PYEVL\textsc{O\textunderscore VOLVE}: Automating Frequent Code Changes in Python ML Systems

Malinda Dilhara
University of Colorado Boulder, USA
malinda.malwala@colorado.edu

Danny Dig
University of Colorado Boulder, JetBrains Research, USA
danny.dig@colorado.edu

Ameya Ketkar
Uber Technologies Inc., USA
ketkara@uber.com

Abstract—Because of the naturalness of software and the rapid evolution of Machine Learning (ML) techniques, frequently repeated code change patterns (CPATs) occur often. They range from simple API migrations to changes involving several complex control structures such as for loops. While manually performing CPATs is tedious, the current state-of-the-art techniques for inferring transformation rules are not advanced enough to handle unseen variants of complex CPATs, resulting in a low recall rate. In this paper, we present a novel, automated workflow that mines CPATs, infers the transformation rules, and then transplants them automatically to new target sites. We designed, implemented, evaluated, and released this as a tool, PYEVL\textsc{O\textunderscore VOLVE}. At its core is a novel data-flow, control-flow aware transformation rule inference engine. Our technique allows us to automate the state-of-the-art for transformation-by-example tools; without it, 70\% of the code changes that PYEVL\textsc{O\textunderscore VOLVE} transforms would not be possible to automate. Our thorough empirical evaluation of over 40,000 transformations shows 97\% precision and 94\% recall. By accepting 90\% of CPATs generated by PYEVL\textsc{O\textunderscore VOLVE} in famous open-source projects, developers confirmed its changes are useful.

I. INTRODUCTION

The naturalness of software [24], [25], [29], [34] leads to repeated code changes, both within and across projects. Because programmers employ the same coding idioms [26], [28] and best practices (e.g., linting [83]), they change code similarly, resulting in repeated code changes. These repeated changes are fine-grained, recurring at the method level, and have the same semantics.

Listing 1 shows an example of such a change in project NifTK/NiftyNet, an open-source convolutional neural network platform. The developers replaced a for loop that sums the list elements with \texttt{numpy.sum}, which is a best practice for improving performance. This change involves programming idioms [27], [28] and occurs within a specific method. As it is repeated at multiple locations in multiple commits, we call this a code change pattern (CPAT).

Listing 1: Commit c8b28432 in GitHub repository NifTK/NiftyNet: Replace for loop with NumPy sum

```python
- result = 0
- for elem in elements:
  - result = elem + result
+ result = \texttt{numpy.sum}(elements)
```

Writing program transformations to automate a huge variety of such changes (e.g., more than 28,000 kinds as shown in [21]) is difficult, as evidenced by research [64]–[82]. There are many reasons. (i) AST rewriting poses a significant barrier to entry, (ii) Matching control/data flows for real code is tedious to develop since there is too much noise and syntactic variance to account for, (iii) Moreover, as these best practices evolve, they are difficult to maintain, thus it requires a community effort. To overcome these challenges, in this paper we rely on a unique insight: if we mine the many examples of CPATs in the open-source community and infer transformation rules, we can feed them to automated program transformation systems, with no burden on the developer. Thus, we present an end-to-end solution, PYEVL\textsc{O\textunderscore VOLVE}, which mines CPATs from the open-source community, it automatically infers the transformation rules in a data-and-control-flow manner, and transplants and applies them at new target sites. We show its effectiveness by applying it on Python ML projects.

According to the GitHub 2021 annual report [42] of programming language usage, Python is one of the top two most used. Moreover, it has become the lingua franca for machine learning (ML) and data science development [8]–[12]. Despite its prominence and community needs, tools for evolving Python code are significantly behind other languages [10]. To improve the tools available for Python, researchers recently developed tools to mine CPATs in software systems. R-CPATMINER [21], [22] is one such tool that we developed for mining CPATs in Python systems. In that previous work [21], we further conducted a large-scale empirical study on CPATs in 1,000 Python ML systems and found that the complexity of CPATs in ML systems ranged from basic API migrations to changes involving complex control structures like for loops. Developers perform CPATs for several reasons: performance (e.g., for loop \rightarrow vectorization), using advanced language features (e.g., for loop \rightarrow list comprehension), better resource management (e.g., using with statement), and library migrations (e.g., numpy.mean() \rightarrow torch.mean()). Manually searching and applying variations of such changes to several locations is error-prone. Moreover, developers might overlook sites that require the same edit. Indeed, over 75\% of the respondents of a survey [21] with 97 developers indicated they needed these CPATs to be automated. However, the existing CPAT automation tools are not yet able to handle them.

Despite the existence of many program transformation systems [64]–[82], their main impediment to adoption is the need for programmers to write sophisticated program transforma-
tion rules. In recent years, we have seen an emerging trend of tools and techniques that infer transformation rules using example code edits [1], [15]–[19], [33], [48]–[52], [54], [86], [87]. These techniques infer transformation rules from before and after edits of human-adaptations, then use the inferred rules to transform target codes. This is called “Transformation by Example”. Despite the potential of such techniques to significantly ease code evolution, they have so far been primarily used for API migrations, such as replacing obsolete API calls with modern ones from the Android SDK [1], [5], [15], [17], [19], [54], the Linux kernel [40], [41], and Type migration of Java systems [33]. Although existing techniques work well when replacing an API call with another, they under-perform on more complex coding idioms such as the one in Listing 1. In diverse codebases, this CPAT will have many variations in terms of data- and control-flow.

Listing 2: The repository hachmannlab/chemm uses a for loop to compute the sum of an array

```python
n_diff = 0
to_eval = getEvalArray()
for dif in to_eval.getDiff():
    total = n_diff + dif
    n_diff = 0
```

For example, Listing 2 is semantically equivalent to Listing 1 but differs in data- and control-flow due to assignments and how the accumulator variable computes the sum. Therefore, existing “Transformation by Example” techniques struggle because the rule they inferred for Listing 1 cannot be used to transform the target code in Listing 2, resulting in a low recall rate. Because existing techniques are so syntax-centric, the transformation rules are prone to overfitting to the input examples. That is, the transformation rule may work well on the given examples but may be unable or erroneously transform unseen data- or control flow variants outside of the examples. This demonstrates a major limitation of current transformation rule inference methodologies.

To overcome these challenges, in this paper we present a novel data- and control-flow aware technique that infers transformation rules and adapts them to transform even unseen variations in the target codes. Our novel technique enables automating even unseen variations of the CPAT by preserving data- and control-flow relations. In this paper, we present a fully end-to-end pipeline that mines and automates Python CPATs. Our pipeline consists of four major steps: (i) mine CPATs from version histories, (ii) infer transformation rules, (iii) identify the new sites to apply the CPATs, (iv) adapt the transformation rules to the new sites. To do so, we leverage and further extend four state-of-the-art techniques: (i) to mine CPATs we use R-CPATMiner [21], [22], (ii) to infer initial transformation rules from example instances in CPATs we use InferRule [33] (iii) to identify new sites to apply CPATs we use fine-grained program dependence graphs fgPDG [34]; and (iv) to apply the CPATs at the new sites, even for unseen variants, we re-infer the transformation rules based on data- and control-flow relations in the fgPDG. Lastly, we use ComBy [32] to declaratively rewrite programs according to the re-inferred transformation rules.

