# 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 8), equivalence classes and possible refactorings (Section 2), the study of regex representations in Python projects (Section 4), and the regex understandability study using human participants (Section 5). We discuss results in Section **??**, implications in Section 7, and conclude in Section 9.

[1] https://regex101.com/

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

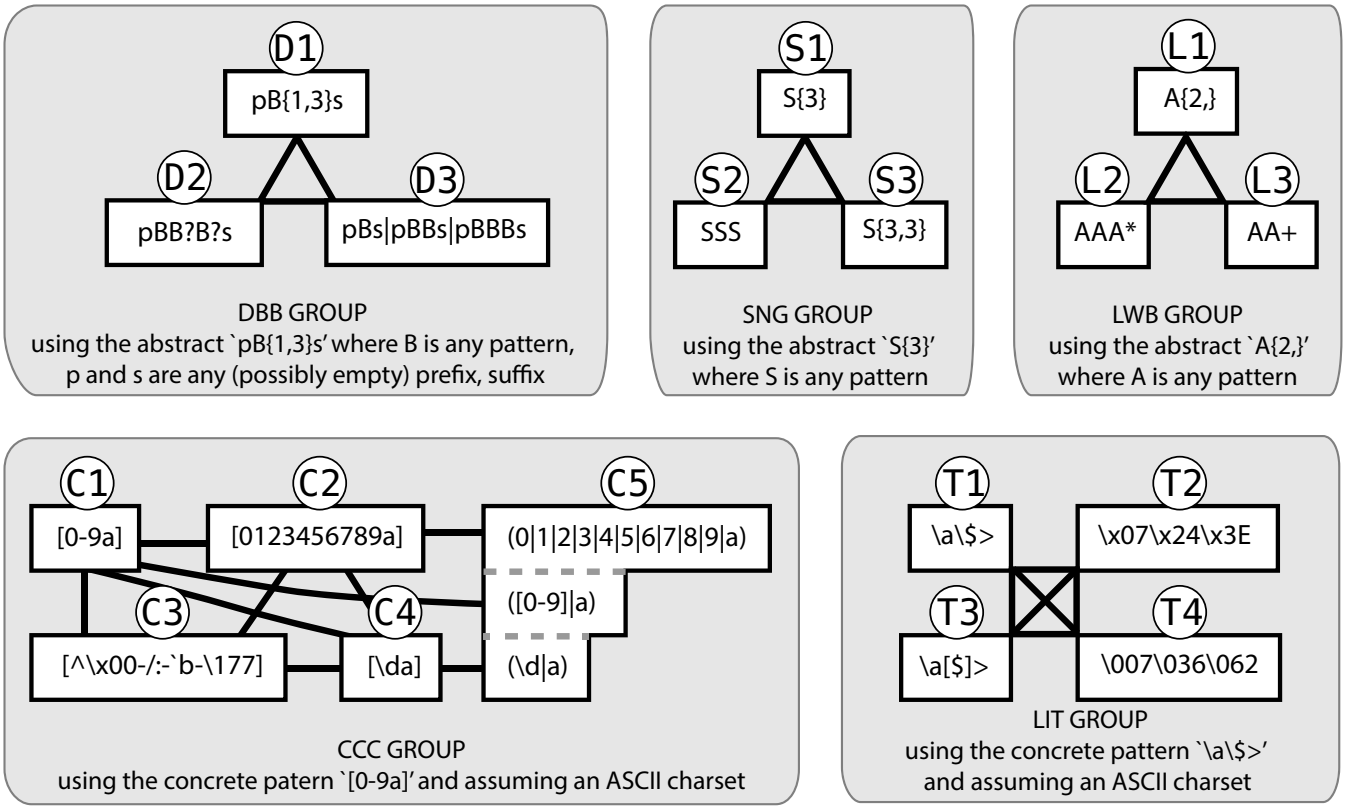


**Figure 1: Equivalence classes with various representations of semantically equivalent refactorings within each class. DBB = Double-Bounded, SNG = Single Bounded, LWB = Lower Bounded, CCC = Custom Character Class and LIT = Literal**

## 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 **??**.

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. 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.

