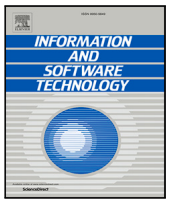
Information and Software Technology 178 (2025) 107592

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



Detecting and Explaining Python Name Errors

Jiawei Wang [a,*], Li Li [b], Kui Liu [c], Xiaoning Du [d]

[a] Faculty of Information Technology, Monash University, Melbourne, 3168, Australia
[b] School of Software, Beihang University, Beijing, 100191, China
[c] Software Engineering Application Technology Lab, Huawei, Hangzhou, China
[d] Faculty of Information Technology, Monash University, Melbourne, 3800, Australia

ARTICLE INFO

Keywords:
Static analysis
Python code
Tools
Empirical studies

ABSTRACT

Python has become one of the most popular programming languages nowadays but has not received enough attention from the software engineering community. Many errors, either fixed or not yet, have been scattered in the lifetime of Python projects, including popular Python libraries that have already been reused. NameError is among one of those errors that are widespread in the Python community, as confirmed in our empirical study. Yet, our community has not put effort into helping developers mitigate its introductions. To fill this gap, we propose in this work a static analysis-based approach called *DENE* (short for **D**etecting and **E**xplaining **N**ame **E**rrors) to automatically detect and explain name errors in Python projects. To this end, *DENE* builds control-flow graphs for Python projects and leverages a scope-aware reaching definition analysis to locate identifiers that may cause name errors at runtime and report their locations. Experimental results on carefully crafted ground truth demonstrate that *DENE* is effective in detecting name errors in real-world Python projects. The results also confirm that unknown name errors are still widely presented in popular Python projects and libraries, and the outputs of *DENE* can indeed help developers understand why the name errors are flagged as such.

1. Introduction

Python is an interpreted high-level, general-purpose programming language with design philosophies emphasizing features such as code readability, execution efficiency, etc. With the recent fast development of data science (machine learning and deep learning in particular), Python has become one of the most popular programming languages nowadays. However, the fast-increasing number of Python projects has not received enough attention from the research community. There are not many advanced approaches proposed to help developers implement high-quality Python code. Subsequently, a large number of low-quality Python projects or libraries have been scattered in the ecosystem. Indeed, as empirically revealed by Wang et al. [1], many of their studied Python code snippets (within a set of publicly released Jupyter notebooks) contain poor quality code such as having unused variables or accessing deprecated functions. The authors further argue that there is a strong need to programmatically analyze Python code to ensure the reliability of publicly released Python code.

Among many of the issues impacting the reliability of Python projects, NameError is one of the most popular issues that will be thrown if a given name (e.g., of a variable or a method) cannot be correctly interpreted by Python. The corresponding error message will be "*name is not defined*". NameError is often considered not too complicated to be fixed by developers, especially when they are able to reproduce and debug the error. However, the fact that many Python projects (even popular ones with hundreds of thousands of stars on GitHub) have been reported as containing name errors shows that name errors are non-trivial to mitigate. On the one hand, because Python is an interpreted language that does not involve a compilation process, certain name errors (even the naive ones) could persist in the (inadequately tested) projects for a long time and will only be revealed in practice (at the production stage) when their corresponding code is reached. On the other hand, some name errors could be complex to fix. They may involve multiple execution paths in numerous methods across different modules and could even involve external libraries. For example, as shown in the discussion history of issue *#19340* [2] on the GitHub repository of *pandas*, one of the most popular libraries in Python, the "`_converter` is not defined" NameError has been widely reported by users and confirmed by the *pandas* maintainers as an real issue of *pandas*. The maintainers also mention when discussing the issue

* Corresponding author.
E-mail address: jiawei.wang1@monash.edu (J. Wang).

https://doi.org/10.1016/j.infsof.2024.107592
Received 18 May 2024; Received in revised form 29 September 2024; Accepted 30 September 2024
Available online 11 October 2024

Table 1
A summary of top 20 popular libraries and the number of issues queried using "NameError" as the keyword[a].

Name	# Commits	# Stars	# Forks	# Dependents	# Contributors	# Files	# Test Scripts	# Issue reports
requests	6.28k	45.8k	8.4k	1.01M	593	35	9	5
Flask	4.44k	56.3k	14.5	808k	627	75	29	2
sphinx	18.06k	4.1k	1.5k	144k	574	548	89	9
numpy	31.40k	17.9k	5.7k	756k	1,175	581	132	14
Pillow	12.24k	8.8k	1.7k	521k	328	275	119	2
pytest	14.17k	7.6k	1.8k	334k	632	240	994	7
Jinja	2.62k	7.9k	1.4k	–	253	60	19	5
matplotlib	40.19k	14k	6k	398k	1,102	887	81	8
pandas	28.94k	30.7k	12.9k	486k	2,402	1,359	815	13
pyyaml	0.59k	1.6k	337k	338	35	82	0	0
coverage	5.07k	1.7k	229	152k	94	148	30	2
scikit-learn	29.46k	46.8	21.8	246k	2,068	880	212	2
dateutil	1.57k	1.6k	376	530k	113	38	19	1
six	0.53k	815	219	927k	56	4	3	1
urllib3	4.04k	2.7k	871	615k	249	74	32	1
clic k	1.98	11.2k	1.2k	563k	282	63	13	2
Werkzeug	4.80k	5.8k	1.6k	564k	382	133	24	3
aiohttp	8.17k	11.5k	1.6k	103k	557	130	60	1
tqdm	1.99k	19.1k	987	189k	104	67	2	3
sqlalchemy	18.24k	4.1k	657	290k	434	591	4	8

[a] We would like to remind the readers that NameErrors (threw at runtime) should not be confused with naming issues that aim at finding better variable names (e.g., to follow a certain naming convention). For instance, He et al. [4] have offered our community a learning-based technique called Namer to spot naming issues (e.g., variable num_or_processors should be named as num_of_processors). Although such changes could (accidentally) fix NameErrors, it essentially targets a different problem compared with ours and most likely will introduce more NameErrors at runtime.

report that this issue is not easy to reproduce. Similar discussions can also be found on the popular StackOverflow website [3].

The aforementioned evidence strongly suggests that there is a need to invent automated approaches for helping Python developers mitigate name errors in their projects. To handle the limitation of insufficient tests (also due to no compilation involved), we argue that those automated approaches need to **(1) be able to detect name errors statically without executing the code**. To cope with complex name errors, we argue that the automated approaches should also **(2) be able to help in explaining why the detected name errors are flagged as such**. Unfortunately, despite the fact that more and more Python-focused research studies are presented in the community, none of the existing approaches be leveraged to achieve the aforementioned objectives, i.e., automatically detecting and explaining name errors in Python projects. Type checking systems in theory should identify all instances of the same variable including both usages and their definitions. However, as shown in the experimental results, industry-leading products such as *pyre* and *pytype* are still incapable of detecting this execution error. [1] Though the error occurrence reason looks simple that the identifier has not been defined at a program point. However, as Python code can be executed without compilation, making the name error can be potentially hidden behind the successful execution of insufficient test inputs. Moreover, Python def-use information extraction can be challenging as it offers various syntax sugars to reduce development efforts while increasing the difficulty of data flow information (such as one-line for-loop statement in container comprehension). In addition, to the best our knowledge, there is not existing work to explore the sound def-use relations of Python programs except Wang et al.'s work [5] on field-sensitive variable def-use relations across Python based Jupyter notebooks, which ignores the nested structure of Python scopes.

In fact, it has been pointed out that existing static analysis techniques for Java and C designed for the past decades are difficult to transfer for due to unique languages features and lack of tools for Python CFG IR extraction [6]. To this end, we propose to fill this research gap by introducing to the research community a prototype tool called *DENE*. Given a Python project, *DENE* first scans the code to simplify some complicated syntax that may introduce difficulties to static analysis. It then builds scope-enhanced control-flow graphs for the project to support subsequent reaching definition analysis for identifying name errors. For the detected errors, *DENE* goes one step further to also record the evidence demonstrating why they are flagged as such. Experimental results show that *DENE* is effective in detecting name errors in real-world Python projects, including popular Python libraries, and the explanation module is also useful for helping developers understand the reason why there are name errors. To summarize, this paper makes the following major contributions.

