Refactoring Regular Expressions

Carl Chapman
Department of Computer Science
Iowa State University
carl1976@iastate.edu

Kathryn T. Stolee
Department of Computer Science
North Carolina State University
ktstolee@ncsu.edu

ABSTRACT

Regexes are hard to understand. Let me tell you how.

1. INTRODUCTION

Regular expressions are used frequently by developers for many purposes, such as parsing files, validating user input, or querying a database. 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., `'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 the opinions of 180 study participants. Our results identify canonical representations for four of the five equivalence classes based on mutual agreement between community standards and understandability. 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 canonical regex representations 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 the related work in regexes (Section 6), equivalence classes and possible refactorings (Section 2), the study of regex representations in Python projects (Section 3.1), and the regex understandability study using human participants (Section 3.2). We discuss results in Section 4, implications in Section 5, and conclude in Section 7.

[1] https://regex101.com/

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

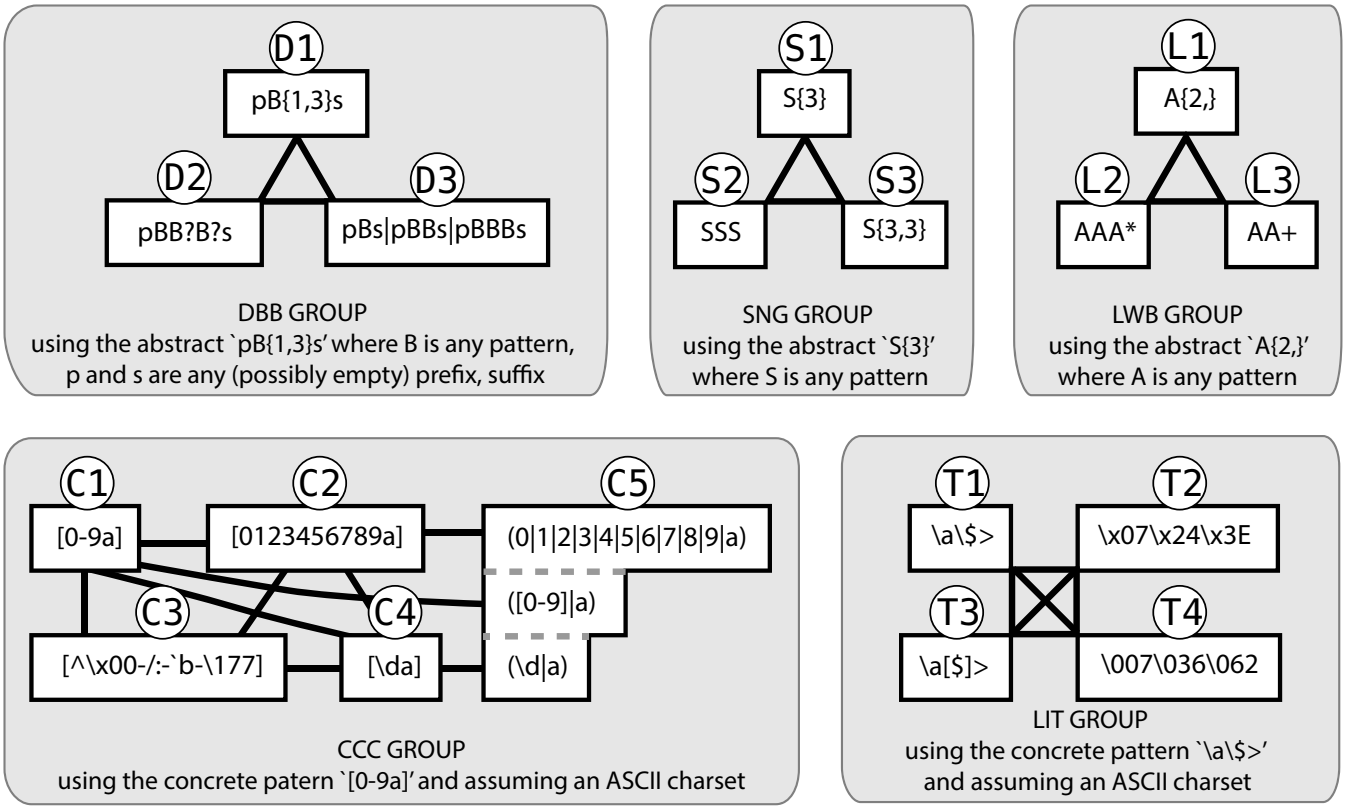

Figure 1: Equivalence classes with various representations of semantically equivalent refactorings within each class. DBB = Double Bounded, SNG = Single Bounded, LWB = Lower Bounded, CCC = Custom Character Class and LIT = Literal

2. REFACTORINGS

After studying over 13,000 distinct regex strings from nearly 4,000 Python projects[2], 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 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 4.

