Support Python in RefDetect

VLADISLAV MUKHACHEV, University of Twente, The Netherlands

Refactoring is an essential technique required by every codebase at some point in its lifecycle. Its purpose is to improve the project's structure while preserving its functionality. However, after multiple changes, it can be challenging to clearly identify what changes were made and how they were implemented. Refactoring detection tools have been developed to address this issue by informing users of the types of refactorings performed between two project versions. One such tool, and the subject of this paper, is RefDetect.

RefDetect employs a unique, language-agnostic approach, allowing it to be extended to support any programming language. This paper describes the unique details and challenges encountered during the development process to support Python in RefDetect. We explain RefDetect's structure, examine the differences between its primary language, Java, and Python, describe the approach used to overcome these differences, and evaluate its detection capabilities using a set of test cases. This paper not only focuses on the tool's implementation but also provides a comparison of object-oriented programming principles among robust languages like Java, C++, Kotlin, and Python.

Additional Key Words and Phrases: RefDetect, Python, refactoring detection, refactoring

1 INTRODUCTION

Refactoring is a well-known technique used to improve the design of a system and prepare it for new extensions while maintaining the software's behaviour intact [4]. It is primarily performed in a semi-automated way — manually, with tool support — and the scope of changes can range from modifications to a single function to alterations spanning multiple documents.

Verification of the changes requires a complete understanding of the system, code reading skills, and knowledge of the benefits of refactoring. Due to that only the most experienced developers are qualified enough for this task and would need to spend a lot of time to not miss anything in a hurry. This step not only requires an experienced developer's attention but also significantly increases the time of the whole process. Furthermore, researchers whose work is based on the benefits of refactoring need to apply the process of verification on hundreds of projects which could take months of manual work.

To address this issue, tools for detecting refactoring have been developed. Their main purpose is to find changes in between two project versions and classify them on refactoring types. Such a tool would reduce the required expertise from a developer on the stage of verification by pointing to and naming the changes.

There are multiple tools that can detect refactoring, of which this paper focuses on RefDetect. The tool has been developed by Hemati Moghadam et al. [9] and is based on a novel language-neutral technique. Its accuracy of refactoring detection on the corpus of Java projects has been compared to the current at the time of the study state-of-the-art tool and showed better results, which proves the usefulness of the novel approach. In this paper, the tool is extended to support Python — the most popular object-oriented programming (OOP) language, according to a survey conducted among developers on the well-known developer community website, Stack Overflow [10]. This extension process involves developing a new language-specific component for the tool — a parser to extract all necessary information from the source code of Python projects.

The paper begins by providing background knowledge about RefDetect in Section 2. Section 3 addresses related work on refactoring tools for Python and section 4 introduces the research question of the study. Section 5 delves into the challenges encountered and the solutions developed during the parser's development. Section 6 covers the evaluation of the parser's reliability and accuracy in information collection. Finally, Section 7 outlines potential future improvements to further enhance the tool and concludes the research paper.

TScIT 41, 5 July 2024, Enschede, The Netherlands
© 2024 University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

2 BACKGROUND

RefDetect consists of two parts: an interchangeable language-dependent part which parses the source code of the input project, in this study we will refer to it as the analyser and language-independent differential algorithm which relies on output of the analyser to determine changes between two program versions, we will refer to it as detector. The tool was initially created in Java for Java projects but was designed to be language-agnostic. You can see the process and structure of RefDetect in Figure 1. Theoretically, an analyser can be developed for any programming language. The implementation of an analyser for one language can differ significantly from its implementation for another, as it highly depends on the available technologies for extracting information from the code and developer preferences. The only requirement is that it produces an accurate SourceInformation object.

2.1 SourceInformation object

The SourceInformation object serves as the final output of the analyser and the main input required by the detector for refactoring detection. It consists of multiple data structures that hold various information about the project structure, including details such as package names, class names, inheritance relationships among classes, declared methods and fields within classes, connections between classes through method calls, field accesses, and instantiation of classes within methods, and string representations of classes.