- We conduct a preliminary study about popular Python libraries and experimentally demonstrate that the most popular Python libraries, although contributed and maintained by hundreds of contributors, have been reported to contain name errors.
- We design and implement a prototype tool called *DENE*, which performs scope-aware reaching definition analysis to statically detect name errors and further records how the errors can be triggered to assist developers in understanding why the errors are reported as such.
- We demonstrate the effectiveness and usefulness of *DENE* through large-scale field experiments over carefully crafted ground-truth datasets, as well as popular Python libraries and open-source GitHub projects and a user study.

Open science. Source code and datasets are all available in our artifact package (https://github.com/DENE-dev/dene-dev) and will be made publicly accessible on acceptance.

2. Preliminary study

As experimentally disclosed by Pimentel et al. [7], their large-scale study of over 1.1 million Jupyter notebooks empirically uncovered that NameError is the second most common exception thrown by the Python scripts recorded in the notebooks. The most common exception type is ImportError, which occurs when an import statement fails to successfully import the specified module. This exception is often linked to the execution environment and is considered a significant issue. Various automated approaches have been proposed by researchers to help Python developers address it [8–11]. However, another important

[1] The number of dependent projects for Jinja is unavailable from its GitHub page.

exception that warrants attention is NameError. This error has been extensively studied in the context of computer science education. For instance, NameError is reported as the primary error category encountered by high school students [12] and is a frequent syntactic error among novice programmers [13]. Despite its prevalence and impact on learning, to the best of our knowledge, NameError has not yet been specifically targeted by automated approaches. Given its significance, we believe that developing automated methods to mitigate NameError is crucial for improving both educational outcomes and the overall development experience. Nonetheless, to the best of our knowledge, the issue has not been targeted yet, we believe it is also an important error type that should be mitigated through automated approaches.

To better motivate the need to invent promising approaches for automatically detecting and explaining name errors in Python projects, we conduct a preliminary study aiming at understanding the prevalence of NameError-related issues reported to open-source Python projects as the NameError can propagate from the open-source tools to client programs. For instance, the name error in the motivating example (Fig. 1) will not be triggered until developers invoked the methods after defining an object of the class in their own application. To this end, we resort to the command line GitHub client tool named *gh* [14] to crawl the source code repositories and their corresponding issue reports of the top-20 Python libraries [15] on GitHub. Note that we end up checking more than 20 libraries because some libraries in the original top-20 list are excluded as they do not have their source code hosted on GitHub or do not allow users to add issue reports (the feature is explicitly disabled).

Table 1 illustrates the final 20 libraries considered in this preliminary study. All of them generally come with a large number of files (i.e., could be as many as 1359 Python files) and commits (i.e., could be as many as 50 thousand), have been stared and forked by many other developers (cf. columns 3–4). Those libraries are all developed through collaborative efforts with at least 35 developers. The *scikit-learn* project has involved over 2000 developers. All of the aforementioned shreds of evidence confirm that the selected libraries are indeed popular ones. As indicated in column 6, those libraries have also been used by many other projects, ranging from 300+ to 1,000,000+ projects. This evidence, while confirming the popularity of the selected libraries, further suggests that the quality of these libraries is also important (a single issue in a library could be propagated to thousands of its dependents). Unfortunately, as indicated by column "# Test Scripts", these libraries have only been equipped with a limited number of test scripts (often much smaller than the number of Python files), resulting in inadequate test coverage of the libraries and thereby leaving a lot of quality issues hidden in the released library versions. Indeed, as suggested by the last column, 19 out of the 20 selected libraries have been reported containing NameError issues.[2] The fact that even the most popular Python projects (with many contributors and a large number of commits, stars, and developer forks), which have already been leveraged by thousands of other projects, are insufficiently tested and contain quite a number of NameError issues suggests that there is a strong need to invent automated approaches to help Python developers detect and mitigate name errors in their Python projects.

We now present a concrete example identified during our preliminary study to help readers better understand this work. The example is shown in Fig. 1, which is extracted from the *tqdm* library that is frequently leveraged to implement extensible progress bars. This code snippet has been reported as an issue (ID 559[3]) containing a potential name error indicating that the name `IntProgress` is not defined (cf. line 20). This issue will be triggered if the `tqdm_notebook` class is initialized after the module is imported.

When importing the module, its body code will be executed to bind the defined names in the local scope. In particular, the code in lines 2–14 will be fully executed at this stage, while the definition of class and methods in lines 16-24 will be recorded. In the motivating example, the definition of `IntProgress` is done at this stage. After that, when class `tqdm_notebook` is initialized by the client code, the construction method (i.e., `__init__` at line 23) will directly trigger the execution of the `status_printer` method (cf. line 24), and subsequently trigger the accesses of name `IntProgress` (cf. lines 19 and 21). Unfortunately, under certain conditions, the `IntProgress` could not be defined (at line 14) during the module importing phase. Subsequently, the aforementioned reference of `IntProgress` (cf. lines 19 and 21) will result in a name error, i.e., `IntProgress` is not defined.

Conventionally, this sort of code defect can be discovered by type-checking tools. However, existing tools such as *pyre* by Meta and *pytype* by Google, are still incapable of detecting this execution error. According to the documentation and code implementation, *pyre* lacks support for nested scopes and *pytype* does not analyze path conditions. Moreover, both tools encounter issues when dealing with complex syntax sugars such as call chains or container access. The root cause of this type of bug is the late name-binding strategy in Python language design. In static-typed Java or C++, the strategy of name binding is called early binding. When variable types are determined at compile-time, and the compiler checks for type correctness and existence of variables before generating the executable code. If a variable or a method name does not exist, the compile-time error is reported, preventing the program from running. In contrast, Python is a dynamically typed language, and its name resolutions are determined at runtime.

Summary

NameError is one of the most common exceptions happening in Python projects. As empirically disclosed in this work, even the most popular Python libraries have propagated various such exceptions to their client dependents, as explicitly reported by the developers. This evidence further suggests that the current regression tests applied to those projects are not yet adequate, which is also known to be non-trivial to achieve full coverage through dynamic testing. Therefore, we argue that there is a need to complement such tests through static analysis approaches, allowing continuous analyses towards detecting and mitigating potential name errors.

3. Methodology

In this section, we first present the challenges in the code analysis for detecting name errors in Python. We then provide our proposed approach *DENE* to detect and explain Python name errors, which consists of three processes as shown in Fig. 2: ❶ Syntax Simplification, ❷ Scope-enhanced Control Flow Graphs (CFGs) Construction, ❸ Scope-aware Reaching Definition Analysis.

3.1. Syntax simplification

Python language is easy to use and convenient to build prototypes for new blooming ideas as it offers a wide range of unique features for practitioners. However, these features (e.g., the syntactic sugar[4]) may hinder the control flow analysis for Python code in a traditional way and hence may lead to unsound static analysis results.

Fig. 3 summarizes four of such cases. Case (1) concerns the `container comprehensions feature` [16] in Python, `for`-loop and conditional statements are wrapped into one-line code (cf., line 2), which makes it difficult to straightforwardly extract the included control flows. Case (2) and Case (3) respectively concern the nested

[2] Given a repository, we go through each of its issue reports to check if it contains *nameerror* or *'.*' is not defined* keywords. If so, we will count it as a NameError issue.

[3] https://github.com/tqdm/tqdm/issues/559

[4] https://thereaderwiki.com/en/Syntactic_sugar

```
1  # tqdm/_tqdm_notebook.py
2  try:
3   if IPY == 32:
4      from IPython.html.widgets import IntProgress, HBox, HTML
5      IPY = 3
6   else:
7      from ipywidgets import IntProgress, HBox, HTML
8  except ImportError:
9   try:
10     from IPython.html.widgets import IntProgressWidget as
       IntProgress
11     ...
12     IPY = 2
13  except ImportError:
14     IPY = 0
15
16 class tqdm_notebook(tqdm):
17  def status_printer(_, total=None, desc=None):
18      if total:
19         pbar = IntProgress(min=0, max=total)
20      else:
21         pbar = IntProgress(min=0, max=1)
22      ...
23  def __init__():
24      self.status_printer(...)
```

Fig. 1. An example of Python code containing a name error. The code snippets are extracted (and simplified) from the Python library *tqdm*.