We use concrete regexes in the representations to more clearly illustrate examples of the representations. In reality, the `A`'s in the LWB group could be any character or character class that has a lower bound on repetitions. We chose the lower bound repetition threshold of 2 for illustration; in practice this could be any number, including zero. **TODO.NOW: is this true with how we counted?** Next, we describe each group, the representations, and possible transformations in detail:

[2]same dataset used in prior work [**?**]

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) belongs to the C1 node. **TODO.NOW: would [0-9] be part of C1? It's semantically identical to "slash-d" but is also wrapped in a custom class.**

C2: Any custom character class that does not contain any shorthand representations, specifically ranges or defaults, but expresses all characters explicitly.

C3: Any character classes expressed using negation, which is indicated by a caret (i.e., `^`) followed by some other pattern. For example the pattern `[^ao]` matches every character *except* `a` or `o`. Note that any non-negative character class can be represented as a negative char-

acter class if the set of characters is known by negating the set of characters not found in the non-negative character class, and visa versa. For example, matching all lower-case ... Katie is here.

C4: Any pattern using some default character class such as `\d` or `\W` within a character class belongs to the C4 node. Note that a pattern can belong to both C1 and C4, like the example `[a-f\d]`. 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 (or visa-versa). This transformed version would only belong to the C1 node.

C5: contains all custom character classes expressed as an OR of length-one sequences, including defaults or other CCCs. For example the character class `[abc]` is equivalent to the OR `(a|b|c)`.

DBB Group.

The double-bounded group contains all regex patterns that use some repetition defined by a lower and upper boundary. For example the pattern `l(ol){1,3}` represents an 'l' followed by one to three sequential 'ol' patterns. This will match 'lol', 'lolol' and 'lololol'. Because `l(ol){1,3}` uses the curly brace repetition with a lower and upper bound, it belongs to the D1 node.

Note that `l(ol){1,3}` can become `lol(ol){0,2}` by pulling the lower bound out of the curly braces and into the explicit sequence (or visa versa). The same functional pattern can be represented as `lol(ol)?(ol)?`, because the questionable (QST) modifier is used. Note how in general, this proceedure is simply pulling out N QST groups from a curly brace style repetition with a zero lower bound and an upper bound of N. One question mark is equivalent to the curly brace style with a lower bound of 0, and upper bound of 1, so `X?` is equivalent to `X{0,1}`, so we can express `X{0,2}` as `X?X?`. Any regex using the QST modifier belongs to the D2 node.

Whenever a repetition has a lower and upper boundary, it is possible to explode the pattern into an OR. So using the same example, `l(ol){1,3}` would become `lol|lolol|lololol`, and this pattern belongs to the D3 node. Again note that a pattern can belong to multiple nodes in the DBB group, as in the example pattern `(a|aa)X?Y{2,4}`

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 regex languages, the user 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. Any pattern using hex tokens belongs to the T2 node, and any pattern using octal tokens belongs to the T4 node. (note we should probably mention unicode somewhere in here) It is completely valid to wrap any literal character in brackets to form a custom character class of size one, like `[x][y][z]`, and this style is used most often to avoid using a backslash for a special character that requires escaping because of its significance in the regex language, like `[{]` which must otherwise be escaped like `\{`. Any literal wrapped in this way belongs to T3. Patterns that do not use any wrapped characters, octal or hex characters and also do use some literal character belong to the T1 node.

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

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. Similarly, 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.

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`. The representation in *S1*, `S{3}`, states that S appears exactly three times in sequence. This can alternately be expressed with a double-bound where the upper and lower bounds are the same, as in *S3*. Removing the repetition symbols in either *S1* or *S3* yields *S2* which is represented explicitly as `SSS`.

Using an example from a Python project, the regex `'[^ ]*\.[A-Z]{3}'` is a member of *S1* and could be refactored to *S3* as `'[^ ]*\.[A-Z]{3,3}'` or to *S2* as `'[^ ]*\.[A-Z][A-Z][A-Z]'`, depending on programmer preferences.