As for the detector, it is unaffected by the specific programming language analysed, as long as the analyser correctly fills the SourceInformation object. Assuming all implementation requirements are met, the combined new analyser and detector should accurately

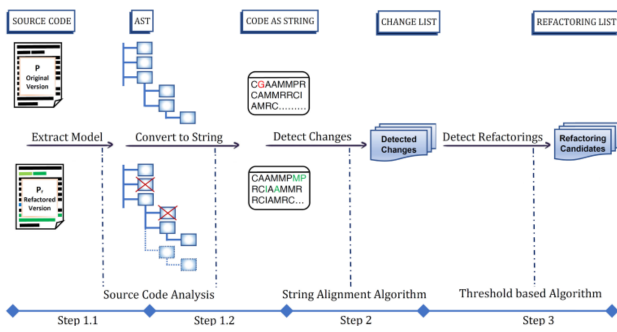

Fig. 1. Flowchart of RefDetect, taken from original study [9]

detect refactorings, thereby expanding the range of supported languages.

The language-agnostic design was validated through functional versions for both C++ [9] and Kotlin [7, 8], with all three versions achieving impressive results and establishing themselves as valid competitors to state-of-the-art refactoring tools for these languages. Importantly, the implementation of analysers for these languages did not encounter significant unresolved complications, underscoring the tool's adaptability. However, it is important to note that all three languages strictly adhere to traditional OOP principles, using clear and enforced mechanisms for encapsulation, inheritance, polymorphism, and abstraction. They rely on static typing and compile-time checking, which enforces a disciplined and predictable structure in code. This structured and predictable nature of C++, Kotlin, and the original source language facilitated the development and integration of their respective analysers.

In contrast to Java, the Python version of the tool presents unique challenges and opportunities due to Python's approach to OOP. While Python supports OOP principles, it does so with much greater flexibility. It allows for dynamic typing, duck typing, and multiple inheritance, and it relies on conventions rather than strict enforcement. This versatility facilitates its use in various contexts but can result in less predictable and more flexible code structures. Furthermore, these structural differences introduce numerous challenges in the implementation of the analyser, which must be addressed to ensure reliable and accurate information extraction for subsequent analysis by the detector.

3 RELATED WORK

Since the functioning of the detector component of the tool is beyond the scope of this paper, we will refrain from discussing the specific algorithms employed by other refactoring detection tools in general. Instead, our focus will be on Python-specific tools, the results obtained, and the methodologies used to extract information from the source code of Python projects.

PyRMiner [3], developed by Tsantalis *et al.*, was the first attempt to create a refactoring detection tool for the Python language. Their approach is unique; they first convert Python code to Java code and then feed this Java code into another refactoring detection tool for Java — RMiner [13]. This technique, in theory, should allow PyRMiner to support all the refactoring types that RMiner supports. However, in their paper, the tool was evaluated only for method-level and refactoring types, despite RMiner being capable of detecting class-level refactoring types as well. This limitation arises because PyRMiner was specifically designed for machine learning Python projects, and as a result, the tool does not adequately cover OOP projects. Consequently, we have not adopted the techniques or technology from this paper as RefDetect must be capable of working with OOP projects.

Atwi *et al.* introduced PyRef [2] as the first native Python refactoring detection tool. While inspired by RMiner [13], unlike PyRMiner [3], PyRef does not rely on program language translation or third-party refactoring detectors. To extract information from project source code, they utilise the built-in `ast` module [11] and adopt a similar information modelling approach as RMiner. They evaluated their tool against PyRMiner using three randomly selected projects, achieving improved results. However, like PyRMiner, PyRef only supports method-level refactoring types and does not address type inference, a gap which our study aims to explore.

