An empirical study of regular expression use in practice, sampling from Python projects on Github, leading to new concepts for refactoring regular expressions for readability.

by

Carl Allen Chapman

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Kathryn Stolee, Major Professor
Samik Basu
Tien Nguyen

Iowa State University
Ames, Iowa
2016

DEDICATION

I would like to dedicate this thesis to my mother, who believed in me and supported me through many years on a long winding road leading to a satisfying career. I'd also like to thank my wife Chien Wen Hung who endured the tough winters and the peripheral stress of living with a student in Ames, IA.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. First and foremost, Dr. Kathryn Stolee for her guidance, patience and support throughout this research and the writing of this thesis. I would also like to thank my committee members for their efforts and contributions to this work: Dr. Samik Basu and Dr. Tien Nguyen.

ABSTRACT

Though regular expressions (regex) are baked into every major language, have inspired several tools and research projects, and have been around since the first days of Unix (1960?), no one has ever formally studied how they are used in practice, or what can be done to make them easier to understand. This thesis presents the original work of studying a sample of regex taken from Python projects pulled from Github, determining what features are used most often, defining some categories that illuminate common use cases, and identifying areas of significance for tool builders. Furthermore, this thesis defines an equivalence class model used to explore comprehension of regex, identifying the most common and most understandable representations of semantically identical regex, suggesting several refactorings and preferred representations. Opportunities for future work include the novel and rich field of regex refactoring, semantic search of regexes, and further fundamental research into regex usage and understandability.

CHAPTER 1. OVERVIEW

This is the opening paragraph to my thesis which explains in general terms the concepts and hypothesis which will be used in my thesis.

With more general information given here than really necessary.

1.1 Introduction

Here initial concepts and conditions are explained and several hypothesis are mentioned in brief.

1.1.1 Hypothesis

Here one particular hypothesis is explained in depth and is examined in the light of current literature.

1.1.1.1 Parts of the hypothesis

Here one particular part of the hypothesis that is currently being explained is examined and particular elements of that part are given careful scrutiny.

1.1.2 Second Hypothesis

Here one particular hypothesis is explained in depth and is examined in the light of current literature.

1.1.2.1 Parts of the second hypothesis

Here one particular part of the hypothesis that is currently being explained is examined and particular elements of that part are given careful scrutiny.

1.2 Criteria Review

Here certain criteria are explained thus eventually leading to a foregone conclusion.

CHAPTER 2. REVIEW OF LITERATURE

This is the opening paragraph to my thesis which explains in general terms the concepts and hypothesis which will be used in my thesis.

With more general information given here than really necessary.

2.1 Introduction

Here initial concepts and conditions are explained and several hypothesis are mentioned in brief.

[**?**], [**?**] and [**?**] did the initial work in this area. But in Struss' work [[**?**]] the definitive model is seen.

2.1.1 Hypothesis

Here one particular hypothesis is explained in depth and is examined in the light of current literature.

2.1.1.1 Parts of the hypothesis

Here one particular part of the hypothesis that is currently being explained is examined and particular elements of that part are given careful scrutiny.

2.1.2 Second Hypothesis

Here one particular hypothesis is explained in depth and is examined in the light of current literature.

2.1.2.1 Parts of the second hypothesis

Here one particular part of the hypothesis that is currently being explained is examined and particular elements of that part are given careful scrutiny.

2.2 Criteria Review

Here certain criteria are explained thus eventually leading to a foregone conclusion.

CHAPTER 3. Feature Use

Exploring Regular Expression Usage and Context in Python

by

Carl Allen Chapman

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Kathryn Stolee, Major Professor
Samik Basu
Tien Nguyen

Iowa State University
Ames, Iowa
2016

Abstract

Due to the popularity and pervasive use of regular expressions, researchers have created tools to support their creation, validation, and use. However, little is known about the context in which regular expressions are used, the features that are most common, and how behaviorally similar regular expressions are to one another.

In this paper, we explore the context in which regular expressions are used through a combination of developer surveys and repository analysis. We survey 18 professional developers from a small software company about their regular expression usage and pain points. Our results indicate that developers frequently use regular expressions in their programming practices, often composing regular expressions at least weekly. Then, we analyzed nearly 4,000 open source Python projects from GitHub and extracted nearly 14,000 unique regular expression patterns, focusing the analysis on how often features are used. We map the most common features used in regular expressions to those features supported by four common regular expression engines from industry and academia: brics, Hampi, RE2, and Rex. Using similarity analysis of regular expressions across projects, we identify six common behavioral clusters that describe how regular expressions are often used in practice. This is the first rigorous examination of regex usage and it provides empirical evidence to support design decisions by regex tool builders. It also points to areas of needed future work, such as refactoring regular expressions to increase refactoring understandability, support for migrating regexes between language, and context-specific tool support for common regexes usages.

3.1 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 using a general purpose language like Python, validating content entered into web forms using Javascript, and searching text files for a particular pattern using tools like grep, vim or Eclipse.

Although regexes are powerful and versatile, they can be hard to understand, maintain, and debug, resulting in tens of thousands of bug reports [**?**].

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 regex creation [**?**] or to allow visual debugging [**?**]. Other research has focused on learning regular expressions from text [**?** , **?**], avoiding human composition altogether. Researchers have also explored applying regexes to test case generation [**?** , **?** , **?** , **?**], as specifications for string constraint solvers [**?** , **?**] and using regexes as queries in a data mining framework [**?**]. Regexes are also employed in critical missions like MySQL injection prevention [**?**] and network intrusion detection [**?**], or in more diverse applications like DNA sequencing alignment [**?**].

Regex researchers and tool designers must pick what features to include or exclude, which can be a difficult design decision. Supporting advanced features may be more expensive, taking more time and potentially making the project too complex and cumbersome to execute well. A selection of only the simplest of regex features limits the applicability or relevance of that work. Despite extensive research effort in the area of regex support, 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 and similarities of regular expressions found in Python[1] *projects.*

First, we survey professional developers about how they use regexes and their pain points. Second, we gather a sample of regexes from Python projects and analyze the frequency of

[1] *Python is the fourth most common language on GitHub (after Java, Javascript and Ruby) and Python's regex pattern language is close enough to other regex libraries that our conclusions are likely to generalize.*

feature usage (e.g., kleene star: `*` and the end anchor: `$` are features). Third, we investigate what features are supported by four large projects that aim to support regex usage (brics [**?**], hampi [**?**], Rex [**?**], and RE2 [**?**]), and which features are not supported, but are frequently used by developers. Finally, we cluster regular expressions that appear in multiple projects by behavior, investigating high-level behavioral themes in regex usage.

Our results indicate that regexes are most frequently used in command line tools and IDEs. Capturing the contents of brackets and searching for delimiter characters were some of the most apparent behavioral themes observed in our regex clusters, and developers frequently use regexes to parse source code. The contributions of this work are:

- A survey of 18 professional software developers about their experience with regular expressions,
- An empirical analysis of regex feature usage in nearly 14,000 regular expressions in 3,898 open-source Python projects, mapping of those features to those supported by common regex tools and survey results showing the impact of not supporting various features,
- An approach for measuring behavioral similarity of regular expressions and qualitative analysis of the most common behaviorally similar clusters, and
- An evidence-based discussion of opportunities for future work in supporting programmers who use regular expressions, including refactoring regexes, developing regex similarity analyses, and providing migration support between languages.

3.2 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 [**?**] or to allow visual debugging [**?**]. 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 [**?** , **?**].

Regarding applications, regular expressions have been used for test case generation [**?** , **?** , **?** , **?**], and as specifications for string constraint solvers [**?** , **?**]. Regexes are also employed

in MySQL injection prevention [**?**] and network intrusion detection [**?**], or in more diverse applications like DNA sequencing alignment [**?**] or querying RDF data [**?** , **?**].

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

Research tools like Hampi [**?**], and Rex [**?**], and commercial tools like brics[**?**] and RE2 [**?**], all support the use of regular expressions in various ways. Hampi was developed in academia and uses regular expressions as a specification language for a constraint solver. Rex was developed by Microsoft Research and generates strings for regular expressions that can be used in applications such as test case generation [**?** , **?**]. 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 an efficient regex engine.

Mining properties of open source repositories is a well-studied topic, focusing, for example, on API usage patterns [**?**] and bug characterizations [**?**]. Exploring language feature usage by mining source code has been studied extensively for Smalltalk [**?** , **?**], JavaScript [**?**], and Java [**?** , **?** , **?** , **?**], and more specifically, Java generics [**?**] and Java reflection [**?**]. 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 [**?**], but have not been the focus of the mining activities. Surveys have been used to measure adoption of various programming languages [**?** , **?**], and been combined with repository analysis [**?**], but have not focused on regexes.

3.3 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 regex appears in source code. We detect utilizations by statically analyzing source code and recording calls to the `re` module in Python.

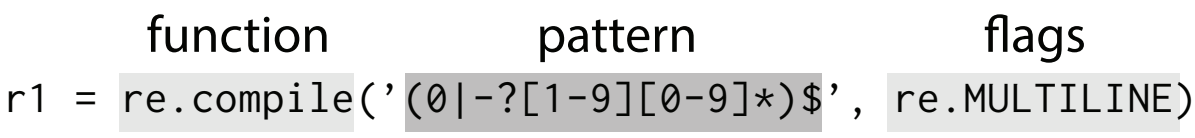

Figure 3.1 Example of one regex utilization

Within a source code file, a utilization is composed of a function, a pattern, and 0 or more flags. Figure 4.2 presents an example of one regex utilization, with key components labeled. The function call is `re.compile`, `(0|-?[1-9][0-9]*)$` is the regex string, or pattern, and `re.MULTILINE` is an (optional) flag. When executed, this utilization will compile a regex object in the variable `r1` from the pattern `(0|-?[1-9][0-9]*)$`, with the `$` token matching at the end of each line because of the `re.MULTILINE` flag. Thought of another way, a regex utilization is one single invocation of the `re` library.

Pattern: A *pattern* is extracted from a utilization, as shown in Figure 4.2. In essence, it is a string, but more formally it is an ordered series of regular expression language feature tokens. The pattern in Figure 4.2 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. Next, we describe the research questions, data set collection and analysis.

3.3.1 Research Questions

To understand the contexts in which regexes are used and feature usage, we perform a survey of developers and explore regular expressions found in Python projects on GitHub. We aim to answer the following research questions:

RQ1: In what contexts do professional developers use regular expressions?

We designed and deployed a survey about when, why, and how often they use regular expressions. This was completed by 18 professional developers at a small software company.

RQ2: How is the `re` module used in Python projects?

We explore invocations of the `re` module in 3,898 Python projects scraped from GitHub.

RQ3: 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 Table 3.4 with examples. We then map the feature coverage for four common regex support tools, brics, hampi, RE2 and Rex, and explore survey responses regarding feature usage for some of the less supported features.

RQ4: How behaviorally similar are regexes across projects?

As this is a first step in understanding behavioral overlap in regexes, we measure similarity between pairs of regexes by overlap in matching strings. For each regex, matching strings are generated and then evaluated against each other regex to compute pairwise similarity. Then we use clustering to form behaviorally similar groupings.

3.3.2 Survey Design and Implementation

To understand the context of when and how programmers use regular expressions, we designed a survey, implemented using Google Forms, with 40 questions. The questions asked

about regex usage frequency, languages, purposes, pain points, and the use of various language features.[2]Participation was voluntary and participants were entered in a lottery for a $50 gift card.

Our goal was to understand the practices of professional developers. Thus, we deployed the survey to 22 professional developers at Dwolla, a small software company that provides tools for online and mobile payment management. While this sample comes from a single company, we note anecdotally that Dwolla is a start-up and most of the developers worked previously for other software companies, and thus bring their past experiences with them. Surveyed developers have nine years of experience, on average, indicating the results may generalize beyond a single, small software company, but further study is needed.

3.3.3 Regex Corpus

Our goal was to collect regexes from a variety of projects to represent the breadth of how developers use the language features. Using the GitHub API, we scraped 3,898 projects containing Python code. We did so by dividing a range of about 8 million repo IDs into 32 sections of equal size and scanning for Python projects from the beginning of those segments until we ran out of memory. At that point, we felt we had enough data to do an analysis without further perfecting our mining techniques. We built the AST of each Python file in each project to find utilizations of the `re` module functions. In most projects, almost all regex utilizations are present in the most recent version of a project, but to be more thorough, we also scanned up to 19 earlier versions. The number 20 was chosen to try and maximize returns on computing resources invested after observing the scanning process in many hours of trial scans. 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).

