all but the top-k edges for each node. After exploring the quality of the clusters using various tuning parameter combinations, the best clusters (by inspection) were found using an inflation value of 1.8 and $k=83$. The end result is clusters of highly semantically similar regular expressions. The top 100 clusters are categorized by inspection into six categories of behavior (see Section 5.3).

We note that there was an operational error in pulling patterns from our database prior to the similarity analysis and clustering, so that 224 patterns (2.3%) of the 9,727 patterns were omitted. These were duplicate patterns that were quoted differently (for example ‘\W’ and “\W”). The result

	A	B	C	D
A	1.0	0.0	0.9	0.0
B	0.2	1.0	0.8	0.7
C	0.6	0.8	1.0	0.2
D	0.0	0.6	0.1	1.0



	A	B	C	D
A	1.0			
B	0.1	1.0		
C	0.75	0.8	1.0	
D	0.0	0.65	0.15	1.0

Figure 4: Creating a similarity graph from a similarity matrix

of this error is a slight underestimate in number of projects per pattern (and per cluster), and a slight over-estimate in the pattern, file and project statistics shown in Table 7. We do not believe that this error affects our conclusions.

5. RESULTS

In this section, we present the results of each research question.

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

To address this research question, we look at regex utilizations, flags, and the most frequently observed pattern strings. We measure the frequency of usage for calls to the 8 functions of the `re` module (i.e., `compile`, `search`, `match`, `split`, `findall`, `finditer`, `sub` and `subn`). We also measure usage of the 8 `re` flags (i.e., `DEFAULT`, `IGNORECASE`, `LOCALE`, `MULTILINE`, `DOTALL`, `UNICODE`, `VERBOSE` and `DEBUG`).

5.1.1 Regex Utilizations per Project

Out of the 3,898 projects scanned, 42.2% (1,645) contained at least one regex utilization. For context about how saturated these projects were with utilizations, we consider how many utilizations were observed per project, how many files the average scanned project contained, how many of those files contained utilizations, and how many utilizations occurred per file, as shown in Table 5.

The average project contained 32 utilizations, and the maximum number of utilizations was 1,427. The project with the most utilizations is a C# project³ that maintains a collection of source code for 20 Python libraries, including larger libraries like `pip`, `celery` and `ipython`. These larger Python libraries contain many utilizations. From Table 5, we also see that each project had an average of 11 files containing any utilization, and each of these files had an average of 2 utilizations.

5.1.2 Usage Frequency of `re` Module Functions

The number of projects that use each of the `re` functions is shown in Figure 5. The y-axis denotes the total utilizations, with a maximum of 53,894. The `re.compile` function encompasses 57.6% of all utilizations, presumably because each usage of those functions can accept a regex object compiled using `re.compile` as an argument.

²<http://micans.org/mcl/>

³<https://github.com/Ouroboros/Arianrhod>

Table 5: How Saturated are Projects with Utilizations? (RQ1)

source	Q1	Avg	Med	Q3	Max
utilizations per project	2	32	5	19	1,427
files per project	2	53	6	21	5,963
utilizing files per project	1	11	2	6	541
utilizations per file	1	2	1	3	207

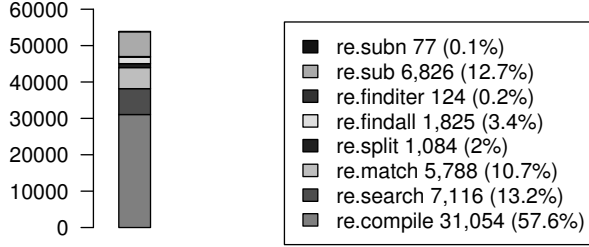


Figure 5: How often are re functions used? (RQ1)

5.1.3 Usage Frequency of re Module Flags

When considering flag use, we excluded the default flag, which is built into the `re` module, and present internally whenever no flag is used. Of all utilizations, 87.3% had no flag, or explicitly specified the default flag. The debug flag, which causes the `re` regex engine to display extra information about its parsing process, was never observed.

Figure 6 presents the number of projects in which each flag appears. Of all behavioral flags used, `ignorecase` (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.

