

MASTER'S THESIS 2023

User-Centric Study and Enhancement of Python Static Code Analysers

Steven Chen

Elektroteknik
Datateknik

ISSN 1650-2884

LU-CS-EX: 2023-31

DEPARTMENT OF COMPUTER SCIENCE

LTH | LUND UNIVERSITY



EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2023-31

**User-Centric Study and Enhancement of
Python Static Code Analysers**

Användarcentrerad Studie och Förbättring
av Python Statiska Kodanalyser

Steven Chen

User-Centric Study and Enhancement of Python Static Code Analysers

(Advancing Pylint Functionality within Visual Studio Code)

Steven Chen

`steven.chen.1750@student.lu.se`

June 1, 2023

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisors: Emma Söderberg, `emma.soderberg@cs.lth.se`
Alan McCabe, `alan.mccabe@cs.lth.se`

Examiner: Martin Höst, `martin.host@cs.lth.se`

Abstract

Despite the growing integration of code analysis tools into developer workflows, usability challenges persist in many aspects. Previous research, primarily focused on static languages and professional programmers, has overlooked the needs of novice Python programmers. This thesis investigates the experiences of beginner Python programmers with static code analysis tools. We aim to understand how these beginners interact with and perceive these tools, with a focus on identifying usability pain points. The insights derived from this study were used to enhance the Pylint extension in Visual Studio Code, incorporating additional quick-fixes to improve user experience. This research contributes to the field by providing a user-centric perspective on the design and functionality of Python code analysis tools for novice programmers.

Keywords: Code analysis, Python, Visual Studio Code, Usability challenges, Pylint

Acknowledgements

I would like to express my sincere gratitude to Emma Södeberg and Alan McCabe. Their guidance, mentoring, and expertise throughout the execution of this thesis have been invaluable. Their consistent support during the research process, as well as their insightful feedback during the report writing phase, has immensely contributed to the successful completion of this work. Their knowledge, patience, and dedication have inspired and challenged me, for which I am truly grateful.

Contents

1	Introduction	7
1.1	Objectives	8
1.2	Delimitations	8
1.3	Risks	8
1.4	Contribution	9
2	Background	11
2.1	Static Code Analysis in Python	11
2.2	Usability Challenges in Code Analysis	12
3	Method	15
3.1	Review of the State of the Art	15
3.1.1	Literature Review	16
3.1.2	Python Code Analysers Review	17
3.2	Python Analyser User Experience	17
3.2.1	Survey	18
3.2.2	Interview	19
3.3	Prototype Development and Evaluation	19
4	Related Work	21
4.1	User-Centric Studies about Analysers	21
4.2	Python Code Analysers Review	22
5	Survey and Interview Findings	23
5.1	Survey	23
5.1.1	Basic Information	23
5.1.2	User's Experience with Code Analysers	23
5.1.3	Previous Knowledge about Analysers	24
5.2	Semi-Structured Interview	24
5.2.1	Positive Experiences with Code Analysers	24

5.2.2	Negative Experiences with Code Analysers	25
5.2.3	Understanding Code Analyser's Message	25
5.2.4	False Positives	26
5.2.5	Quick-fix Solutions to Problematic Code	26
5.2.6	Suggestions	27
5.2.7	Discussion	28
6	Prototype Development	29
6.1	Pylint	30
6.1.1	Message Severity	30
6.1.2	Message Control	30
6.2	Language Server Protocol	32
6.3	Pylint Extension on VS Code	34
6.4	Current Quick-Fix Features	34
6.5	Adding Ignore Features to C0103	39
6.5.1	Disabling Through Annotations	40
6.5.2	Disabling Through Pylintrc	42
7	Prototype Evaluation	45
7.1	Evaluation Setup	45
7.2	Results	46
7.2.1	Experience of the Exercise	46
7.2.2	Adding Annotation	47
7.2.3	Dealing with Pylintrc	47
7.2.4	Discussion	47
8	Threats to Validity	49
8.1	Internal Validity	49
8.2	External Validity	50
9	Conclusion	51
9.1	Future Work	51
	References	53
	Appendix A Query Strings, Survey and Interview Questions	59
	Appendix B Survey Results	67
	Appendix C Code Listing	75

Chapter 1

Introduction

Code analysers are tools that inspect the program source code for potential problems, such as syntax errors, performance inefficiencies, and coding standards violations, among others. In the context of static code analysis, these analysers operate without executing the program code.

Code analysers are often integrated into the development process in different ways [23]. Some analysers are plugged into integrated development environments (*IDEs*) to provide programmers with immediate feedback as the code is written down. Others may need to be executed separately in the command-line interface (*CLI*).

One of the main advantages of static code analysis is that it helps to identify issues early in the development process, which can be easier and less expensive to fix. For example, identifying a security vulnerability in the source code during development can be much less costly than finding it after the code has been deployed and is in use.

While static code analysers can be a powerful tool for programmers, they also have limitations. One of the main challenges is the high false positives rate [11], where the analysis tool flags a piece of code as problematic when it is actually correct. Moreover, usability issues can be problematic. Some tools may require a significant amount of configuration and setup, or may produce messages that are difficult to understand [23, 24]. These issues can make it challenging for programmers to integrate static code analysers into their development process and get benefits from them.