We implemented our novel technique in PYEVLolve. We evaluated its effectiveness and usefulness on a corpus of CPATs that had previously been shown to be diverse in terms of size, frequency, authors, and projects [21]. We conducted replication case studies comprising a broad variety of 40,000 transformation trials. Using cross-validation, we tested PYEVLove’s ability to correctly transform CPATs. We found that PYEVLove achieves 97% precision and 94% recall when replicating developer-performed changes. In addition, we discovered that 70% of these changes are data or control-flow variants that cannot be automated using existing techniques, thus PYEVLove advances the state-of-the-art significantly. To evaluate PYEVLove’s usefulness, we submitted pull requests to highly-rated, best-in-class projects such as TensorFlow, PyTorch, Scikit-Image, and Keras, totaling 181 CPAT instances. At the time of this writing, their developers have already accepted 163 (90%) CPATs.

This paper makes the following contributions:

1. We introduce a novel data- and control-flow aware rule inference that effectively transforms even unseen variants that cannot be handled by existing rule inference techniques.
2. We designed and implemented our technique in PYEVLove. It mines CPATs from projects, infers transformation rules, and generates patches for Python projects. To the best of our knowledge, this is the first such pipeline developed for Python and it assists ML developers and other Python developers in keeping up with rapidly evolving best practices.
3. Our empirical evaluation of PYEVLove on 40,000 transformations shows that PYEVLove is effective (97% precision, 94% recall), needed (it enables automating 70% more CPATs than existing tools), and its patches are useful (developers accepted 90% of 181 PYEVLove-generated CPATs).
4. Our tool and evaluation dataset is open-source and available for others to reuse [44].

II. MOTIVATING EXAMPLES

To illustrate the challenges of using existing “Transformation by Example” techniques we use the real-world code changes, shown in Table I. The first column (Code Before—Code After) shows the code fragments before and after the code change, while the second column (Rule) presents the rules encoding the code change, while the second column (Rule) presents the rules encoding the code change using the syntax of Comby [20], a state-of-the-art code rewrite tool. The third column shows the Guards for each rule in column-2. The last column shows examples of new target sites that we would like to transform. First, we describe a success scenario that existing techniques automate with a high recall rate, followed by three scenarios that they struggle to automate (as the complexity of the code rises), but PYEVLove succeeds.

Table 1, row 1 depicts an example of a CPAT mined from TensorFlow. To open a file, the code changes from open, which is native to Python, to GFile in TensorFlow. This adaptation is represented by the rewrite rule `{{v1}}=open({{v0}}} → `{{v1}}}=tf.gfile.GFile{{v0}}}). This rule transforms any
target code similar to the before-change shown in column-1-Table I. The rule has a left side (indicating the “before” the change) and a right side (indicating the “after” the change) separated by an arrow. The left side contains Python statements with template variables (e.g., \( v_0 \)) that bind to AST nodes from the actual source code (e.g., \( v_0 \) binds to `file.csv`). The right side of the rule also contains Python statements with template variables, where each template variable denotes the code fragments that will be used after the change is applied. Because the change in row 1 swaps one API invocation (\( \text{open} \)) for another (\( tf.gfile.GFile \)), the likelihood of finding semantically equivalent different variants (in terms of data-flow or control-flow) in new target sites is very low. Therefore, the example transformation (in columns 1&2) is sufficient to represent many variants and would be applicable to many new target sites. For these API migrations, the existing Program-by-Example techniques [15], [17]–[19], [33], [48]–[52] perform well (i.e., they achieve high recall rate).

However, many real-world CPATs involve multiple method invocations [21]. For example, as shown in row 3, in the project SciPy, developers transform error-ignoring code that uses `try finally` to a \( \text{with} \) statement. The new target site in column-4, on the other hand, has an extra method invocation, making it a control-flow variant. Now let us consider the example in row 4, which also appeared in Section I. The target site in column-4 has an additional assignment statement and variable assignment, which makes it both data- and control-flow variant. Since these variants were not seen as change exemplars during rule inference, existing “Transformation by Example” techniques fail to change these new target sites, but PYEVOLVE changes them correctly.

We observed many cases where existing techniques inferred rules that are prone to over-fitting the examples and are unable to account for even minor variants. These variants are frequently employed in ML code bases that perform advanced numerical computations. Perhaps these variants make it easier to debug. For example, even though the code in Listing 2 uses a redundant variable \( \text{total} \) (lines 4 and 5 can be easily combined), a developer can still use the variable \( \text{total} \) by inserting a debug point in Line 4 to test the addition of two consecutive array elements. Despite the abundance of such variants, existing “Transformation by Example” techniques fail to change these new target sites, but PYEVOLVE changes them correctly.

### Table I: Motivating Examples

<table>
<thead>
<tr>
<th>Code Before</th>
<th>Rule</th>
<th>Guard</th>
<th>New Target Site</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>f=open(&quot;f.csv&quot;)</code></td>
<td><code>:[[v1]]=open(:[[v0]])</code></td>
<td><code>type:=:[[v1]] -&gt; TextIO</code></td>
<td><code>f=open(&quot;data.csv&quot;)</code></td>
</tr>
<tr>
<td><code>f=tf.gfile.GFile(&quot;f.csv&quot;)</code></td>
<td><code>-</code></td>
<td><code>-</code></td>
<td><code>-</code></td>
</tr>
<tr>
<td><code>X=numpy.dot(A,B)</code> <code>Y=numpy.dot(X,C)</code></td>
<td><code>:[[v3]]=:([[[v1]],:([v2])].dot(:([v3]),:([v4])))</code></td>
<td><code>type:=:[[v1]] -&gt; ndarray</code></td>
<td><code>prod = numpy.dot (numpy.dot(A,B),C)</code></td>
</tr>
<tr>
<td><code>Y=numpy.linalg._multi_dot((A,B,C))</code></td>
<td><code>:[[v5]]=:([v6]).linalg.multi_dot (:([v1]),:([v2]),:([v3])))</code></td>
<td><code>import:=:[[v6]] -&gt; numpy</code></td>
<td><code>-</code></td>
</tr>
<tr>
<td><code>olderr = np.seterr(divide='ignore')</code> <code>try:</code> <code>try:</code> <code>try:</code> <code>try:</code> <code>try:</code></td>
<td><code>:[[l1]]=:([[[l3]].seterr(divide='ignore'))</code> <code>finally:</code> <code>finally:</code> <code>finally:</code> <code>finally:</code> <code>finally:</code></td>
<td><code>import:=:[[l3]] -&gt; numpy</code></td>
<td><code>olderr = np.seterr(divide='ignore')</code></td>
</tr>
<tr>
<td><code>with np.errstate(divide='ignore'):</code> <code>actual=logit(a)</code></td>
<td><code>with :[[l3]].errstate(divide='ignore'):</code> <code>n_diff = 0</code></td>
<td><code>n_diff = 0</code></td>
<td><code>n_diff = 0</code></td>
</tr>
<tr>
<td><code>for elem in elems;</code> <code>res = 0</code></td>
<td><code>for :[[v1]] in :[v2];</code> <code>for :[[v0]] = :[[v1]] + :[[v1]]</code></td>
<td><code>type:=:[[v0]] -&gt; int</code></td>
<td><code>to_eval = getEvalArray()</code></td>
</tr>
<tr>
<td><code>res = numpy.sum(elems)</code></td>
<td><code>-&gt; :[[v0]]=numpy.sum(:[[v2]])</code></td>
<td><code>type:=:[[v1]] -&gt; int</code></td>
<td><code>n_diff = total</code></td>
</tr>
</tbody>
</table>
target sites. Figure 1 provides the high-level overview of our technique, which consists of four major phases:

**Phase 1.** PyEVOVE mines **CPATs** from the version history of Python projects. To do so, we use **R-CPATMiner** [21], the state-of-the-art **CPAT** mining tool for Python systems.