RefDetect's language-neutral technique is based on using a language-specific parsing algorithm to get information from code and package it in a SourceInformation object. This is then handled as input for a differential algorithm, which manipulates data from SourceInformation to detect 27 different types of refactoring. This structure allows to expand list of supported languages by only implementing a new parsing algorithm, without the need to modify the rest of the tool, as far as SourceInformation is being filled accurately. In the original study [9], it was made for Java and, to demonstrate its language-agnostic properties was subsequently evaluated on C++ projects and performed well compared to state-of-the-art on both languages. However, in the following dataset [8] and comparative study [7] the goal of which was to make it possible to use RefDetect on Kotlin, some implementation problems were present. The reason was the lack of a good parser for this language and the difference in language features such as type inference. It increased the complexity of the tool and required some meticulous add-ons to the implementation of the parsing algorithm. Nevertheless, RefDetect still performed better than the state-of-the-art, which again proves its flexibility.

4 PROBLEM STATEMENT

The goal of this study is to extend RefDetect to support Python. Despite its popularity among developers [10], there are currently only two notable refactoring detection tools available for Python: PyRef [2] and PyRMiner [3], with PyRef being considered state-of-the-art. However, these tools support fewer refactoring types compared to RefDetect's capability in its Java version, with PyRef and PyRMiner supporting 9 and 18 types respectively, while RefDetect supports 27. All these factors make Python an ideal candidate for inclusion as a next supported language in RefDetect. Nonetheless, Python's significant differences from Java, even more noticeable than those with Kotlin [7], present considerable implementation challenges. These challenges require a precise set of requirements for project analysis to ensure the tool functions accurately.

Based on that, the research question of this study is:

To what extent can RefDetect support refactoring detection in Python object-oriented projects compared to its functionality in Java?

This main research question can be answered with the following two sub-questions:

(1) To what extent can a Python object-oriented project be accurately represented by a filled Java-based 'SourceInformation' object?
(2) How many types of refactorings can be detected in Python object-oriented projects compared to Java and what are the classification metrics of the tool?

5 FROM JAVA TO PYTHON

In contrast to Java, the Python version of the tool presents unique challenges and opportunities due to Python's approach to OOP. While Python supports OOP principles, it does so with much greater flexibility. It allows for dynamic typing, duck typing, and multiple inheritance, and it relies on conventions rather than strict enforcement. This versatility facilitates its use in various contexts but can result in less predictable and more flexible code structures.

5.1 Requirements for input project

These less predictable and more flexible code structures consequently hinder the precise representation of Python's OOP concepts within the SourceInformation object. Some aspects of Python's OOP implementation do not align accurately with the framework provided by SourceInformation. Acknowledging this problem, and since the implementation of the detector falls outside the scope of this study, we could not modify the structure of SourceInformation. Therefore, to develop a compatible version of the analyser for Python, we had to omit certain language functionalities, specifically those that conflict with traditional OOP principles. If the input project's structure heavily depends on these functionalities, the tool may fail to accurately grasp its structure due to its inability to recognize these relationships. For the tool to be used correctly, we have established a set of project requirements to ensure it accurately captures the project's structure. Additionally, as a precaution, failing to follow this guideline could potentially result in errors during the analyser's execution, serving as a clear warning to users about serious issues.

5.1.1 Module Concept and Module-Level Entities. In Python, the file structure revolves around the concept of modules. Each file serves as a module that can encompass multiple module-level entities such as classes, functions, and variables. This approach diverges from typical OOP languages where all entities are encapsulated within classes. Consequently, this structure poses challenges for representation within SourceInformation. As a result, we exclude module imports unless they relate to external libraries and avoid analysing entities outside the class scope to ensure reliability and consistency in our analysis approach.

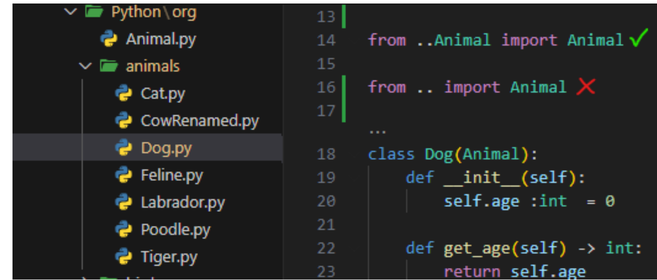

Fig. 2. Module import and class import statements syntax

As an example of module import, Figure 2 includes 2 import statements; we will focus on the bottom one initially. Although syntactically correct, this statement imports the entire module Animal. To access the Animal class from this module within the imported file, one would need to call `Animal.Animal`. In contrast, the statement above in the same figure imports the Animal class directly, allowing the class to be accessed simply by calling `Animal`.