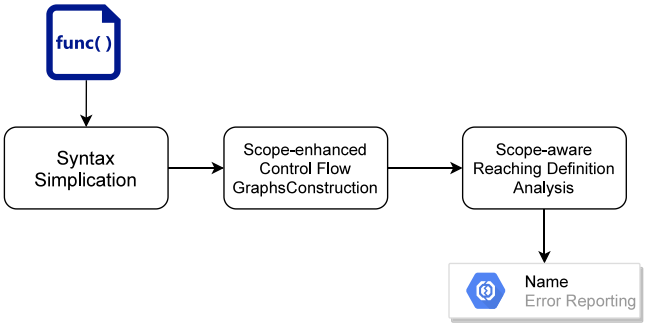


Fig. 2. The overview of *DENE*.

function calls (cf., line 11) and subscription assignment (cf. line 16) that will also make it complicated to extract data dependencies, when statically interpreting the code. Finally, the last case (Case (4)) concerns the famous lambda expression assignment (cf. line 27), which should not be treated as a regular variable assignment as it is semantically equivalent to a standalone function definition. These wrapped code conventions, while simplifying the code-writing tasks for developers, do hinder static analysis approaches to examine the code's control flows and data dependencies.

To mitigate the impact of such complicated Python features and towards achieving more complete and effective control flow analysis for Python code, we propose, as the first step of *DENE*, to simplify the wrapped code fragments by expending the syntactic sugar into flat code fragments with traditional constructs, which should be semantically equivalent to the original code fragments. Fig. 3 further illustrates (e.g., marked -> Unfolded Code) the de-sugared results, demonstrating how the complicated syntax is simplified.

3.2. Scope-enhanced control flow graphs construction

Detecting Python name errors relies on checking name usages and their impacted scope in terms of the corresponding control flows, which plays a critical role in *DENE*'s data flow analysis. To this end, the second module of *DENE* aims at constructing control-flow graphs for the Python code simplified by the first module. To the best of our knowledge, our community has not yet achieved a sound and robust control flow graph generation approach. The lack of tools converting source code to CFGs leaves the analysis techniques and algorithms designed over the past decades inapplicable to Python code analysis [6]. For instance, the CFG module introduced by FuzzyBook [17] handles 13 of all statements and 5 control statements. Control statements need to be specifically represented on CFGs because they may change the code's execution flows. For example, the `if` control flow statement will yield two branches representing two independent execution flows. As stated by the official Python documentation, there are eight control flow statements [18] (i.e., `for`, `if`, `while`, `break`, `continue`, `try`, `ExceptHandler` and `with`). In addition, the Python library *python-graphs* by Google Research [19] deals with 13 of all statements and 7 control statements. We believe these existing tools are not competent for program analysis for large Python real-world projects. Therefore, we decide to implement the control-flow graph generation module from scratch with the aim of dealing with all the statements.

Considering the complexity of CFGs, *DENE* builds the basic CFG unit at the function level. The detailed construction process recursively visits all the statement nodes in the abstract syntax trees (ASTs) of given functions. Each of the code blocks represents the sequence of statements without any branch.

Once a control statement is met, we will create additional nodes to place the target code blocks that this control statement jumps to. For those statements that direct the program to the exit (e.g. raise statement), the block will be linked to a special exit node. The links between these code blocks are based on the control flow semantics. In this work, we extend beyond *FuzzyBook* and *python-graphs* by carefully taking into account all possible types of control flow statements as

```
# Case (1): Normal Container Comprehension
x = [ int(a) for x in numbers if len(x) != 0 ]
# -> Unfolded Code
x = []
for x in numbers:
    if len(x) != 0:
        x.append(int(a))
# Case (2): Nested Function Call
s = fun(fun(), name='func')
# -> Unfolded Code
_val = fun()
s = fun(_val, name='func')
# Case (3): Subscription Assignment
s = [ str(a) for a in lst if a%2 == 0 ][0:3]
# -> Unfolded Code
_s = [ str(a) for a in lst if a%2 == 0 ]
s = _s[0:3]
# Case (4): Lambda Expression Assignment
fun = lambda x:x**2
# -> Unfolded Code
def fun(x):
    return x**2
```

Fig. 3. Examples of unfolding wrapped code fragments at the syntax level.

well as all the remaining statements to obtain the precise identifier usage analysis. For instance, in both Fuzzy's work and python-graph's implementation, the with statement is not considered, whereas the statement is frequently used by Python programs.

Furthermore, in addition to building CFGs for traditionally defined functions (e.g., directly under modules or within classes), there are three types of specific functions that also need to be considered. The first type is related to *module's static code*, which is directly written in Python modules. Technically speaking, this code is not within any function. However, they will be executed before any of the other functions or classes are defined in the same module. The corresponding names declared in this code block will also be visible to the functions or classes declared in the module. The second type is related to *class's static code*, which is directly written in Python classes. Similar to that of *module's static code*, *class's static code* will also be executed before (and the defined names will also be visible to) the class's other functions. The third type is *function's enclosing functions*, which are actual Python functions defined within another function. The names involved in the outer functions are visible to the enclosing functions, but not the other way around. Because these special functions will introduce visibility changes of names, they have to be handled in order to support sound name error detection. In this work, we have carefully taken into account all three types of special functions when constructing CFGs.

After modeling the aforementioned special functions, *DENE* goes one step deeper to further record the visibilities among different CFGs within the same module, with respect to the so-called LEGB (short for Local, Enclosing, Global, Built-in) rule [20]. The LEGB rule essentially helps in resolving names in given Python programs. In this work, we enhance the existing CFGs by connecting them through the following rules: (1) the names defined in the module's static code is visible to all the classes/functions defined in the module, (2) the names defined in the class's static code is visible to all the functions defined in the class, and (3) the names defined in a function is visible to all its enclosed functions. Taking Fig. 1 again as an example, after parsing the code, *DENE* will build four CFGs (i.e., *_tqdm_notebook module static code*, *tqdm_notebook class static code*,[5] *_init_*, and *status_printer*), as shown in Fig. 4. The CFGs are further connected through dotted single arrow lines (in blue) based on their scope visibilities. The generated scope-enhanced CFGs indicates that all the names declared in *_tqdm_notebook module static code* will be visible (and hence accessible) to both *_init_* and *status_printer* functions. However, the names declared in function *_init_* will not be visible to function *status_printer*, and vice versa.

3.3. Scope-aware reaching definition analysis

Based on the constructed scope-enhanced CFGs, the last module aims at traversing them to pinpoint name errors, i.e., name identifiers are used but not defined. Specifically, for each name identifier involved in a CFG, the idea of the last module is to backtrace all the possible execution paths (with Python scope considered) that can reach the identifier to check if it is defined in all the involved paths. We define the NameError in Python as follows.

Definition 1 (*NameError*)**.** Given a name n referenced in the basic block[6] B_j of a control-flow graph (CFG) g, there exists an execution path $P = B_0 ... B_j$ in g, such that $\forall B_i \in P, n \notin gen(B_i)$, or $\exists B_k, n \in del(B_k)$ but $\forall k \leq m < j, n \notin gen(B_m)$, where $gen(B_i)$ and $use(B_i)$ and $del(B_i)$ represents the set of names that are generated, used and deleted by block B_i, respectively. The path P records a sequence of blocks that represent an execution path starting from the entry block (i.e., B_0) in g to the current block B_j.

In the next, we present a flow-sensitive and path-sensitive analysis approach to achieve this purpose, which includes three steps: (1) Gen, use and del set construction, (2) backward path-sensitive tracking, and (3) scope analysis.

[5] *tqdm_notebook class static code* is kept as empty as there is no code defined as such.