**Phase 2.** PyEVOVE infers transformation rules from the previously mined **CPATs**. To do so, we adapted **InferRule** [33], which is a state-of-the-art tool for inferring transformation rules in Java (that we adapted to Python).

**Phase 3.** PyEVOVE identifies target sites in the code where previously mined **CPATs** can be applied. To do so, we use a graph-based matching technique (described in Section III-B3).

**Phase 4.** PyEVOVE adapts the rules inferred in Phase2 to the target sites and their contexts (see Section III-B4). This allows it to be effective for a wider range of unseen **CPAT** variants. These adapted rules take into account the data-and control-flow relations. Finally, it applies them to the target sites observed in Phase3.

The key idea of making **PyEVOVE** resilient to both seen and unseen variants relies on a graph-based matching technique that involves two main steps: (i) We first build graphs for the target code and initial transformation rule, then mine isomorphic sub-graphs to those of the rule in the target code. This makes it possible to identify potential sites for **CPATs**, taking into account both seen and unseen data-and control flow variants (Section III-B3). Then (ii) using the sub-graphs, we adapt the transformation rules to the target site and its context. This makes it possible to create rules that take into account the target code’s data-and-control flow (Section III-B4).

The rest of this section is organized as: (i) Section III-A defines fundamental terms used throughout the paper, (ii) Section III-B1 describes how we mine **CPATs**, (iii) Section III-B2 describes our graph representation for the rule and target code, and (iv) Section III-B3 describes how to mine potential sites to apply **CPAT** by mining sub-graphs, finally (v) Section III-B4 how to infer the final transformation rule.

### A. Basic Concepts

We now formally define the fundamental terms.

**Definition III.1 (**Template Expression**; **T**).** This is a generic and lightweight way of searching and matching syntactic structures in the **AST** of a function. In Python, a template expression is made up of expressions and statements, as well as template variables (or **holes**) that match to a program **AST**.

Our template expressions adhere to the **ComBY** syntax [32] - a state-of-the-art, lightweight multi-language syntax transformation technique for declaratively rewriting syntax. Rules written in the **ComBY** language are human readable and modifiable by developers, who can further customize the changes. Thus, they will appeal to developers who might not like hard-coded refactoring tools that automate code changes [45], [84] without any intervention from the developer. Details of **ComBY**’s syntax can be found on its website [20].

**Definition III.2 (**Template Variable**; **V**).** This corresponds to one or more program elements and is also known as **Holes** [5] in the “Transformation by Example” domain. According to **ComBY** [20], **v[n]** binds the source code to a template variable **n**. A template variable can match all characters lazily up to its suffix (like .*? in regex) within its level of balanced delimiters. **ComBY** supports mainly two template variables: (i) **v[1]** matches an identifier, analogous to \w+ in regex; we denote it as **V1**, (ii) **v[n]** matches one or more alphanumeric characters and "_". We denote it as **Vn**.

**Definition III.3 (RULE: **T** LHST → **T** RHST).** This defines how to transform the input **AST** to an output **AST** using **template expressions**. In our setting, **template expression** (**T** LHST) on the right side of the **Rule** contains **template variable** (**V**) that denote the substitution with an appropriate fragment of the program **AST**, as matched on the left side. For example, once the rule in row 4-Table I is applied to the code that sums the elements in Listing 1, it will be rewritten as **np.sum(elements)**.

1) **Transformation Rules (TR)**: Modern “Transformation by Example” techniques employ transformation rules that correspond to the examples defined over a predefined Domain-Specific Language (DSL). **PyEVOVE** inherits the DSL of **TCInfer** [33], a tool that infers rules for Java systems, and extends it to the language shown in Figure 2. In this DSL, a program transformation rule is a pair of **Guard** and **Rule**. Essentially, the **Guard** validates which code fragments should be transformed while the **Rule** describes how those code fragments should be transformed. The **Guard** is composed of a set of conjunctive predicates (**Pred**) on the attributes (e.g., **Type**, **Kind**, **Value**) of the template variables. The **Guard** evaluates where a template variable satisfies its predicate(s) and returns a Boolean value accordingly. Column 2 of Table I shows the **Rule** inferred from the code changes in column 1, and column 3 shows the relevant **Guard**. For example, the
Rule in row 1 describes how the method call open should be transformed (Rule), but only in the places where a string is passed as an argument and TestILO is returned (Guard).

2) Fine-Grained Program Dependence Graph (fgPDG): In their recent work, Nguyen et al. [34] presented fgPDG, a sufficiently generalized program dependency graph that can be used to mine semantically equivalent program fragments with differing data-and control relations. Researchers utilize fgPDGs for many applications. For example, Nguyen et al. [34], Smirnov et al. [46], and Dilhaara et al. [21], used fgPDG on Java and Python systems to mine semantically equivalent repeated code changes, while Noda et al. [47] used fgPDG to mine repeated bug fixes and repair unified similar bugs. Our key idea for determining if the target code contains an equivalent code for the LHS (see Definition III.3) regardless of data-and control-flow is to construct a fgPDG for the LHS and determine if it is a subgraph of the target code’s fgPDG. Then, we adapt the inferred rule to match the target code. For example, to match the target code in Listing 2, the learned rule in row 4-Table I must be adapted to Listing 3.

Listing 3: Adapted transformation rule to match with Listing 2
```
:([v0]) = 0  \Rightarrow  :([v3])
:([v3]) = :([v0]) - numpysum(:([v2]))
for :([v1]) in :([v2]):
  :([v3]) = :([v0]) + :([v1])
  :([v0]) = :([v4])
```

To achieve this, we extend fgPDG with two new nodes: IdentifierHole and AlphaHole, which represent template variables in template expressions.

Definition III.4 (IDENTIFIERHOLE: I), \( V_I ([v_i]) \) represents a variable identifier in code (analogous to w+ in regex). To denote \( V_I \) in fgPDG, we add the new node, IdentifierHole.

Definition III.5 (ALPHAHOLE: A), \( V_A ([v]) \) represents an expression (e.g., method call \(-np.dot(), \) list\([-1,2], \) and dictionary \(-{'one':1}, \) or a statement (e.g., an assignment). To denote \( V_A \) in fgPDG, we use the new node, AlphaHole.