[2]survey link removed for anonymity

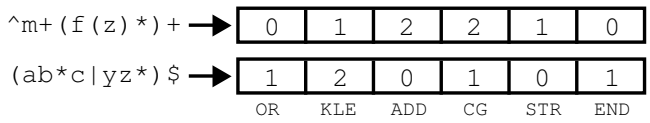

Figure 3.2 Two patterns parsed into feature vectors

The remaining 80.8% (43,525) of the utilizations were collapsed into 13,711 distinct pattern strings. 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 parser[3]. This parser was unable to support 0.5% (73) of the patterns due to unsupported unicode characters. Another 0.2% (25) of the patterns used regex features that we chose to exclude because they appeared very rarely (e.g., reference conditions). An additional 0.1% (16) of the patterns were excluded because they were empty or otherwise malformed so as to cause a parsing error.

The 13,597 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*.

3.3.4 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 in the tree. For a simple example, consider the patterns in Figure 3.2. The pattern ‘^m+(f(z)*)+’ contains four different types of tokens. It has 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. A list of all features and abbreviations is provided in Table 3.4.

Once all patterns were transformed into vectors, we examined each feature independently for all patterns, tracking the number of patterns and projects that the each feature appears in at least once.

[3] 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 3.3 A similarity matrix created by counting strings matched

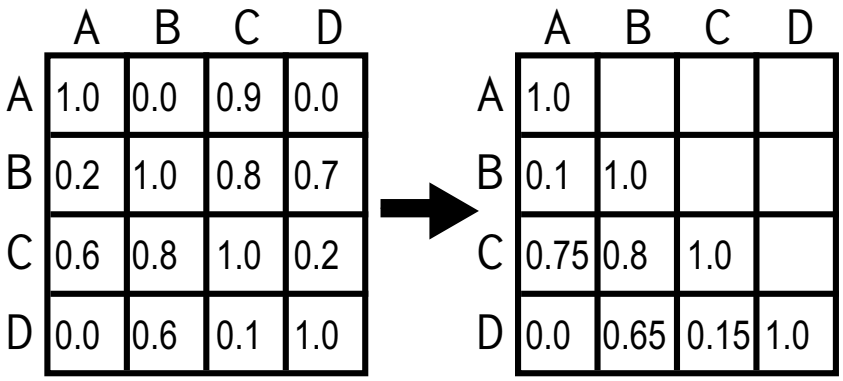

Figure 3.4 Creating a similarity graph from a similarity matrix

3.3.5 Clustering and Behavioral Similarity

An ideal analysis of regex behavioral similarity would use subsumption or containment analysis. However, we struggled to find a tool that could facilitate such an analysis. Further, regular expressions in code libraries (e.g., for Python, Java) are not the same as regular languages in formal language theory. Some features of regular expression libraries, such as backreferences, make the libraries more expressive than regular languages. This allows a regular expression pattern to match, for example, repeat words, such as "cabcab", using the pattern `([a-z]+)\1`. However, building an automaton to recognize such a pattern and to facilitate containment analysis, is infeasible. For these reasons, we developed a similarity analysis based on string matching.

Our similarity analysis clusters regular expressions by their behavioral similarity on matched strings. Consider two unspecified patterns `A` and `B`, a set `mA` of 100 strings that pattern `A` matches, and a set `mB` of 100 strings that pattern `B` matches. If pattern `B` matches 90 of the 100 strings in the set `mA`, then `B` is 90% similar to `A`. If pattern `A` only matches 50 of the strings in `mB`, then `A` is 50% similar to `B`. We use similarity scores to create a similarity matrix as shown in Figure 3.3. 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.

Once the similarity matrix is built, the values of cells reflected across the diagonal of the matrix are averaged to create a half-matrix of undirected similarity edges, as illustrated in Figure 3.4. This facilitates clustering using the Markov Clustering (MCL) algorithm[4]. We chose MCL 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*.

In the implementation, strings are generated for each pattern using Rex [**?**]. Rex generates matching strings by representing the regular expression as an automaton, and then passing that automation to a constraint solver that generates members for it[5]. If the regex matches a finite set of strings smaller than 400, Rex will produce a list of all possible strings. Our goal is to generate 400 strings for each pattern to balance the runtime of the similarity analysis with the precision of the similarity calculations.

For clustering, we prune the similarity matrix to retain all similarity values greater than or equal to 0.75, setting the rest to zero, and then using MCL. This threshold was selected based on recommendations in the MCL manual. The impact of lowering the threshold would likely result in either the same number of more diverse clusters, or a larger number of clusters, but is unlikely to markedly change the largest clusters or their summaries, which are the focus of our analysis for RQ4 (Section 3.4.4), but further study is needed to substantiate this claim. We also note that MCL can also 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 top 100 clusters are categorized by inspection into six categories of behavior.

The end result is clusters and categories of highly behaviorally similar regular expressions, though we note that this approach has a tendency to over-approximate the similarity of two regexes. We measure similarity based on a finite set of generated strings, but some regexes match an infinite set (e.g., `ab*c`), so measuring similarity based on the first 400 strings may lead to an artificially high similarity value. To mitigate this threat, we chose a large number

[4] http://micans.org/mcl/
[5] http://research.microsoft.com/en-us/projects/rex/

Table 3.1 Survey results for number of regexes composed per year by technical environment (RQ1)

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

of generated strings for each regex, but future work includes exploring other approaches to computing regex similarity.

3.4 Results

Next, we present the results of each research question.

3.4.1 RQ1: How do developers use regexes?

The survey was completed by 18 participants (82% response rate) that identified as software developer/maintainers. Respondents have an average of nine years of programming experience ($\sigma = 4.28$). On average, survey participants report to compose 172 regexes per year ($\sigma = 250$) and compose regexes on average once per month, with 28% composing multiple regexes in a week and an additional 22% composing regexes once per week. That is, 50% of respondents uses regexes at least weekly. Table 3.1 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 five (28%) do this 6-10 times per year. For command line usage in tools such as grep, 6 (33%) participants use regexes 51+ times per year. Yet, regexes were rarely used in query languages like SQL. Upon further investigation, it turns out the surveyed developers were not on teams that dealt heavily with a database.

Table 3.2 Survey results for regex usage frequencies for activities, averaged using a 6-point likert scale: Very Frequently=6, Frequently=5, Occasionally=4, Rarely=3, Very Rarely=2, and Never=1 (RQ1)

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

Table 3.2 shows how frequently, on average, the participants use regexes for various activities. Participants answered questions using a 6-point likert scale including very frequently (6), frequently (5), occasionally (4), rarely (3), very rarely (2), and never (1). Averaging across participants, among the most common usages 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 regexes with the same frequency as they test their code (average response was 5.2, which is between frequently and very frequently). Half of the developers indicate that they use external tools to test their regexes, and the other half indicated that they only use tests that they write themselves. Of the nine developers using tools, six mentioned online composition aides such as `regex101.com` where a regex and input string are entered, and the input string is highlighted according to what is matched.

When asked an open ended question about pain points encountered with regular expressions, we observed three main categories. The most common, "hard to compose," was represented in 61% (11) responses. Next, 39% (7) developers responded that regexes are "hard to read" and 17% (3) indicated difficulties with "inconsistency across implementations," which manifest when using regexes in multiple languages. These responses do not sum to 18 as three developers provided multiple parts in their answers.

Table 3.3 How saturated are projects with utilizations? (RQ2)

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

Summary - RQ1: Common uses of regexes include locating content within a file, capturing parts of strings, and parsing user input. 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 regexes being hard to read and hard to write.

3.4.2 RQ2: How is the `re` module used?

We explore regex utilizations and flags used in the scraped Python projects. Out of the 3,898 projects scanned, 42.2% (1,645) contained at least one regex utilization. To illustrate how saturated projects are with regexes, we measure utilizations per project, files scanned per project, files contained utilizations, and utilizations per file, as shown in Table 3.3.

Of projects containing at least one utilization, the average utilizations per project was 32 and the maximum was 1,427. The project with the most utilizations is a C# project[6] 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 3.3, 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.

The number of projects that use each of the `re` functions is shown in Figure **??**. The y-axis denotes the total utilizations, with a maximum of 53,894. The `re.compile` function encompasses 57.6% of all utilizations. Note that compiled objects can also be used to call functions of the re module, ie `compiledObject.findall(...)`, but we ignore these utilizations

[6] https://github.com/Ouroboros/Arianrhod

so that our analysis is easier to automate, and because we are primarily interested in extracting the patterns which these 8 functions contain.

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, was never observed. This may be because developers use it for debugging and choose not to commit it to their repositories.

Summary - RQ2: Only about half of the Python 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.

3.4.3 RQ3: Regex language feature usage

We count the usages of each feature per project and as compared to all distinct regex patterns in the corpus.

3.4.3.1 Feature Usage

Table 3.4 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 *example* column provides a short example of how the feature can be used. The next four columns, (i.e., *brics*, *hampi*, *Rex*, and *RE2*), map to the four major research projects chosen for our investigation (see Section 3.4.3.2). 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 final 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 13,597 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 1,645 projects scanned that contain at least one utilization.

One notable omission from Table 3.4 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 97.7% of patterns. We consider the literal feature to be necessary for any regex related tool to support, and so exclude it from Table 3.4 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. CG is more commonly used in patterns than the highest ranked feature (ADD) by a wide margin (over 8%), even though they appear in similar numbers of projects.

3.4.3.2 Feature Support in Regex Tools

While there are many regex tools available, in this work, we focus on the feature support for four tools, brics, hampi, Rex and RE2, which offer diversity across developers (i.e., Microsoft, Google, open source, and academia) and applications. Further, as we wanted to perform a feature analysis, these four tools and their features are well-documented, allowing for easy comparison.

To create the tool mappings, we consulted documentation for each tool. For brics, we collected the set of supported features using the formal grammar[7]. For hampi, we manually inspected the set of regexes included in the `lib/regex-hampi/sampleRegex` file within the hampi repository[8] (this may have been an overestimation, as this included more features than specified by the formal grammar[9]). For RE2, we used the supported feature documentation[10]. For Rex, we collected the feature set empirically because we tried to parse all scraped patterns with Rex for the behavioral analysis (Section 3.4.4), and Rex provides comprehensive error feedback for unsupported features.

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

[7] http://www.brics.dk/automaton/doc/index.html?dk/brics/automaton/RegExp.html
[8] https://code.google.com/p/hampi/downloads/list
[9] http://people.csail.mit.edu/akiezun/hampi/Grammar.html
[10] https://re2.googlecode.com/hg/doc/syntax.html

projects support the 8 most commonly used features except brics, which does not support STR or END. 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 not.

3.4.3.3 Survey Results for Feature Usage

The pattern language for Python, which is used to compose regexes, supports default character classes like the ANY or dot character class: `.` meaning, '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, which can create semantically equivalent regexes. For example the decimal character class: `\d` is equivalent to a CCC containing all 10 digits: `\d` ≡ `[0123456789]` ≡ `[0-9]`. 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 3.5, with 67% (12) indicating that they use default the most. Participants who favored CCC indicated that "it is more explicit," whereas the participants who favored default character classes said, "it is less verbose" and "I like using built-in code."

To further explore how participants use various regex features, participants were asked five questions about how frequently they use specific related groups of features, chosen based on the tool feature support explored in Section 3.4.3.2. Results are shown in Table 3.6, indicating that lazy repetition and look-ahead features are rarely used and capture groups and endpoint anchors are occasionally to frequently used.

Summary - RQ3: The eight most common features are found in over 50% of the projects. Shown in Table 3.4, the STR and END features are present in over half of the scanned projects containing utilizations. In our survey, over half (56%) of the respondents answered that they

use endpoint anchors frequently or very frequently, and none of them claimed to never use them.

The LZY feature is present in over 36% of scanned projects with utilizations, and yet was not supported by two of the four major regex projects we explored, brics and RE2. 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.

When survey participants were asked if they prefer to always use numbered (BKR) or named (BKRN) back references, 66% (12) of survey participants said that they always use BKR, and the remaining 33% (6) said "it depends." No participants preferred named capture groups. BKR is present in 5% of scanned projects, while BKRN is present in only 1.7%, which corroborates our findings that numbered are generally preferred over named capture groups.

3.4.4 RQ4: Regex behavioral similarity

In clustering the regular expressions, we are most interested in observing behavior of regexes found in multiple projects. Starting with the 13,597 patterns of the corpus, we discarded 10,015 (74%) patterns that were not found in multiple projects. Then we excluded an additional 711 (5%) patterns that contain features not supported by Rex. We studied the remaining 2,871 (21%) patterns using our similarity analysis technique. The impact is that 923 projects were excluded from the data set for the similarity analysis. Omitted features are indicated in Table 3.4 for Rex.