[6] Also known as a node in the control flow graph. Often, a code block represents a sequence of statements without any branch.

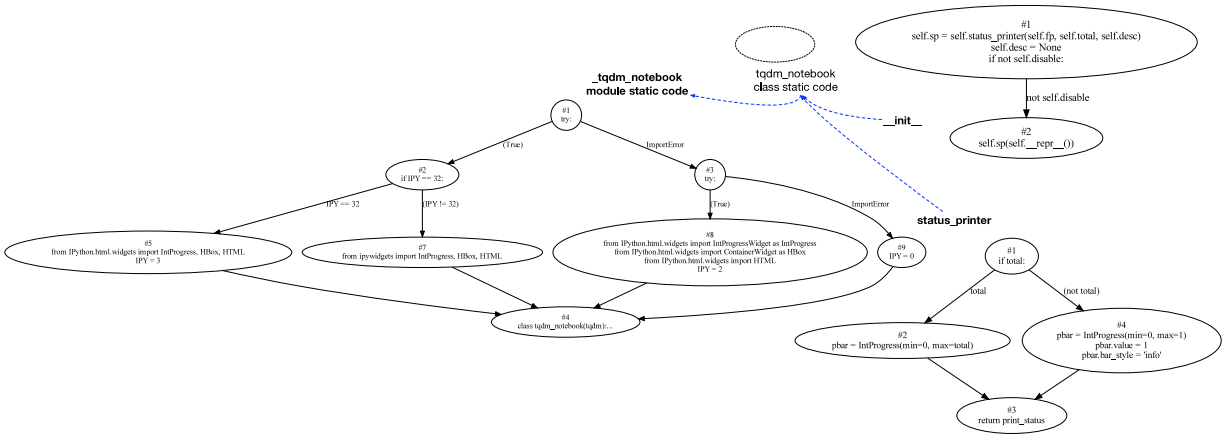


Fig. 4. The scope-enhanced CFGs constructed for the motivating example (Fig. 1). The dotted single arrow lines record the scope visibility of the constructed graphs (Names in targeted graphs are visible to the source graph and its succeeding graphs).

Step 1: Gen, Use and Del Set Construction. For all the previously constructed CFGs, this step goes through each of their blocks to generate the set of produced name identifiers and the set of consumed name identifiers. When identifying the produced identifiers, in addition to considering the traditional assignment statements (i.e. variable is stored), we also consider the following three cases to form the identifier generation set. The three cases are (1) function or class names declared in the block (Python allows to define functions or classes in a given function), (2) names introduced in import statements (import statements can also happen in the middle of a given function's code), (3) names introduced in *as expressions*. For example, f is considered as a generated name for the following statement *"with open("input.txt") as f"*. All the other name identifiers that are appeared in the block but are not considered as generated ones will be added to the consumed set (except for the formal parameters involved in function definition statements). Moreover, all the names processed by delete statement will be added to the del set.

Step 2: Backward Path-Sensitive Tracking. After generating the three sets required for all the blocks in all the constructed CFGs, *DENE* performs backward path-sensitive tracking for each block to check if the definitions of its consumed identifiers can be reached backwardly in the corresponding CFG. More specifically, for a given name identifier n in block B_i's consumed set $n \in Use(B_i)$, *DENE* first needs to identify all the paths that can reach backwardly from block B_i to the entry block in the CFG. Subsequently, *DENE* checks if n is defined in all the identified paths. If there is a path P showing n is not defined, we consider there is a potential name error happening in B_i due to path P, i.e., $\exists n$ and $\exists P = \{B_0, \ldots, B_m, \ldots, B_i\}, n \in use(B_i) \wedge n \notin gen(B_m), B_m \in P$ or there exists B_m such that $n \in del(B_m)$ and n is contained in a del set along this path before it is included in a gen set.

In this step, *DENE* repeats the aforementioned process for all the blocks of all the constructed CFGs to recognize undefined names (described in algorithm 1). The number of visited blocks for a given Python project could be huge and hence time-consuming to be fully visited. Yet, each block may involve a large number of execution paths that may further explode the backward tracking space.

To mitigate this, so as to achieve a realistic time performance for analyzing Python projects, we propose to utilize the concept of dominance properties [21] to reduce the search space. Notationally, we define there is a **dominant relationship** between two blocks (B_x, B_y) in a given CFG as $B_x \gg B_y$ (reading as B_x dominates B_y), meaning every path from entry node to B_y must go through B_x. In practice, when conducting the backward path-sensitive tracking for a given block, *DENE* will first generate the block's dominators and directly check if the block's used name identifiers are defined (or used[7]) in its any dominator (in backward order as well). If so, the name identifiers will be ignored from further analysis as they will not introduce name errors (c.f. line 18–19 of algorithm 1). Since the majority of name identifiers should be defined, this process will significantly reduce the number of visited blocks and paths. In this work, the dominant relationships are iteratively computed [21,22] by satisfying the following data flow equation [22],

$$Dom(B_n) = \begin{cases} \{B_n\} & B_n = B_{entry} \\ \{B_n\} \cup (\cap_{B_p \in preds(B_n)} Dom(B_p)) & B_n \neq B_{entry} \end{cases} \quad (1)$$

where B_{entry} is the entry node in the given CFG and B_p is one of the predecessors of given node B_n. In practice, since there is no need to visit the block itself, we will remove it from its dominators as by definition every node dominates itself.

Step 3: Scope Analysis. Now that we have detailed our methodology of identifying undefined names in one control flow graph using path constraints based backward tracking, which should have confirmed that the majority of name identifiers will not cause name errors. For the remaining name identifiers, for which *DENE* cannot locate their definitions within the CFG (i.e., in the local scope), we need to go one step further to check if the identifiers are defined the enclosing scopes. Recall that Python relies on the so-called LEGB rule to resolve undefined identifiers, for which the scope visibility has been recorded when constructing the scope-enhanced CFGs (cf. Section 3.2).

In this work, when finishing the visit of a CFG, *DENE* steps into its enclosed CFGs to further search for definitions of the unresolved identifiers. In other words, the name error candidates from the enclosed scope are taken to the current scope for further analysis (cf. line 17 in algorithm 1).

For example, when jumping out from the CFG of function *status_printer* in Fig. 4, *DENE* will highlight that the *IntProgress* identifier is not yet defined. Following the scope-enhanced CFGs, *DENE* will then visit the function's class scope (i.e., the *tqdm_notebook class static code*, which is empty in this example) and then the module scope (i.e., the *_tqdm_notebook module static code*). Unfortunately, at this stage, the *IntProgress* identifier still cannot be fully resolved because it is not defined in the following path $\{\#1, \#3, \#9, \#4\}$.

After that, *DENE* further checks whether *IntProgress* is one of Python's reserved keywords (i.e., the Built-in scope). If still not matched, *DENE* will consider it as a name error and will flag it as such. To help developers better understand the name error, *DENE* will further output the location where the name error happens (i.e., block #2 in function *status_printer*) and the full path showing the identifier is not defined from the block per se to the module entry in backward order.

Feasibility of Path Exploration in *DENE*. It is worthwhile to note our approach tries to enumerate all the possible execution paths to

[7] The problem will be propagated to the dominator block.

Algorithm 1 Path-sensitive backward tracking.