B. Data-flow Control-flow aware rule inference

PYEVEOLVE generates transformation rules for applying CPATs to target code. First, it uses INFERRULE [33] to infer the transformation rule for the input CPATs, then adapts the rule based on the data and control flow in target codes, which we call data-and control-flow aware rule inference, formally:

Definition III.6 (DATA-AND CONTROL-FLOW AWARE RULE INERENCE). For given input code changes \( \{i_0 \rightarrow o_0, \ldots, i_n \rightarrow o_n\} \), the existing “Transformation by Example” techniques infer transformation rule \( TR \) such that \( TR_k(i_k) = o_k, \) \( k \in \{0...n\} \). Control- and data-flow aware rule inference generates an adapted \( TR_k^A \) for each \( TR_k \) that matches a target code.

1) Input: In our previous research, we introduced R-CPATMINER [21], a tool for collecting CPATs that developers performed in Python systems. Our previous research also conducted an empirical study of diverse 2,500 CPATs, which revealed the existence of four kinds of frequently occurring CPATs in Python ML systems: (i) dissolve for loops to domain-specific abstractions (e.g., row-4-Table I) (ii) update API usage (e.g., row-2-Table I), (iii) transform to context managers (e.g., row-3-Table I), and (iv) use advanced language features (e.g., Python list comprehension). We use the same CPATs as input to PYEVEOLVE.

2) Generating fgPDGs: A fgPDG is a directed graph consisting of three types of nodes and two types of edges. (i) Data Nodes (\( N^D \)) represent variables, field accesses, and constants. For example, the variable \( n_{-diff} \) in Figure 3(a) is a data node. (ii) Action Nodes (\( N^A \)) represent operations on data, e.g., array accesses, and method calls. For example, in Figure 3(a), the methods calls \( getEvalArray() \) is an Action node, and (iii) Control Nodes (\( N^C \)) represent control statements, such as \( if \) for branching, \( for \) for looping (see Figure 3(a)). (i) Control edges represent control relations between the statements/operations and the control nodes on which their executions depend, (ii) Data edges represent the data flow of fgPDG nodes. Figure 3(a) shows labels on each edge, indicating the type of data flow as defined by Nguyen et al. [34], such as para, def, or cond.

We first generate an fgPDG for the target code. For example, Figure 3(a) shows the fgPDG generated for the target code in Listing 2. One of the difficulties in generating an fgPDG for Python code is the lack of type information at compile time, which is critical for mining semantically equivalent code. To overcome this challenge, we employ type inference, a technique for inferring the type information of program elements based on data-flow information available at compile time. For this purpose, we use PYTYPE [37], developed by Google, which is widely used by the Python community.

Second, we generate an fgPDG for the left side of the transformation rule (\( T_{LHS} \)). Figure 3(b) shows an example fgPDG generated for the \( T_{LHS} \) in row 4-Table I. In addition to the three nodes used by Nguyen et al. [34], we added two more nodes to the fgPDG — IdentifierHole (see Definition III.4) and AlphaHole (see Definition III.5) — to generate the fgPDG of the \( T_{LHS} \). For example, the fgPDG nodes, :([v2]) and :([v0]) in Figure 3(b) are examples of AlphaHole and IdentifierHole.

3) Identifying potential sites: We now describe how we identify potential target code sites to apply CPATs. Conceptually, given the fgPDG of the target code (fgPDG\(^T\) and
Figure 4: Steps in Algorithm 2 that are followed by Listing 2 and the rule in row 4-Table 1 to be adapted as a rule in Listing 3.

Algorithm 1 Match nodes from fgPDG\textsuperscript{T} and fgPDG\textsuperscript{R}

1: function MATCHEDNODE(cNode, tNode)
2:   if NODETYPE(cNode) == IdentifierHole then
3:     if cNode.ASTNode == SimpleName then
4:       if GUARDSMEETS(cNode, tNode) then
5:         return True
6:   else if NODETYPE(tNode) == AlphaHole then
7:     if cNode.ASTNode==Expression or Statement then
8:       if GUARDSMEETS(cNode, tNode) then
9:         return True
10:  else if NODETYPE(cNode) == NODETYPE(tNode) then
11:    if NODETYPE(cNode) == N\textsuperscript{D} Or N\textsuperscript{C} Or N\textsuperscript{A} then
12:      if GUARDSMEETS(cNode, tNode) then
13:        return True
14:   return False

The `fgPDG` of the `T\textsubscript{LHS}` (fgPDG\textsuperscript{R}), `PYEVEOLVE` determines whether the fgPDG\textsuperscript{R} is a sub-graph of fgPDG\textsuperscript{T} and if it is, it reports the matching nodes as the locations to apply the rule. This differs from typical sub-graph mining problems in two ways: (i) AlphaHole in fgPDG\textsuperscript{R} can match against one or more nodes in fgPDG\textsuperscript{T}, and (ii) data nodes in fgPDG\textsuperscript{R} can match transitively to data nodes in fgPDG\textsuperscript{T} via Data edges [34]. Therefore, the matched sub-graph can be a disconnected graph.

Figure 3(a) (fgPDG\textsuperscript{T}) depicts the fgPDG generated for the target code Listing 2, whereas Figure 3(b) (fgPDG\textsuperscript{R}) depicts the fgPDG generated for the `T\textsubscript{LHS}` of rule in row 4-Table 1.

Algorithm 1 describes how we match two nodes from fgPDG\textsuperscript{R} and fgPDG\textsuperscript{T} based on node kinds and the related `Guard`. If both nodes are in same category, i.e., N\textsuperscript{D}, N\textsuperscript{A}, or N\textsuperscript{C} (see Section III-B2), the operation matches them based on Guards (line 12). For example, in Figure 3, the N\textsuperscript{D}, number(0) in fgPDG\textsuperscript{T}, matches the N\textsuperscript{D}, number(0), in fgPDG\textsuperscript{R}. The node kind IdentifierHole matches the identifiers, hence the operation determines if the target node’s AST kind is SimpleName and whether the Guard matches (see 4). For example, the IdentifierHole, :\{v0\} in Figure 3(b) matches with the N\textsuperscript{D}, n\textsuperscript{diff}, which has type->int as the Guard. AlphaHole, matches with any expression (e.g., method invocation) or statement (e.g., for, statement) if they meet the Guard. Hence, AlphaHole can match one or many nodes in fgPDG\textsuperscript{T}. The :\{v2\} in Figure 3(b) matches two nodes in the fgPDG\textsuperscript{T} that are relevant to the method call to_eval.getDiff().

We recursively walk through nodes and their child nodes, and then perform the operation MATCHEDNODE to obtain a matched fgPDG (G\textsuperscript{M}), which is a sub-graph of the fgPDG\textsuperscript{T}. In order to match with data flow variants, we also check whether the Data nodes of fgPDG\textsuperscript{R} are transitively matched through Data edges to Data nodes in fgPDG\textsuperscript{T}. For example, the target code in Listing 2 adds the two values, n_diff, and diff, together and assigns it to a variable total, and then it is reassigned to n_diff whereas the _T\textsubscript{LHS} directly assigns the value to the template variable, :\{v9\}. These data flow variants can be matched if we transitively match the code through the Data edges. Therefore, the nodes int(n_diff), Assignment, and number(0) in Figure 3(b) are matched transitively and are disconnected from the main graph. Finally, we obtain the matched graph (G\textsuperscript{M}), a disconnected sub-graph which primarily contains information about the locations where the nodes in CPAT can be applied.