3. STUDY

After defining the equivalence classes and potential regex refactorings as described in Section 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 and understandable. As prior work has shown that regexes are difficult to read [], we seek to define refactorings toward understandability.

We define understandability three ways. First, assuming that common programming practices are more understandable than uncommon practices, we explore the frequencies of each representation from Figure 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 overall research questions are:

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

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

RQ3: What is the overlap in the refactoring suggested by RQ1, RQ2, and RQ3?

3.1 Community Study (RQ1)

To determine how common each of the 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.

3.1.1 Artifacts

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). The remaining 80.8% (43,525) of the utilizations were collapsed into 13,711 distinct pattern strings. Each of the pattern strings was preprocessed 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. After removing all problematic patterns as described, 13,597 distinct patterns remained to be used in this study.

3.1.2 Metrics

We measure community support using the frequency of regex patterns that match each of the representations (nodes) in Figure 1. We also measure the proportion of sampled projects to contain at least one pattern belonging to a node. To determine how often each representation appears in the wild, we extract regex patterns from source code and measure if a representation matches (part of) the pattern.

[3]https://github.com/bkiers/pcre-parser

1.

What is your gender?	n	%
Male	149	83%
Female	27	15%
Prefer not to say	4	2%

2. **What is your age?**
$\mu = 31$, $\sigma = 9.3$

3.

Education Level?	n	%
High School	5	3%
Some college, no degree	46	26%
Associates degree	14	8%
Bachelors degree	78	43%
Graduate degree	37	21%

4.

Familiarity with regexes?	n	%
Not familiar at all	5	3%
Somewhat not familiar	16	9%
Not sure	2	1%
Somewhat familiar	121	67%
Very familiar	36	20%

5. **How many regexes do you compose each year?**
$\mu = 67$, $\sigma = 173$

6. **How many regexes (not written by you) do you read each year?**
$\mu = 116$, $\sigma = 275$

Figure 2: Participant Profiles, $n = 180$

function pattern flags

```
r1 = re.compile('(0|-?[1-9][0-9]*)$', re.MULTILINE)
```

Figure 3: Example of one regex utilization

Patterns.

A regex utilization is one single invocation of a regex library. Figure 3 presents an example of one regex utilization from Python, the language used in our artifact analysis (Section 3.1), with key components labeled. The *function* called is `re.compile`. The *pattern* used to define what strings this utilization will match is `(0|-?[1-9][0-9]*)$`. The *flag* `re.MULTILINE` modifies the rules used by the regex engine when matching. When executed, this utilization will compile a regex object in the variable `r1` from the pattern `(0|-?[1-9][0-9]*)$`, with the `$` token matching at the end of each line because of the `re.MULTILINE` flag.

A *pattern* is extracted from a utilization, as shown in Figure 3. In essence, it is a string, but more formally it is an ordered series of regular expression language feature tokens. The pattern in Figure 3 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 `-`).

Projects.

The process for deciding if a particular pattern belongs to a particular node is described in detail in Section 3.1.3. For this frequency analysis, we focus on patterns and the number of projects the patterns appear in.

3.1.3 Analysis

TODO.NOW: Katie, please look at this section about how patterns are chosen into nodes, editing and making notes if more info is needed. For the nodes that only require a particular feature to be present, 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 DBB repetition, D2 requires QST repetition, S1 requires SNG repetition, L1 requires LWB repetition, L2 requires KLE repetition, L3 requires ADD repetition, and C3 requires the negated custom character class NCCC.

For some nodes, the presence of a feature is not enough to determine membership, but the presence of a feature and a simple regex executed directly on the pattern can decide membership. The nodes decided in this way are as follows: D3 requires an OR and a match of the regex `"(?<=[|(]|^)([^ \\]+)\|\1\1"` which captures some (non-space or slash) characters at the beginning of a group, within an OR, or at the beginning of a pattern, then sees an OR-bar and then sees the captured characters repeated twice.

This technique was developed by trial and error and then the results were verified by hand looking both at the set of patterns identified by the regex for false positives, and also at the set of rejected patterns for false negatives. No false results in either set were found (TODO-verify this again later?).

S3 requires both the DBB feature and a match of the regex `\{(\d+),\1\}` which identifies curly brace repetition with identical lower and upper bounds. These results were also hand-verified with no false results.