5.1.2 Naming convention and multiple classes. Additionally, as a consequence of the aforementioned requirement, each file in the implementation of the analyser should exclusively contain a single class at the module level. We heavily rely on name matching between the file and the class within our implementation. The reason for this comes from the Java structure, where each file can contain only one public class, i.e., accessible from outside the file, and its name must match the name of the file. Therefore, a single file cannot contain multiple classes as this would lead to name mismatches and potential inconsistencies in our analysis process.

5.1.3 Multiple Inheritance. In SourceInformation, while analysing a class, we store its parent class if it exists. The data structure used for this purpose maintains a one-to-one mapping, with the current class as the key and its direct parent as the corresponding value. Singleton inheritance is standard in traditional OOP, allowing for the addition of interfaces if greater abstraction is necessary. However, Python supports multiple inheritance, and the concept of interfaces is not inherent in basic Python. Consequently, we only capture the first parent of a class in our analysis and omit any additional parents. Therefore, input projects should aim to avoid classes with multiple parents.

Besides the requirements outlined above for the input project, several additional challenges were encountered during the implementation of the Python analyser.

5.2 Parsing Python code

The pivotal component in developing an analyser is the technology used to extract information from the source code of the input project. In the Java implementation, the Spoon library was utilised, which offers a high-level API for parsing. This facilitated a less meticulous implementation of the analyser by providing access to complex relationships such as binding context and type inference. For the Python implementation, an ideal technology would have similar API functionality. However, it was not identified during the research

phase of the study. We initially considered employing ANTLR [12] alongside its Python grammar [1] for our analysis. However, we determined that this approach would be overly complex, prompting us to explore alternative solutions. Subsequently, we decided to utilize the Python `ast` module as our preferred option [11]. Evaluating this decision requires an examination of its distinct strengths and weaknesses, as detailed below:

Pluses:

- Provides access to first-hand code information, essentially exposing the internal data structures that the Python compiler utilises to interpret and process the code.
- Supported by Python language maintainers, ensuring ongoing support and updates of grammar aligned with language developments.
- Implementation is flexible, readable and the module is built into the Python standard library, making it widely accessible and straightforward to integrate.
- Parsing with the `ast` module is highly efficient, resulting in minimal execution time.

Minuses:

- Requires a deep understanding of Python's abstract syntax and semantic rules, often with sparse or limited documentation examples for guidance.
- Requires the implementation of visitor patterns for consistent traversal of the abstract syntax tree (AST) across different node types.
- Lacks a higher-level API, necessitating workaround solutions for detecting complex relationships in the code structure.

With this module, we were able to parse files, produce an AST tree of nodes and extract all the basic information from them. However, information stored inside nodes is limited and to completely fill the SourceInformation object the biggest challenge of implementation followed. It is similar to a problem that was faced in the implementation of the analyser for Kotlin [7]. Namely, we had to find a solution for binding context and type inference.

5.2.1 Type inference. As Python uses dynamic typing, information about types is not accessible in the nodes from the produced AST. We anticipated this issue from the start, as it is a well-known property of the language and was mentioned in the study of the state-of-the-art as an unresolved issue [2]. However, type information is heavily utilised by the detector, so the ambiguity of the saved information, such as assigning the type "Any" to all entities and function returns, would impact the accuracy of the tool.

We added a check for type annotations in assignment expressions and function signatures and could specify it as a requirement for the input project, but this is only a partial solution. To address this, we developed an algorithm for type approximation based on initialisation to constants. This algorithm defines the type of the constant and assigns it to the variable information. If the variable was accessed in the same context, such as for initialising another variable, its type could be inferred and used.