Especially for novice programmers or the Python language, code analysis tools might be difficult to use or require a steep learning curve. If the tools are not intuitive or require extensive training, programmers may be less likely to use them or may use them incorrectly, leading to inaccurate results. Therefore, the usability experience of users with static code analysers is a critical aspect of this research. By examining the usability of these tools and identifying areas of enhancement, we can aim to improve the effectiveness of the tools and experience of the users.

1.1 Objectives

The overall objective of the thesis is to investigate the precision and performance of Python static code analysers and to design and implement improved features that can enhance the efficiency and fluency of the programmers' workflow. To achieve this objective, we seek answers to the following research questions:

- **RQ1:** What does previous research say about Python static code analysers and their usability problems?

This research question helps us understand the current state of Python static code analysers, their limitations, and the challenges faced by users. By reviewing previous research, we can identify areas of improvement and build upon existing knowledge.

- **RQ2:** What functionalities do current Python analysers offer in commonly used development environments such as IDEs or notebook editors?

By exploring the functionalities of existing Python analysers in different development environments, we can identify gaps in their features and usability. This information will guide us in designing and implementing enhanced features for Python static code analysers.

- **RQ3:** What experience do novice Python programmers have with static analysis?

Python has become a popular language of choice for many users whose primary focus is not programming. Industries such as AI, ML or data analysis are fast-growing fields where Python has gained popularity. This is why understanding the experiences and challenges faced by novice Python users when using static analysis tools is important for improving the tools' usability. This research question aims to gather insights into these specific issues, allowing us to incorporate these findings into the design of more user-friendly features.

- **RQ4:** In order to increase the usability of static analysis results, how can the interaction with Python code analysers be improved?

This research question focuses on identifying ways to improve the interaction between users and Python code analysers. By exploring potential improvements, we can enhance the user experience and help programmers make better use of static analysis tools in their workflow.

1.2 Delimitations

The research studies conducted in this thesis are limited to a user sample that primarily consists of students from Lund University. The development and enhancements made to the chosen Python analyser were based on the feedback gathered from the sample users and were developed locally.

1.3 Risks

Risks and challenges arose during the research process, and it was important to be prepared to address them. One potential challenge was finding a sufficient number of participants

for data collection. To minimise this risk, the research carefully designed the questionnaire items to make it easier for people to participate while still providing meaningful feedback.

Another potential challenge was the possible lack of relevant open-source Python code analysers. Additionally, there was a challenge in developing enhanced features for these analysers within the short project timeline. To address this, the research carefully considered which features to develop and ensured their viability within the time constraints.

It should be noted that deviations from the original plan did occur, and we had to make adjustments as the project evolved. To stay on track, we conducted weekly reviews. We also established a mid-project report milestone to track the progress of the work and plan more precisely for the second half of the project.

1.4 Contribution

This research project represents a journey undertaken primarily by a single individual: the author. Throughout the project, the author benefited from the guidance and mentorship of two supervisors, who helped refine the research roadmap.

The author was responsible for the subsequent data collection phase. This required designing and conducting surveys, carrying out interviews, and effectively gathering critical data. This information then became the base of the subsequent research and analysis.

Progressing to the next phase, the author undertook the design and implementation of the prototype. This task required careful consideration and application of the insights and knowledge gained from the research process.

This thesis reflects the author's dedicated involvement at each stage of the project. From conceptualisation and data gathering to the implementation of the prototype, each step was guided by the author's decisions and effort. The guidance from the supervisors was crucial in shaping this project, but the final product stands as a testament to the author's individual journey through the research process.

Chapter 2

Background

2.1 Static Code Analysis in Python

Static and dynamic programming languages are fundamentally different in how they approach type checking. In static languages like C and Java, type checking occurs during the compilation process, allowing many errors to be caught before the code is executed. Dynamic languages like Python, on the other hand, perform type checking during run time, which means that some errors may only be detected when the code is executed.

While Python's dynamic type system and simple syntax make it an accessible programming language, ideal for learning purposes as initially intended [12, 33], it also introduces the risk of type-related bugs in source code. Therefore, type annotations were introduced in Python via PEP-484 [8] and static type checkers like Mypy [22] have emerged as important tools in recent updates of the language. They enable tracking of type-related issues within the code and catch potential issues before they cause problems for end-users.

On the other hand, dynamic language's type system makes the adaption of static analysis method challenging. Nonetheless, research has demonstrated that many techniques utilised in static languages such as *pattern matching*, abstract syntax trees (AST) matchers, *symbolic execution*[9] and *abstract interpretation* [1, 6] can be applied to Python as well.

Static code analysis is critical for improving code quality and reducing the likelihood of defects in any programming language. In the context of dynamic languages like Python, research and development of effective static analysis methods and tools are important to ensure code correctness, maintainability, and security. Considering Python's significant popularity and use [33], it is essential to understand how users perceive and interact with static analysers in dynamic languages. In Chapter 4, we discuss several previous works related to Python code analysis.