T2 requires a LIT feature and must match the regex `(\\x[a-f0-9A-F]{2})` which reliably identifies hex codes within a pattern. Similarly T4 requires a LIT 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. One false positive was identified which was actually the lower end of a hex range using the literal `\0`.

T3 requires that a character is wrapped in a CCC, so the pattern `(?<=\[).(?=\])` was used to find characters wrapped in brackets. T1 requires that no characters are wrapped in brackets or are hex or octal characters, so the regex `(\\x[a-f0-9A-F]{2})|((\\0\d*)|(\\\d{3}))` was used to find any hex or octal codes. Any pattern that matched either of these regexes was not accepted as belonging to node T1, otherwise all nodes containing any LIT feature were accepted into T1.

For the remaining nodes, the sequence of tokens produced by the PCRE parser was transformed into a bullet-delimited string so that a regex could be executed on the token sequence itself, avoiding some pitfalls with special characters and pattern nesting. S2 requires any element to be repeated at least once, so the regex `(ELEMENT•[^ ]+)\1` was used to find a repeated element. Node C1 requires that a character class contains a range, and so a regex on the token sequence was prepared that recognizes that situation: `CHARACTER_CLASS•DOWN•(((.|\\0x..)|(\\d|\\D|\\s|\\S|\\w|\\W))•)*(RANGE•DOWN•(.|\\0x..)•(.|\\0x..)•UP•)((.|\\0x..)|(\\d|` This regex works by recognizing a character class using `CHARACTER_CLASS•DOWN•`, then allowing zero or more single literal characters or hex literals or default character classes before seeing the `RANGE•DOWN•(.|\\0x..)•(.|\\0x..)` pattern which specifies the lower and upper boundaries of a range. Then zero or more additional single chars, default char classes or ranges can be seen before the last `UP•` ends the `CHARACTER_CLASS` element.

C2 requires that there are no character classes using ranges or defaults (all characters are explicit so no shorthand is allowed), so a regex was composed that recognizes any character class with a range or a default present. If a pattern has the character class feature, but does not match this regex, then it belongs to the C2 node. The regex is: `CHARACTER_CLASS•DOWN•(((\\0x..)|.)•)*((RANGE•DOWN•((\\0x..)|.)•` and works in similar way to the regex to identify C1 nodes: first it requires the `CHARACTER_CLASS•DOWN•` sequence, then it allows zero or more literals, then requires one default or range, then allows zero or more literals, defaults or ranges before ending the `CHARACTER_CLASS` element. C4 requires the presence of a default character class, so using this regex: `"CHARACTER_CLASS•DOWN•(((.|\\0x..)|(RANGE•DOWN•(.|\\0x..)•(.|\\` we again recognize the start of the character class with `CHARACTER_CLASS•DOWN•`, then allow zero or more literals or ranges, then requiring one default, and finally allowing zero or more literals, defaults or ranges before ending the `CHARACTER_CLASS` element. C5 requires an OR of length-one sequences, and so using this regex: `OR•DOWN•(ALTERNATIVE•DOWN•ELEMENT•` we recognize the start of an OR, then allow one or more literals, defaults or ranges before the OR is completed. This addressed *RQ1*.

3.2 Understandability Study (RQ2)

To gauge the understandability of regexes, we designed and implemented a study on Amazon's Mechanical Turk with 180 participants to collect the matching and composition metrics.

3.2.1 Platform

Amazon's Mechanical Turk (MTurk) is a crowdsourcing platform in which requestors 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.

3.2.2 Metrics

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

Matching.

Given a regex and a set of strings, a participant determines which strings will be matched by the regex. The percentage of correct responses is the matching score. For example, consider regex `'RR*'` and five strings, which comes from our study, shown in Table 1, and the responses from two participants in the *Response 1* and *Response 2* columns. The oracle has the first three strings matching since they each contain at least one `R` character. *Response 1* answers correctly for the first three strings but incorrectly thinks the fourth string matches, so the matching score is $4/5 = 0.80$. *Response 2* misses the second string, so they also scored $4/5 = 0.80$.

Composition.