However, for complex types such as objects, lists, dictionaries, and so on, this method did not work well, as types could be reliably defined only from constants. To improve this, we utilised an external library, Jedi [6], which can infer types over the file. If there was a specification of a type for the selected variable, Jedi would return this type. Despite this, the issue was not fully resolved, as it was not uncommon for the library to return empty results for variables, and inferring deeper into inter-file relations would be too costly in terms of execution time.

We were unable to find another approach to this problem, and with the deadline approaching, we decided to proceed with this method and compare the results of the approach with and without type annotations.

Fig. 3. Example of inheritance

5.2.2 Binding context. In SourceInformation, we save the relationships of class entities being called in other parts of the project. When an entity is accessed through an object, the necessity for Python to import utilised external files in the current file becomes advantageous. We check what the object was initialised to, and if it is a class, we examine the import statement to precisely define which class it is. This avoids ambiguity when several classes in the project have the same name. Consequently, we save this relationship in the SourceInformation.

Furthermore, one of the main concepts of any OOP language is inheritance. To determine if an entity, which can be either a field or a method, was inherited, after parsing all the project files, we iterate over saved in SourceInformation parent-child relationships. For that, we saved the names of the classes and their parents in a one-to-one relationship. For each class, we check if an entity called through `self.` is defined in the parent class. If so, then the entity was actually called from the parent class. You can see an example in Figure 3 where class A is a parent class with field f1 and on the left side under the "Before" label, access to field f1 in child class B results in access to parent. In other cases, if a method or a field was initialized in the child class itself, we consider it overwritten. This would mean that `self.` corresponds to the child class, instead of the parent class, you can refer to the right side of an example in Figure 3 under the "After" label.

No.	Class-Level Refs.	Description
1	Rename Class	Change the name of a class to a new name, and update its references.
2	Move Class	Move a class to a more relevant package.
3	Extract Class	Create a new class and move relevant features to the new class.
4	Inline Class	Move all features of a class to another class and delete the empty class.
5	Extract Subclass	Add a new subclass to a class and push down relevant features to it.
6	Extract Superclass	Add a new superclass to class and pull up relevant features to it.
7	Collapse Hierarchy	Move all features of a class to its super or sub class and delete it.
8	Extract Interface	Create a new interface from selected classes, and make the selected classes inherit from the newly created interface.
	Field-Level Refs.	
9	Rename Field	Change the name of a field to a new name, and update its references.
10	Push Down Field	Move a field from a class to those subclasses that require it.
11	Pull Up Field	Move a field from some class(es) to their immediate superclass.
12	Move Field	Move a field from a class to another one which uses the field most.
	Method-Level Refs.	
13	Rename Method	Change method name to a new name, and update its references.
14	Push Down Method	Move a method from a class to those subclasses that require it.
15	Pull Up Method	Move a method from some class(es) to their immediate superclass.
16	Move Method	Create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation, or remove it.
17	Extract Method	Move a code fragment in an existing method to a newly created method and replace the old code with a call to the new method.
18	Inline Method	Replace calls to the method with the method's content and delete the method itself.
19	Change Method Parameters	Including insert, remove, and reorder of method's parameters, and also change method's parameters type.
Composite Refs.		
20. Extract & Move Method, 21. Move & Inline Method, 22. Move & Change Method Parameter, 23. Move & Rename Class, 24. Move & Rename Method, 25. Move & Rename Field, 26. Inline to an Inline Method, 27. Extract from an Extract Method.		