5.1.4 Most Frequently Observed Patterns

Table 6 contains the most frequently used patterns, ordered by the number of projects in which the pattern appears. The patterns are quoted to show the presence of spaces, if any. We elaborate on the top four patterns and reference the feature set in Table 7.

The first pattern, `'\s+'`, uses the WSP character class feature. This character class represents one or more whitespace characters, defined as spaces, tabs, newlines, vertical whitespace, carriage returns or form feeds. The `+` (the ADD

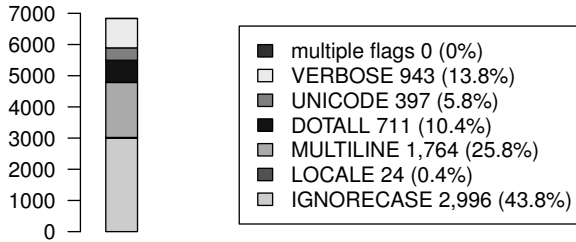


Figure 6: Which behavioral flags are used? (RQ1)

Table 6: Top 10 Patterns by nProjects (RQ1)

pattern	nProjects
'\x1b\ ((?:. ;)*?) (\x07)'	2
'\x1b\ [\d+m'	1
'^[A-Z]'	2
'^#format \w*\n'	2
'(,\n [\n-])+'	1
'^[a-df-z]\d*\$'	1
'[a-zA-Z_:][\w\.-_:]*Z'	1
'^?(.*)'	8
'Video: \w+'	1
'\.\..'	9

feature) at the end of the pattern means that it must match one or more whitespace characters. The pattern `'\s+'` is often used to split sentences into separate words which may have more than one space between them, or contain tabs or other types of whitespace.

Interestingly, the second most common pattern, `'\s'`, also uses the WSP character. In this case, it does not match several whitespace characters (though it would match the first of several).

The third pattern, `\d+`, uses the DEC character class, which is composed of the digits from 0 to 9. Like the first pattern, the third uses the ADD feature to match one or more digits.

The fourth pattern, `[\x80-\xff]`, uses the CCC feature to create a custom character class, and the RNG feature to specify a range of characters. When hexadecimal values are used within a character class like this (and the UNICODE flag is not active), it signifies the corresponding characters in an ASCII lookup table.

5.1.5 Summary

Only about half of the projects sampled contained any utilizations. Most utilizations used the `re.compile` function to compile a regex object before actually using the regex to find a match. Most utilizations did not use a flag to modify matching behavior. The most frequently observed patterns were used to match whitespace and digits.

5.2 RQ2: Which regular expression language features are most commonly used in python?

To measure feature usage, we count the number of usages of each feature per project, per file and as a percent of all distinct regular expression patterns.

5.2.1 Feature Usage

Table 7 displays feature usage from the corpus and relates it to four major regex related projects. Only features appearing in at least 10 projects are listed. The first column, *rank*, lists the rank of a feature (relative to other features) in terms of the number of projects in which it appears. The next column, *code*, gives a succinct reference string for the feature, and is followed by a *description* column that provides a brief comment on what the feature does. The *exam-*

Table 7: How Frequently do Features Appear in Projects, and Which Features are Supported By Four Major Regex Projects? (RQ2)