4) *Inferring transformation Rules* (TR\textsuperscript{A} : T\textsubscript{LHS} \rightarrow T\textsubscript{RHS}): Now we know the code elements in the target code that correspond to nodes in T\textsubscript{LHS}, the next step is to infer the adapted TR\textsuperscript{A} (Definition III.6) that can be applied to automate the program transformation. The higher level intuition is to: (i) capture the structure of the target code that is matched to fgPDG\textsuperscript{T}, and (ii) infer an adapted transformation rule (T\textsubscript{LHS} \rightarrow T\textsubscript{RHS}) by making matched nodes in the target code as holes, if necessary, and creating or deleting the new holes in the T\textsubscript{LHS} or T\textsubscript{RHS} to transform the matched nodes and the context. We now explain the steps we follow to adapt the rule in row 4-Table 1 (inferred from Listing 1) to the rule in Listing 3, which can be applied on Listing 2, an unseen variant.

In order to generate the adapted transformation rule (TR\textsuperscript{A}), we must first rename the matched nodes with the matched Template Variable in fgPDG\textsuperscript{R}. RENAMETEMPLATEVARS in Algorithm 2 renames the matched nodes in the target code using G\textsuperscript{M} which contains which template variables are matched to which target code nodes. As shown in Figure 4(a), we rename, :\{v0\} \rightarrow n\_diff, and :\{v1\} \rightarrow diff. AlphaHole (:\{v2\}) is used for to_eval.getDiff() because it matches several nodes, resulting in the renaming of an entire expression.

The next step of generating TR\textsuperscript{A} is to rename the target code’s context, i.e., unmatched code nodes, using appropriate template variables. We use the template variable \forall\textsuperscript{A} (:\{v\}) to rename statements (e.g., for loop), whereas \forall\textsuperscript{I} (:\{v\}),
is used to rename identifier names. The nodes that have been renamed thus far in the operation in line 2 are taken into consideration when deciding whether the template variable to employ. We use a new $V^A$, if none of the nodes in a statement have been renamed. For example, $: [v3]$ in Figure 4(b) is used to rename $toEval = getEvalArray()$. We use $V^I$ to rename identifiers that are still intact after part of a statement has already been replaced. Here, we use the same $V^I$ for the same identifier. For example, in Figure 4(b), we use the $[:[v4]]$ to rename the identifier $total$.

The next step in Algorithm 2 is to use the operation SUBSTITUTE_NODES on $T^A_{LHS}$ to generate $T^A_{RHS}$ which performs two actions: (i) replaces the nodes $T^A_{LHS} \cap T^A_{LHS}$ with $T^A_{RHS}$ (line 8). The replacing statements will be ordered as specified in $T^A_{RHS}$, so that the final transformation resembles the input code change example, (ii) builds the context of the code using the new template variables $(T^A_{LHS} - T^A_{LHS})$. Here, we discard the transitivity matched nodes (i.e., $[:[v4]]$) identified in the sub-graph mining algorithm described in Section III-B3. This ensures that the template expression, $T^A_{LHS}$, matches the data variants in the code and then transforms it to the code defined in $T^A_{RHS}$. The example in Figure 4(c) shows the two actions performed by SUBSTITUTE_NODES. It first substitutes the nodes $T^A_{LHS} \cap T^A_{LHS}$ in $T^A_{LHS}$, i.e., the for loop and its body from $: [v0] = np.sum(:[v2])$. Then, it uses the nodes $T^A_{LHS}$ - $T^A_{LHS}$, i.e., $: [v3]$ to create the context. Finally, the algorithm outputs the adapted rule, $T^A_{LHS} \rightarrow T^A_{RHS}$.

Handling variations on the $T^A_{RHS}$: When PYEVOLVE generates $T^A_{RHS}$, there may be more than one way to place the $T^A_{RHS}$ along with the context in the target code. One may assume that PYEVOLVE can output all the possible forms of $T^A_{RHS}$ that can be created and then seek the developer to choose the final rule. However, this becomes impractical as the number of possible variations of $T^A_{RHS}$ increases with the size of $T^A_{RHS}$ and context. Therefore, PYEVOLVE always puts the $T^A_{RHS}$ in the place of the last matched data node, and allows the developer to change the rules if they need a different variant. Given that most CPATs often replace a more complex idiom on the left side (involving several statements) with one or two statements on the right side, handling variations on the right side would not occur often in practice.

5) Eliminating Unsafe Transformations: To safely transform code, refactoring researchers [30], [31], [63] use preconditions that must be satisfied before transforming a target site. In a similar spirit, the operation ISSAFE in Algorithm 2 evaluates lightweight preconditions on the identified transformable code locations ($G^M$) before applying the rule to a target site. PYEVOLVE uses the following preconditions:

**Precondition 1:** Some CPATs may involve the deletion of variables used in $T_{LHS}$. For example, in Listing 2, the loop variable $dif$ and the local variable $total$ will be deleted and will no longer be available once $np.sum$ replaces the original for loop. However, in Python, loop and local variables can be used outside the scope of the loop or block where they are initialized, e.g., they can be used further down in the code after the for loop. If that was the case, transforming the loop in Listing 2 to $np.sum$ and deleting those two variables would be unsafe. PYEVOLVE checks to see if any variables that are marked for deletion are later used in the code. If this is the case, PYEVOLVE does not proceed with the transformation.

**Precondition 2:** The $fgPDG$-based matching algorithm, as described in Section III-B3, can identify target sites in a program that contain extra statements within scopes defined in $T_{LHS}$. This would make the transformation unsafe. For example, suppose that Listing 2 contained an additional statement such as $print(dif)$ inside the for loop body. The matching algorithm would identify this as a potential site to transform. However, this target site should not be replaced by $numpy.sum$ because it does not preserve semantics. To prevent transforming such target sites and ensure safety, PYEVOLVE discards target sites that contain going edges from action nodes (i.e., $N^A$) or control nodes (i.e., $N^C$) in the matched graph (i.e., $G^M$) to the nodes in $fgPDG$ (but $\notin G^M$).

**Precondition 3:** The $T_{RHS}$ of CPATs might contain APIs from third-party libraries (e.g., TensorFlow). Transplanting such CPATs into projects that do not use those libraries would break the code. To ensure the transformation’s safety, PYEVOLVE checks whether it can fully resolve the API invocations in the $T_{RHS}$. For example, if the $tf.sum$ is in the $T_{RHS}$, PYEVOLVE checks the project’s requirements.txt [88] to see if the TensorFlow is listed as one of the project’s libraries. If it is not included, it will discard the transformation.

Defining a complete list of preconditions is challenging, but it can be addressed by capturing more context. Therefore, our implementation is flexible to express additional preconditions as they emerge. We further elaborate on safety in Section V.

### IV. Evaluation

We empirically evaluate PYEVOLVE and we answer the following research questions:

**RQ1. What is the effectiveness of PYEVOLVE in generating correct code transformations?** We conduct cross-validation to determine that PYEVOLVE correctly transforms code by replicating real-world CPATs. We report PYEVOLVE’s overall effectiveness by its precision and recall.