Given a regex, a participant composes a string that it matches. If the participant is accurate and the string indeed is matched by the regex, then a composition score of

Table 1: Matching metric example

String	'RR*'	Oracle	Response 1	Response 2	Res
1	"ARROW"	✓	✓	✓	
2	"qRs"	✓	✓	×	
3	"R0R"	✓	✓	✓	
4	"qrs"	×	✓	×	
5	"98"	×	×	×	
	Score	1.00	0.80	0.80	

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

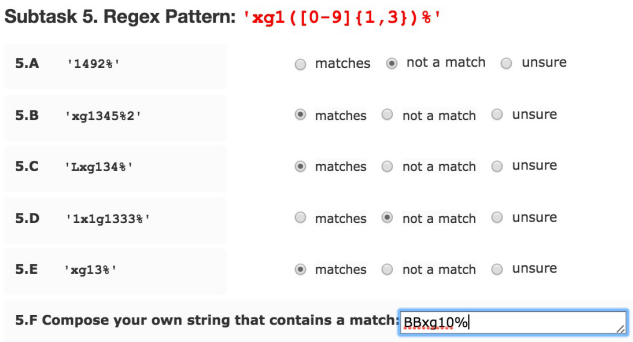

Figure 4: Example of one HIT Question

1 is assigned, otherwise 0. For example, given the regex `'(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.

Each pattern was compiled using the *java.util.regex* library. Composed strings were grouped by the pattern participants were attempting to match, and a Matcher `m` 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.

3.2.3 Design

TODO.NOW: needs to be updated with respect to no C1,T1 nodes Using the regexes in the corpus as a guide, we created 60 regex patterns that were grouped into 26 equivalence groups, where 18 groups had two equivalent regexes each and eight groups had three equivalent regexes each. For example, one of the groups of size two had regexes, `([0-9]+)\.([0-9]+)'` and `(\d+)\.(\d+)'`. One of the groups of size thee contained `((q4f)?ab)'`, `(q4fab|ab)'`, and `((q4f){0,1}ab)'`.

For each of the 26 groups of regexes, created five strings, **TODO.NOW: how many matched and how many did not match?**. These were used for computing the composition metric.

Once all the regexes and matching strings were collected, we created tasks for the MTurk participants as follows: randomly select a regex from ten of the 26 groups. Randomize the order of the regexes, as well as the order of the matching strings for each regex. After adding a question asking the participant to compose a string that the regex 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 HITs.

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

Workers were paid $3.00 for successfully completing a HIT, and were clearly instructed to complete only one HIT. The average completion time for accepted HITs was 682 seconds (11 mins, 22 secs). A total of 241 HITs were submitted - of those 55 were rejected, and 6 duplicates were ignored, always using the first accepted submission so as to obtain a value for each of the 180 distinct tasks. Of the 55 rejected HITs, 48 were rushed through by one person leaving many answers blank and spending an average of 454 seconds per hit, 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.

3.2.4 Participants

In total, there were 180 different 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. Fittingly, participants read more regexes than they write with an average of 116 and a range from 0 to 2000. Figure 2 summarizes the self-reported participant characteristics from the qualification survey.

3.2.5 Analysis

For each of the 60 regexes, an average matching score was computed using the metrics in Table 1. The average composition metric was measured using the process described in Section **??**. This addresses *RQ2* and *RQ3*.

TODO.NOW: How to deal with unsure responses? How many were there? Carl's analysis goes here.

4. RESULTS

4.1 RQ1: Community Support

Figure 2 presents the frequencies with which each representation appears in a regex pattern and in a project scraped from GitHub. Recall that the patters 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, representation C1 matches when a custom character class uses ranges and it appears in 2,479 (18.2%) of all the patterns but 810 (52.5%) of the projects. Representation D1 appears in 367 (2.7%) of the patterns but only 242 (15.7%) of the projects. In contrast, representation T3 appears in 60 *fewer* patterns but 26 *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 $C2 \Rightarrow C1$ refactoring. Table **??** presents these

Table 2: How frequently is each alternative expression style used?