rank	code	description	example	brics	hampi	Rex	RE2	nPatterns	% patterns	nProjects	% project
1	ADD	one-or-more repetition	z+	●	●	●	●	5,889	44	1,197	72.8
2	CG	a capture group	(caught)	●	●	●	●	6,965	52.1	1,182	71.9
3	KLE	zero-or-more repetition	.*	●	●	●	●	5,882	44	1,084	65.9
4	CCC	custom character class	[aeiou]	●	●	●	●	4,389	32.8	1,014	61.6
5	ANY	any non-newline char	.	●	●	●	●	4,537	33.9	990	60.2
6	STR	start-of-line	^	○	●	●	●	3,507	26.2	836	50.8
7	RNG	chars within a range	[a-z]	●	●	●	●	2,575	19.2	836	50.8
8	END	end-of-line	\$	○	●	●	●	3,112	23.3	816	49.6
9	NCCC	negated CCC	[^qwx^f]	●	●	●	●	1,871	14	765	46.5
10	WSP	\t \n \r \v \f or space	\s	○	●	●	●	2,804	21	752	45.7
11	OR	logical or	a b	●	●	●	●	2,068	15.5	689	41.9
12	DEC	any of: 0123456789	\d	○	●	●	●	2,272	17	687	41.8
13	WRD	[a-zA-Z0-9_]	\w	○	●	●	●	1,393	10.4	644	39.1
14	QST	zero-or-one repetition	z?	●	●	●	●	1,836	13.7	641	39
15	LZY	as few reps as possible	z+?	○	●	○	●	1,262	9.4	590	35.9
16	NCG	group without capturing	a(?:b)c	○	●	○	●	776	5.8	391	23.8
17	PNG	named capture group	(?P<name>x)	○	●	○	●	891	6.7	352	21.4
18	SNG	exactly n repetition	z{8}	●	●	●	●	573	4.3	335	20.4
19	NWSP	any non-whitespace	\S	○	●	●	●	476	3.6	270	16.4
20	DBB	$n \leq x \leq m$ repetition	z{3,8}	●	●	●	●	363	2.7	232	14.1
21	NLKA	sequence doesn't follow	a(?:!yz)	○	○	○	○	130	1	183	11.1
22	LWB	at least n repetition	z{15,}	●	●	●	●	92	0.7	157	9.5
23	LKA	matching sequence follows	a(?:=bc)	○	○	○	○	110	0.8	157	9.5
24	NWRD	non-word chars	\W	○	●	●	●	92	0.7	154	9.4
25	WNW	word/non-word boundary	\b	○	○	○	●	239	1.8	152	9.2
26	OPT	options wrapper	(?i)CasE	○	●	○	●	225	1.7	149	9.1
27	NLKB	sequence doesn't precede	(?<!x)yz	○	○	○	○	89	0.7	132	8
28	LKB	matching sequence precedes	(?<=a)bc	○	○	○	○	77	0.6	118	7.2
29	ENDZ	absolute end of string	\Z	○	○	○	●	89	0.7	90	5.5
30	BKR	match the i^{th} CG	\1	○	○	○	○	59	0.4	83	5
31	NDEC	any non-decimal	\D	○	●	●	●	36	0.3	57	3.5
32	BKRN	references PNG	\g<name>	○	●	○	○	17	0.1	28	1.7
33	VWSP	matches U+000B	\v	○	○	●	●	13	0.1	15	0.9
34	NWNW	negated WNW	\B	○	○	○	●	4	0	10	0.6

ple column provides a short example of how the feature can be used. The next four columns map to the four major research projects chosen for our investigation (see Section ??). We indicate that a project supports a feature with the ‘●’ symbol, and indicate that a project does not support the feature with the ‘○’ symbol.

The next four columns contain two pairs of usage statistics. The first pair contains the number and percent of *patterns* that a feature appears in, out of the 13,912 patterns that make up the corpus. The second pair of columns contain the number and percent of *projects* that a feature appears in out of the 1645 projects scanned that contain at least one utilization.

One notable omission from Table 7 is the literal feature, which is used to specify matching any specific character. An example pattern that contains only one literal token is the pattern ‘a’. This pattern only matches the lowercase letter ‘a’. The literal feature was found in 97.6% of patterns. We consider the literal feature to be ubiquitous in all patterns, and necessary for any regex related tool to support, and so exclude it from Table 7 and the rest of the feature analysis.

The eight most commonly used features, ADD, CG, KLE, CCC, ANY, RNG, STR and END, appear in over half the projects. The remaining 26 features appear in less than half of the projects containing utilizations. CG is more commonly used in patterns than the highest ranked feature (ADD) by a wide margin (over 8%). CG is also more commonly used with respect to how many files it appears in by 2.3%, while only being present in 12 fewer projects (0.7%) than ADD.

5.2.2 Feature Support in Regex Tools

We mapped the features from the corpus to those features supported by the four regular expression engines described in Section 2: brics, hampi, RE2, and Rex. To create the tool mappings, we consulted documentation for each of the selected regular expression engines. For brics, we collected the set of supported features using the formal grammar⁴. For hampi, we manually inspected the set of regexes included in the `lib/regex-hampi/sampleRegex` file within the hampi repository⁵ (this may have been an overestimation, as this included more features than specified by the formal grammar⁶). For RE2, we used the supported feature documentation⁷. For Rex, we were able to use trial and error because we tried to parse all patterns with Rex, and Rex provides good error feedback when a feature is unsupported.