**RQ2. What is the contribution of data- and control-flow aware rule inference to overall effectiveness?** To perform this analysis, we report the number of changes that would be impossible to perform without the features in PYEVOLVE.

**RQ3. How do developers find PYEVOLVE’s changes useful?** To answer this, we submit pull requests to open-source projects containing patches generated by PYEVOLVE and record the developers’ responses.

#### A. RQ1: What is the effectiveness of PYEVOLVE in generating correct code transformations?

To answer this question, we replicate with PYEVOLVE thousands of transformations that open-source developers applied manually on their projects. We conducted cross-validation to test whether PYEVOLVE correctly transforms real-world CPATs. We divided the human adaptations published by Dillahara et al. [21] into training and testing sets. The training...
set is used to learn the initial transformation rules, while the test set is used to apply the CPATs learned from the training set. We assessed PyEVOlve’s effectiveness by comparing the syntactic and semantic equivalence of Python-transformed codes to those that developers manually performed. We report the overall effectiveness by computing precision and recall.

1) Dataset: Dilhara et al. [21] studied 2,500 CPATs that occurred in 1000 top-rated ML repositories. The authors released CPAT dataset which is shown to be diverse with respect to size, frequency, authors, and projects. With a survey of 650 developers, the authors further confirm developers’ desire to have the identified CPATs automated in their code. Hence, we use the same dataset to perform cross validation. Section III-B1 provides further details on the CPAT dataset.

2) Experimental setup: Each CPAT, denoted as $CP_k$, consists of three or more instances of code changes, denoted as $\{k_1^{cp_k}, k_2^{cp_k}, ..., k_m^{cp_k}\}$. We split the change examples in each CPAT into training and test sets for every cross validation iteration. One iteration of our cross-validation process selects one instance (e.g., $k_1^{cp_k}$) from which to learn the initial transformation rule, and PyEVOlve then generates adapted rules to apply the CPAT to the test data (e.g., $\{k_2^{cp_k}, k_3^{cp_k}, ..., k_m^{cp_k}\}$). We identify one transformation trial as inferring a rule from $k_1^{cp_k}$ and applying it to $k_j^{cp_k}$ where $j \neq i$. Applying PyEVOlve to our dataset yielded over 40,000 trials, providing a high degree of confidence in the effectiveness of results.

To evaluate the effectiveness of transformations performed by PyEVOlve, we use the CPATs performed by developers as the ground-truth (the oracle). We use PyEVOlve to replicate CPATs in Oracle and compare them to the changes made by the original developer. We compute precision inside one iteration as the percentage of PyEVOlve-applied transformations (i.e., trials) that are correct (i.e., equivalent to the ground-truth). We compute recall within one iteration as the percentage of all transformations from the ground-truth that PyEVOlve was able to transform. We obtain mean values over all iterations once we have precision and recall for each iteration.

To determine if a PyEVOlve-applied transformation is correct, we have both a manual and an automated validation. Since we validate 40,000 transformation trials, manually checking them is tedious. Therefore, we chose a statistically significant (95% confidence level) random sample of transformation trials and manually validated the semantic correctness by comparing them to the human-performed transformations. For the automated validation, we were inspired by the steps that Noda et al. [47] used to evaluate automated bug patches. We denote the AST nodes that represent the $T_{LHS}^A$ and $T_{RHS}^A$ as $A_{LHS}$ and $A_{RHS}$, respectively. We judge that the transformed code is syntactically correct, if the transformed code: (i) contains all $N \in A_{RHS} - A_{LHS}$ (new nodes), (ii) does not contain all $N \in A_{LHS} - A_{RHS}$ (deleted nodes), and (iii) contains all $N \in A_{LHS} \cap A_{RHS}$ (unchanged nodes). Conditions (1) and (2) ensure that pattern code is successfully inserted and deleted. Condition (3) assures that no excessive changes are made.

3) Results: In our replication of actual CPATs from open-source projects, PyEVOlve performed 40,000 transformation trials in total, and we discovered that it achieved 97% precision and 94% recall. We manually validated a statistically significant sample of 381 instances for semantic validity, and it achieved 95% precision and 91% recall, which is slightly less than the automated validation. This is because the automated validation checks whether the CPAT nodes are fully transplanted but does not check their semantic validity. We primarily identified three causes for why PyEVOlve either failed to perform transformations at all or did so incorrectly.

1) Python union types: We employ type inference to obtain the type information of a Python program element at compile time, which is critical in mining semantically equivalent codes. Python allows a single variable to hold values of multiple types through the use of `Union` types. For example, the accumulator variable $n_{diff}$ in Listing 2 can be assigned to a `String` before the for loop, making $n_{diff}$ type of `Union[int, str]`. Algorithm 1 refers to node types for Guards with types to determine whether two `fgPDG` nodes are equal. Due to the fact that it searches for $n_{diff}$ of type `int` rather than `Union[int, str]`, PyEVOlve will not transform such cases.

2) Abstracting over one example: We use `PyInfer` to infer initial transformation rules, which abstract over one example, and might generate over-specialized transformation rules. For example, `m.atl.getM()` can be generalized either to `::[[v0]].getM()` or `::[[v0]].::[[v1]].getM()`. To decide this, other existing techniques use multiple examples, which may also result in over generalization. Our matching technique is dependent on the `fgPDG` nodes, and thus on the rule’s generalization when matching semantic variants in the target codes. Hence, PyEVOlve missed cases where it did not match the generalization of the rule. However, developers evolve Python in a Pythonic way [56], [57], and they evolve idioms mostly in the same way. Hence, despite learning from one example, PyEVOlve achieved a high precision and recall.

3) Semantically nonidentical instances in CPAT: PyType infers the type `Any` for program elements for which there is insufficient information to determine the correct type. This may result in unrelated examples being grouped into the same CPAT. PyEVOlve under-performs when we learn rules from such cases or apply other rules to them.

PyEVOlve achieves an overall precision of 97% and recall of 94% in the cross-validation evaluation, confirming its effectiveness in inferring rules: $T_{LHS}^A \rightarrow T_{RHS}^A$.

B. RQ2: What is the contribution of data- and control-flow aware rule inference to overall effectiveness?

To answer this, we use PyEVOlve to apply CPATs on new target sites, and then we perform a sensitivity analysis to check the impact of our novel contribution. We examined how many data-and control-flow variants PyEVOlve can automate that would be impossible to automate using prior tools.

1) Dataset: We chose highly rated, best-in-class 20 Python ML projects like TensorFlow, Pytorch, Keras, and Scikit-Learn, etc. Finding opportunities for applying CPATs in these high-quality, professionally-maintained projects shows that
We selected eight CPATs randomly from the list provided by Dilhara et al. [21] and used PYEVOLVE to apply them to new target sites in the top-rated projects. We only considered a subset of CPATs from Dilhara et al.’s list, as manual validation of the entire corpus was not possible. To determine if the modified code is a data or control-flow variant, we manually compared the modified code to the original input used by PYEVOLVE.

3) Results: Table II shows the evaluation results. Columns: (i) Transformation Rule—shows the initial transformation rule that the InferRule [33] generated, (ii) Guard—shows the Preds relevant to rules, (iii) N—shows the total number of CPATs instances transformed by PYEVOLVE, and (iv) V—shows the number of data or control-flow variants transformed by PYEVOLVE and (v) I—shows the number of variants that PYEVOLVE made possible as a percentage of all transformations.