2.2 Usability Challenges in Code Analysis

In spite of the importance of the code analysis in the programming world, there are still numerous challenges that hinder users from integrating analysers into their workflow. In recent years, several studies [23, 24] have identified the main problems that lead to poor communication between analyser tools and users. To summarise the issues related to usability, most of the barriers can be grouped in six different challenges:

- **Incomprehensible messages:** After identifying a problematic code segment, the tool must explain to the user the reasons for flagging the code as an issue. The warning message displayed by the tool should be both clear and intuitive; however, numerous aspects could be enhanced in this regard. Some tools might not provide any explanation as to why specific code is considered problematic, whereas others may inundate users with excessive information. The challenge lies in determining the appropriate level of detail for the warning message, considering that users may possess varying levels of expertise and knowledge. The message should be comprehensible to the vast majority of users, ensuring that it is neither inadequately explained nor be overwhelmingly informative.
- **Auto-fix support:** This category relies heavily on the technical aspects of the tool. After detecting a problem with the code, some analysers lack options for fixing the issue. Since the tool is capable of detecting the error, it should also provide some type of hint on how to solve it. While many commercial *IDEs* offer quick-fix functionalities, there is still a shortage of this type of feature in general. Other useful related functions include the option of previewing the refactored code before applying the fix, providing examples of similar problems and solutions, and more.
- **False positives:** One of the biggest reasons why programmers may not trust code analysers is the existence of false positives. This problem is highly dependent on the technical implementation of the analyser, similar to the issue of fixing support. When the rate of false positives is high, the tool can become more annoying than useful. One possible solution to address this problem from a usability perspective could be to collect users' feedback and improve the quality of reports based on it [23]. Users could also inform the tool about false positives or mark warnings as not useful, which could help the tool learn from users' knowledge and preferences and adapt accordingly for programmers.
- **User feedback integration:** The lack of customisation options is a common issue with code analysers. Users should be able to incorporate their feedback into the tool to adapt it to their specific requirements. Some analysers offer options to adjust warning severity, customise or create rules, or temporarily suppress alerts. Validating true positives could also serve as a valuable form of feedback, assisting analysers in adapting and improving, similar to marking false positives.
- **Workflow integration:** It is important to integrate the tool seamlessly into the developer's coding workflow. Some analysers are standalone tools that require installation

and configuration, while others may already be integrated into the development environment. If executing the analyser requires a complicated and disjointed process, programmers are less likely to utilise the tool [11].

- **User interface:** The interface serves as the link between the analyser and the user. The tool should provide a clear overview of errors and display warnings in visually intuitive manners. Additionally, it should offer features that guide users through the issues, enable them to examine them in more detail, track their progress, and provide unambiguous visual feedback. Moreover, the timeliness aspect of when to present or request user information is another subject to investigation [23].

While dividing each problem category into many subcategories is possible, addressing all of them would make code analysers overly complex and go beyond this project's scope. Instead, we will focus on studying users' real experiences and evaluating the technology's user aspect.

Empirical research over the past decade has looked into programmers' user experiences to understand why they need or use static code analysers [3, 5]. These studies also aim to identify the issues and barriers that stop programmers from using program analysers [11]. Most of these studies use surveys and interviews with professional programmers who have prior knowledge of code analysers. Some use them to improve code quality, while others employ the tools for policy and security reasons within their company.

In this project, we focus on the experiences of novice Python users. As Python has rapidly grown within the industry, and considering the usage characteristics of the language as discussed in RQ3, there's a notably higher percentage of non-programmers who are less familiar with code analysis tools. This target group differs from those of previous studies, as most beginners are early in their careers or students. They may not be concerned about false positives, and some might not even be aware of code analysis, even though they encounter it within their coding environments, such as *IDEs* or notebooks.

The first step in this project will be to detect the barriers relevant to this user group. We will observe and identify usability issues from the perspective of novice Python programmers, aiming to improve the tool in this aspect. By understanding the challenges and requirements of this target group, we can contribute to the development of more effective and user-friendly static code analysers for Python. This will, in turn, help novice programmers build a strong foundation in code quality, correctness, maintainability, and security as they progress in their careers.

Chapter 3

Method

In this Chapter we will discuss in detail the methodologies employed to address the different raised research questions.

The first step was a learning exercise about static code analysers, conducted by reviewing existing literature. This enabled us to address RQ1 by gaining an understanding of static code analysers and other key features of these tools. The findings and insights derived from this exercise are presented in Section 3.1.

After having a general idea of the topic, to address RQ2, we investigated open-source Python code analysers by conducting a literature review and also experimenting with the available analysers (section 3.1.2). Static code analysers could be present in *IDEs* or notebook editors, so the main objective of this step was to determine what functionalities or features these different tools present.

To answer RQ3, we gathered data from volunteer Python novice programmers via a survey and interview (section 3.2). This involved the design and implementation of a questionnaire (section 3.2.1) and semi-structured interview (section 3.2.2) aimed at gathering data on the user experiences of analysers.

To answer RQ4, we used the insights and information gathered from addressing the previous research questions to inform the design of an enhanced tool (section 3.3) that incorporates enhanced features of Python static code analysers.

Finally, the enhanced tool underwent a final evaluation using a different group of volunteer Python users.

3.1 Review of the State of the Art

In this section we describe how we reviewed related literature. The methods aim to find useful information to help us understand the context of our thesis and explore available resources about Python code analysis tools.