Input: Current Block B_j, entry block B_{entry} and variable name *name*.
Output: name error candidates
procedure DFS_FOR_NAME(B_j, B_{entry}, *name*)
    if $name \in gen(B_j)$ **then**
        return False ▷ Error-free with this path
    else if $name \in del(B_j)$ **then**
        return True ▷ Name used after deleting
    else if $B\_j == B_{entry}$ **then**
        return True ▷ Path ends
    else
        for B_k in B_j.predecessors **do**:
            return dfs_for_name(B_k, B_{entry}, name)
        end for
    end if
end procedure

Input: A control flow graph cfg for a scope (function/class definition.)
Output: name error candidates
procedure BACKWARD_TRACKING(cfg)
    $name\_error\_candidates \leftarrow \phi$
    $B\_entry \leftarrow$ cfg.entry_block
    for $B_i \in cfg.blocks$ **do**
        $enclosed\_scope\_name\_candidates \leftarrow \phi$
        if has_inner_cfg(B_i) **then** ▷ If this block contains function/class definition.
            es_cfg ← get_enclosed_scoped_cfg(B_i)
            $enclosed\_scope\_name\_candidates \leftarrow$ backward_tracking(es_cfg)
        end if
        name_candidates ← $use(B_i) \cup enclosed\_scope\_name\_candidates$
        for $name \in name\_candidates$ **do**
            $B_{df} \leftarrow$ get_dominator_blocks(B_i) ▷ visiting dominator blocks
            if $name \in gen(B_{df})$ **then**
                continue
            end if
            if dfs_for_name(B_i, B_{entry}, name) **then**
                $name\_error\_candidates.add(name)$
            end if
        end for
    end for
end procedure

flag name errors. However, this will not cause scalability problem due to : (1) Our algorithm terminates if any single path that can trigger name error; (2) Most of names are defined within the current block or its dominator blocks, which largely reduces the search space. The experimental results for the evaluation of execution efficiency can be found in Section 4.3.

4. Experiments

To evaluate the effectiveness of our approach and provide insights for NameError in real-world open-source Python projects, we propose to answer the following research questions.

- **RQ1: How effective is *DENE* in detecting name errors in Python programs?** This first RQ aims at checking to what extent the *DENE* is capable of statically pinpointing name errors in Python projects. Based on a carefully crafted ground truth dataset, we experimentally demonstrate that *DENE* is quite effective in uncovering name errors in Python programs, with precision and recall at 94.0% and 97.6%, respectively.
- **RQ2: To what extent are name errors hidden in popular Python projects?** After demonstrating the effectiveness of *DENE*, the second RQ attempts to apply it to detect name errors in popular Python projects in the wild, and understand to what extent is name errors (still) scattered in the Python community. Our experimental results confirm that majority of top-100 Python libraries still contain a number of *unknown* name errors .
- **RQ3: Can *DENE* help in understanding why are name errors presented in Python programs?** By answering the previous RQs, we experimentally find that name errors are a severe issue in the Python community, for which even publicly released popular libraries contain various such issues. Towards helping Python developers mitigate name errors , in addition to detecting name errors , we have further introduced a module in *DENE* to generate hints for explaining why the errors are flagged as such. With this RQ, we intends to check its usefulness. Our experimental results, via a user study, demonstrate that *DENE* can not only effectively detect name errors but also provide useful information for helping developers understand why the errors are identified, reducing their time spent on debugging.

4.1. RQ1: Effectiveness of dene

In this first research question, we evaluate the effectiveness of *DENE* by examining if *DENE* is capable of statically pinpointing name errors in Python projects. We resort to fulfill this objective through two independent experiments.

Experiment #1. We apply *DENE* to a ground-truth dataset that is carefully prepared to satisfy the purpose. The Python scripts included in this dataset all contain a name error, which has been confirmed and labeled at the place where it happens. The dataset is dervied from a set of Python Jupyter notebooks recently released by Wang et al. [10]. All the notebooks will encounter various execution issues at runtime, and 54[8] among them are demonstrated to contain name errors (i.e., with 208 labeled name errors in total). For each notebook, we extract its Python source code snippets using *nbformat* [23] and rewrite them into a Python file following the sequential execution order. When manually labeling the generated Python files, we find that a small number of them are written based on Python 2 while the majority of them are based on Python 3. To keep the consistency of the dataset, we decide to stick to Python 3 syntax. Hence, for Python 2 syntax code, we use the tool named *2to3* [24] to convert them automatically to Python 3 syntax. Fortunately, *2to3* successfully transformed all the cases. As a result, we are able to form the ground-truth dataset with 54 test samples with each sample containing a Python file and a label indicating how the name error will be triggered.

After preparing the ground truth, we apply *DENE* to analyze all the included test samples. In total, *DENE* reports 216 name errors, covering 203 of the labeled ones (hence missing to report five labeled errors meanwhile yielding 13 false-positive results), giving a precision and recall rate at 94.0% and 97.6%, respectively. Moreover, our manual observation over the five failed cases reveals that two are related to dependencies of external Python code (i.e., one depends on a Python library while another depends on an independent Python script that is not included in the test sample). After adding the dependent snippets to the test samples, *DENE* can successfully identify the name errors in both of them.

The remaining three failed cases are mainly due to limitations of the scope analysis of *DENE*. At the moment, *DENE*'s scope analysis is restricted by complex Python interpretation mechanisms, e.g., some store and load contexts are not reflected directly by the syntax features, which may lead to inaccurate results. Let us take Fig. 5 as an example,

[8] One notebook is excluded as it contains R code

```
1 for root, dirs, files in os.walk('tsv_files/'):
2     tsv_list = [file for file in files if file[-3:]=="tsv"]
3 whole_class_info=[[],[],[]]
4 with open(root+"assess_pregnancy.tsv") as cur_tsv:
```

Fig. 5. A simplified code example containing a name error that cannot be successfully detected by *DENE*.

for which the code snippet is extracted from one of the three aforementioned failed cases. Its runtime execution indicates that there is a name error at line 4, as the variable "`root`" is not defined. Observant readers may find this issue is strange as the variable "`root`" is actually defined in the for loop statement in line 1. Syntactically speaking, the readers' observation is correct, which is also why *DENE* regard it as such. However, at runtime, when function `os.walk()` returns empty, the assignment to variable "`root`" (in line 1) will not be triggered, thus leading to a name error when "`root`" is referenced at line 4. Unfortunately, *DENE*, at the moment, cannot handle such a complicated case.

Experiment #2. To further validate the correctness of *DENE* in detecting name errors, we go one step further to check if *DENE* can successfully identify the historical name errors that have already been fixed by developers. We achieve this by leveraging Github API to search for the top-1000 commits in Python projects based on keywords "fix" and "NameError". After the data collection from the open source community, we filter these commits to form a dataset with the following criteria: (1) Removing duplicated commits in the search results using the commits' hash values; (2) Removing commits that contain more than one parent commit (mainly due to merges of different branches) as we want to ensure that the parent commit is the one witnessing name errors; (3) Removing commits that update source files that contain "`import *`" statements as it introduces additional identifiers at execution that cannot be determined statically. For example, if "`from numpy.random import *`" is used in the client program, it can import all the available functions and variables defined in the sub-module "`numpy.random`" of the Numpy library.

Eventually, we are able to harvest 626 valid name error fix commits[9] from 356 Python projects. For each fix commit, we apply *DENE* to analyze the library's source files at versions both before and after the fix commit is applied.

We then check if *DENE* can successfully identify the name error that has subsequently been fixed by Python developers (i.e., the name error appearing in the parent commit but not in the subject commit version). Among the retained 626 subject commits, *DENE* is able to highlight 504 of them relevant to name error fixes (i.e., name error does exist in their parent commit versions). For the remaining 122 commits, *DENE* fails to detect the name error in their parent commits. Our manual investigation from randomly chosen 20 failure cases reveals that these false negatives are mainly caused by the following reasons: (1) the commit is not relevant to NameError, (2) the error is propagated from external libraries and cannot be analyzed alone from the client source code (see example in Fig. 6), (3) it involves dynamical features such as built-in names that are unavailable in some specific versions, and (4) there are some complex syntax feature that cannot be handle by *DENE*. We then launch *DENE* on the updated files at the subject commit version to test to what extent *DENE* can detect the fix resulted from the fix commit.

To confirm whether the flagged 504 cases are related to the fix commit, we further scan the project source code patched with the fix commit. If any of the flags falls down, it means the identified name error is associated with the fix commit. Among the 504 cases, 418 of them have name errors disappeared (i.e., fixed) after the subject

[9] There are 716 commits retained at this stage, among which 90 of them are further ignored because *DENE* encounters syntax errors on their corresponding source codes.

```
1 try:
2     response = urllib2.urlopen(req)
3     return simplejson.load(response)
4 finally:
5 +     try:
6           response.close()
7 +     except NameError:
8 +         pass
```

Fig. 6. The subject commit handles a NameError thrown by an external function call `response.close()` that is not implemented in this program.

commits are incorporated. For remaining cases that still have name errors retained, our further investigation (on sampled cases) reveals that developers have often attempted to resolve the issues but cannot fully resolve them, demonstrating again the complicity of resolving name errors in Python projects. This experimental result also confirms the high effectiveness of *DENE* in detecting name errors in real-world Python programs.