As seen in Table II, running PYEVOLVE on 20 projects transformed a total of 185 instances. This shows that even in the top-rated projects, there are many instances to apply the best practices, and developers often overlook these instances.

Table II shows the data-or control flow variants for each rule. Of the 185 instances, 128 (70%) are data-flow or control-flow variations of the examples used to build the initial transformation rules. These instances would not be possible to transform without PYEVOLVE. For example, Listing 4 shows a target code that the PYEVOLVE was able to automate, which uses a for loop to compute the cumulative sum when np.cumsum should have been used instead. This is an unseen variant of the LHS of the rule given in row-3—Table II because: (i) there is an additional assignment statement, and (ii) it accumulates cur_len differently than the rule does. The target code was previously unseen during the rule inference, yet PYEVOLVE was able to successfully automate it. This shows the significant contribution of data-and control-flow aware rule inference, and how PYEVOLVE improves over the previous state-of-the-art for automating CPATs in Python ML systems.

Listing 4: The project Dialogue from Baidu uses a for loop to compute cumulative sum of seq_lens

```python
lod = []
cur_len = 0
seq_lens = [len(ids) for ids in data_ids]
for l in seq_lens:
    cur_len = cur_len + 1
    lod.append(cur_len)
```

70% of PYEVOLVE’s code transformations are data or control-flow variants that cannot be transformed using existing techniques, thus improving the state-of-the-art.

C. RQ3: How do developers find PYEvolve’s patches useful?

To determine how useful PYEVOLVE is for real-world developers, we automatically applied frequent CPATs from our corpus to well known open source projects using PYEVOLVE, and then submitted these patches as pull requests.

1) Dataset: We chose 35 best-in-class projects like Tensor-Flow, Keras, PyTorch, and Scikit_Learn, and apply the CPATs. Dilhara et al. [10] performed an empirical study on diverse corpus of CPATs and revealed the dominant CPAT kinds in Python systems. We chose the same set of CPATs which covers

<table>
<thead>
<tr>
<th>Transformation Rule</th>
<th>Guard</th>
<th>N</th>
<th>V</th>
<th>I (V/N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>:([v0]) = 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>for :([v1]) in :([v2]):</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:([v0]) = np.sum()</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:([v0]) == 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>for :([v1]) in :([v2]):</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:([v0]) = :([v1]).join(:([v2]))</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:([v3]) == ()</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>for :([v1]) in :([v2]):</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:([v0]) = np.cumsum(:([v2]))</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:([v0]) == 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>for :([v1]) in :([v2]):</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:([v0]) = :([v1]) + :([v0])</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:([v1]).append(:([v0]))</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:([v0]) == 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>for :([v1]) in :([v2]):</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:([v0]) = np.dot(:([v1]), :([v3]))</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>np.dot(:([v0]), :([v3]))</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| :([v0]) == :([v1]).apply(tf
| .batch_and_drop_remainder(:([v2]))) |       |   |   |         |
| if :([v2]) in :([v3]): |       |   |   |         |
|   :([v1]) = :([v3])[:([v2])].strip() |       |   |   |         |
| else:               |       |   |   |         |
|       :([v1]) = :([v4]) |       |   |   |         |
| sum (:([v1]))/len (:([v1])) |       |   |   |         |
| :([v0]) == 0       |       |   |   |         |
| for :([v1]), :([v2]) in zip(:([v3]), :([v4])) |       |   |   |         |
|       :([v0]) = np.dot(:([v3]), :([v4])) |       |   |   |         |
| :([v0]) == :([v1]) |       |   |   |         |
| for :([v1]) in :([v2]): |       |   |   |         |
|       :([v0]) = :([v1]) + :([v2]) |       |   |   |         |
| :([v0]) == 0       |       |   |   |         |
| for :([v1]) in :([v2]): |       |   |   |         |
|       :([v0]) = :([v1]) + :([v2]) |       |   |   |         |
| :([v0]) == :([v1]) |       |   |   |         |
| for :([v1]) in :([v2]): |       |   |   |         |
|       :([v0]) = :([v1]) + :([v2]) |       |   |   |         |

N: Transformed CPAT instances  V: Number of data and control variants of the original rule  I: number of variants as a percentage of N
all the kinds of CPATs revealed by the authors. Section III-B1 provides an explanation of the identified CPATs kinds.

2) Experimental setup: PYEVOLVE transformed 181 instances of CPATs that improve the performance and quality of the affected Python code. These patches updated 116 source code files and affected 1028 SLOC. After PYEVOLVE applied the CPATs in each project, we ran all the test cases of projects to ensure that the changes did not break the code. We then notified the open-source project maintainers via pull requests.

3) Results: Even the highly rated and optimized codes, like Keras, PyTorch, and TensorFlow, accepted our pull requests. We submitted 40 pull requests with 181 CPAT instances. At the time of writing the paper, 28 (70%) PRs containing 163 CPAT instances were accepted, 4 pull requests were rejected, and the rest are still under review.

Developers found that our changes either enhance performance or code quality, or both. A developer from the project Prosodics, an NLP meta-library, mentioned that “Well done, your changes are cleaner and either faster or equivalently faster.” Another developer from the ML library Transferlearning applauded the PR: “Your changes improve the efficiency. I did not pay attention to this efficiency before.” Developers confirmed that they were aware of the best practices, but a tool like PYEVOLVE is able to identify opportunities that even the experts missed. For example, a developer from the ML library Amn-benchmark remarked that “The changes look good, I am not sure why we didn’t write it that way before.” We submitted several CPAT instances where the developers should have used efficient ML libraries that were already being imported as dependencies to the project, but instead they were employing inefficient Python constructs. For example, in 83 (37%) instances, projects had libraries as project dependencies, and in 62 (28%) cases, they already had the library imported in the changed file, yet they still missed the opportunity to use the ML library at its fullest potential.

We discovered four major reasons for pull request rejections.

1) Some CPATs are dependent on matrix shape. The performance of matrix operations is affected by their shape. For example, we submitted a pull request to project the Mne-python to replace multiple calls to np.dot with np.linalg.multi_dot. The developers rejected this because while multi_dot improves performance on non-square matrices, it degrades performance on square matrices. To address this issue, we must update the Guards for this CPAT to account for the matrix shape. However, to the best of our knowledge, there are no tools that can infer matrix shapes in Python at compile time.

2) Dependencies on Hardware. ML library optimizations depend on hardware platforms. For example, Pytorch is optimized for GPU use, whereas NumPy is not. A pull request submitted to Pytorch to replace a for loop with NumPy APIs was rejected because NumPy is not optimized for GPUs.

3) Functions are already optimized with NumBa. NumBa [39] is a library that translates annotated Python code to optimized machine code at runtime. Because the optimization occurs at runtime, the code is faster than it looks at compile time, even if the code employs inefficient constructs like Python loops. Our patch submitted to Pyndescent was rejected because their code was already optimized at runtime.

4) Deprecated code no longer updated. Developer from the project Basenji rejected our patch because it changed a deprecated function that they will remove in a later release.