3.1.1 Literature Review

Our main approach for reviewing existing literature was to query the Scopus database [2] using relevant keywords. The primary objective was to identify studies related to code analysis, with a focus on the user experience perspective. We experimented with different combinations of keywords and collected the results of each query in separate spreadsheets that were combined into a single data sheet file. In total, we tried six different searches A.1, and recorded each query in an index sheet that specified the sheet where the data was saved and the number of results for that search.

The next step we took was to extract the EID¹ column of all papers and merge it into a new sheet under the same column. We then removed the duplicated rows.

Once we filtered unique papers' EIDs in the same column, based on the identifier, we extracted three additional columns with information of: which query the paper originated from, paper's title, link to the main page of the paper. At this stage, we obtained around 710 rows of papers, which needed to be reduced. The next step was a manual selection of interesting work titles by highlighting them.

Aiming to find answers to the research questions RQ1, RQ2, and RQ4, we defined a set of selection criteria, which were applied at the filtering stages of our research. These criteria are as follows:

- **C1:** The work in question must be related to code analysis. This was our primary requirement in all filtering phases.
- **C2:** A more specific requirement was that the paper had to be related to user experience or the usability of the code analysis tool itself.
- **C3:** We looked for empirical works that investigate why and how programmers use or choose not to use code analysers.
- **C4:** We attempted to identify studies focusing specifically on Python code analysers, although this criterion did not yield many results.

Applying these criteria, we identified several relevant papers. For instance, criterion C1 and C2 led us to some empirical works that investigate the use of code analysers. However, with respect to criterion C4, we only found a few papers focusing specifically on Python code analysers.

Many times the titles were not informative enough, so we reviewed the abstracts and proceeded to download those that still looked relevant to our work. This resulted in 14 papers, which was a manageable amount of works to check and study. It's worth mentioning that we found a lot of interesting and related literature on the topic but not relevant given the criteria used. We highlighted those works in another color in the spreadsheet so that we can revisit them in the future.

The final step involved a detailed reading of each of the 14 papers, focusing on the ones that specifically addressed our research areas. These include papers on the usability of code analysis tools, programmers' utilisation and perception of these tools, and specialised research on Python code analysers. Through this review process, we filtered and excluded those papers that did not directly contribute to our research questions.

¹The EID is a unique academic work identifier assigned in Scopus bibliographic database [34]

At this final step, we had identified 7 relevant papers. The strings used to query on Scopus can be found in the Appendix A.1.

3.1.2 Python Code Analysers Review

The literature review identified several relevant papers that provided different perspectives on the topic of code analysis. While all of them were valuable in understanding the field, only two studies were directly related to Python.

To gain a better understanding of Python code analysis tools, we examined the survey study conducted by Hristina and Zoltán in 2019 [9]. The research evaluated the performance of popular Python code analysers in identifying typical Python errors and bugs.

In addition, we came across a large-scale study of code analysis tool usability issues [24]. While the study evaluated analysers for multiple languages, it included 15 Python code analysers.

Following the literature review, we explored Python code analysers in different *IDEs* and notebooks such as Pycharm, VS Code, and Jupyter Notebook. We thoroughly reviewed the documentation and explored the features offered by the built-in analysers. Moreover, we researched the potential integration of extensions or external Python analysers into these coding environments.

3.2 Python Analyser User Experience

For the data gathering phase of our research, we utilised two different methodologies. First, we designed a questionnaire to gather the opinions and experiences of students regarding Python code analysers. The main target of this project was users with some basic knowledge of Python, with a focus on novice programmers who have just started learning the programming language. The questionnaire consisted of a series of multiple-choice and open-ended questions that covered topics such as familiarity with different code analysers, perceived usefulness of code analysers, and challenges encountered while using code analysers. To reach a wide range of novice users, we distributed the questionnaire both offline and online.

Additionally, we conducted semi-structured interviews with a subset of participants who expressed interest in contributing further insights on the topic. The interviews were generally conducted via video conferencing and followed a flexible structure that allowed for in-depth exploration of participants' experiences and perspectives. The interviews focused on topics such as the impact of code analysers on participants' coding practices, their preferred features and functionalities of code analysers, and their suggestions for improving code analyser usability.

In total, we received 39 completed questionnaires and conducted 6 interviews. The data gathered from both methodologies provided valuable insights into students' experiences and perceptions of Python code analysers, and helped to inform the development design of our project.

3.2.1 Survey

Before administering the survey, we obtained informed consent from all respondents, permitting the use of their information for research purposes. The survey was structured in three sections:

1. **Basic information:** We asked respondents how long they have been working with Python, what type of programming experience they have, and what environment they usually work in.
2. **Users' experience with code analysers:** In this section, we presented figures about code analysers in different *IDEs*. We aimed to guide respondents to recall their coding routines and identify situations where they have interacted with static code analysers. The core of this section is to explore the reasons behind every possible interaction between users and the tool. We aimed to understand why programmers pay attention to tool warnings and highlights, how they perceive the displayed messages, whether they understand them or not, and whether they think the tool is significant to their coding routine.
3. **Previous knowledge:** Finally, we wanted to know whether respondents had previous knowledge of static code analysers, even in other languages.

Most of the questions were multiple-choice, with an “Others” option where respondents could provide further comments.

Before distributing the survey to collect answers, an initial version was tested with five pilot users, who provided constructive feedback about the questionnaire. The feedback from the testing phase was used to enhance the structure and questions of the survey.