To better understand the false alarms in this study, we randomly selected 50 (10%) for closer examination. Our analysis revealed that 6 were false positives and 44 were true positives. When multiple name errors were detected for a single file by *DENE*, we investigated all reported errors. Consequently, we estimate a true positive rate of 88% and a false positive rate of 12%. :

Comparison with State-of-the-art. We also take the opportunity to compare *DENE* with the industry products. The most closely related works to ours could be *pyre* and *pytype*, proposed by Meta (also known as Facebook) and Google, for type checking and type inference respectively. These two tools, nevertheless, can report unbound names and so-called *name-error*, which could be counted as runtime name errors. We therefore compare our approach with these two tools by launching them all on the 626 test cases leveraged in Experiment #2. *pyre* and *pytype* can only report 123 and 353 name errors, which are much smaller than that reported by *DENE*, showing that our approach is capable of outperforming state-of-the-art tools to precisely detect more name errors in real-world Python programs.

✍ Answers to RQ1

DENE is effective in statically pinpointing name errors in our well-crafted ground-truth dataset, giving a precision and recall rate at 94.0% and 97.6%. Experimental results over a large set of popular Python projects further demonstrate the effectiveness of *DENE* when applied to deal with real-world Python programs. Our design also shown to outperform existing state-of-the-art products.

4.2. RQ2: NameErrors in the wild

The Section 4.1 experimentally shows that *DENE* is highly effective in detecting name errors in Python programs. In the second research question, we are interested in leveraging *DENE* to check if name errors do appear in released popular Python libraries, and if so, to what extent do they exist? We focus on popular Python libraries because those libraries, on the one hand, have been well developed and continuously maintained, while on the other hand, should have been leveraged (i.e., tested) by many client projects, for which many name errors could

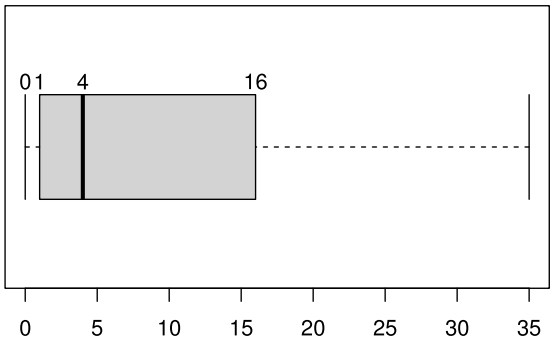


Fig. 7. The distribution of number of NameErrors per libraries where the median and mean values are 4 and 15.3 respectively.

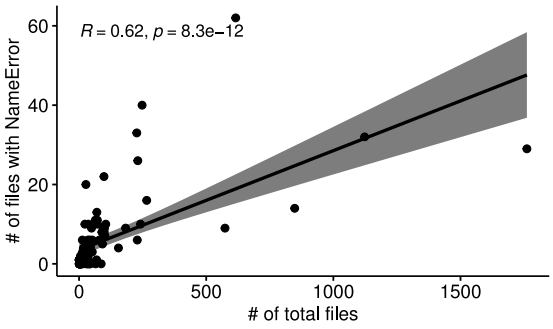


Fig. 8. The correlation between the number of total files and those containing NameErrors.

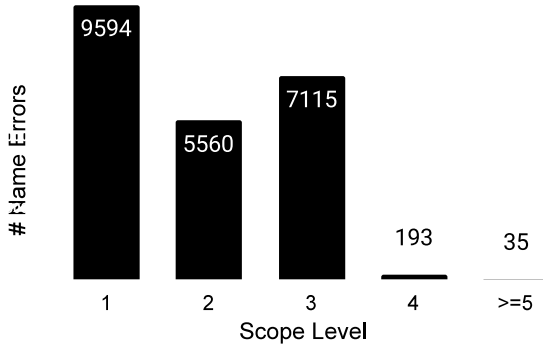


Fig. 9. The distribution of NameError occurrence by scope level.

have already been extensively spotted and fixed. Therefore, if those libraries still contain a number of name errors, we could conclude that (1) name errors are hard to address and (2) our community should pay more attention to mitigate these name errors.

To fulfill the aforementioned study, we resort to the *libraries.io* website to identify the top-100 popular Python libraries by SourceRank [25]. We then leverage the *PyPI* repository to download the source code of those libraries. To ensure 100 libraries to be collected, we have ended up queried slightly more than 100 libraries (in the list maintained on *libraries.io*) because a small number of libraries are not equipped with Python wheel files on PyPI and hence are ignored.

Among the selected 100 libraries, 77 of them have been identified to include name errors. Of the total selected 9452 source files, 6.2%(583) of them include at least one name error. Fig. 7 shows the distribution of the number of files including name errors in every libraries. This indicates these popular Python libraries may potential propagates the code defects to downstream developers, harming the re-usability of Python open source software.

We further compute the Pearson coefficient to understand the correlation between the number of total files and the number of files containing name errors. As shown in Fig. 8, the correlation coefficient $r = 0.62$ suggests that there is a moderate positive correlation between the two variables. That is, the more Python files involved in a library, the more files will contain name errors. This evidence further demonstrates the importance of having automated approaches to help the community mitigate (unknown) name errors in Python projects.

✍ Answers to RQ2

Over 70% of popular python libraries, which have already been used by many Python projects, still contain a number of unknown name errors that could break the execution of their client projects, resulting in unfriendly developer experiences including increased debugging efforts to understand why such name errors are triggered at runtime.

4.3. RQ3: NameErrors explanation

By answering the previous two research questions, we have experimentally demonstrated that *DENE* is effective in detecting unknown name errors in Python projects. In the last research question, we evaluate the explainability of *DENE*, i.e., to what extent can *DENE* help developers understand the name errors reported by *DENE*. Recall that *DENE* has been designed to not only pinpoint potential name errors but also recorded the location where name errors will arise and the execution path triggering the errors.

To fulfill this purpose, we resort to the 356 real-world Python projects used in **Experiment #2** of Section 4.1 again and relaunch *DENE* to analyze their latest code versions (after the latest commit). The latest versions of the 356 projects contain in total 54,957 Python source files, among which 8304 (15.1%) of them are flagged as containing name errors. In total, *DENE* reports 22,497 name errors.

Understanding NameError Location. Based on the detected name errors, we first look at the location where they will be thrown. Specially, we report the location based on the errors' scope level, which is defined as the number of scopes enclosed from the module level. For example, the statements in the module's static code are at scope level 1. The static code in a class defined in a module is considered at scope level 2, while the code in the class's functions will be regarded as at level 3. Fig. 9 illustrates the distribution of scope levels associated with the reported name errors. Clearly, most errors happened at level 1 and level 3, where most module static code and member function of class definitions (i.e., the module's statements and the class's inner functions, respectively) reside. There are also a significant number of errors that happened at the second level, although these errors do not involve complicated scope changes. They should be relatively easier to notice and fix than those in other levels. This result suggests that Python developers have not yet taken effective approaches to mitigate name errors, not even mention that there are also many errors located in scope level 4 and beyond that may require substantially more efforts to identify and fix.