PYEVOLVE transformed 181 CPAT instances, of which 163 have been approved as of writing.

D. Threats To Validity

1) Internal Validity: Does our tool produce valid results? We thoroughly evaluated the accuracy of the transformations produced by PYEVOLVE. To understand if the inferred rules can be trusted, the authors both automatically and manually validate the transformations to identify non-conforming ones. Furthermore, we develop a comprehensive setup that semi-automatically validates the application of transformation rules for a large and diverse set of CPATs.

2) External Validity: Do our results generalize? Although PYEVOLVE is effective on the evaluated data-set, it may not perform well on other subjects. To address this issue, we chose a large dataset that previous researchers [21] used. It covers different scenarios and has been shown to be diverse in terms of frequency, size, authors, and projects. All our subjects are open-source, and we have yet to evaluate proprietary codes.

We performed manual steps in RQ2 (identifying the variants) and RQ3 (submitting patches). These manual steps prevented us from using all the CPATs for the evaluation of RQ2 and RQ3, so we only used a subset of CPATs. This could impact the generalizability to other CPAT kinds. To mitigate this, we used a randomly selected subset of CPATs.

3) Verifiability: The collected data, source code, and executable of PYEVOLVE are publicly available [44].

V. DISCUSSION

1) Safety and soundness of our approach: The “Transformation by Example” systems are not intended to replace developers, nor are they designed to be sound [89]. Our PYEVOLVE (like other “Transformation by Example” systems) sits in the middle between a regex-based find-replace tool and a refactoring tool. It has the expressivity of a find-and-replace tool, while being syntax-aware. In contrast, a refactoring tool or a compiler optimization tool have hard-coded, task-specific rules that make them safer to use, but they are expensive to develop. Determining the safety of such transformations require deep analysis of the context, hence we do not recommend blindly accepting patches from PYEVOLVE. However, in our workflow, we trade-off safety for broader applicability by relying on the developer’s insight in determining whether it is safe to perform the transformation. In their study, Ketkar et al. [33] observed that rules must sometimes be manually vetted to ensure their safety and soundness. If this is the case, our human-comprehensible rules make the process easier. In our empirical evaluation, PYEVOLVE achieved 97% precision and it did not require any intervention from us to achieve this.
VI. RELATED WORK

We group the related work in two areas: (i) inferring and applying changes, and (ii) Python code idioms.

**Inferring rules and applying changes:** Researchers have developed an array of advanced program transformation systems that automatically generate programs according to given input-output examples, so called “Transformation by Example”. This technique has been used in a variety of applications, including (i) Java Type Migration [33], (ii) API migration [1], [5], [15], [19], [54], (iii) String manipulation [6], [7], and (iv) Data Structure Transformation [3]. LASE [53], REFAZER [35], SPDIFF [4], TCINFER [33], AppEvolve [15], and APIFIX [5] learn from multiple examples, determining how to abstract the adaptations based on their commonalities and differences, whereas SYDIT [49], A4 [54] and MEDITOR [19] abstract over individual examples. These works aim to generate a properly generalized rule by varying the number of examples and its properties. However, they do not account for previously unseen data- and control-flow variations of the examples, resulting in low recall.

Several tools have been developed to address control flow variants to some extent, with SPINNER [40] being the most well-known among them. SPINNER [40] supports some control variations through the use of the “...” operator to represent any arbitrary number of unrelated/noise statements between statements in the rule. However, it learns the potential locations to insert “...” based on multiple input examples. This can cause problems if the input examples do not cover all potential locations for arbitrary statements, making it unable to handle many unseen variants. Moreover, tools such as TCINFER [33], LASE [53] decompose code changes into edit actions, allowing them to handle some control-flow variations. However, they do not consider control-flow constraints, causing potential incorrect changes. Additionally, none of these tools are capable of addressing data-flow variants. Therefore, we suggest a novel graph-based technique that captures both data- and control-flow unseen variants, then adapts the generated rules for the new contexts and correctly handles complex CPATs.

**Studies involving Python idioms:** Researchers studied Python idioms and how they were used in Python systems. Phan-udom et al. [55] recommend 58 non-idiomatic and 55 idiomatic changes. Alexandru et al. [56] present a non-exhaustive list of Python idioms gleaned from a developer survey. Sakulniwat et al. [57] studied the evolution of Python with statements over time, while Wang et al. [62] studied Python code smells. However, none of these fully automate code transformations. Using Pevolve, we can infer the rules and fully automate all these idioms. Recently, Zhang et al. [23] employed AST rewriting to automatically refactor nine Python idioms. However, they must hard-code the transformations. In contrast, Pevolve mines and automates idioms, allowing for future-proof handling of emerging idioms. This will make it much easier for Python-ML developers—the dominant ecosystem in Python [11]—to keep up with the rapidly advancing ML techniques.

VII. CONCLUSIONS AND FUTURE WORK

Despite Python’s and ML’s meteoric rise [58]–[61], support for automated code evolution is still in its infancy. To advance the science and tooling for automating code changes in Python, we built Pevolve, which infers transformation rules and then applies them. Unlike existing tools that are hard-coded for specific transformations, Pevolve automates a wide range of best practices using “Transformation by Example”.

Our thorough empirical evaluation of a diverse, representative corpus of 40,000 transformation trials from real-world projects shows that Pevolve is effective. It has a 97% precision and 94% recall, and 70% of Pevolve transformations would be impossible to automate without our technique. Developers accepted 90% of the 181 CPATs that Pevolve produced, and their feedback shows Pevolve’s usefulness.

We anticipate these future advancements for Pevolve:

1) **Version Awareness:** CPATs use language and library constructs that continuously evolve based on their release versions. We will extend the transformation rules to be version aware.

2) **Community repository of transformation rules:** The rules that Pevolve learns may change over time; some rules may become obsolete while new rules are emerging. For this reason, the researchers have called for a community-maintained central database of CPATs [34] and their respective rules [2], [33] that would need to be properly versioned, maintained, and evolved. The likelihood of having data-flow or control-flow variations, however, rises as the CPAT gets bigger. If we add more rules to the database in order to represent all variants, it would significantly increase the size of the database and, in turn, expand the search space with significant slow downs. Our novel approach uses a single rule that transforms all other data- and control-flow variants, thus it significantly reduces the number of rules while increasing the speed.

3) **Expanding to other domains and languages:** As new idioms emerge, Pevolve is future-proof and will continue to handle new idioms. We think Pevolve will aid ML developers in keeping up with the rapidly advancing ML technologies. While the corpus of programs we use in this paper is ML systems, Pevolve is readily applicable to identify, recommend, and automate CPATs for any kind of Python software system. Moreover, the core ideas presented here are easily transferable to other programming languages. Given how well they work even for a dynamically-typed language like Python, we think they would work even better for statically-typed languages. By making our design, implementation, and datasets available to the community [44], we hope this would inspire many others to advance the field.

VIII. ACKNOWLEDGEMENTS

We would like to thank the CUPLV group at CU Boulder, and the ICSE reviewers for their insightful and constructive feedback for improving the paper. This research was partially funded through the NSF grants CNS-1941898, CNS-2213763, and the Industry-University Cooperative Research Center on Pervasive Personalized Intelligence.