From 2,871 distinct patterns, MCL clustering identified 186 clusters with 2 or more patterns, and 2,042 clusters of size 1. The average size of clusters larger than size one was 4.5. Each pattern belongs to exactly one cluster.

Table 3.7 provides an example of a behavioral cluster containing 12 patterns (four longer patterns omitted for brevity). Patterns from this cluster are present in 31 different projects. All patterns in this cluster share the literal ':' character. The smallest pattern, `':+'`, matches one or more colons.

We observe that the smallest pattern in a cluster provides insight about key characteristic that all the patterns in the cluster have in common. A shorter pattern will tend to have less

extraneous behavior because it is specifying less behavior, yet, in order for the smallest pattern to be clustered, it had to match most of the strings created by Rex from many other patterns within the cluster, and so we observe that the smallest pattern is useful as a representative 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 any member of the cluster appears in, so the cluster in Table 3.7 will be represented as `':+'(31)`. This representation is not an attempt to express all notable behavior of patterns within a cluster, but is a useful and meaningful abbreviation. Other regexes in the cluster may exhibit more diverse behavior, for example the pattern `'([^: ]+):(.*)'` requires a non-colon character to appear before a colon character.

We manually mapped the top 100 largest clusters based on the number of projects into 6 behavioral categories (determined by inspection). The largest cluster was left out, as it was composed of patterns that trivially matched almost any string, like `'b*'` and `'^'`. The remaining 99 clusters were all categorized. These clusters are briefly summarized in Table 3.8, showing the name of the category and the number of clusters it represents, patterns in those clusters, and projects. The most common category is *Multi Matches*, which contains clusters that have alternate behaviors (e.g., matching a comma or a semicolon, as in `',|;'(18)`). Each cluster was mapped to exactly one category. Next, we describe the categories, ordered by the number of projects the regex patterns map to.

3.4.4.1 Multiple Matching Alternatives

The patterns in these clusters match under a variety of conditions by using a character class or a disjunctive `|`. For example: `'(\W)'(89)` matches any alphanumeric character, `'(\s)'(89)` matches any whitespace character, `'\d'(58)` matches any numeric character, and `',|;'(18)` matches a comma or semicolon. Most of these clusters are represented by patterns that use default character classes, as opposed to custom character classes. This provides further support for our survey results to the question, *Do you prefer to use custom character classes or default character classes more often?*, in which a majority of participants indicated they use the default classes more than custom.

3.4.4.2 Specific Character Must Match

Each cluster in this category requires one specific character to match, for example: `‘\n\s*’(42)` matches only if a newline is found, `‘:+’(31)` matches only if a colon is found, `‘%’(22)`, matches only if a percent sign is found and `‘}’(14)` matches only if a right curly brace is found. Table 3.7 presents a cluster that falls under this category. The commonality of this cluster category contrasts with the survey in Section 3.4.1 in which participants reported to very rarely or never use regexes to check for a single character (Table 3.2).

3.4.4.3 Anchored Patterns

Each of the clusters uses at least one endpoint anchor to require matches to be absolutely positioned, for example: `‘(\w+)$’(35)` captures the word characters at the end of the input, `‘^\s’(16)` matches a whitespace at the beginning of the input, and `‘^-?\d+$’(17)` requires that the entire input is an (optionally negative) integer. These anchors are the only way in regexes to guarantee that a character does (or does not) appear at a particular location by specifying what is allowed. As an example, `^[-_A-Za-z0-9]+$` says that from beginning to end, only `[-_A-Za-z0-9]` characters are allowed, so it will fail to match if undesirable characters, such as `?`, appear anywhere in the string.

3.4.4.4 Content of Brackets and Parenthesis

The clusters in this category center around finding a pair of characters that surround content, often also capturing that content. For example, `‘\(.*\)’(29)` matches when content is surrounded by parentheses and `‘".*"’(25)` matches when content is surrounded by double quotes. The cluster `‘<(.+)>’(23)` matches and captures content surrounded by angled brackets.

3.4.4.5 Two or More Characters in Sequence

These clusters require several characters in a row to match some pattern, for example: `‘\d+\.\d+’(30)` requires one or more digits followed by a period character, followed by one or

more digits. The cluster `‘  ’(17)` requires two spaces in a row, and `‘@[a-z]+’(9)` requires the at symbol followed by two or more lowercase characters, as in a twitter handle.

3.4.4.6 Code Search and Variable Capturing

These clusters show a recognizable effort to parse source code or URLs. For example, `‘^https?://’(23)` matches a web address, and `‘(.+)=(.+)’(9)` matches an assignment statement, capturing both the variable name and value. The cluster `‘\$\{([\w\-]+)\}’(11)` matches an evaluated string interpolation and captures the code to evaluate.

Summary - RQ4: We identified six main categories that define regex behavior at a high level: matching with alternatives, matching literal characters, matching with sequences, matching with endpoint anchors, parsing contents of brackets or braces, or searching and capturing code.

3.5 Discussion

In this section, we discuss the implications of these empirical findings and opportunities for future work.

3.5.1 Implications For Tool Designers

The results have implications for regex tool designers.

3.5.1.1 Finding Specific Content

Two categorical clusters, *Specific Characters Must Match* (Section 3.4.4.2) and *Two or More Characters in Sequence* (Section 3.4.4.5), deal with identifying the presence of specific character(s). While multiple character matching subsumes single character matching, the overarching theme is that these regexes are looking to validate strings based on the presence of very specific content, as would be done for many common activities listed in Table 3.2, such as, “Locating content within a file or files.” More study is needed into what content is most frequently searched for, but from our cluster analysis we found that version numbers, twitter

or user handles, hex values, decimal numbers, capitalized words, and particular combinations of whitespace, slashes and other delimiters were discernible targets.

3.5.1.2 Capturing Specific Content Near A Delimiter

The survey results from Section 3.4.1 indicate that capturing parts of strings is among the most frequent activities for which developers use regexes. From a feature perspective, the capture group (CG) is the most frequently used in terms of patterns (Table 3.4). This feature has two functions: 1) logical grouping as would be expected by parenthesis, and 2) retrieval of information in one logical grouping. As mentioned in Section 3.4.4, capturing content was a primary goal evident in several cluster categories. The fourth-largest category is based entirely on capturing the content between brackets or parentheses (Section 3.4.4.4).

Many uses of CG also use the ANY and KLE features, eg. `(.*){(.*)}(.*)` and `\\s*([^: ]*)\\s*:(.*)`. This type of usage frequently revolves around an important delimiter character such as `:` or `\`. This use case is well supported by existing tools for ASCII characters, but future tools should consider the centrality of this use case and its implications for non-English users of regex tools. For example, Unicode characters like 'U+060D' the Arabic Date Separator, or 'U+1806' the Mongolian Todo Soft Hyphen may be used to locate segments of text that a user would want to capture.

3.5.1.3 Counting Lines

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 13,597 patterns in the corpus, 3,410 (25%) contained ANY followed by KLE (i.e., `'.*'`), 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. It may be valuable for tool builders to include support for common activities such as line counting.

3.5.2 Opportunities For Future Work

There are many opportunities for future work.

3.5.2.1 Refactoring Regexes

The survey showed that users want readability and find the lack of readable regexes to be a major pain point. This provides an opportunity to introduce refactoring transformations to enhance readability or comprehension. As one opportunity, certain character classes are logically equivalent and can be expressed differently, for example, `\d` ≡ `[0123456789]` ≡ `[0-9]`. While `\d` is more succinct, `[0-9]` may be easier to read, so a refactoring for *default to custom character classes* could be introduced. Human studies are needed to evaluate the readability and comprehension of various regex features in order to define and support appropriate regex refactorings.

Another avenue of refactoring could be for performance. Various implementations of regex libraries may perform more efficiently with some features than others. An evaluation of regex feature implementation speeds would facilitate semantic transformations based on performance, similar to performance refactorings for LabVIEW [**?** , **?**].

Additionally, some developers may *find* specific content with a regex and then subsequently *capture* it with string parsing, which may be more error prone than using a capture group and indicates a missed opportunity to use the full extent of regex libraries. Future work will explore source code to identify the frequency of such occurrences and design refactorings to better utilize regex library features.

3.5.2.2 Migration Support for Developers

Within standard programming languages, regular expressions libraries are very common, yet there are subtle differences between language libraries in the supported features. For example, Java supports possessive quantifiers like `'ab*+c'` (here the '+' is modifying the '*' to make it possessive) whereas Python does not. Differences among programming language implementations was identified as a pain point for using regular expressions by 17% of the

survey participants. This provides a future opportunity for tools that translate between regex utilizations in various languages.

3.5.2.3 Similarity Beyond String Matching

There are various ways to compute similarity between regexes, each with different trade-offs. While the similarity analysis we employ over-approximates similarity when compared to containment analysis, it may under-approximate similarity in another sense. For example, two regexes that have dissimilar matching behavior could be very similar in purpose and in the eyes of the developer. For example, `commit:\[(\d+)\] - (.*)` and `push:\[(\d+)\] - (.*)` could both be used to capture the id and command from a versioning system, but match very different sets of strings. Future work would apply abstractions to the regex strings, such as removing or relaxing literals, prior to similarity analysis to capture and cluster such similarities.

From another perspective, our regex similarity measure, and even containment analysis, could treat behaviorally identical regexes as the same, when their usage in practice is completely different. For example, in Table 3.7, the regexes `':+'` and `'(:+)'` are behaviorally identical in that they match the same strings, except the latter uses a capture group. In practice, these may be used very differently, where the former may be used for validation and the latter for extraction. This usage difference could be observed by code analysis, and is left for future work.

3.5.2.4 Automated Regex Repair

Regular expression errors are common and have produced thousands of bug reports [**?**]. This provides an opportunity to introduce automated repair techniques for regular expressions. Recent approaches to automated program repair rely on mutation operators to make small changes to source code and then re-run the test suite (e.g., [**?** , **?**]). In regular expressions, it is likely that the broken regex is close, semantically, to the desired regex. Syntax changes through mutation operators could lead to big changes in behavior, so we hypothesize that using the semantic clusters identified in Section 3.4.4 to identify potential repair candidates could efficiently and effectively converge on a repair candidate.

3.5.2.5 Developer Awareness of Best Practices

One category of clusters, *Content of Brackets and Parenthesis*, parses the contents of angle brackets, which may indicate developers are using regexes to parse HTML or XML. As the contents of angle brackets are usually unconstrained, regexes are a poor replacement for XML or HTML parsers. This may be a missed opportunity for the regex users to take advantage of more robust tools. 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.

3.5.2.6 Tool-Specific Regex Exploration

In some environments, such as command line or text editor, regexes are used extensively by the surveyed developers (Section 3.4.1), but these regular expressions do not persist. Thus, using a repository analysis for feature usage only illustrates part of how regexes are used in practice. Exploring how the feature usage differs between environments would help inform tool developers about how to best support regex usage in context, and is left for future work.

3.6 Threats to Validity

The following threats impact our results and conclusions:

Reliability of Measures: The validity of our survey results is dependent on the clarity of the questions. The authors went through several iterations of the survey and included examples for all the regex feature descriptions to improve understandability.

The similarity measure between regexes used in the cluster algorithm is computed empirically rather than analytically, and the more Rex-generated strings used to compute the similarity measure, the more likely it is to be accurate. Our experiments used 400 strings to balance performance and precision, but a higher number could lead to more cohesive clusters. Additionally, regex patters that use any feature not supported by Rex were omitted from the cluster analysis. Last, the threshold of 0.75 was chosen based on the MCL recommendation, but it may not create optimal clusters.

Instrumentation: Regular expression patterns were clustered using strings generated by the Rex tool. We assume that the strings generated by Rex are reasonably diverse to help characterize the regex behavior. To mitigate this threat, Rex generated 400 strings per regex and we inspected strings randomly to ensure diversity.

Implementation errors are a risk for research involving repository analysis. To combat this, we have tested our code and made the repository publicly available[11].

Selection: We mined 3,898 Python projects from GitHub, which is small compared to all available projects with Python code. The projects were mined using the GitHub API which sorts the projects by creation date. By using the API, the goal was to reduce sampling bias from the researchers.

We also 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 regex utilization was added and then removed, leading to fewer utilizations in our final data set.