The survey was first distributed offline in the Physics building of Lund University for two days. We set up a stand with free pastries and coffee to encourage student participation. The advantage of gathering data with this methodology is the possibility of having direct contact with respondents, which encourages them to provide good-quality data. This method also ensures that they fully understand the questions and do not answer randomly just to receive the reward.

On the other hand, valuable insights may also arise from informal conversations. For example, one student mentioned that for their university work purposes, the performance and formality of their code might sometimes be irrelevant as long as the script can execute and provide the right output.

To increase the number of responses, we distributed the same questionnaire in different social media platforms. We used the survey tool from Lund University and published the anonymous survey in relevant student groups on Facebook, WeChat, and WhatsApp. The advantage of publishing the survey in different groups is that it diversifies the academic backgrounds of the respondents, while decreasing the influence of other factors such as nationality, gender, age, and so on. The designed survey can be found in the Appendix A.

In total, we collected 39 responses, with 13 from the offline distribution, 21 from the online distribution, and 5 from the pilot subset.

3.2.2 Interview

To gain further insights and perspectives from users, we conducted a second phase of exploration into their experiences with Python static code analysers. Although we provided options for respondents to express their opinions during the survey beyond the established questions, we aimed to delve deeper and establish direct conversations to explore more insightful data and experiences through semi-structured interviews.

After completing the questionnaire, we asked respondents if they were willing to participate in an interview. Through this method, we successfully contacted six participants and scheduled interviews with them.

The structure of the interviews followed a general guide of questions to discuss different aspects of users' experiences with the tool. We began by asking about their previous knowledge and understanding of the technology, followed by their positive and negative experiences with Python code analysers. We concluded by encouraging them to provide suggestions for enhancing the tool, whether from a user interface or functional perspective. The interviews were flexible and did not necessarily follow a strict question-and-answer format, with the goal of encouraging participants to speak freely and share their ideas.

We initiated all interviews by first seeking the informed consent of the participants for recording the conversation and using the information for our research purposes. Of the interviews conducted, five were carried out remotely, and one was conducted face-to-face. After the interviews, we used *Descripti* [4] to code the conversation, manually checked the content, and highlighted relevant quotes. We then cleaned the quotes and summarised the interesting content by grouping it according to the participants.

After reviewing the cleaned transcriptions, we organised the information into categories that correspond to various topics in Section 5.2. This categorisation helps us to organise the data and facilitate the analysis process.

3.3 Prototype Development and Evaluation

One of the most commonly used working environments identified in Section 5.1 was VS Code. As a result, we decided to focus on VS Code and chose a Python code analyser compatible with the platform. After evaluating several options, we opted to use the Pylint [18, 27] plugin extension for the notebook [20]. Pylint is a popular and well-supported Python code analyser with a variety of features and customisation options.

Based on user feedback presented in Section 5.2, we decided to work on the invalid-name [26] issue of Pylint, as it was a recurring problem that users found annoying from a usability perspective. The invalid-name issue occurs when Pylint raises a warning when a name doesn't conform to naming rules associated to its type (constant, variable, class...).

The main development process consisted of cloning the repository, understanding the open-source project, and using it as the starting point. This required us to comprehend and master concepts like VS Code extension integration and LSP (language server protocol). It was a complex learning process that demanded extensive reading of API documentation [16] and the official repository of the Pylint extension on VS Code, which was based on Python, TypeScript, and JavaScript.

During the learning process, besides the previously mentioned concepts, we also became

familiar with essential Python packages like *pygls* [31, 32] and *lsprotocol*[21].

After grasping how the LSP works as a server, interacts with the VS Code client, and exchanges information with Pylint, we added new features and options for ignoring the invalid-name issue raised by the tool. This allowed users greater flexibility to interact with and provide feedback to the analyser.

We then proceed to evaluated the enhanced tool. To recruit volunteers for this phase, we posted announcements on Facebook and, this time, also in a Python-based course at Lund University. Unfortunately, we only received responses from one person from each source, which led us to consider involving previously interviewed users. Eventually, we experimented the tool with four volunteer programmers and asked them to solve two different exercises using both the classic and enhanced versions of the Pylint extension on VS Code. Following this, we conducted a semi-structured interview to understand their experiences with the new feature of the tool. We discuss the specifics of the experiment setup in Section 7.1. The general guidance and code used for the experiment can be found in Appendix A, C.

Chapter 4

Related Work

Similar to our work, many other studies [3, 5, 10, 11] have focused on the user experience of code analysers using empirical methodologies. Most of these studies conducted interviews with professional programmers within organisations to identify the barriers that hinder users from utilising code analysers. A common characteristic of these studies is that participants are typically experienced software programmers who primarily work with statically typed programming languages.

On the other hand, there are relatively few studies [9, 25] that investigated code analysis topics specifically related to the Python language.

4.1 User-Centric Studies about Analysers

In a study by Christakis and Bird [3], they conducted surveys and interviews at Microsoft, aiming to understand the desired functionalities, barriers, and pain points associated with using code analysers. They addressed non-functional features the tool should have and how the tool should display and represent its results. The study highlighted the significant impact of false positives and recommended tools to not enable all rules by default. It proposed adding specifications or annotations in the code to enhance analyser performance. It was also shown that team policy is often the driving factor behind actively using code analysers, which differs from our studied sample since they are mostly students.