Of the four projects selected for this analysis, RE2 supports the most studied features (28 features) followed by hampi (25 features), Rex (21 features), and brics (12 features). All projects support the 8 most commonly used features except brics, which does not support STR or END. All projects support NCC, OR, and the four less common repetition features: QST, SNG, DBB and LWB. RE2 is the only project to support the WNW, ENDZ and NWNW features.

No projects support the four look-around features LKA, NLKA, LKB and NLKB. RE2 and hampi support the LZY, NCG, PNG and OPT features, whereas brics and Rex do

⁴<http://www.brics.dk/automaton/doc/index.html?dk/brics/automaton/RegExp.html>

⁵<https://code.google.com/p/hampi/downloads/list>

⁶<http://people.csail.mit.edu/akiezun/hampi/Grammar.html>

⁷<https://re2.googlecode.com/hg/doc/syntax.html>

Table 8: An example cluster (RQ3)

index	pattern	nProjects	index	pattern	nProjects
1	‘\s*,\s*’	54	7	‘,.*\$’	3
2	‘,’	30	8	‘(\S+)\s*,\s*’	2
3	‘\s*,’	16	9	‘,+’	1
4	‘,\s*’	13	10	‘,\?’	1
5	‘*,*’	12	11	‘\s*(,)\s*’	1
6	‘,\s’	5	12	‘\s*\,\s*’	1

not. Brics is the only project that does not support any of the six default character classes (WSP, DEC, WRD, NWSP, NWRD, NDEC) - the rest of the projects support all of those features. RE2 also supports the PNG feature (which allows you to name capturing groups), but does not support the BKRN feature which is necessary to refer back to the named capture group.

5.2.3 Summary

We found that the eight most common features are found on over 50% of the projects. We also identify brics as the project supporting the fewest features and RE2 as the project supporting the most features, and identify groups of features supported or not supported by the four regex projects.

5.3 RQ3: What is the impact of *not* supporting various regular expression features?

When tool designers are considering what features to include, data about usage in practice is valuable. Semantic similarity clustering helps to discern these behaviors by looking beyond the structural details of specific patterns and seeing trends in actual matching behavior. We are also able to find out what features are being used in these behavioral trends so that we can make assertions about why certain features are important.

From 9,503 distinct patterns, the MCL clustering technique identified 514 clusters with 2 or more patterns, and 7,214 clusters of size 1. Recall that only pairs of patterns with a similarity level of 0.75 were included in the matrix passed to MCL. The average size of clusters larger than size one was 3.7. Each pattern belongs to exactly one cluster.

Table 8 provides an example of a behavioral cluster representing 13 patterns with at least one pattern from this cluster present in 100 different projects. At first glance this cluster may seem to revolve around the ‘\s*’ parts of these patterns, but actually this cluster was formed because each of these patterns has a comma literal, and other details did not interfere with matching the Rex-generated strings with commas in them.

The smallest pattern in Table 8 is the single comma literal ‘,’ at index 1. This smallest pattern gives the a good idea of what all the patterns within it have in common. A shorter pattern will tend to have less extraneous behavior because it is specifying less behavior. And yet in order for the smallest pattern to be clustered with other patterns, it had to match most of the strings created by Rex from another pattern within the cluster, and so we assume that *the smallest pattern is a good representation of the cluster*.

For the rest of this paper, a cluster will be represented by one of the shortest patterns it contains, followed by the number of projects a member of the cluster appears in, so the cluster in Table 8 will be represented as ‘,’(100).

We manually mapped the top 100 clusters into each of 6 behavioral categories (determined by inspection), omitting 40 clusters that did not fit into any of these categories. ...including 29 clusters composed of long strings (example: `set_fabric_sense_len\()\()` which were in the top 100 because 28 projects that were scanned were forked linux kernels.

Next, we define the six categories and provide examples from the relevant clusters.

5.3.1 Single Literal Characters

This category contains 19 clusters. Each of the clusters center around a single literal character. For example, three of the top clusters in this category include: ‘\’ (110), ‘,’ (100), and ‘:’ (91). This is in contrast to the survey in Section 3 in which participants reported to very rarely or never use regexes to check for a single character (Table 2).