Recall bias: Survey participants were asked to reflect on their past behavior, which may not represent actual behavior. To mitigate this, we designed the survey to lead participants to think about their behavior before summarizing (e.g., asking how often they use regexes in each of several environments before asking about usage frequency).

Interaction of Selection and Treatment: Our survey participants were software developers from a small startup company and may not be representative of developers who use regexes. Given that the average participant has nine years of development experience, their responses likely pull from a variety of experiences with regular expression usage, but replication with a more diverse set of developers is needed.

Interaction of Setting and Treatment: We only explore regular expressions in Python projects so these results may be coupled with the activities performed using Python and not generalize to other languages. The regex usage context reported by survey participants, however, includes information on regex usage in a variety of settings and languages. Future work will replicate this study in other languages.

[11] GitHub link removed for anonymity

3.7 Conclusion

In this work, we have explored the contexts in which regular expressions are used as well as the features and behavioral similarities of regexes found in open source Python projects. In a survey of 18 professional developers, we find that 50% compose regular expressions at least weekly. The most common purposes are locating content within a file or capturing parts of strings. The most difficult parts about working with regular expressions were reported to be composing and reading them. In a study of regular expression usage in nearly 4,000 Python projects, we find that over 42% of projects contain a regular expression. We present an approach to measure behavioral similarity between regexes by generating strings that match one regex and pairwise testing the remaining regexes against it. This similarity measure is used to form cross-project behavioral clusters. In the top 100 largest clusters, we find that capturing the contents of brackets, searching for delimiter characters and matching alternative values were common behaviors. These results have implications for tool designers and for future research aimed at better supporting developers in using regular expressions.

Acknowledgment

Special thanks to the Dwolla developers for their survey participation. This work is supported in part by NSF SHF-1218265, NSF SHF-EAGER-1446932, and the Harpole-Pentair endowment at Iowa State University.

Table 3.4 How frequently do features appear in projects? (RQ3)

rank	code	description	example	brics	hampi	Rex	RE2	nPatterns	% patterns	nProjects
1	ADD	one-or-more repetition	`z+`	●	●	●	●	6,003	44.1	1,204
2	CG	a capture group	`(caught)`	●	●	●	●	7,130	52.4	1,194
3	KLE	zero-or-more repetition	`.*`	●	●	●	●	6,017	44.3	1,099
4	CCC	custom character class	`[aeiou]`	●	●	●	●	4,468	32.9	1,026
5	ANY	any non-newline char	`.`	●	●	●	●	4,657	34.3	1,005
6	RNG	chars within a range	`[a-z]`	●	●	●	●	2,631	19.3	848
7	STR	start-of-line	`^`	○	●	●	●	3,563	26.2	846
8	END	end-of-line	`$`	○	●	●	●	3,169	23.3	827
9	NCCC	negated CCC	`[^qwxf]`	●	●	●	●	1,935	14.2	776
10	WSP	\t \n \r \v \f or space	`\s`	○	●	●	●	2,846	20.9	762
11	OR	logical or	`a|b`	●	●	●	●	2,102	15.5	708
12	DEC	any of: 0123456789	`\d`	○	●	●	●	2,297	16.9	692
13	WRD	[a-zA-Z0-9_]	`\w`	○	●	●	●	1,430	10.5	650
14	QST	zero-or-one repetition	`z?`	●	●	●	●	1,871	13.8	645
15	LZY	as few reps as possible	`z+?`	○	●	○	●	1,300	9.6	605
16	NCG	group without capturing	`a(?:b)c`	○	●	○	●	791	5.8	404
17	PNG	named capture group	`(?P<name>x)`	○	●	○	●	915	6.7	354
18	SNG	exactly n repetition	`z{8}`	●	●	●	●	581	4.3	340
19	NWSP	any non-whitespace	`\S`	○	●	●	●	484	3.6	270
20	DBB	$n \leq x \leq m$ repetition	`z{3,8}`	●	●	●	●	367	2.7	238
21	NLKA	sequence doesn't follow	`a(?!yz)`	○	○	○	○	131	1	183
22	WNW	word/non-word boundary	`\b`	○	○	○	●	248	1.8	166
23	NWRD	non-word chars	`\W`	○	●	●	●	94	0.7	165
24	LWB	at least n repetition	`z{15,}`	●	●	●	●	91	0.7	158
25	LKA	matching sequence follows	`a(?=bc)`	○	○	○	○	112	0.8	158
26	OPT	options wrapper	`(?i)CasE`	○	●	○	●	231	1.7	154
27	NLKB	sequence doesn't precede	`(?<!x)yz`	○	○	○	○	94	0.7	137
28	LKB	matching sequence precedes	`(?<=a)bc`	○	○	○	○	80	0.6	120
29	ENDZ	absolute end of string	`\Z`	○	○	○	●	89	0.7	90
30	BKR	match the i^{th} CG	`\1`	○	○	○	○	60	0.4	84
31	NDEC	any non-decimal	`\D`	○	●	●	●	36	0.3	58
32	BKRN	references PNG	`\g<name>`	○	●	○	○	17	0.1	28
33	VWSP	matches U+000B	`\v`	○	○	●	●	13	0.1	15

Table 3.5 Survey results for preferences between custom character and default character classes (RQ3)

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 3.6 Survey results for regex usage frequencies, averaged using a 6-point likert scale: Very Frequently=6, Frequently=5, Occasionally=4, Rarely=3, Very Rarely=2, and Never=1 (RQ3)

Group	**Code**	**Frequency**
endpoint anchors	(STR, END)	4.4
capture groups	(CG)	4.2
word boundaries	(WNW)	3.5
lazy repetition	(LZY)	2.9
(neg) look-ahead/behind	(LKA, NLKA, LKB, NLKB)	2.5

Table 3.7 Sample from an example cluster (RQ4)

index	pattern	nProjects	index	pattern	nProjects
1	`':+'`	8	5	`'[:]'`	6
2	`'(:)'`	8	6	`'([^:]+):(.*)'`	6
3	`'(:+)'`	8	7	`'\s*:\s*'`	4
4	`'(:)(:*)'`	8	8	`'\:'`	2

Table 3.8 Cluster categories and sizes (RQ4)

Category	**Clusters**	**Patterns**	**Projects**
Multi Matches	21	237	295
Specific Char	17	103	184
Anchored Patterns	20	85	141
Content of Parens	10	46	111
Two or More Chars	16	40	120
Code Search	15	27	92

Bibliography

CHAPTER 4. Refactoring

Abstract

Regular expressions (regex) are powerful tools employed across many tasks and platforms. Regex can be complex, so optimizing understandability of regex is desirable for maintainers. Because of a rich feature set, there is more than one way to compose a regex to get the same desired behavior. We define five equivalence classes where the same behavior can be achieved with multiple representations. With the goal of finding refactorings that improve understandability, we analyze regexes in GitHub to find community standards, and obtain understandability metrics from an empirical study with 180 participants to find out which representations are more understandable to programmers. We found, for example, that patterns requiring one or more of some character expressed using kleene star such as `‘::*’` are more understandable when expressed using the plus: `‘:+’`. We identify strongly preferred transformations in three of the five equivalence classes and identify opportunities for future work in improving regex refactoring.

4.1 Introduction

Regular expressions are used frequently by developers for many purposes, such as parsing files, validating user input, or querying a database. Regexes are also employed in MySQL injection prevention [**?**] and network intrusion detection [**?**]. However, recent research has suggested that regular expressions (regexes) are hard to understand, hard to compose, and error prone [**?**]. Given the difficulties with working with regular expressions and how often they appear in software projects and processes, it seems fitting that efforts should be made to ease the burden on developers.

Tools have been developed to make regexes easier to understand, and many are available online. Some tools will, for example, highlight parts of regex patterns that match parts of strings as a tool to aid in comprehension.[1] Others will automatically generate strings that are matched by the regular expessions [**?**]. Other tools will automatically generate regexes when given a list of strings to match [**?** , **?**]. The commonality of such tools provides evidence that people need help with regex composition and understandability.

In software, code smells have been found to hinder understandability of source code [**?** , **?**]. Once removed through refactoring, the code becomes more understandable, easing the burden on the programmer. In regular expressions, such code smells have not yet been defined, perhaps in part because it is not clear what makes a regex smelly.

As with source code, in regular expressions, there are multiple ways to express the same semantic concept. For example, the regex, `'aa*'` matches an `a` followed by zero or more `a`'s, and is is equivalent to `'a+'` , which matches one or more `a`'s. What is not clear is which representation, `'aa*'` or `'a+'`, is preferred. Preferences in regex refactorings could come from a number of sources, including which is easier to maintain, easier to understand, or better conforms to community standards, depending on the goals of the programmer.

In this work, we introduce possible refactorings in regular expressions by identifying equivalence classes of regex representations and transformations between the representations. These equivalence classes provide options for how to represent double-bounds in repetitions (e.g.,

[1] https://regex101.com/

‘a{1,2}’ or ‘a|aa’), single-bounds in repetitions (e.g., ‘a{2}’ or ‘aa’), lower bounds in repetitions (e.g., ‘a{2,}’ or ‘aaa*’), character classes (e.g., ‘[0-9]’ or ‘[\d]’), and literals (e.g., ‘\a’ or ‘\x07’). We suggest directions for the refactorings, for example, from ‘aa*’ to ‘a+’, based on two high-level concepts: which representation appears most frequently in source code (conformance to community standards) and which is more understandable by programmers, based on comprehension tests completed by 180 study participants. Our results identify preferred representations for four of the five equivalence classes based on mutual agreement between community standards and understandability, with three of those being statistically significant. For the fifth group on double-bounded repetitions, two recommendations are given depending on the goals of the programmer.

Our contributions are:

- Identification of equivalence classes for regular expressions with possible transformations within each class,
- Conducted an empirical study with 180 participants evaluating regex understandability,
- Conducted an empirical study identifying opportunities for regex refactoring in Python projects based on how regexes are expressed, and
- Identified preferred regex representations and refactorings that are the most understandable and conform best to community standards, backed by empirical evidence.

To our knowledge, this is the first work to apply refactoring to regular expressions. Further, we approach the problem of identifying preferred regex representations by looking at thousands of regexes in Python projects and measuring the understandability of various regex representations using human participants. The rest of the paper describes equivalence classes and possible refactorings as well as our two empirical studies, one using source code artifacts and another using human participants.

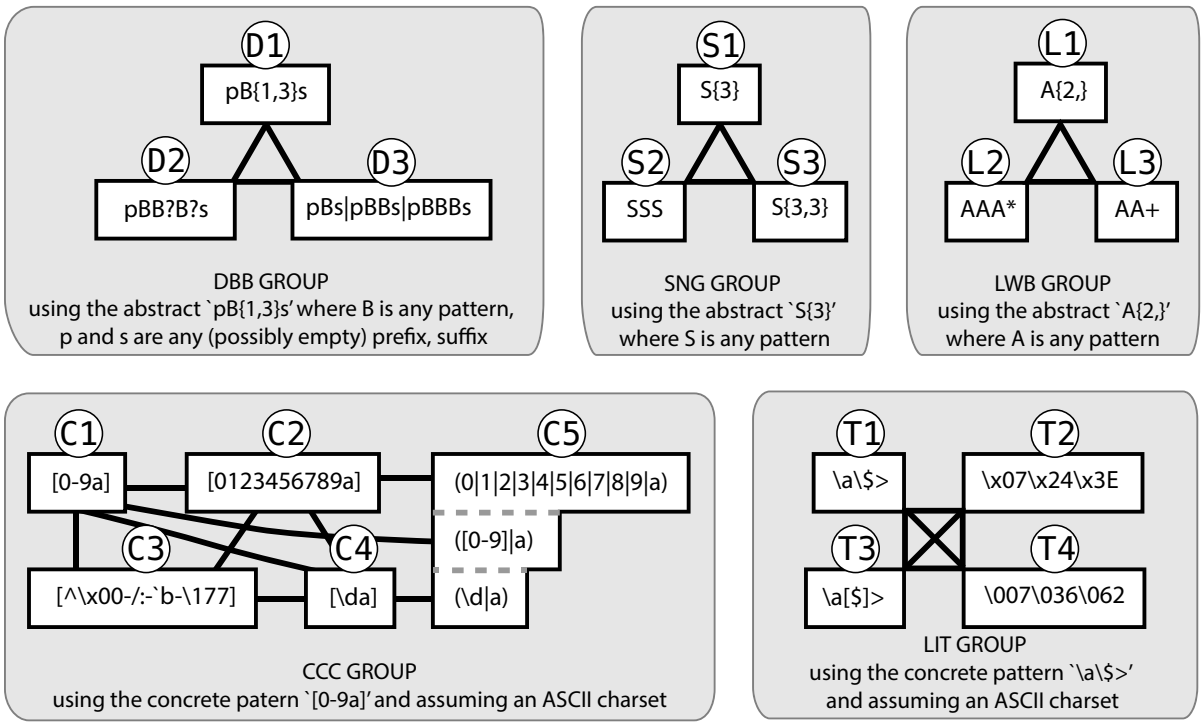

Figure 4.1 Equivalence classes with various representations of semantically equivalent refactorings within each class. DBB = Double-Bounded, SNG = Single Bounded, LWB = Lower Bounded, CCC = Custom Character Class and LIT = Literal

4.2 Refactorings

After studying over 13,000 distinct regex strings from nearly 4,000 Python projects, we have defined a set of equivalence classes for regexes with refactorings that can transform among members in the classes. For example, `AAA*` and `AA+` are semantically identical, except one uses the star operator (indicating zero or more repetitions) and the other uses the plus operator (indicating one or more repetitions). Both match strings with two or more `A`'s.