In addition, we further look at the reasons behind the name errors' occurrences. Specifically, we check if the name error occurs (1) due to unresolved names for which the name is not defined in the entire program or (2) because of uncovered paths for which the name has indeed been defined in the project, but the definition could be bypassed, resulting in names used but not defined. In our experiments, among the 22,497 name errors, we find that the majority of them (i.e., 84.5%) are related to the first case. Our further investigation reveals that such errors are mainly introduced by forgetting to import other modules, external libraries, and name typos. Fig. 10 illustrates such an example. The variable named "BallInImage" defined in line 2 is misspelled as "BallsInImage" at line 7. The fact that such a naive mistake still exists in Python projects, for which their developers have actively fixed name

```
1 #bit-bots/bitbots_behavior
2 import rospy
3 from humanoid_league_msgs.msg import BallInImage,
    BallInImageArray
4 from trajectory_msgs.msg import JointTrajectory,
    JointTrajectoryPoint
5 from bitbots_ros_patches.rate import Rate
6
7 def run():
8     pub_ball = rospy.Publisher("ball_in_image", BallsInImage,
      queue_size=1)
```

Fig. 10. Typo mistake causes NameError. "BallInImage" (line 2) is misspelled as "BallsInImage" (line 7).

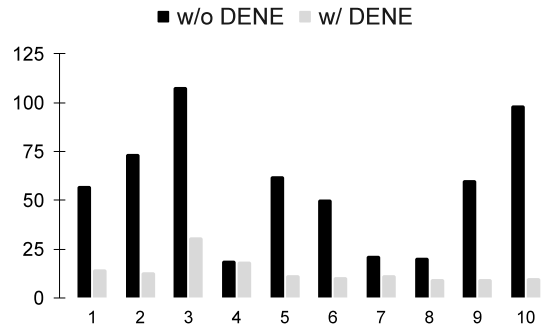


Fig. 11. Time spent (in seconds) on each of ten test cases by two group of participants on locating name error execution path.

errors, shows that name errors are common in Python and yet hard to fix. This result hence demonstrates the necessity to have automated approaches such as *DENE* to mitigate name errors so as to improve the reliability of Python projects.

User study on *DENE*'s explainability. In addition to reporting the location where a name error may arise, *DENE* also endeavors to generate the execution path indicating how an error is triggered from the module entry. In this work, we resort to a user study to check if such a path is helpful for developers to understand the corresponding name error. To this end, we randomly choose ten name errors happening respectively in ten Python source files. And then, we recruit four graduate students who use Python as their primary language for their daily work. All students are divided into groups of two and all students are given the source code and the location (line number) of a potential name error. Additionally, one group of them is provided with the execution path (produced by *DENE*). Before the experiment is conducted, all participants are given one introductory toy example with a name error to get familiar with the topic. During the experiments, all the participants are allowed to use any IDE/editors that they are familiar with. Participants are also given breaks when they finish one test case to avoid burnouts.

Fig. 11 illustrates the experimental results in terms of the average time spent by each group of participants to understand the error. Clearly, with the help of *DENE*, the participants can achieve the purpose much faster than her counterpart. Indeed, *DENE* can largely reduce developers' efforts of locating name errors by lowering the average time spent on 10 test cases from 57 s to 14 s. The maximum and minimum time for two group of participants are (98.5, 19) and (31, 9.5) seconds, respectively. All participants also positively confirmed the usefulness of the generated execution paths and the subject participant agrees that given path is valid to trigger the NameError in our following-up meetings after the controlled experiments.

Execution Time We also take the opportunity of this experiment to evaluate the execution efficiency of *DENE*. We count the total execution time for processing the 54,957 from 356 projects containing more than 12 million lines of code. We use the Linux command line tool "time" to obtain the wall execution time. The experimental result shows that it takes on average 0.05 s to process a single source file, showing our tool is feasible to be applied in a real development context.

✍ Answers to RQ3

DENE can not only effectively detect name errors in real-world open-source, popular and well-maintained Python projects but also provide useful explanations for helping developers understand why the flagged name errors are reported as such.

5. Threats to validity

The main threat to validity of this work is related to the datasets selected for experiments, which may not be representative of the whole Python community. However, we have attempted to mitigate this by only selecting the most popular ones. The validity of this work may also be impacted by the manual work involved in verifying the experimental results, as this can be error-prone. To mitigate this threat, the first two authors of this paper have cross-validated the results. Furthermore, the performance of *DENE* may be impacted by the known limitations of static analysis techniques. For example, there is no guarantee that the statically inferred execution paths, which are deemed to yield name errors, can actually be triggered at runtime, although statically speaking, the paths are valid. Moreover, *DENE* cannot handle dynamically declared functions such as the one defined via *exec* i.e., *exec("def(a):...")*, which may impact *DENE*'s accuracy. In addition, the results in RQ2 may not reflect the how name errors are scattered in real world because our approach, like all static analysis tools, cannot achieve both soundness and completeness. That is to say, there are both false alarms and false negatives in our results. However, as demonstrated in RQ1, our approach is effective in spotting real-world name errors. Nevertheless, we especially find that this type of code is uncommon in Python, and hence the corresponding impact should be limited.

6. Related work

The fast-growing popularity of Python language raises an increasing interests in Python programs by our software engineering researchers. In this section, we present the existing studies in Python code analysis, Python projects, and libraries.

Python Code Analysis Our work is closely related to the def-use relation of Python code analysis. To the best of our knowledge, Wang et al. [1] firstly presents an approach of utilizing contextual variable usage information to detect unused variables from Python code embedded in computational notebooks. Furthermore, the authors also propose approaches to model the cell dependencies among Python code cells in Jupyter notebooks [5,26]. However, the approach is field sensitive as it does not take into account the scope of functions. In addition, our fellow researchers also provide the fundamental analysis

tool *pycg*, a static Python call graph generator [27]. The implementation of *pycg* is based on computing the assignment relation among identifiers of functions, classes via inter-procedural analysis. Moreover, the naming mistakes among in Python programs are also targeted by existing work [28]. He et al.'s work designed a machine learning algorithm to spot naming issues in Python and Java source code. However, this work is not applicable to detecting Python NameErrors as NameErrors occur due to incomplete testing coverage rather than naming issues alone.

Furthermore, the studies on Python programs also involve type inference for Python variables with the interests in probabilistic methods, and learning-based approaches [29–32]. Xu et al. leverage the natural language information as well as dataflow constraints such as attribute access to build a probabilistic inference model for type inference [29]. In addition, Hellendoor et al. [33] have applied a deep learning based approach by feeding recurrent neural network models with program tokens. Following the learning-based paradigm, Michael et al. [31] propose to use both commentary text and source code tokens to combine both the natural language properties as well as programming language information from source files. Our fellow researchers also present non-RNN based approaches such as using graph neural networks [34] and deep similarity learning to reduce the type vocabulary issue [30].

Python library study Moreover, our software engineering also values Python API and ecosystem studies. Recent years have seen an increasing interests in Python API studies [35,35–39]. For instance, Wang et al.'s work reported the fragmentation of Python API deprecation system [36]. Zhang et al. [37] presents empirical discoveries of Python API evolution patterns and compare them with it in Java libraries. Following these findings, Aparna et al. invented an tool, APIScanner, to automatically locate deprecated APIs in Python libraries [39]. Apart from empirical studies on APIs, He et al. [35] invented a tool named PyART for real-time Python library API recommendation.

Furthermore, our community also contributes to the runtime dependency issues of Python libraries to automatic dependency generation [8–10,40]. For instance, Horton et al. [8]leverage a pre-built knowledge base to detect dependencies for a given Python gist file. However, the approach is limited by the lack of version information of the dependency entries as the client API may be removed over the evolution. Therefore, the authors further propose dynamical testing support to improve by collecting crash log information of runtime dependency issues to regenerate more reliable results [9]. Moreover, Wang et al. [10] have built an API database named API bank to record the API change profiles and by querying the database using API analysis results from the given Python file to generate final dependency entries with version constraints.

To summarize, our community has a large body of research to give insights into the surging number of Python applications. Most of the existing work focuses on eco-system, empirical findings, and type inference issues. There is a gap in control flow analysis and automatic bug detection, which our work aims to fill in.

7. Conclusion

In this paper, we have presented to the community the first static analysis approach called *DENE* for automatically detecting and explaining NameErrors in Python programs. *DENE* achieves this objective by performing scope-aware reaching definition analysis over well-crafted control-flow graphs statically built for Python programs. The large-scale experiments show that *DENE* is effective in pinpoint unknown name errors in both popular Python libraries and popular open-source Python projects. The explanation module provided by *DENE* is also useful for assisting developers in understanding why the name errors are reported as such. To help others to replicate our study and reuse our tool, *DENE* is available at https://github.com/DENE-dev/dene-dev.