5.3.2 Default Character Classes

This category contains 12 clusters. Each of the clusters revolves around the use of a default character class. For example, three of the top clusters in this category are: ‘\s’ (277), ‘\w’ (208), and ‘\d’ (193). This corroborates our survey results to the question, *Do you prefer to use custom character classes or default character classes more often?*, in which 56% (10) of the participants indicated they use the default classes more than custom.

5.3.3 User-Defined Character Classes

This category contains 10 clusters. Each of the clusters center around user defined character classes. For example, three of the top clusters in this category are: ‘[a-zA-Z]’ (138), ‘[-]’ (122), and ‘[<]’ (50). This further supports our survey results in which 33% (6) participants indicated they use the custom classes more than default. The remaining two participants use both equally.

5.3.4 Matching Whole Strings

This category contains 8 clusters. Each of the clusters begins with the STR anchor and ends with the END anchor, requiring the entire input string to match the pattern. For example, three of the top clusters in this category include: ‘^d+\$’ (78), ‘^w+\$’ (74), and ‘^s*\$’ (59).

5.3.5 Parsing Angle Bracket Contents

This category contains 5 clusters. Each of the clusters contains a pair of angle brackets that contain a repeating character class. It appears that these clusters are being used to recognize or capture the contents of the angle brackets. For example, three of the top clusters in this category include: ‘<.+>’ (63), ‘<!\s+([<>]*)>’ (35), and ‘<([<>]*)/>’ (35).

5.3.6 Capturing Variable Assignments

This category contains 4 clusters. Each of the clusters contain an equals symbol and some pattern on either side of it, which appears to be a variable on the left of the equals sign and a value on the right. This type of cluster is very likely used to capture the value of the variable assignment when

parsing source code. For example, three of the top clusters in this category are: ‘\nmd5_data = {\n([^\n]+)}’ (69), ‘.*rlen=([0-9]+)’ and ‘coding[:]=\s*([-\.w.]+)’ (48).

5.4 Summary

We used the behavior of individual patterns to form clusters, and identified six main categories that clusters belonged to. Overall, we see that many clusters are defined by the presence of particular tokens, such as the comma for the cluster in Table 8. These six categories define what users are doing with regexes at a high level: using default character classes, defining their own character classes, matching single characters, parsing variable assignments, parsing the contents of brackets, or matching whole lines. The higher frequency of default character class matching than custom character class matching corroborates our developer survey responses. One of the six common cluster categories, *capturing variable assignments*, has a very specific purpose of parsing source code files. This shows a very specific and common use of regular expressions in practice.

6. DISCUSSION

In this section, we discuss the implications of these empirical findings on tool designers and users of regex tools and opportunities for future work.

6.1 Implications For Tool Designers

We observe that, although the clusters were generated based on behavioral similarity, they are often centered around the presence of certain features. Thus, omitting those features in a tool’s implementation often omits an entire space of regular expression behaviors.

6.1.1 STR, END

The endpoint anchor features STR and END are useful for specifying how a pattern relates to the beginning or end of a line, or an entire line from beginning to end. In our survey, over half (56%) of the respondents answers that they use endpoint anchors frequently or very frequently, and none of them claimed to never use them. Some common use cases require the endpoint anchors, such as specifying that some content is first or last in an input. In multi-line mode, endpoint anchors match the beginning and ending of each line. For example the regex `(?m)^\s*$` will match every line that is only whitespace.

From Table 7 we know that STR and END features are present in over half of the scanned projects containing utilizations - further evidence of the importance of these features. The brics library does not support this feature, which is a missed opportunity for many developers who could otherwise have used brics to model their regexes that use STR and END.

6.1.2 LZY

The LZY feature modifies the behavior of the repetition features (i.e., ADD, KLE, QST, DBB, LWB, SNG) by forcing them to use as few repetitions as possible for a match. The default behavior for the repetition features is to use as many repetitions as possible for a match. Consider trying to find the shortest sequence of binary characters starting and ending with a 1 within the binary sequence: 1010001. The pattern ‘1.*1’ will match the entire sequence whereas the pattern ‘1.*?1’ will match 101 because the KLE repetition

feature was modified by the LZY feature. There is no way to obtain shortest matches without the LZY feature. LZY is present in over 36% of scanned projects with utilizations (see Table 7), and yet was not supported by two of the four major regex projects we looked at. In our developer survey, 11% (2) of participants use this feature frequently and 6 (33%) use it occasionally, showing a modest impact on potential users.