Figure 4.1 displays the five equivalence classes in grey boxes and various semantically equivalent *representations* of a regex are shown in white boxes. For example, LWB is an equivalence class with representations that all have a lower bound on repetitions. Regexes `AAA*` and `AA+` are both members of this class mapping to representations L2 and L3, respectively, along with the L1 representation, `A{2,}`. The undirected edges between the representations define possible refactorings. Identifying the best direction for each arrow in the possible refactorings is discussed in Section 3.4.

We use concrete regexes in the representations to more clearly illustrate examples of the representations. However, the `A`'s in the LWB group abstractly represent any pattern that could be operated on by a repetition modifier (literal characters, character classes, groups, etc.). We chose the lower bound repetition threshold of 2 for illustration; in practice this could be any number, including zero. Next, we describe each group, the representations, and possible transformations in detail:

CCC Group The Custom Character Class (CCC) group has regex representations that use the custom character class language feature or can be represented by such a feature. A custom character class enables a programmer to specify a set of alternative characters, any of which can match. For example, the regex `'c[ao]t'` will match both the string "cat" and the string "cot" because, between the `c` and `t`, there is a custom character class, `[ao]`, that specifies either `a` or `o` (but not both) must be selected. We use the term *custom* to differentiate these classes created by the user from the default character classes, : `\d`, `\D`, `\w`, `\W`, `\s`, `\S` and `.`, provided by most regex libraries. Next, we provide descriptions of each representation in this equivalence class:

C1: Any pattern using a range feature like `[a-f]` as shorthand for all of the characters between 'a' and 'f' (inclusive) within a (non-negative) character class belongs to the C1 node.

C2: Any pattern that contains at least one (non-negative) custom character class without any shorthand representations, specifically ranges or defaults. For example, `'[012]'` is in C2, but `'[0-2]'` is not.

C3: Any character classes expressed using negation, which is indicated by a caret (i.e., `^`) followed by a custom character class specification. For example, the pattern `[^ao]` matches every character *except* `a` or `o`. If the applicable character set is known (e.g., ASCII, UTF-8, etc.), then any non-negative character class can be represented as a negative character class. For example, assuming an ASCII charset that has 128 characters: `\x00-\x7f`, a character class representing the lower half: `[\x00-\x3f]` can be represented by negating the upper half: `[^\x40-\x7f]`.

C4: Any pattern using a default character class such as `\d` or `\W` within a (non-negative) character class belongs to the C4 node.

C5: While not expressed using a character class, these representations can be transformed into custom character classes by removing the ORs and adding square brackets (e.g., `(\d|a)` in C5 is equivalent to `[\da]` in C4). All custom character classes expressed as an OR of length-one sequences, including defaults or other CCCs, are included in C5. Note that because an OR cannot be directly negated, it does not make sense to have an edge between C3 and C5 in Figure 4.1, though C3 may be able to transition to C1, C2 or C4 first and then to C5.

A pattern can belong to multiple representations. For example, `[a-f\d]` belongs to both C1 and C4. The edge between C1 and C4 represents the opportunity to express the same pattern as `[a-f0-9]` by transforming the default digit character class into a range. This transformed version would only belong to the C1 node.

DBB Group The Double-Bounded (DBB) group contains all regex patterns that use some repetition defined by a (non-equal) lower and upper boundary. For example the pattern `pB{1,3}s` represents a `p` followed by one to three sequential `B` patterns, then followed by a single `s`. This will match "pBs", "pBBs", and "pBBBs".

D1: Any pattern that uses the curly brace repetition with a lower and upper bound, such as `pB{1,3}s`, belongs to the D1 node. Note that `pB{1,3}s` can become `pBB{0,2}s` by pulling the lower bound out of the curly braces and into the explicit sequence (or visa versa). Nonetheless, it would still be part of D1, though this within-node refactoring on D1 is not discussed in this work.

D2: Any pattern that uses the questionable (i.e., `?`) modifier implies a lower-bound of zero and an upper-bound of one, and belongs to D2. For example, when a double-bounded regex has zero on the lower bound, as is the case with `pBB{0,2}s` in D1, transforming it to D2 involves replacing the curly braces with n questionable modifiers, where n is the upper bound, creating `pBB?B?s`.

D3: Any pattern that has a repetition with a lower and upper boundary and is expressed using ORs is part of D3. The example, `pB{1,3}s` would become `pBs|pBBs|pBBs` by expanding on each option in the boundaries. Note also that a pattern can belong to multiple nodes in the DBB group, for example, `(a|aa)X?Y{2,4}` belongs to all three nodes.

Note that a pattern can belong to multiple nodes in the DBB group, for example, `(a|aa)X?Y{2,4}` belongs to all three nodes: `Y{2,4}` maps it to D1, `X?` maps it to D2, and `(a|aa)` maps it to D3.

LIT Group All patterns that are not purely default character classes have to use some literal tokens to specify what characters to match. In Python and most other languages that support regex libraries, the programmer is able to specify literal tokens in a variety of ways. In our example we use the ASCII charset, in which all characters can be expressed using hex and octal codes like `\xF1`, and `\0108`, respectively. This group defines transformations among various representations of literals.

T1: Patterns that do not use any hex characters (T2), wrapped characters (T3) or octal (T4), but use at least one literal character belong to the T1 node.

T2: Any pattern using hex tokens, such as `\x07`, belongs to the T2 node.

T3: Any literal wrapped in square brackets belongs to T3. Literal character can be wrapped in brackets to form a custom character class of size one, such as `[x][y][z]`. This style is used most often to avoid using a backslash for a special character in the regex language, for example, `[{]` which must otherwise be escaped like `\{`.

T4: Any pattern using octal tokens, such as `\007`, belongs to the T4 node.

Patterns often fall in multiple of these representations, for example, `abc\007` includes literals `a`, `b`, and `c`, and also octal `\007`, thus belonging to T1 and T4.

LWB Group The lower-bounded (LWB) group contains all patterns that specify only a lower boundary on the number of repetitions required for a match. This can be expressed

using curly braces with a comma after the lower bound but no upper bound, for example `A{3,}` which will match 'AAA', 'AAAA', 'AAAAA', and any number of A's greater or equal to 3.

L1: Any pattern using this curly braces-style LWB repetition belongs to node L1.

L2: The kleene star (KLE) means zero-or-more of something, and so `X*` is equivalent to `X{0,}`. Any pattern using KLE belongs to the L2 node.

L3: One of the most commonly used regex features is additional repetition (ADD), for example `T+` which means one-or-more T's. This is equivalent to `T{1,}`. Any pattern using ADD repetition belongs to the L3 node.

Regex patterns often belong to multiple nodes, for example, with `A+B*`, `A+` maps it to L3 and `B*` maps it to L2. We note that the refactorings from L1 to L3 and L2 to L3 are not always possible, specifically when the lower bound is zero and the pattern is not repeated in sequence (e.g., `'A*'` or `'A{0,}'`).

SNG Group This equivalence class contains three representations of a regex that deal with repetition of a single element in the regex, represents by `S`.

S1: Any pattern with a single repetition boundary in curly braces belongs to S1. For example, `S{3}`, states that S appears exactly three times in sequence.

S2: Any pattern that is explicitly repeated two or more times and could use repetition operators is part of S2.

S3: Any pattern with a double-bound in which the upper and lower bounds are same belong to S3. For example, `S{3,3}` states `S` appears a minimum of 3 and maximum of 3 times.

The important factor distinguishing this group from DBB and LWB is that there is a single finite number of repetitions, rather than a bounded range on the number of repetitions (DBB) or a lower bound on the number of repetitions (LWB).

Example Regular expressions will often belong to many representations in the equivalence classes described here, and often multiple representations within an equivalence class. Using

an example from a Python project, the regex `'[^ ]*\.[A-Z]{3}'` is a member of S1, L2, C1, C3, and T1. This is because `'[^ ]'` maps it to C3, `'[^ ]*'` maps it to L2, `'[A-Z]'` maps it to C1, `'\.'` maps it to T1, and `'[A-Z]{3}'` maps it to S1. As examples of refactorings, moving from S1 to S2 would be possible by replacing `'[A-Z]{3}'` with `'[A-Z][A-Z][A-Z]'` and moving from L2 to L1 would replace `'[^ ]*'` with `'[^ ]{0,}'`, resulting in a refactored regex of: `'[^ ]{0,}\.[A-Z][A-Z][A-Z]'`.

4.3 Research Questions

After defining the equivalence classes and potential regex refactorings as described in Section 4.2, we wanted to know which representations in the equivalence classes are considered desirable and which might be smelly. Desirability for regexes can be defined many ways, including maintainable, understandable, and performance. We focus on refactoring for understandability.

We define understandability two ways. First, assuming that common programming practices are more understandable than uncommon practices, we explore the frequencies of each representation from Figure 4.1 using thousands of regexes scraped from Python projects. Second, we then present people with regexes exemplifying some of the more common characteristics and ask them comprehension questions along two directions: determine which of a list of strings are matched by the regex, and compose a string that is matched by the regex. Our research questions are:

RQ1: Which refactorings have the strongest *community support* based on how frequently each representation appears in regexes in open source Python projects?

RQ2: Which refactorings have the strongest support based on *understandability* as measured by matching strings and composing strings?

RQ3: Which regex representations are most desirable based on both community support and understandability?

Next, we present the analysis and results for each question in turn, followed by a unified discussion in Section 4.7.

Table 4.1 How frequently is each alternative expression style used?

Node	Description	Example	nPatterns	% patterns	nProjects
C1	char class using ranges	`'^[1-9][0-9]*$'`	2,479	18.2%	810
C2	char class explicitly listing all chars	`'[aeiouy]'`	1,903	14.0%	715
C3	any negated char class	`'[^A-Za-z0-9.]+'`	1,935	14.2%	776
C4	char class using defaults	`'[-+\d.]'`	840	6.2%	414
C5	an OR of length-one sub-patterns	`'(@|<|>|-|!)'`	245	1.8%	239
D1	curly brace repetition like {M,N} with M¡N	`'^x{1,4}$'`	346	2.5%	234
D2	zero-or-one repetition using question mark	`'^http(s)?://'`	1,871	13.8%	646
D3	repetition expressed using an OR	`'^(Q|QQ)\<(.+)\>$'`	10	.1%	27
T1	no HEX, OCT or char-class-wrapped literals	`'get_tag'`	12,482	91.8%	1,485
T2	has HEX literal like `\xF5`	`'[\x80-\xff]'`	479	3.5%	243
T3	has char-class-wrapped literals like [$]	`'[$][{]\d+:([^}]+)[}]'`	307	2.3%	268
T4	has OCT literal like `\0177`	`'[\041-\176]+:$'`	14	.1%	37
L1	curly brace repetition like {M,}	`'(DN)[0-9]{4,}'`	91	.7%	166
L2	zero-or-more repetition using kleene star	`'\s*(#.*)?$'`	6,017	44.3%	1,097
L3	one-or-more repetition using plus	`'[A-Z][a-z]+'`	6,003	44.1%	1,207
S1	curly brace repetition like {M}	`'^[a-f0-9]{40}$'`	581	4.3%	340
S2	explicit sequential repetition	`'ff:ff:ff:ff:ff:ff'`	3,378	24.8%	861
S3	curly brace repetition like {M,M}	`'U[\dA-F]{5,5}'`	27	.2%	32

4.4 Community Support Study (RQ1)

The goal of this study is to understand how frequently each of the regex representations appears in source code. Based on the results, we identify preferred representations using popularity in source code.