name	description	example	nPatterns	% patterns	nProjects	% projects
C1	char class using ranges	`'^[1-9][0-9]*$'`	2,479	18.2%	810	52.5%
C2	char class explicitly listing all chars	`'[aeiouy]'`	1,283	9.4%	551	35.7%
C3	any negated char class	`'[^A-Za-z0-9.]+'`	1,935	14.2%	776	50.3%
C4	char class using defaults	`'[-+\d.]'`	840	6.2%	414	26.8%
C5	an OR of length-one sub-patterns	`'(@|<|>|-|!)'`	245	1.8%	239	15.5%
D1	curly brace repetition like {M,N} with M<N	`'^x{1,4}$'`	367	2.7%	242	15.7%
D2	zero-or-one repetition using question mark	`'^http(s)?://'`	1,871	13.8%	646	41.8%
D3	repetition expressed using an OR	`'^(Q|QQ)\<(.+)\>$'`	10	.1%	27	1.7%
T1	no HEX, OCT or char-class-wrapped literals	`'get_tag'`	12,482	91.8%	1,485	96.2%
T2	has HEX literal like `\xF5`	`'[\x80-\xff]'`	479	3.5%	243	15.7%
T3	has char-class-wrapped literals like [$]	`'[$][{]\d+:([^}]+)[}]'`	307	2.3%	268	17.4%
T4	has OCT literal like `\0177`	`'[\041-\176]+:$'`	14	.1%	37	2.4%
L1	curly brace repetition like {M,}	`'(DN)[0-9]{4,}'`	91	.7%	166	10.8%
L2	zero-or-more repetition using kleene star	`'\s*(#.*)?$'`	6,017	44.3%	1,097	71.0%
L3	one-or-more repetition using plus	`'[A-Z][a-z]+'`	6,003	44.1%	1,207	78.2%
S1	curly brace repetition like {M}	`'^[a-f0-9]{40}$'`	581	4.3%	340	22.0%
S2	explicit sequential repetition	`'ff:ff:ff:ff:ff:ff'`	3,378	24.8%	861	55.8%
S3	curly brace repetition like {M,M}	`'U[\dA-F]{5,5}'`	27	.2%	32	2.1%

recommendations for each pair of representations within each equivalence class. The *Comm* column is populated based on the findings of *RQ1*. The findings for *RQ2* and *RQ3* are in the *Match* and *Compose* columns, respectively.

4.2 RQ2: Understandability

Table 3 shows average understandability. **TODO.NOW: add more info about understandability results**

4.3 RQ3: Overlap in Refactorings

To determine the overall trends in the data, we created total orderings on the representation nodes in each equivalence class (Figure 1) with respect to community standards (RQ1) and understandability (RQ2). 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 (10 in total), we performed a topological sort to get total node orderings.

The graphs for community support are based on Table 2 and the graphs for understandability are based on Table 3.

4.3.1 Building the Graphs

In the community standards graph, we represent a directed edge from $C2 \rightarrow C1$ when nPatterns(C1) > nPatterns(C2) *and* nProjects(C1) > nProjects(C2). When there is a conflict between nPatterns 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 is used. This is to represent that there was no clear winner based on the two metrics used in the community standards analysis. After considering all pairs of nodes in each equivalence class that also have an edge in Figure 1, we have created a graph, for example Figure 5a, that represents the frequency trends among the community artifacts. Note that with the CCC group, there is no edge between C3 and C5 because there is no straightforward refactoring between those representations, as discussed in Section 2.

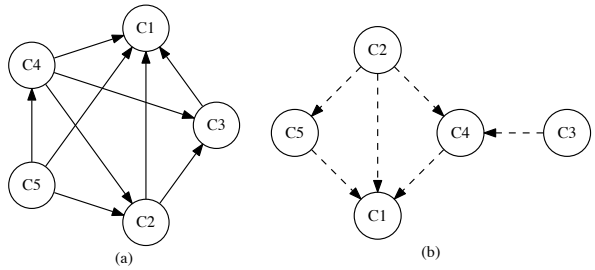

Figure 5: Trend graphs for the CCC equivalence graph. Solid lines represent the artifact analysis. Dashed lines represent the understandability analysis.

In the understandability graph, we represent a directed edge from $C2 \rightarrow C1$ 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 is used. When one metric has a tie, as is the case with compose(C1) = compose(C5), we resort to the matching to provide the direction, in this case, C5 → C1. An example understandability graph is provided in Figure 5b with the dashed arrows.