**C2:** Any pattern that contains at least one 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 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 character class, and visa versa. **TODO.NOW: any exceptions to this?** For example, matching any lower-case

consonant in the English alphabet can be expressed as `[bcdfghjklmnpqrstvwxyz]` in C2 or with negation, `[^aeiou]` in C3.

**C4:** Any pattern using a 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, such as `[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:** 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. **TODO.NOW: Mention the lack of edge between C3 and C5**

**TODO.NOW: Is the example in Figure 1 of C3 also a member of C1 because it contains a range? Or does the carat trump this and throw it in C3 only?**

*DBB Group.*

The Double-Bounded (DBB) group contains all regex patterns that use some repetition defined by a 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. The challenge with identifying membership in this node is recognizing the opportunity to replace the ORs with double-boundaries, which we discuss in Section **??**. 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 wrapped characters (T3), octal (T4) or hex (T3) characters 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 instead of using a backslash escape, 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. **TODO.NOW: is this true?**

**TODO.NOW: mention unicode**

*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.

**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 zero repetitions is possible (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 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. **TODO.NOW: Is this really a member of L2?** 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]'`.

# 3. RESEARCH QUESTIONS

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 two 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 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?

# 4. COMMUNITY SUPPORT 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.

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

**Figure 2: Example of one regex utilization**

## 4.1 Artifacts

A regex utilization is one single invocation of a regex library. Figure 2 presents an example of one regex utilization from Python, the language used in our artifact analysis (Section 4), 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. The pattern in Figure 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 `-`).

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.,

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

**Table 1: How frequently is each alternative expression style used?**

| name | description | example | nPatterns | % patterns | nProjects | % projects |
|---|---|---|---|---|---|---|
| C1 | char class using ranges | `'^[1-9][0-9]*$'` | 2,479 | 18.2% | 810 | 52.5% |
| C2 | char class explicitly listing all chars | `'[aeiouy]'` | 1,903 | 14.0% | 715 | 46.3% |
| C3 | any negated char class | `'[^A-Za-z0-9.]+'` | 1,935 | 14.2% | 776 | 50.3% |
| C4 | char class using defaults | `'[-+\d.]'` | 840 | 6.2% | 414 | 26.8% |
| C5 | an OR of length-one sub-patterns | `'(@|<|>|-|!)'` | 245 | 1.8% | 239 | 15.5% |
| D1 | curly brace repetition like {M,N} with M<N | `'^x{1,4}$'` | 346 | 2.5% | 234 | 15.2% |
| 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% |

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 from 1,544 projects remained to be used in this study.

## 4.2 Metrics

We measure community support by matching each regex in the corpus to the representations (nodes) in Figure 1 and counting the number of *patterns* that contain the representation and the number of *projects* that contain the representation. A *pattern* is extracted from a utilization, as shown in Figure 2. 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 13,711 distinct regex patterns that represent 43,525 regex utilizations across the projects.

## 4.3 Analysis

To determine how many of the representations match regex 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[4]. Next, we describe the process in detail:

### 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.

### 4.3.2 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 and the presence of 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. **TODO.NOW: Say this with words and intuition, not complex regexes and algorithms**

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 character is wrapped in a custom character class. **TODO.NOW: always of size 1? Is a member of T3 also 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.

### 4.3.3 Token Stream

The following representations were identified by representing the regex patterns as a sequence of tokens. Identifying S2 requires any element to be repeated at least twice. This element could be a character class, a literal, or a collection

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

of things encapsulated in parentheses. Identifying simply C1 requires that a character class contains a range.

Identifying C2 requires that there exists a character classes that does not use ranges or defaults. Identifying C4 requires the presence of a default character class, specifically, `\d`, `\D`, `\w`, `\W`, `\s`, `\S` and `.`. Identifying C5 requires an OR of length-one sequences/ We recognize the start of an OR, then allow one or more literals, defaults or ranges before the OR is completed.

## 4.4 Results

Table 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 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 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, 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 C2 $\Rightarrow$ C1 refactoring. Based on patterns along, 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 would be more desirable based on community standards.