4.4.1 Artifacts

To determine how common each regex representations is in the wild, we collected regexes from GitHub projects. We specifically targeted Python projects as it is a popular programming language with a strong presence on GitHub. Further, Python is the fourth most common language on GitHub (after Java, Javascript and Ruby) and Python's regex pattern language is close enough to other regex libraries that our conclusions are likely to generalize.

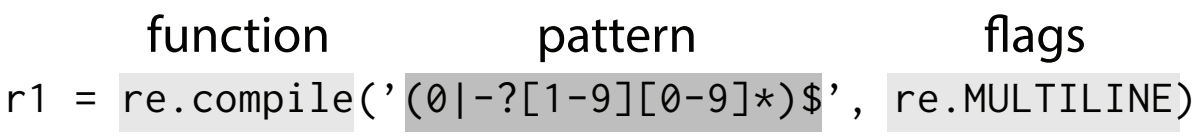

Figure 4.2 Example of one regex library invocation

We collected and analyzed static invocations to the Python `re` library. Figure 4.2 presents an example from Python with key components labeled. The *function* called is `re.compile`. The *pattern* defines what strings will be matched and the *flag* `re.MULTILINE` modifies the rules used by the regex engine when matching. When executed, a regex object `r1` is created and it 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 `-`).

Our goal was to collect regex patterns from a variety of projects to represent the breadth of how developers use regexes. We scraped 3,898 projects containing Python code using the GitHub API. This was done by systematically selecting repository IDs, checking the repository for Python files, and retaining the project if Python was found. After dividing eight million repository IDs into 32 groups, we scanned from the beginning until we had collected approximately four thousand Python projects. At that point, we felt we had enough data to do an analysis without further perfecting our mining techniques.

To identify invocations of the `re` module, we built the AST of each Python file in each project. In most projects, almost all `re` invocations are present in the most recent version of a project, but to be more thorough, we also scanned up to 19 earlier versions. All regex patterns were obtained, sans duplicates. In the end, we observed and recorded 16,088 non-duplicate patterns in 1,645 projects.

In collecting the set of distinct patterns for analysis, we ignore the 12.7% of `re` invocations using flags, which can alter regex behavior. An additional 6.5% of `re` invocations contained patterns that could not be compiled because the pattern was non-static (e.g., used some run-time variable). This parser was unable to support 0.8% (114) due to error. After removing all problematic patterns as described and collapsing on duplicates, we ended up with 13,597 distinct patterns from 1,544 projects remained to be used in this study.

4.4.2 Metrics

We measure community support by matching each regex in the corpus to the representations (nodes) in Figure 4.1 and counting the number of *patterns* that contain the representation and the number of *projects* that contain the representation. Note that a regex can belong to multiple representations, and a regex can belong to multiple projects since we collapsed duplicates and only analyze the distinct regex patterns.

4.4.3 Analysis

To determine how many of the representations match patterns in the corpus, we performed an analysis using the PCRE parser and by representing the regexes as token streams, depending on the characteristics of the representation. Our analysis code is available on GitHub[2]. Next, we describe the process in detail:

4.4.3.1 Presence of a Feature

For the representations that only require a particular feature to be present, such as the question-mark in D2, the features identified by the PCRE parser were used to decide membership of patterns in nodes. These feature-requiring nodes are as follows: D1 requires double-bounded repetition with different bounds, D2 requires the question-mark repetition, S1 requires single-bounded repetition, S3 requires double-bounded repetition with the same bounds, L1 requires a lower-bound repetition, L2 requires the kleene star (`*`) repetition, L3 requires the add (`+`) repetition, and C3 requires a negated custom character class.

4.4.3.2 Features and Pattern

For some representations, the presence of a feature is not enough to determine membership. However, the presence of a feature and properties of the pattern can determine membership.

Identifying D3 requires an OR containing at least two entries - some sequence present in one entry repeated N times, and then the same sequence present in another entry repeated N+1 times. This is a hard pattern to detect directly, but we identified candidates by looking

[2] https://github.com/softwarekitty/regex_readability_study

for a sequence of N repeating groups with an OR-bar (ie. `|`) next to them on one side (either side). This produced a list of 113 candidates which we narrowed down manually to 10 actual members.

Identifying T2 requires a literal feature that matches the regex `(\\x[a-f0-9A-F]{2})` which reliably identifies hex codes within a pattern. Similarly T4 requires a literal feature and must match the regex `((\\0\d*)|(\\\d{3}))` which is specific to Python-style octal, requiring either exactly three digits after a slash, or a zero and some other digits after a slash. Only one false positive was identified which was actually the lower end of a hex range using the literal `\0`.

Identifying T3 requires that a single literal character is wrapped in a custom character class (a member of T3 is always a member of C2). T1 requires that no characters are wrapped in brackets or are hex or octal characters, which actually matches over 91% of the total patterns analyzed.

4.4.3.3 Token Stream

The following representations were identified by representing the regex patterns as a sequence of dot-delimited tokens. Identifying S2 requires any element to be repeated at least twice. This element could be a character class, a literal, or a collection of things encapsulated in parentheses. Identifying C1 requires that a non-negative character class contains a range. Identifying C2 requires that there exists a custom character class that does not use ranges or defaults. Identifying C4 requires the presence of a default character class within a custom character class, specifically, `\d`, `\D`, `\w`, `\W`, `\s`, `\S` and `.`. Identifying C5 requires an OR of length-one sequences (literal characters or any character class).

4.4.4 Results

Table 4.1 presents the frequencies with which each representation appears in a regex pattern and in a project scraped from GitHub. The *node* column references the representations in Figure 4.1 and the *description* column briefly describes the representation, followed by an *example* from the corpus. The *nPatterns* column counts the patterns that belong to the representation, followed by the percent of patterns out of 13,597. The *nProjects* column counts the projects

that contain a regex belonging to the representation, followed by the percentage of projects out of 1,544. Recall that the patterns are all unique and could appear in multiple projects, hence the project support is used to show how pervasive the representation in across the whole community. For example, 2,479 of the patterns belong to the C1 representation, representing 18.2% of the patterns. These appear in 810 projects, representing 52.5%. Representation D1 appears in 346 (2.5%) of the patterns but only 234 (15.2%) of the projects. In contrast, representation T3 appears in 39 *fewer* patterns but 34 *more* projects, indicating that D1 is more concentrated in a few projects and T3 is more widespread across projects.

Using the pattern frequency as a guide, we can create refactoring recommendations based on community frequency. For example, since C1 is more prevalent than C2, we could say that C2 is smelly since it could better conform to the community standard if expressed as C1. Thus, we might recommend a $\overrightarrow{C2C1}$ refactoring. Based on patterns alone, the winning representations per equivalence class are C1, D2, T1, L2, and S2. With one exception, these are the same for recommendations based on projects. The difference is that L3 appears in more projects than L2, so it is not clear which is more desirable based on community standards. Section 4.6 explores these results more deeply.

4.5 Understandability Study (RQ2)

The overall idea of this study is to present programmers with one of several representations of semantically equivalent regexes and ask comprehension questions. By comparing the understandability of semantically equivalent regexes that have different representations, we aim to understand which representations are more desirable and which are more smelly. This study was implemented on Amazon's Mechanical Turk with 180 participants. Each regex pattern was evaluated by 30 participants. The patterns used were designed to belong to various representations in Figure 4.1.

4.5.1 Metrics

We measure the understandability of regexes using two complementary metrics, *matching* and *compostition*.

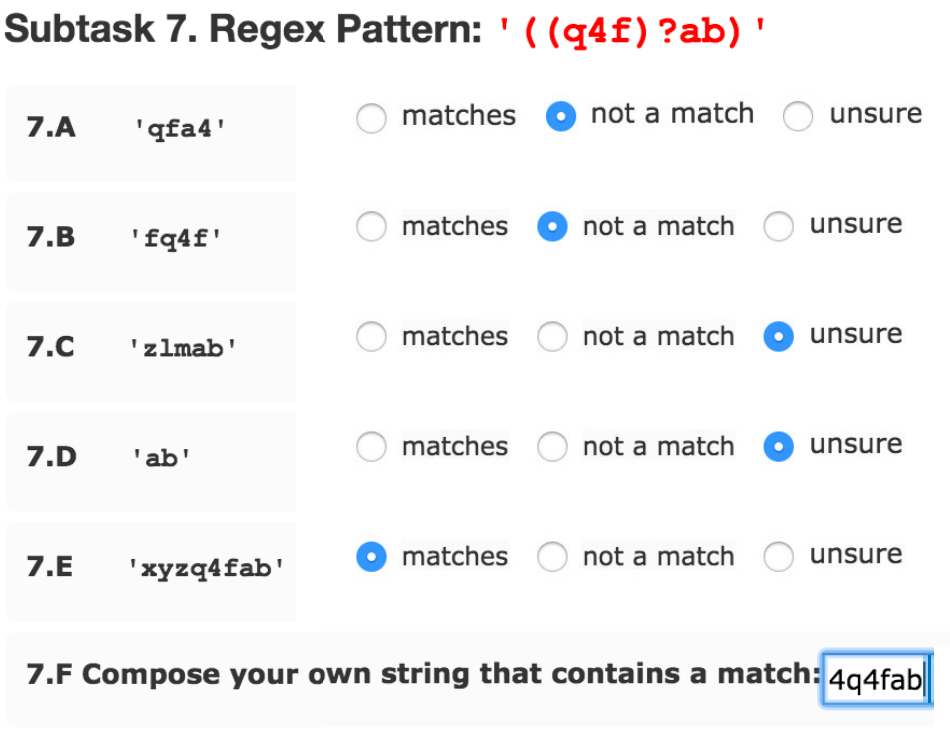

Figure 4.3 Example of one HIT Question

Table 4.2 Matching metric example

String	'RR*'	Oracle	P1	P2	P3	P4
1	"ARROW"	✓	✓	✓	✓	✓
2	"qRs"	✓	✓	×	×	?
3	"R0R"	✓	✓	✓	?	-
4	"qrs"	×	✓	×	✓	-
5	"98"	×	×	×	×	-
	Score	1.00	0.80	0.80	0.50	1.00

✓= match, ×= not a match, ? = unsure, – = left blank

Matching: Given a pattern and a set of strings, a participant determines which strings will be matched by the pattern. There are four possible responses for each string, *matches*, *not a match*, *unsure*, or blank. An example from our study is shown in Figure 4.3.

The percentage of correct responses, disregarding blanks and unsure responses, is the matching score. For example, consider regex pattern `'RR*'` and five strings shown in Table 4.2, and the responses from four participants in the *P1*, *P2*, *P3* and *P4* columns. The oracle has the first three strings matching since they each contain at least one `R` character. *P1* answers correctly

for the first three strings but incorrectly thinks the fourth string matches, so the matching score is $4/5 = 0.80$. *P2* incorrectly thinks that the second string is not a match, so they also score $4/5 = 0.80$. *P3* marks 'unsure' for the third string and so the total number of attempted matching questions is 4 instead of 5. *P3* is incorrect about the second and fourth string, so they score $2/4 = 0.50$. For *P4*, we only have data for the first and second strings, since the other three are blank. *P4* marks 'unsure' for the second matching question so only one matching question has been attempted, and it was answered correctly so the matching score is $1/1 = 1.00$.

Blanks were incorporated into the metric because questions were occasionally left blank in the study. Unsure responses were provided as an option so not to bias the results when participants were honestly unsure of the answer. These situations did not occur very frequently. Only 1.1% of the responses were left blank and only 3.8% of the responses were marked as unsure. We refer to a response with all blank or unsure responses as an 'NA'. Out of 1800 questions, 1.8%(32) were NA's (never more than 4 out of 30 per pattern).

Composition: Given a pattern, a participant composes a string they think it matches. If the participant is accurate and the string indeed is matched by the pattern, then a composition score of 1 is assigned, otherwise 0. For example, given the pattern `'(q4fab|ab)'` from our study, the string, "xyzq4fab" matches and would get a score of 1, and the string, "acb" does not match and would get a score of 0.

To determine a match, each pattern was compiled using the *java.util.regex* library. A *java.util.regex.Matcher* `m` object was created for each composed string using the compiled pattern. If `m.find()` returned true, then that composed string was given a score of 1, otherwise it was given a score of 0.

4.5.2 Design

This study was implemented on the Amazon's Mechanical Turk (MTurk), a crowdsourcing platform in which requesters can create human intelligence tasks (HITs) for completion by workers. Each HIT is designed to be completed in a fixed amount of time and workers are compensated with money if their work is satisfactory. Requesters can screen workers by requiring each to complete a qualification test prior to completing any HITs.