4.3.2 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

Table 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	$H_0 : \mu_{comp1} = \mu_{comp2}$
E1	T1 – T4	2	0.80	0.60	0.001	0.87	0.37	**<0.001**
E2	D2 – D3	2	0.78	0.87	**0.011**	0.88	0.97	0.085
E3	L2 – L3	3	0.86	0.91	**0.032**	0.91	0.98	0.052
E4	C2 – C5	4	0.85	0.86	0.602	0.88	0.95	**0.063**
E5	C2 – C4	1	0.83	0.92	**0.075**	0.60	0.67	0.601
E6	D1 – D2	2	0.84	0.78	0.120	0.93	0.88	0.347
E7	C1 – C2	2	0.94	0.90	0.121	0.93	0.90	0.514
E8	T2 – T4	2	0.84	0.81	0.498	0.65	0.52	0.141
E9	C1 – C5	2	0.94	0.90	0.287	0.93	0.93	1.000
E10	T1 – T3	3	0.88	0.86	0.320	0.72	0.76	0.613
E11	D1 – D3	2	0.84	0.87	0.349	0.93	0.97	0.408
E12	C1 – C4	6	0.87	0.84	0.352	0.86	0.83	0.465
E13	C3 – C4	2	0.61	0.67	0.593	0.75	0.82	0.379
E14	S1 – S2	3	0.85	0.86	0.776	0.88	0.90	0.638

sorting algorithm to obtain a total ordering on the nodes, as shown in Algorithm 1. 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.

Algorithm 1 Modified Topological Sort

1: $L \leftarrow []$
2: $S \leftarrow []$
3: Remove all undirected edges (creates a DAG)
4: Add all disconnected nodes to L and remove from graph. If there are more than one, mark the tie.
5: Add all nodes with no incoming edges to S. If there are more than one, mark the tie.
6: **while** S is non-empty **do**
7: remove a node n from S
8: add n to L
9: **for** node m such that e is an edge from $n \rightarrow m$ **do**
10: remove e
11: **if** m has no incoming edges **then**
12: add m to S
13: **end if**
14: **end for**
15: remove n from graph
16: If multiple nodes were added to S in this iteration, mark those as a tie
17: **end while**
18: For all ties in L, use a tiebreaker.

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 5b, there is a tie between C3 and C2 since both have no incoming edges, so they are marked as a tie on Line 4. Further, when $n = C2$ on line 7, both C5 and C4 are added to S on Line 12, thus the tie between them is parked on line 16. In these cases, a tiebreaker is needed.

Breaking ties on the community standards graph comes down to choosing the representation that appears in a larger number of projects, since it is more widespread across the community. Breaking ties in the understandability graph is trickier. Using Table 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 **TODO.LAST: C2 – C4**, **TODO.LAST: C1 – C4** and **TODO.LAST: C3 – C4** with an overall average matching score of 0.81 and composition score of 24.3. C5 appears in **TODO.LAST: C1 – C5** and **TODO.LAST: C2 – C5** with an average matching of 0.87 and composition of 28.28. Thus, C5 is favored to C4 and appears higher in the sorting.

After running the topological sort in Algorithm 1, we have a total ordering on nodes for each graph. After breaking ties as described, the topological sorts for all graphs are shown in Table 4. For example, given the graphs in Figure 5a and Figure 5b, the topological sorts are `C1 C3 C2 C4 C5` and `C1 C5 C4 C2 C3`, respectively.

4.3.3 Summary

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 understandability 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.

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.

CCC and DBB are shuffled quite differently, and LBW and SNG don't have enough information from the understandability analysis since there is just one edge. DBB is an odd one as the orderings are completely reversed depending on the analysis.

Table 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

5. DISCUSSION

5.1 Further Refactoring Opportunities

TODO.MID: discuss opportunities

5.2 Threats to Validity

5.2.1 *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.

TODO.MID: what about the threat of too few examples per node?

We treated unsure responses as omissions and did not count those against the participants. Thus, if a participant answered two strings correctly with match/not match, and marked the other three strings as unsure, then this was 2/2 correct, not 2/5.

5.2.2 *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 we used in the evaluation were inspired by those commonly 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.

Our community analysis only focuses on the Python language. 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.

6. RELATED WORK

TODO.MID: We are building on the survey that indicates regexes are hard to read, and the apparent lack of any regex readability refactoring attempts. Many papers have talked about refactoring, basically it is changing the form but not the behavior.

Prior work that surveyed developers about regex usage found that in a small software company, the 18 surveyed developers compose an average of 172 regexes per year. This is 48% higher than the number of regexes composed annually by MTurk participants in this work, which may be due to the nature of the jobs performed by the two populations.

7. CONCLUSION

Acknowledgements

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