# 5. 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.

## 5.1 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.

## 5.2 Metrics

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

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 3: Participant Profiles,** $n = 180$

### *Matching.*

Given a regex and a set of strings, a participant determines which strings will be matched by the regex. The ratio of correct responses to attempted matching questions is the matching score. For example, consider regex `'RR*'` and five strings, which comes from our study, shown in Table 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 about the first and second matching questions, 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 $1/1 = 1.00$.

### *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 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

Table 2: Matching metric example

| String | 'RR*' | Oracle | P 1 | P 2 | P 3 | P 4 |
|---|---|---|---|---|---|---|
| 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

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.

## 5.3 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.

## 5.4 Participants

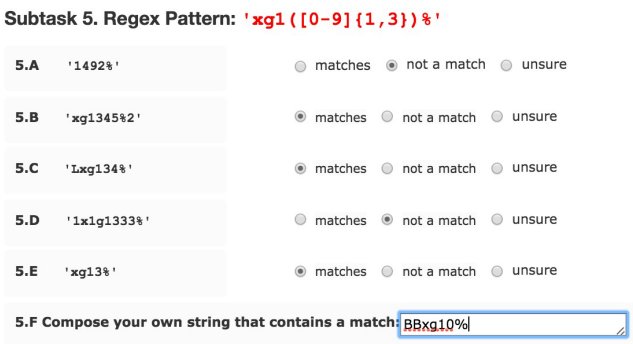


**Figure 4: Example of one HIT Question**

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 3 summarizes the self-reported participant characteristics from the qualification survey.

## 5.5 Analysis

For each of the 60 regexes, an average matching score was computed using the metrics in Table 2. 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.**

## 5.6 Results

**TODO.NOW: add more info about understandability results**

# 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 1) with respect to community standards (RQ1) and understandability (RQ2).

## 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 (10 in total), we performed a topological sort to get total node orderings.

The graphs for community support are based on Table 1 and the graphs for understandability are based on Table 3.

### 6.1.1 *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.

**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 |

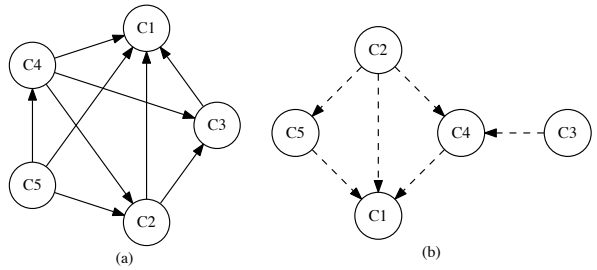


**Figure 5: Trend graphs for the CCC equivalence graph. Solid lines represent the artifact analysis. Dashed lines represent the understandability analysis.**

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.

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.

### 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. 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 ← []
2: S ← []
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 → 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.

## 6.2 Results

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.

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.

# 7. DISCUSSION

## 7.1 Further Refactoring Opportunities

A particular community (like Mozilla or some startup) may want to include regex refactoring into their coding standards. We anticipate that having one preferred way to express a particular regex may increase regex understandability in the codebase for maintainers, leading to a reduction in errors related to misunderstanding regexes.

We looked at 5 groups that cover the most frequently used features, but other refactorings exist that deserve further study. Here are a few:

**Single line option** `'''(.|\n)+'''` is equivalent to `(?s)'''(.)+'''`

**Multi line option** `(?m)G\n` is equivalent to `(?m)G$`

**Multi line option** `(?i)[a-z]` is equivalent to `[A-Za-z]`

**Backreferences** `(X)q\1` is equivalent to `(?P<name>X)q\g<name>`

**Word Boundaries** `\bZ` is equivalent to `((?<=\w)(?=\W)|(?<=\W)(?=\w))Z`

One terribly obvious application of our refactorings is to search existing codebases for regex patterns that are candidates for refactoring, allowing a person to quickly go through a list of possible changes and improve the understandability of the regexes in their codebase.

Maintainers of code that is intentionally obfuscated for security purposes may want to develop regexes that they understand and then automatically transform them to equivalent but less understandable regexes.

## 7.2 Threats to Validity

### 7.2.1 *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.

### 7.2.2 *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.

# 8. 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.

# 9. CONCLUSION

**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 |


## Acknowledgements

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