CRediT authorship contribution statement

Jiawei Wang: Writing – review & editing, Writing – original draft, Validation, Methodology, Conceptualization. **Li Li:** Supervision, Conceptualization. **Kui Liu:** Writing – review & editing. **Xiaoning Du:** Writing – review & editing, Supervision, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

[1] J. Wang, L. Li, A. Zeller, Better code, better sharing: on the need of analyzing jupyter notebooks, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results, 2020, pp. 53–56.
[2] Pandas-Dev, dataframe plot method throws an error · Issue #19340 · pandas-dev/pandas, GitHub, URL: https://github.com/pandas-dev/pandas/issues/19340.
[3] ts.plot() and dataFrame.plot() throwing error: " NameError: name '_converter' is not defined", Stack Overflow (2018) URL: https://stackoverflow.com/questions/48341233/ts-plot-and-dataframe-plot-throwing-error-nameerror-name-converter-is.
[4] J. He, C.-C. Lee, V. Raychev, M. Vechev, Learning to find naming issues with big code and small supervision, in: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, 2021, pp. 296–311.
[5] J. Wang, K. Tzu-Yang, L. Li, A. Zeller, Assessing and restoring reproducibility of jupyter notebooks, in: 2020 35th IEEE/ACM International Conference on Automated Software Engineering, ASE, IEEE, 2020, pp. 138–149.
[6] Y. Yang, A. Milanova, M. Hirzel, Complex python features in the wild, in: 2022 IEEE/ACM 19th International Conference on Mining Software Repositories, MSR, IEEE.
[7] J.F. Pimentel, L. Murta, V. Braganholo, J. Freire, A large-scale study about quality and reproducibility of jupyter notebooks, in: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories, MSR, IEEE, 2019, pp. 507–517.
[8] E. Horton, C. Parnin, Dockerizeme: Automatic inference of environment dependencies for python code snippets, in: 2019 IEEE/ACM 41st International Conference on Software Engineering, ICSE, IEEE, 2019, pp. 328–338.
[9] E. Horton, C. Parnin, V2: Fast detection of configuration drift in python, in: 2019 34th IEEE/ACM International Conference on Automated Software Engineering, ASE, IEEE, 2019, pp. 477–488.
[10] J. Wang, L. Li, A. Zeller, Restoring execution environments of jupyter notebooks, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering, ICSE, IEEE, 2021, pp. 1622–1633.
[11] S. Mukherjee, A. Almanza, C. Rubio-González, Fixing dependency errors for python build reproducibility, in: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2021, pp. 439–451.
[12] T. Kohn, The error behind the message: Finding the cause of error messages in python, in: Proceedings of the 50th ACM Technical Symposium on Computer Science Education, SIGCSE '19, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450358903, 2019, pp. 524–530, http://dx.doi.org/10.1145/3287324.3287381.
[13] T. Kohn, B. Manaris, Tell me what's wrong: A python IDE with error messages, in: Proceedings of the 51st ACM Technical Symposium on Computer Science Education, 2020, pp. 1054–1060.
[14] Manual, GitHub CLI, URL: https://cli.github.com/manual/.
[15] Libraries - The Open Source Discovery Service, Libraries.io, URL: https://libraries.io/pypi.
[16] Comprehensions, Comprehensions - Python 3 Patterns, Recipes and Idioms, URL: https://python-3-patterns-idioms-test.readthedocs.io/en/latest/Comprehensions.html.
[17] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, C. Holler, Control flow graph, in: The Fuzzing Book, CISPA Helmholtz Center for Information Security, 2020, URL: https://www.fuzzingbook.org/html/ControlFlow.html, Retrieved 2020-10-13 15:12:26+02:00.
[18] ast - Abstract Syntax Trees - Python 3.9.7 documentation, URL: https://docs.python.org/3/library/ast.html.

[19] A library for generating graph representations of Python programs, python-graphs 1.0.1, URL: https://pypi.org/project/python-graphs/.
[20] M. Lutz, Learning Python: Powerful Object-Oriented Programming, O'Reilly Media, Inc. 2013.
[21] R.T. Prosser, Applications of boolean matrices to the analysis of flow diagrams, in: Papers Presented At the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference, 1959, pp. 133–138.
[22] K.D. Cooper, T.J. Harvey, K. Kennedy, A simple, fast dominance algorithm, Softw. Pract. Exp. 4 (1–10) (2001) 1–8.
[23] The Jupyter Notebook Format - nbformat 5.1 documentation, URL: https://nbformat.readthedocs.io/en/latest/.
[24] 2to3 - Automated Python 2 to 3 code translation - Python 3.9.7 documentation, URL: https://docs.python.org/3/library/2to3.html.
[25] Libraries.io Documentation, Overview, URL: https://docs.libraries.io/overview.html#sourcerank.
[26] J. Wang, T.-y. Kuo, L. Li, A. Zeller, Restoring reproducibility of jupyter notebooks, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings, 2020, pp. 288–289.
[27] V. Salis, T. Sotiropoulos, P. Louridas, D. Spinellis, D. Mitropoulos, PyCG: Practical call graph generation in python, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering, ICSE, IEEE, 2021, pp. 1646–1657.
[28] J. He, C.-C. Lee, V. Raychev, M. Vechev, Learning to find naming issues with big code and small supervision, in: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450383912, 2021, pp. 296–311, http://dx.doi.org/10.1145/3453483.3454045.
[29] Z. Xu, X. Zhang, L. Chen, K. Pei, B. Xu, Python probabilistic type inference with natural language support, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016, pp. 607–618.
[30] A.M. Mir, E. Latoskinas, S. Proksch, G. Gousios, Type4py: Deep similarity learning-based type inference for python, 2021, arXiv preprint arXiv:2101.04470.
[31] M. Pradel, G. Gousios, J. Liu, S. Chandra, Typewriter: Neural type prediction with search-based validation, in: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020, pp. 209–220.
[32] S. Cui, G. Zhao, Z. Dai, L. Wang, R. Huang, J. Huang, PYInfer: Deep learning semantic type inference for python variables, 2021, arXiv preprint arXiv:2106.14316.
[33] V.J. Hellendoorn, C. Bird, E.T. Barr, M. Allamanis, Deep learning type inference, in: Proceedings of the 2018 26th Acm Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018, pp. 152–162.
[34] J. Wei, M. Goyal, G. Durrett, I. Dillig, Lambdanet: Probabilistic type inference using graph neural networks, 2020, arXiv preprint arXiv:2005.02161.
[35] X. He, L. Xu, X. Zhang, R. Hao, Y. Feng, B. Xu, PyART: Python API recommendation in real-time, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering, ICSE, IEEE, 2021, pp. 1634–1645.
[36] J. Wang, L. Li, K. Liu, H. Cai, Exploring how deprecated python library apis are (not) handled, in: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020, pp. 233–244.
[37] Z. Zhang, H. Zhu, M. Wen, Y. Tao, Y. Liu, Y. Xiong, How do python framework APIs evolve? an exploratory study, in: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering, SANER, IEEE, 2020, pp. 81–92.
[38] S.A. Haryono, F. Thung, D. Lo, J. Lawall, L. Jiang, Characterization and automatic update of deprecated machine-learning API usages, 2020, arXiv preprint arXiv:2011.04962.
[39] A. Vadlamani, R. Kalicheti, S. Chimalakonda, APIScanner-Towards automated detection of deprecated APIs in python libraries, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings, ICSE-Companion, IEEE, 2021, pp. 5–8.
[40] Y. Wang, M. Wen, Y. Liu, Y. Wang, Z. Li, C. Wang, H. Yu, S.-C. Cheung, C. Xu, Z. Zhu, Watchman: Monitoring dependency conflicts for python library ecosystem, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020, pp. 125–135.