Fig. 4. Refactoring types supported by RefDetect [9]

6 EVALUATION

To evaluate the tool's ability to detect refactorings we created a set of test cases. An integral aspect of these test cases is their coverage of all supported class-level, method-level and field-level refactoring types by RefDetect, in total 18 refactoring types. The types are presented in Figure 4. The number differs from the aforementioned 27 types as we do not include separate test cases for composite refactorings. Additionally, since Python lacks support for interfaces, we do not include the Extract Interface refactoring type. For each type, we stipulated a minimum of 3 test cases to account for subtle variations that might be overlooked during implementation. To streamline the process of generating examples for each refactoring type, we opted to leverage test cases from the Java version of the program. To answer the first sub-question we have identified the structural differences of Java and Python test cases to list them as concepts which are not supported either because Python does not have a workaround to replicate refactoring type or a SourceInformation is not capable of representing Python structure. Furthermore, to answer the second sub-question, we decided to use the recall, precision, and F1-score of the detected refactorings. We calculate them using these formulas.

$$\text{Precision} = \frac{\text{\# of correct refactorings}}{\text{\# of recommended refactorings}}$$

$$\text{Recall} = \frac{\text{\# of correct refactorings}}{\text{\# of true refactorings}}$$

$$\text{F-score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

6.1 First sub-question

By comparing versions of test cases in Java and Python, we observed that, for the majority, the structure could be extracted correctly. However, some test cases under the "Inline Class" refactoring type involved inner classes and a combination of public and private classes within the same file. We were unable to accurately interpret the structure of these test cases due to the requirements outlined in Section 5.1.2 of this paper. Additionally, we excluded the "Extract Interface" refactoring type, as Python does not natively support interfaces, as mentioned in Section 5.1.3. We decided not to incorporate workarounds for this limitation within the scope of this study. Furthermore, we did not add test cases utilizing Python's multiple inheritance feature, as they could not be represented in the SourceInformation, and the current version of the detector does not support this concept.

To provide a general answer to the first sub-question, a Python OOP project can be fully and accurately represented if the specified requirements are met. Future versions of the analyser and detector may include support for the missing concepts and reduce the requirements for input projects. concepts and reduce requirements for the input project.

6.2 Second sub-question

In total, we ran the tool on 63 test cases, covering 18 refactoring types, as the Python version does not support Extract Interface refactoring type. We have calculated the chosen classification metrics and obtained the following results. Precision was 0.951, as three test cases resulted in the detection of incorrect refactoring types. Recall was 0.983, as one test case had no refactorings detected. Finally, the

F-score was 0.966. Therefore, despite some limitations in handling certain Python-specific constructs, the tool demonstrates robust capability in detecting refactorings with a high degree of accuracy. However, the tool still requires evaluation on real projects for the results to be more reliable for end users.

7 CONCLUSION

The implementation of the language-dependent part of RefDetect [9] relies significantly on the OOP rules of the target language and the available technology for parsing the source code of the input project. Despite the challenges encountered in the implementation for Python, we successfully developed a working prototype and provided a detailed explanation of the methods used to address these challenges. The use of the `ast` module [11] with the Visitor pattern [5] resulted in efficient and readable code, comprising no more than 1,000 lines.

The tool is designed to support a certain number of refactoring types, yet it still requires evaluation on real large-scale applications to be compared with state-of-the-art tools, a task that could not be covered by this study. Nonetheless, we have identified potential improvements to enhance the tool and approach for even better results. Future work should focus on conducting comprehensive evaluations to validate the effectiveness and robustness of the tool in diverse, real-world scenarios, as well as on these improvements:

- **Support for Module concept**: Implementing support for the module concept would enable the tool to recognize the use of inner classes, multiple classes within the same file, and global variables and methods. This enhancement would eliminate the necessity of the requirements outlined in Sections 5.1.1 and 5.1.2, consequently broadening the scope of acceptable input projects. However, achieving this would necessitate modifications to both the analyser and detector components of the tool.
- **Support for Java Concepts Present but Not Enforced by Python**: Implementing support for class variables, which are similar to Java static fields, would enhance the tool's functionality. Additionally, adding support for Abstract Base Classes (ABC) interface declaration, which was previously avoided due to it not being a norm for Python developers, would be beneficial. Incorporating the ability to recognize modifiers based on naming conventions would further improve the tool's capabilities. These changes would require modifications only to the analyser component of the tool.
- **Enhance Recognition of Python-Specific Patterns**: Python has many flexible features that do not have direct analogues in Java, such as list comprehensions, properties, decorators, metaclasses, and more. Incorporating these features would necessitate extensive changes in both components of the tool: the analyser and the detector.
- **Support for Multiple Parent Classes**: Extending the tool to handle multiple inheritance would address a requirement from Section 5.1.3, thus broadening the scope of acceptable input projects. This would require changes in the detector component of the tool and a slight modification of the analyser, as it already saves the information but is unable to set it in the SourceInformation object.
- **Batch Parsing for Enhanced Context and Type Inference**: Instead of parsing files individually, parsing all files simultaneously could provide a more reliable, faster, and straightforward implementation of binding context and type inference, ultimately improving the tool's precision and performance.
- **Documentation for the SourceInformation object**: Specifically for the Python implementation, documentation is no longer necessary since the SourceInformation has been fully populated. However, for future extensions to support additional languages in RefDetect, comprehensive documentation would significantly facilitate the development of new analysers. In the current version, the developer had to manually inspect the output of the Java version to understand the required information, which is not efficient or scalable.

REFERENCES

[1] ANTLR Project. 2024. Python 3 Grammar for ANTLR. https://github.com/antlr/grammars-v4/blob/master/python/python3/Python3Lexer.g4. Accessed: 2024-06-30.
[2] Hassan Atwi, Bin Lin, Nikolaos Tsantalis, Yutaro Kashiwa, Yasutaka Kamei, Naoyasu Ubayashi, Gabriele Bavota, and Michele Lanza. 2021. PYREF: Refactoring Detection in Python Projects. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 136–141. https://doi.org/10.1109/SCAM52516.2021.00025
[3] Malinda Dilhara, Ameya Ketkar, Nikhith Sannidhi, and Danny Dig. 2022. Discovering Repetitive Code Changes in Python ML Systems. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 736–748. https://doi.org/10.1145/3510003.3510225
[4] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley.
[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.
[6] David Halter et al. 2024. Jedi - an awesome autocompletion, static analysis and refactoring library for Python. https://pypi.org/project/jedi/. Accessed: 2024-06-30.
[7] Sandu-Victor Mintuş. 2023. Supporting New Programming Language in RefDetect. http://purl.utwente.nl/essays/96107 Bachelor's thesis, Universiteit Twente, The Netherlands.
[8] Iman Hemati Moghadam, Mohammad Mehdi Afkhami, Parsa Kamalipour, and Vadim Zaytsev. 2024. Extending Refactoring Detection to Kotlin: A Dataset and Comparative Study. In *Proceedings of the 31st IEEE International Conference on Software Analysis, Evolution and Reengineering, ERA Track (SANER)*. https://doi.org/10.5281/zenodo.10465265
[9] Iman Hemati Moghadam, Mel Ó Cinnéide, Faezeh Zarepour, and Mohamad Aref Jahanmir. 2021. RefDetect: A Multi-Language Refactoring Detection Tool Based on String Alignment. *IEEE Access* 9 (2021), 86698–86727. https://doi.org/10.1109/ACCESS.2021.3086689
[10] Stack Overflow. 2023. Stack Overflow Developer Survey 2023: Most Popular Technologies - Programming, Scripting, and Markup Languages. https://survey.stackoverflow.co/2023/#section-most-popular-technologies-programming-scripting-and-markup-languages Accessed: 2024-06-29.
[11] Python Software Foundation. 2008. Python 3 Documentation: `ast` – Abstract Syntax Trees. https://docs.python.org/3/library/ast.html. Accessed: 2024-06-29.
[12] Terence Parr. 2024. *ANTLR: ANother Tool for Language Recognition.* ANTLR Project. https://www.antlr.org Accessed: 2024-06-30.
[13] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2022. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* 48, 3 (2022), 930–950. https://doi.org/10.1109/TSE.2020.3007722