Johnson et al. [11] conducted 20 interviews to find out the reasons why professionals stop using code analysers, which revealed that the false positive issue was the most impactful. The reasons for use addressed in the study generally match some of the discoveries in our research study, associated with the acceleration and automation of finding bugs. The study also highlights the importance of having the analysis tool integrated into their coding environment as a major factor in using the tool.

Another recent study that uses similar methodologies is the work conducted by Do et al. [5]. With the objective of deriving new tool requirements to assist programmers, they

conducted a user-centered survey and interview to understand the motivation behind the usage of code analysis tools. One of the interesting conclusions from the study mentions the importance of being able to integrate programmers' experience into the tool so it can adjust and provide more personalised results.

4.2 Python Code Analysers Review

Hristina and Zoltán [9] conducted a detailed investigation about static analysis tools for Python programs. They mentioned different techniques used by analysis tools, such as pattern matching, AST matchers, and symbolic execution. All of these techniques are heuristics, which could display imperfections leading to false positives and negatives. Another noteworthy aspect of the study was the evaluation of available Python code analysis tools against common Python bugs and errors. This provided a comprehensive overview of the current capabilities and limitations of these tools in the field.

Oliveira et al. [25] analysed 1119 open-source projects with six different types of lint-based warnings. They discovered that a significant percentage of projects contained code with at least one of the six lint-based warnings, and programmers prefer their code without those lints. This last part matches some of the characteristics from our survey results in Section 5.1.2, where many of the questionnaire respondents did not want to see their code being highlighted.

Chapter 5

Survey and Interview Findings

In this chapter, we discuss some of the results and findings from the conducted survey and semi-structured interview.

5.1 Survey

In this section, we will present the results of the distributed survey. The questionnaire aimed to gather insights and feedback from our target users, particularly novice programmers, providing valuable information on their experiences and preferences when using Python static code analysers. By analysing the responses, we seek to better understand the workflows of novice programmers in the context of code analysis tools, shedding light on their needs and potential areas for improvement. This analysis will help us address RQ3, which focuses on the experiences of novice Python programmers with static analysis.

5.1.1 Basic Information

The first section of the survey aimed to understand the programming background of the respondents. In Appendix B.1, we observe that 67 % of the participants have less than two years of programming experience. Figure B.2 shows that almost everyone has used Python in academic courses. Additionally, in Figure B.3, we see that the most commonly used coding environments were VS Code, Pycharm, and Jupyter Notebook. These results are as expected and appropriate since we targeted students.

5.1.2 User's Experience with Code Analysers

In this section, we presented a screenshot scenario featuring several Python code analysis results (see Appendix A, Figure 1) to the respondents, asking if they had encountered similar

situations before. One of the reasons for this was that we suspected most users had experience with code analysers, although they may not have had knowledge of the concept and technology. As expected, in Appendix B.4, we see that the majority of users had indeed worked with analysers. When we examined the environments for Python code analysis, VS Code emerged as the most common platform, followed by Pycharm and Jupyter Notebook (Appendix B.5).

We also investigated the level of interaction users had with the tool, discovering that 59% frequently checked analyser results while only 15% typically ignored the warnings. As Appendix B.7 illustrates, the tool is viewed as beneficial by 69% of users who believe it can enhance their code quality, while 47% expressed frustration upon seeing their code highlighted. Two reasons people ignore and do not check highlighted code by the tool are because they think the warning is useless, which relates to the concept of false positives, and because they are indifferent to warnings or messages (Appendix B.8).

Despite these frustrations, a high percentage (92%) of users who interact with the tool agreed that the analyser positively influences their coding process by detecting errors and bugs early (Appendix B.9). When it comes to the reason for ignoring warning messages after checking, 62% of users felt the warnings didn't apply (false positives), and 47% considered the problem difficult to fix (Appendix B.10). In some cases, users mentioned that the problem looked hard to fix, meaning that the tool did not provide an immediate or adequate solution.

As shown in Appendix B.11, the users' reactions to issues varied: most look for external assistance (from the internet or experienced programmers), while 49% attempted to solve the problems themselves. Finally, most users interacted with the code analyser while they were writing their code, as the highlighting feature appeared during this phase (Appendix B.12).

5.1.3 Previous Knowledge about Analysers

In the last section of the questionnaire, we wanted to verify whether the user had previous knowledge or experience about code analysis, even in other programming languages. Almost everybody answered negatively in this section, as expected.

5.2 Semi-Structured Interview

In this section, we will present the results from the interviews, dividing them into six different topics: positive experiences with the tool, negative experiences with the tool, understanding the tool's message, false positives, quick-fix solutions to problematic code, and suggestions. The participants will be referred to as P1-P6 to respect their anonymity. The insights and analysis from these interviews will serve as valuable input and feedback, informing the design of an enhancement feature for an existing Python static code analyser. By examining the responses, we aim to address the issues faced by novice programmers and improve the overall user experience with code analysis tools, thus answering RQ4.

5.2.1 Positive Experiences with Code Analysers

Among the six users we interviewed, all expressed a positive attitude towards code analysers in general. They admitted that it is a powerful tool that accelerates their coding by detecting errors at an early stage.