6.1.3 PNG and BKR vs CG and BKR

The PNG group (python-style named capture groups) and BKR (back-references: named) are intended to operate together to allow users to name the content that is expected to appear in a capture group. A simple example of how these can be used together with the named capture group matching some vowel is: `(?P<vowel>[aeiou]x(?P=vowel))` which will match the strings 'axa' and 'oxo' but not 'txt'. The same functionality can be obtained with a much shorter expression: `([aeiou])x\1` where the '`\1`' references the CG started by the first left parenthesis found when moving from left to right. When survey participants were asked if they prefer to always use numbered capture groups, always use named capture groups or 'it depends', 66% (12) of survey participants said that they always use BKR, and the remaining 33% (6) said 'it depends'. No one said that they always use named capture groups, and the participants who answered 'it depends' said that they would only use named capture groups when composing a very large regex. BKR is present in 5% of scanned projects, while BKR is present in only 1.7%, which corroborates our findings that numbered capture groups are generally preferred over named capture groups.

6.2 Key User Behaviors to Support

Here we offer a discussion of the main behaviors that are important for tool designers to support.

6.2.1 Capturing Specific Content

The survey results from section 3 indicated that capturing parts of strings was the second most frequent activity that developers used regex for. As mentioned in section 5.2.1, the CG feature is the most frequently used feature in terms of patterns (see Table 7). As mentioned in Section 5.4, capturing values assigned to variables when parsing source code was one of the main categories of clusters observed. The ability to capture some part of a match provides a powerful tool to programmers. The CG feature has two functions: 1. it allows logical grouping as would be expected by parenthesis, and 2. it allows retrieval of information that was in one logical grouping. Although the four regex projects all support the very necessary logical grouping aspect of the feature, none support the BKR feature that retrieves what content was found in an earlier CG. Any non-trivial tool or research that hopes to be applicable to regex use in practice must treat the CG feature as especially important, and must support some way to reason about what information is retrieved by capture groups.

6.2.2 Counting Delimiters and Finding Flags

Text files containing one unit of information per line are common in a wide variety of applications (for example log and csv files). Out of the 13,912 patterns in the corpus, 3444 (24%) contained ANY followed by KLE: '`.*`', often at the

end of the pattern. One reasonable explanation for this tendency to put '`.*`' at the end of a pattern is that users want to disregard all matches after the first match on a single line in order to count how many distinct lines the match occurs on. Survey participants indicated an average frequency of 'Counting lines that match a pattern' and 'Counting substrings that match a pattern' at 3.2 or Rarely/Occasionally.

Delimiters that separate items on one line like '`,`' are also quite common. Although survey participants indicated an average frequency of 1.7 (very rarely or never) for 'checking for a single character', we found 19 clusters whose essential behavior was to search for one or two characters. This makes sense if you consider that the top ranked activity for developers was 'Locating content within a file or files', and usually this content is located using some small set of characters that the user knows will flag that content. Looking closely at that category of cluster, some of the characters being searched for were `-`, `#` and `:` - all common delimiters in different scenarios.

6.3 Opportunities For Future Work

Based on our findings, there are many opportunities for future work.

6.3.1 A Modern WRD Character Class

One surprising result of our clustering is that, behaviorally speaking, the negation of the word class NWRD was used in 208 projects, while the word class itself was used in only 114 projects. After inspecting several projects using the patterns found in this behavioral cluster, we concluded that most users are trying to sanitize arbitrary strings that must conform to a system character set requirement, such as requirements for filenames. For example, a user might replace all NWRD matching characters with the '`_`' to guarantee that an arbitrary string can be used as a filename. We also considered the largest cluster using custom character classes ('`[^~*]`' (122)) and concluded that users are constructing a more permissive version of the NWRD character class, to allow more non-letter, non-digit characters than just the '`_`' in their sanitized strings. More research is needed to determine if a more modern WRD class could be useful, and if so, what characters set is preferred.