4.5.2.1 Worker Qualification

Workers qualified to participate in the study by answering questions regarding some basics of regex knowledge. These questions were multiple-choice and asked the worker to describe what the following patterns mean: `‘a+’`, `‘(r|z)’`, `‘\d’`, `‘q*’`, and `‘[p-s]’`. To pass the qualification, workers had to answer four of the five questions correctly.

4.5.2.2 Tasks

Using the patterns in the corpus as a guide, we created 60 regex patterns that were grouped into 26 semantic equivalence groups. These semantic groups were focused on exploring edges in the equivalence classes. In this way, we can draw conclusions about transformations between representations since the regexes evaluated were semantically equivalent.

For example, a group with regexes `‘([0-9]+)\.([0-9]+)’` and `‘(\d+)\.(\d+)’` is intended to evaluate the edge between C1 and C4. There were 18 groups with two regexes that target various edges in the equivalence classes. The other eight semantic groups had three regexes each. For example, a semantic group with regexes `‘((q4f){0,1}ab)’`, `‘((q4f)?ab)’`, and `‘(q4fab|ab)’` is intended to explore the edges among D1, D2, and D3.

For each of the 26 groups of patterns, we created five strings, where at least one matched and at least one did not match. These strings were used to compute the matching metric.

Once all the patterns and matching strings were collected, we created tasks for the MTurk participants as follows: randomly select a pattern from each of the 10 metagroups. Randomize the order of these 10 patterns, as well as the order of the matching strings for each pattern. After adding a question asking the participant to compose a string that each pattern matches, this creates one task on MTurk. This process was completed until each of the 60 regexes appeared in 30 HITs, resulting in a total of 180 total unique HITs. An example of a single regex pattern, the five matching strings and the space for composing a string is shown in Figure 4.3.

Table 4.3 Averaged Info About Edges (sorted by lowest of either p-value)

Index	Representations	Pairs	Match1	Match2	$H_0 : \mu_{match1} = \mu_{match2}$	Compose1	Compose2	$H_0 : \mu_{comp1}$
E1	T1 – T4	2	0.80	0.60	0.001	0.87	0.37	**¡0.0**
E2	D2 – D3	2	0.78	0.87	**0.011**	0.88	0.97	0.08
E3	L2 – L3	3	0.86	0.91	**0.032**	0.91	0.98	0.05
E4	C2 – C5	4	0.85	0.86	0.602	0.88	0.95	0.06
E5	C2 – C4	1	0.83	0.92	0.075	0.60	0.67	0.60
E6	D1 – D2	2	0.84	0.78	0.120	0.93	0.88	0.34
E7	C1 – C2	2	0.94	0.90	0.121	0.93	0.90	0.51
E8	T2 – T4	2	0.84	0.81	0.498	0.65	0.52	0.14
E9	C1 – C5	2	0.94	0.90	0.287	0.93	0.93	1.00
E10	T1 – T3	3	0.88	0.86	0.320	0.72	0.76	0.61
E11	D1 – D3	2	0.84	0.87	0.349	0.93	0.97	0.40
E12	C1 – C4	6	0.87	0.84	0.352	0.86	0.83	0.46
E13	C3 – C4	2	0.61	0.67	0.593	0.75	0.82	0.37
E14	S1 – S2	3	0.85	0.86	0.776	0.88	0.90	0.63

4.5.2.3 Implementation

Workers were paid $3.00 for successfully completing a HIT, and were only allowed to complete one HIT. The average completion time for accepted HITs was 682 seconds (11 mins, 22 secs). A total of 55 HITs were rejected, and of those, 48 were rushed through by one person leaving many answers blank, 4 other HITs were also rejected because a worker had submitted more than one HIT, one was rejected for not answering composition sections, and one was rejected because it was missing data for 3 questions. Rejected HITs were returned to MTurk to be completed by others.

4.5.3 Participants

In total, there were 180 participants in the study. A majority were male (83%) with an average age of 31. Most had at least an Associates degree (72%) and most were at least somewhat familiar with regexes prior to the study (87%). On average, participants compose 67 regexes per year with a range of 0 to 1000.

4.5.4 Analysis

For each of the 180 HITs, we computed a matching and composition score for each of the 10 regexes, using the metrics described in Section 4.5.1. This allowed us to compute and then average 26-30 values for each metric for each of the 60 regexes (fewer than 30 values were used if all the responses in a matching question were unsure or a combination of blanks and unsure).

Each regex was a member of one of 26 groupings of equivalent regexes. These groupings allow pairwise comparisons of the metrics values to determine which representation of the regex was most understandable and the direction of a refactoring for understandability. Among all the groups, we performed 42 pairwise comparisons of the matching and composition scores (i.e., one comparison for each group of size two and three comparisons within each group of size three). For example, one group had regexes, `RR*` and `R+`, which represent a transformation between L2 and L3. The former had an average matching of 86% and the latter had an average matching of 92%. The average composition score for the former was 97% and 100% for the latter. Thus, the community found `R+` from L3 more understandable. There were two other pairwise comparisons performed between the L2 and L3 group, using regexes pair `zaa*` and `za+'`, and regexes pair `\..*` and `\.+'`. Considering all three of these regex pairs, the overall matching average for the regexes belonging to L2 was 0.86 and 0.91 for L3. The overall composition score for L2 was 0.91 and 0.98 for L3. Thus, the community found L3 to be more understandable than L2, from the perspective of both understandability metrics, suggesting a refactoring from L2 to L3.

This information is presented in summary in Table 4.3, with this specific example appearing in the E3 row. The *Index* column enumerates all the pairwise comparisons evaluated in this experiment, *Representations* lists the two representations, *Pairs* shows how many comparisons were performed, *Match1* gives the overall matching score for the first representation listed, *Match2* gives the overall matching score for the second representation listed, and $H_0 : \mu_{match1} = \mu_{match2}$ uses the Mann-Whitney test of means to compare the matching scores, and presents the p-values. The last three columns list the average composition scores for the representations and the p-value, also using the Mann-Whitney test of means.

Although we had 42 pairwise comparisons, we had to drop six comparisons due to a design flaw since the patterns performed transformations from multiple equivalence classes. For example, pattern `([\072\073])` is in C2 and T4, and was grouped with pattern `(:|;)` in C5, T1, so it was not clear if any differences in understandability were due to the transformation between C2 and C5, or T4 and T1. However, the third member of the group, `([:;])`, could be compared with both, since it is a member of T1 and C2, so comparing it to `([\072\073])` evaluates the transformation between T1 and T4, and comparing to `(:|;)` evaluates the transformation between C2 and C5. The end result is 36 pairwise comparisons across 14 edges from Figure 4.1.

4.5.5 Results

Table 4.3 presents the results of the understandability analysis. A horizontal line separates the first three edges from the bottom 11. In E1 through E3, there is a statistically significant difference between the representations for at least one of the metrics considering $\alpha = 0.05$. These represent the strongest evidence for suggesting the directions of refactoring based on the understandability metrics we defined. Specifically, $\overrightarrow{T4T1}$, $\overrightarrow{D2D3}$, and $\overrightarrow{L2L3}$ are likely to improve understandability.

We note here that participants were able to select *unsure* when they were not sure if a string would be matched by a pattern (Figure 4.3). From a comprehension perspective, this indicates some level of confusion and is worth exploring.

For each pattern, we counted the number of responses containing at least one unsure, representing confusion. We then grouped the patterns into their representation nodes and computed an average of unsures per pattern. A higher number may indicate difficulty in comprehending a pattern from that node. Overall, the highest number of unsure responses came from T4 and T2, which present octal and hex representations of characters. The least number of unsure responses were in L3 and D3. These results also corroborate the refactorings suggested by the understandability analysis for the LIT group (i.e., $\overrightarrow{T4T1}$), the DBB group (i.e., $\overrightarrow{D2D3}$), and the LWB group (i.e., $\overrightarrow{L2L3}$) because the more understandable node has the least unsures of its group.

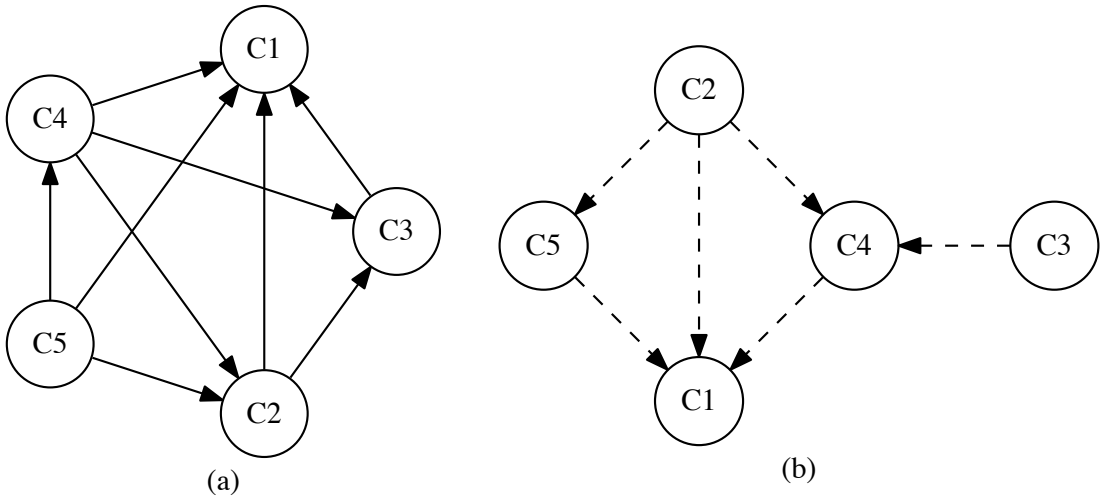

Figure 4.4 Trend graphs for the CCC equivalence graph: (a) represent the artifact analysis, (b) represent the understandability analysis.

4.6 Desirable Representations (RQ3)

To determine the overall trends in the data, we created total orderings on the representation nodes in each equivalence class (Figure 4.1) with respect to the community standards (RQ1) and understandability (RQ2) metrics.

4.6.1 Analysis

At a high level, these total orderings were achieved by building directed graphs with the representations as nodes and edge directions determined by the metrics: patterns and projects for community standards and matching and composition for understandability. Then, within each graph, we performed a topological sort to obtain total node orderings.

The graphs for community support are based on Table 4.1 and the graphs for understandability are based on Table 4.3. The following sections describe the processes for building and and topologically sorting the graphs.

4.6.1.1 Building the Graphs

In the community standards graph, we represent a directed edge $\overrightarrow{C2C1}$ when nPatterns(C1) $>$ nPatterns(C2) *and* nProjects(C1) $>$ nProjects(C2). When there is a conflict between nPat-

terns and nProjects, as is the case between L2 and L3 where L2 is found in more patterns and L3 is found in more projects, an undirected edge $\overline{L2L3}$ is used. This represents that there was no winner based on the two metrics. After considering all pairs of nodes in each equivalence class that also have an edge in Figure 4.1, we have created a graph, for example Figure 4.4a, that represents the frequency trends among the community artifacts.

In the understandability graph, we represent a directed edge $\overrightarrow{C2C1}$ when match(C1) > match(C2) *and* compose(C1) > compose(C2). When there is a conflict between match and compose, as is the case with T1 and T3 where match(T1) is higher but compose(T3) is higher, an undirected edge $\overline{T1T3}$ is used. When one metric has a tie, as is the case with composition in E9, we resort to the matching metric to determine $\overrightarrow{C5C1}$. An example understandability graph for the CCC is shown in Figure 4.4b.

4.6.1.2 Topological Sorting

Once the graphs are built for each equivalence class and each set of metrics, community standards and understandability, we apply a modified version of Kahn's topological sorting algorithm to obtain a total ordering on the nodes, as shown in Algorithm 1. The first modification is to remove all undirected edges since Kahn's operates over a directed graph.

In Kahn's algorithm, all nodes without incoming edges are added to a set S (Line 5), which represents the order in which nodes are explored in the graph. For each n node in S (Line 6), all edges from n are removed and n is added to the topologically sorted list L (Line 8). If there exists a node m that has no incoming edges, it is added to S. In the end, L is a topologically sorted list.

One downside to Kahn's algorithm is that the total ordering is not unique. Thus, we mark ties in order to identify when a tiebreaker is needed to enforce a total ordering on the nodes. For example, on the understandability graph in Figure 4.4b, there is a tie between C3 and C2 since both have no incoming edges, so they are marked as a tie on Line 5. Further, when $n = C2$ on line 7, both C5 and C4 are added to S on Line 12, thus the tie between them is marked on line 15. In these cases, a tiebreaker is needed.