"It's very practical that you get like the underlying, the colored underlying based on the importance of the problem. The first example that comes to mind is when I forget to import a library and a warning pops up reminding me to do so. It's helpful because it directs me to the solution right away." – P1

"Analysers are incredibly helpful because they save a lot of time. Instead of having to run the code, wait for it to break, and then look at the interpreter to figure out what went wrong, you can catch the mistake ahead of time. It's especially helpful when you miss something like a semicolon or a mismatched parentheses. If I don't catch an error like that before running the code, and it breaks, it can be frustrating." – P3

"I get instant feedback when I write something that is not syntactically correct. Correcting bad coding practices and stuff could be helpful. Makes the code nicer in general. You don't have to run it and it speeds up the programming process." – P5

5.2.2 Negative Experiences with Code Analysers

After hearing about the positive aspects of code analysers, we asked the respondents to share their dislikes about the technology. We specifically asked about any pain points or frustrations they had experienced while using code analysers in their work or studies. This part of the survey was an important source of feedback for identifying areas of improvement and designing features that would address those issues.

"Sometimes it tells me that a variable should not be named a certain way, but I feel like I can name my variable however I want." – P1

"You can have code that works, even though the tool doesn't like how it is presented. But that's just the way the code is structured. So I can't change it, or it would take me a long time to change the use of that kind of code everywhere. So I just leave it there and it distracts me from the important things. If it's just a break against some kind of style guide that nobody really cares about, then it's not a big deal." – P3

"Many times you just accidentally click on something and change your code, like maybe you press enter and it automatically changes things. It is really annoying. I tried to get rid of it but I couldn't." – P4

"I really dislike when the analyser suggests a possible error of code. Because I wrote it for a reason, I probably wanna do it. I don't like when it informs me of something that may not be an error." – P6

5.2.3 Understanding Code Analyser's Message

We asked the respondents if they felt the tool's messages were clear enough and what they typically do if they don't understand them. Based on their responses, we found that the tool's messages are usually understandable when the issue is simple, but when it becomes more complex, the messages would be difficult to understand and do not provide enough options to help them further. Most users tended to turn to Google to find a solution.

"Most of the time, the error messages are clear, and the ones that are hard to understand are usually runtime errors that stack up. But overall, editors usually provide helpful and nice error messages." – P1

"I think I rarely read what it says. It's just like, oh, it marks this place. And I quickly realize what is wrong. Almost all the time I understand the message, but when I don't understand it, I really don't understand it. If it marks something and I don't understand, reading it probably won't help me." – P3

"I would try to understand the message. If it has a little help icon I probably would use it, it's easier to press up than to go to Google." – P4

"I think in general it's usually fairly understandable. If I don't understand the message. I would probably run it and see what happens. If it doesn't run, I might get a better error message or then I will Google." – P5

"Mostly, I think they're quite clear. Sometimes you have to do a quick Google search to understand, but, you know, that's not a lot of work." – P6

5.2.4 False Positives

False positives are a major usability problem with code analysers. To better understand the frequency of this issue among students and novice Python programmers, we asked the respondents about their experience with false positives.

"There have been times that I've disagreed with some of these warnings, but it usually ends up that the machine is right." – P1

"I think it happens with typos, for example when you misspell a variable, but it's actually something you wanted to call it in that way on purpose." – P2

"I don't think there have ever been false positives to me that I can remember. If it suggests something, it's probably going to be very technically correct." – P3

"I don't think that's ever happened. My code doesn't usually get too complicated." – P4

"False positives can be very annoying, but I don't think I've experienced it in Python." – P5

5.2.5 Quick-fix Solutions to Problematic Code

We asked the respondents about a specific feature that we believed would be useful in code analysers: providing suggestions to help users solve the problems identified by the tool. We asked for their thoughts on this feature and on the quality of the solutions suggested by the tool.

"As long as it doesn't affect the program's functionality, I usually just say 'Okay, sure' and apply the suggestion" – P1

"Quick fixes for simple things are good, you just click it and change it and it is really nice, I use a shortcut in VS all the time. I don't think it's ever happened that it marks something and the suggestion is bad. Most of the time there is a suggestion that is probably good."
– P3

"I would probably try to 'quick fix' it at the beginning. But if I noticed that they'd rarely work or they're gibberish, I would probably stop using them pretty quickly. I think as long as it's understandable, like as long as it makes sense from a glance, I will be fine with it. But I think if it starts doing things I don't understand, I wouldn't dare use it."
– P5

"I've almost never seen a quick fix that works. The only time I've seen it work is it's like, oh you haven't imported, Numpy or other packages. But I, if it's more complicated than that, I've never seen that actually function in the tools I've used." – P6

5.2.6 Suggestions

In the final section of the interview, we asked the respondents to suggest areas of improvement for the code analyser tool. We encouraged them to provide suggestions for enhancements in various areas, including user interface design and other functionalities that they felt were lacking.