6.3.2 Refactoring Regex for Readability and Performance

The survey showed that users want readability and find the lack of readable regexes to be a major pain point. Certain character classes that are logically equivalent can be expressed differently. One avenue for future exploration is a tool to transform a regex with an character class that is difficult to read into one that has the same effect but is easier to understand. Other similar refactoring techniques may become evident with a more thorough search. A tool that preserved the exact behavior of a regex but optimized for readability could be incorporated into an IDE and relieve some developer pain. More research is needed into why certain character classes are considered more readable, as has been done for other refactoring work (e.g., [19]). Similarly, there are several principals that can be followed to enhance the performance of a regex. Using non-capture groups whenever possible, avoiding backtracking, etc. In theory it seems possible to build a compiler that could compose a regex with

identical behavior but with better readability and performance.

6.3.3 Developer Awareness of Best Practices

One category of 5 clusters in the top 100 contained regex patterns to parse the contents of angle brackets. Because the contents of angle brackets are usually unconstrained, regex are a poor replacement for XML or HTML parsers. Using regex instead of a parser can lead to many programming disasters, as thoroughly discussed on the StackOverflow page dedicated to this topic⁸. More research is needed into how regex users discover best practices and how aware they are of how regexes should and should not be used.

6.3.4 Automated Regex Repair

Given a suite of tests and a high suspiciousness score for lines of code containing regex, an automated system could search for similar regexes to try in place of the faulty one. The same clustering technique that we used to study regex behavior could be used to index groups of similar regexes, speeding up the process of searching for a replacement.

7. THREATS TO VALIDITY

The threats to validity of this work stem primarily from sampling bias, tool limitations, and language selection.

We mined only 3,898 Python projects from GitHub, which is small in comparison with the over 100,000 available Python projects. The projects were mined using the GitHub API which sorts the projects by creation date. By using the API, the goal was to reduce any sampling bias introduced by the researchers.

The survey results are from a set of developers at one company and may not generalize to other developers at that company or at other companies. While the results provide some information on regex usage, repeating the survey to a larger set of diverse developers will increase the generalizability of the results.

We did not scrape all commits in every project for regular expression utilizations, rather, we grabbed each project every 20 commits. It is possible that in between the scanned commits, a regular expression utilization was added and then removed, leading to fewer utilizations in our final data set.

In extracting patterns for analysis, we omit utilizations that contain flags since flags provide refinements on the functionality of the pattern as it is used in the project. Given that only 12.7% of all utilizations used flags, the impact on the clusters should be minimal.

Regular expression patterns were clustered using strings generated by the Rex tool. This introduces two threats to validity. First, we assume that the strings generated by Rex are reasonably diverse to help characterize the regular expression behavior. To mitigate this threat, Rex generated at least 384 strings per regular expression. Second, since the similarity between two regular expressions was calculated empirically using Rex rather than through analysis, we may have over-estimated the similarity. For this reason, in our clustering algorithm, we require a similarity level of 0.75 so that we do not over-estimate the similarity between regular expressions.

The final threat to validity comes from the fact that we only explore regular expressions in Python projects so these results may not generalize to other languages. Future work will replicate this study in other languages and compare the results.

8. CONCLUSION

Regular expressions are used frequently in programming projects. In our analysis of nearly 4,000 Python projects scraped from GitHub, we observed that over 42% contained a regular expression. The most commonly found regular expressions deal with matching whitespace or digits. In analyzing the features used in regular expressions, we find that the most common is the `+` token. When clustering regexes based on semantic similarity, we observe that programmers frequently create identical regular expressions using slightly different patterns. After mapping the features supported by four popular regex tools from academia and industry, we observe that most of the most popular features are also supported. However, those unsupported can have an impact on the tool users.