Algorithm 1 Modified Topological Sort

```
L ← []
S ← []
Remove all undirected edges (creates a DAG)
Add all disconnected nodes to L and remove from graph. If there is more than one, mark the tie.
Add all nodes with no incoming edges to S. If there is more than one, mark the tie.
while S is non-empty do
    remove a node n from S
    add n to L
    for node m such that e is an edge n⃗m⃗ do
        remove e
        if m has no incoming edges then
            add m to S
        end if
    end for
    If multiple nodes were added to S in this iteration, mark the tie
    remove n from graph
end while
For all ties in L, use a tiebreaker.
```

Breaking ties on the community standards graph involves choosing the representation that appears in a larger number of projects, as it is more widespread across the community.

Breaking ties in the understandability graph uses the metrics. Based on Table 4.3, we compute the average matching score for all instances of each representation, and do the same for the composition score. For example, C4 appears in E5, E12 and E13 with an overall average matching score of 0.81 and composition score of 24.3. C5 appears in E4 and E9 with an average matching of 0.87 and composition of 28.28. Thus, C5 is favored to C4 and appears higher in the sorting.

4.6.2 Results

After running the topological sort in Algorithm 1 with tiebreakers, we have a total ordering on nodes for each graph, shown in Table 4.4. For example, given the graphs in Figure 4.4a and Figure 4.4b, the topological sorts are `C1 C3 C2 C4 C5` and `C1 C5 C4 C2 C3`, respectively.

There is a clear winner in each equivalence class, with the exception of DBB. That is, the node sorted highest in the topological sorts for both the community standards and understand-

Table 4.4 Topological Sorting, with the left-most position being highest

	CCC	DBB	LBW	SNG	LIT
Community Standards	C1 C3 C2 C4 C5	D2 D1 D3	L3 L2 L1	S2 S1 S3	T1 T3 T2 T4
Understandability	C1 C5 C4 C2 C3	D3 D1 D2	L3 L2	S2 S1	T1 T2 T4 T3

ability analyses are C1 for CCC, L3 for LBW, S2 for SNG, and T1 for LIT. After the top rank, it is not clear who the second place winner is in any of the classes, however, having a consistent and clear winner is evidence of a preference with respect to community standards and understandability, and thus provides guidance for potential refactorings.

This positive result, that the most popular representation in the corpus is also the most understandable, makes sense as people may be more likely to understand things that are familiar or well documented. However, while L3 is the winner for the LBW group, we note that L2 appears in slightly more patterns.

DBB is different as the orderings are completely reversed depending on the analysis, so the community standards favor D2 and understandability favors D3. Further study is needed on this, as well as on LBW and SNG since not all nodes were considered in the understandability analysis.

4.7 Discussion

Based on our analyses of source code and our empirical study on the understandability of regex representations, we have identified preferred regex representations that may make regexes easier to understand and thus maintain. In this section, we describe the implications of these results.

4.7.1 Interpreting Results

In the CCC equivalence class, C1 (e.g., `[0-9a]`) is more commonly found in the patterns and projects. Representations C2 (e.g., `[0123456789a]`) and C3 (e.g., `[^\x00-/:-‘b-\x7F]`) appear in similar percentages of patterns and projects but there is no significant difference in understandability considering two pairs of regexes tested as part of E13 (Table 4.3). However,

a small preference is shown for C1 over C2 (E7), leading this to to be the winner of both the community support and understandability analyses. Regex length is probably important for understandability, though we did not test for this.

In the DBB group, D3 (e.g., `pBs|pBBs|pBBBs`) merits further exploration because it is the most understandable but least common node in DBB group. This may be because explicitly listing the possibilities with an OR is easy to grasp, but if the number of items in the OR is too large, the understandability may go down. Further analysis is needed to determine the optimal thresholds for representing a regex as D3 compared to D1 (e.g., `pB{1,3}s`) or D2 (e.g., `pBB?B?s`).

In the SNG group, S1 is a compact representation (e.g., `S{3}`), but S2 was preferred (e.g., `SSS`). Similar to the DBB group, this may be do to the particular examples chosen in the analysis, as a large number of explicit repetitions may not be as preferred.

In the LWB group, L1 (e.g., `A{2,}`) is rare, appearing in $< 1\%$ of the patterns. Representations L2 (e.g., `AAA*`) and L3 (e.g., `AA+`) appear in similar numbers of patterns and projects, but there is a significant difference in their understandability, favoring L3.

In the LIT group, T1 (e.g., `\a\$>`) is the typical way to list literals, but the reason to use hex (T2) or oct (T4) types is because some characters cannot be represented any other way, like invisible chars. One main result of our work is that T4 (e.g., `\007\036\062`) is less understandable than T2 (e.g., `\x07\x24\x3E`), so if invisible chars are required, hex is the more understandable representation. Regarding T3 (e.g., `\a[$]>`), initially we thought the square brackets would be more understandable than using an escape character, but we found the opposite. Given a choice between T1 and T3, the escape character was more understandable.

4.7.2 Opportunities For Future Work

There are several directions for future work related to regex study and refactoring.

Equivalence Class Models We looked at five equivalence classes, each with three to five nodes. Future work could consider richer models with more or different classes and nodes. Additional equivalence groups to consider may include:

Multi line option `(?m)G\n` ≡ `(?m)G$`

Case insensitive `(?i)[a-z]` ≡ `[A-Za-z]`

Backreferences `(X)q\1` ≡ `(?P<name>X)q\g<name>`

It might also be the case that there exist critical comprehension differences within a representation. For example, between C1 (e.g., `[0-9a]`) and C4 (e.g., `[\da]`), it could be the case that `[0-9]` is preferred to `[\d]`, but `[A-Za-z0-9_]` is not be preferred to `[\w]`). By creating a more granular model of equivalence classes, and making sure to carefully evaluate alternative representations of the most frequently used specific patterns, additional useful refactorings could be identified.

Regex Migration Libraries We have identified opportunities to improve the understandability of regexes in existing code bases by looking for some of the less understandable regex representations, which can be thought of as antipatterns, and refactoring to the more common or understandable representations. Building migration libraries is a promising direction of future work to ease the manual burden of this process, similar in spirit to prior work on class library migration [**?**].

Regex Programming Standards Many organizations enforce coding standards in their repositories to ease understandability. Presently, we are not aware of coding standards for regular expressions, but this work suggests that enforcing standard representations for various regex constructs could ease comprehension.

Regex Refactoring for Performance The representation of regexes may have a strong impact on the runtime performance of a chosen regex engine. Prior work has sought to expedite the processing of regexes over large bodies of text [**?**]. Refactoring regexes for performance would complement those efforts.

4.7.3 Threats to Validity

Internal We measure understandability of regexes using two metrics, matching and composition. However, these measures may not reflect actual understanding of the regex behavior.

For this reason, we chose to use two metrics and present the analysis in the context of reading and writing regexes, but the threat remains.

Participants evaluated regular expressions during tasks on MTurk, which may not be representative enough of the context in which programmers would encounter regexes in practive. Further study is needed to determine the impact of the experimentation context on the results.

Some regex representations from the equivalence classes were not involved in the understandability analysis and that may have biased the results against those nodes. Repetition of the analysis with more compete coverage of the edges in the equivalence classes is needed.

We treated unsure responses as omissions that did not count against the matching scores. Thus, if a participant answered two strings correctly and marked the other three strings as unsure, then this was 2/2 correct, not 2/5. This may have inflated the matching scores, however, less than 5% of the matching scores were impacted by such responses.

External Participants in our survey came from MTurk, which may not be representative of people who read and write regexes on a regular basis.

The regexes used in the evaluation were inspired by those found in Python code, which is just one language that has library support for regexes. Thus, we may have missed opportunities for other refactorings based on how programmers use regexes in other programming languages.

The results of the understandability analysis may be closely tied to the particular regexes chosen for the experiment. For many of the representations, we had several comparisons. Still, replication with more regex patterns is needed.

4.8 Related Work

Regular expression understandability has not been studied directly, though prior work has suggested that regexes are hard to read and understand since there are tens of thousands of bug reports related to regular expressions [**?**]. To aid in regex creation and understanding, tools have been developed to support more robust creation [**?**] or to allow visual debugging [**?**]. 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 [**?** , **?**].

Regular expression refactoring has also not been studied directly, though refactoring literature abounds [? , ? , ?]. The closest to regex refactoring comes from research toward expediting the processing of regular expressions on large bodies of text [?], which could be thought of as refactoring for performance.

Code smells in object-oriented languages were introduced by Fowler [?]. Researchers have studied the impact of code smells on program comprehension [? , ?], finding that the more smells in the code, the harder the comprehension. This is similar to our work, except we aim to identify which regex representations can be considered smelly. Code smells have been extended to other language paradigms including end-user programming languages [? , ? , ? , ?]. The code smells identified in this work are representations that are not common or not well understood by developers. This concept of using community standards to define smells has been used in other refactoring literature for end-user programmers [? , ?].

Exploring language feature usage by mining source code has been studied extensively for Smalltalk [?], JavaScript [?], and Java [? , ? , ? , ?], and more specifically, Java generics [?] and Java reflection [?]. Our prior work ([?], under review) was the first to mine and evaluate regular expression usages from existing software repositories. The intention of the prior work [?] was to explore regex language features usage and surveyed developers about regex usage. In this work, we define potential refactorings and use the mined corpus to find support for the presence of various regex representations in the wild. Beyond that, we measure regex understandability and suggest canonical representations for regexes to enhance conformance to community standards and understandability.

4.9 Conclusion

In an effort to find refactorings that improve the understandability of regexes, we created five equivalence class models and used these models to investigate the most common representations and most comprehensible representations per class. We found the most common representations per class by both number of patterns and number of projects to be C1, D2, T1 and S2 (L3 has the most patterns, L2 has the most projects). We also identified three strongly preferred transformations between representations (i.e., $\overrightarrow{T4T1}$, $\overrightarrow{D2D3}$, and $\overrightarrow{L2L3}$) according to the results

of our comprehension tests. We combined the results of these two investigations using a version of Kahn's topological sorting algorithm to produce a total ordering of representations within each model. The agreement between Community Standards and Understandability in this analysis validates our results and suggests that indeed one particular representation can be preferred over others in most cases. We can also recommend using hex to represent invisible characters in regexes instead of octal, and to escape special characters with slashes instead of wrapping them in brackets to avoid escaping them. Further research is needed into more granular models that treat common specific cases separately, and that address the effect of length on readability when transforming from one representation to another.

Acknowledgements

This work is supported in part by NSF SHF-EAGER-1446932.

CHAPTER 5. SUMMARY AND DISCUSSION

This is the opening paragraph to my thesis which explains in general terms the concepts and hypothesis which will be used in my thesis.

With more general information given here than really necessary.

5.1 Introduction

Here initial concepts and conditions are explained and several hypothesis are mentioned in brief.

Or graphically as seen in Figure 5.1 it is certain that my hypothesis is true.

5.1.1 Hypothesis

Here one particular hypothesis is explained in depth and is examined in the light of current literature.

As can be seen in Table 5.1 it is truly obvious what I am saying is true.

5.1.1.1 Parts of the hypothesis

Here one particular part of the hypothesis that is currently being explained is examined and particular elements of that part are given careful scrutiny.

5.1.2 Second Hypothesis

Here one particular hypothesis is explained in depth and is examined in the light of current literature.

Table 5.1 This table shows almost nothing but is a sideways table and takes up a whole page by itself

Element	**Control**	**Experimental**
Moon Rings	1.23	3.38
Moon Tides	2.26	3.12
Moon Walk	3.33	9.29

5.1.2.1 Parts of the second hypothesis

Here one particular part of the hypothesis that is currently being explained is examined and particular elements of that part are given careful scrutiny.

5.2 Criteria Review

Here certain criteria are explained thus eventually leading to a foregone conclusion.

Figure 5.1 Durham Centre— Another View

APPENDIX A. Patterns in Python projects from Github

This is now the same as any other chapter except that all sectioning levels below the chapter level must begin with the *-form of a sectioning command.

top 100 clusters used

top 10 by feature group

13,579 patterns: the corpus(1510 per page=9 pages)

Supplemental material.

APPENDIX B. Developer Survey

This is now the same as any other chapter except that all sectioning levels below the chapter level must begin with the *-form of a sectioning command.

Survey Questions

Survey Responses

Survey Statistics

More stuff.

BIBLIOGRAPHY