Acknowledgment

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

9. REFERENCES

- [1] F. Alkhateeb, J.-F. Baget, and J. Euzenat. Extending sparql with regular expression patterns (for querying rdf). *Web Semant.*, 7(2):57–73, Apr. 2009.
- [2] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn. An orchestrated survey of methodologies for automated software test case generation. *J. Syst. Softw.*, 86(8):1978–2001, Aug. 2013.
- [3] A. Arslan. Multiple sequence alignment containing a sequence of regular expressions. In *Computational Intelligence in Bioinformatics and Computational Biology, 2005. CIBCB '05. Proceedings of the 2005 IEEE Symposium on*, pages 1–7, Nov 2005.
- [4] R. Babbar and N. Singh. Clustering based approach to learning regular expressions over large alphabet for noisy unstructured text. In *Proceedings of the Fourth Workshop on Analytics for Noisy Unstructured Text Data, AND '10*, pages 43–50, New York, NY, USA, 2010. ACM.
- [5] R. A. Baeza-Yates and G. H. Gonnet. Fast text searching for regular expressions or automaton searching on tries. *J. ACM*, 43(6):915–936, Nov. 1996.
- [6] F. Beck, S. Gulan, B. Biegel, S. Baltes, and D. Weiskopf. Regviz: Visual debugging of regular expressions. In *Companion Proceedings of the 36th International Conference on Software Engineering, ICSE Companion 2014*, pages 504–507, New York, NY, USA, 2014. ACM.
- [7] A. Begel, Y. P. Khoo, and T. Zimmermann. Codebook: Discovering and exploiting relationships in software repositories. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10*, pages 125–134, New York, NY, USA, 2010. ACM.

⁸<http://stackoverflow.com/questions/1732348>

- [8] T.-H. Chen, M. Nagappan, E. Shihab, and A. E. Hassan. An empirical study of dormant bugs. In *Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014*, pages 82–91, New York, NY, USA, 2014. ACM.
- [9] S. J. Galler and B. K. Aichernig. Survey on test data generation tools. *Int. J. Softw. Tools Technol. Transf.*, 16(6):727–751, Nov. 2014.
- [10] I. Ghosh, N. Shafiei, G. Li, and W.-F. Chiang. Jst: An automatic test generation tool for industrial java applications with strings. In *Proceedings of the 2013 International Conference on Software Engineering, ICSE '13*, pages 992–1001, Piscataway, NJ, USA, 2013. IEEE Press.
- [11] A. Kiezun, V. Ganesh, S. Artzi, P. J. Guo, P. Hooimeijer, and M. D. Ernst. Hampi: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Trans. Softw. Eng. Methodol.*, 21(4):25:1–25:28, Feb. 2013.
- [12] J. Lee, M.-D. Pham, J. Lee, W.-S. Han, H. Cho, H. Yu, and J.-H. Lee. Processing sparql queries with regular expressions in rdf databases. In *Proceedings of the ACM Fourth International Workshop on Data and Text Mining in Biomedical Informatics, DTMBIO '10*, pages 23–30, New York, NY, USA, 2010. ACM.
- [13] Y. Li, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. V. Jagadish. Regular expression learning for information extraction. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing, EMNLP '08*, pages 21–30, Stroudsburg, PA, USA, 2008. Association for Computational Linguistics.
- [14] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk. Mining energy-greedy api usage patterns in android apps: An empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014*, pages 2–11, New York, NY, USA, 2014. ACM.
- [15] A. Møller. dk.brics.automaton – finite-state automata and regular expressions for Java, 2010. <http://www.brics.dk/automaton/>.
- [16] The Bro Network Security Monitor. <https://www.bro.org/>, May 2015.
- [17] RE2. <https://github.com/google/re2>, May 2015.
- [18] 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.
- [19] K. T. Stolee and S. Elbaum. Identification, impact, and refactoring of smells in pipe-like web mashups. *IEEE Trans. Soft. Eng.*, 39(12):1654–1679, 2013.
- [20] N. Tillmann, J. de Halleux, and T. Xie. Transferring an automated test generation tool to practice: From pex to fakes and code digger. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14*, pages 385–396, New York, NY, USA, 2014. ACM.
- [21] M.-T. Trinh, D.-H. Chu, and J. Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS '14*, pages 1232–1243, New York, NY, USA, 2014. ACM.
- [22] M. Veanes, P. d. Halleux, and N. Tillmann. Rex: Symbolic regular expression explorer. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation, ICST '10*, pages 498–507, Washington, DC, USA, 2010. IEEE Computer Society.
- [23] A. S. Yeole and B. B. Meshram. Analysis of different technique for detection of sql injection. In *Proceedings of the International Conference & Workshop on Emerging Trends in Technology, ICWET '11*, pages 963–966, New York, NY, USA, 2011. ACM.