"What annoys me is that sometimes it's hard to apply the suggestions because you have to either select the light bulb next to the text, but it sometimes disappears when you're trying to click on it. Alternatively, you have to click there and use some sort of shortcut to apply the suggestion, but this can be distracting as it makes you forget where you initially were. It would be nice to have an easier way to interact with the suggestions."
– P1

"Analysers usually detect syntax errors. But they don't detect compiling or run run-time errors. For example, when you're dividing by zero, some calculation ends at infinity." – P2

"It would be nice to dismiss some specific warnings. I don't care about this thing on this line, but it usually will recheck every time. Like just dismiss this specific error in this specific situation. So in general, I want the check, but for this time, ignore it." – P3

"I think a lot of code analysis doesn't provide quick fixes or don't for a lot of cases, and I feel often there's pretty easy solutions to things. So getting some more quick fixes would be helpful." – P5

"I think a feature that is quite lacking in a lot of these built in code analysers is they give you a very superficial description of the problem, but there's no option to have them explain more, or dig deeper. Then you have to go to an external source to understand what that means. You only get the superficial explanation from the tool most of the time."
– P6

5.2.7 Discussion

Despite the generally positive experience with code analysers, many users also expressed several pain points that can be frustrating when using the tool.

One common issue is that, when code issues become complex, the tool often lacks resources to help the user understand the message. Similarly, the tool's limited quick-fix solutions can also be a source of frustration.

Interestingly, the mainly novice programmers among our participants did not appear to be significantly affected by false positives. Although a high percentage of respondents in Figure B.10 answered that they ignore warning messages because they believe the warning doesn't apply, none of the users we interviewed could provide an example of frustration with false positives.

In the suggestion section of the interview, many users talked about improving the warning messages, increasing the number and variety of quick-fix features, and improving the user interface and other functional options.

One specific issue that caught our attention is the problem of naming terms that do not follow certain style standards. This type of warning occurs when a user names a variable, for example, that does not follow Python's official PEP8 [7] style standard, even though it is not related to the functionality of the code. In some cases, users may want to name a term or variable in a specific way, regardless of the formatting, capitalization, or case. The fact that the tool does not provide a solution or a fast and intuitive way to ignore the warning can be frustrating for users and may discourage them from using the tool.

We believe that this issue could be a valuable problem to study and address. In the following sections, we aim to provide various possibilities for addressing the issue in specific situations.

Chapter 6

Prototype Development

In this chapter, we describe the selection of Pylint and VS Code for this project, the fundamental concepts related to these tools, and the subsequent enhancement of the Pylint extension for VS Code. Pylint is one of the most extensively used Python code analysers in the VS Code extension market. The clear objective of our enhancement was proposing ignore options to users when Pylint flags a code segment that does not adhere to a naming standard.

To kick off the development process, we cloned the official Pylint repository [18] and followed the contribution guidelines, which included installing all the necessary packages and modules. The project prerequisites consisted of NodeJs 14.19, Npm 8.5.0, and Python 3.7. We then set up a new virtual environment and installed all JavaScript and Python dependencies following the provided guide.

The implementation process was iterative, employing the Extension Development Host (EDH) [19] and the editor's built-in debugging tools for testing and debugging extensions in VS Code. The EDH, a self-contained environment, launches a separate editor instance distinct from the main session, allowing us to evaluate the real-world impact of our modifications.

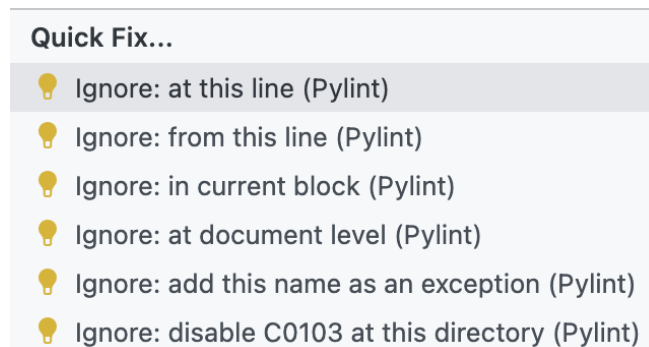


Figure 6.1: Implemented quick-fixes.

Overall, we developed four quick-fix options, providing ways for users to ignore the

invalid-name issue at various levels—single line, from a specific line, within the current block, or at the document level by adding annotations. Additionally, we created two quick-fix options that manage a `pylintrc` configuration file, either disabling the rule entirely in the working directory or adding the treated naming variable as an always accepted term. The final view of our implementation is illustrated in Figure 6.1.

6.1 Pylint

Pylint [27] is an open-source Python static code analyser widely used by programmers to identify and fix potential issues in their code. It checks the source code for programming errors, enforces coding standards, identifies code smells, and suggests improvements. Pylint parses the Python code and applies various checks, rules, and heuristics, producing diagnostic messages as output.

Pylint is highly configurable, allowing programmers to enable or disable specific checks, define custom coding standards and internal rules, and adjust the strictness of its analysis. This flexibility makes it suitable for projects with different requirements and coding styles.

Although Pylint is designed to be called and executed from the command line, in this project, we will focus on integrating the tool into the user's editor. As our aim is to implement ignore options at different levels for naming standard issues, it is important to understand how Pylint handles this type of problem and interacts with the developer through the user interface.

6.1.1 Message Severity

When Pylint analyses code, it can return diagnostic messages with different levels of severity [29]: Fatal, Error, Warning, Convention, Refactor, and Information. Each category includes a default group of problematic code types. The levels of severity for each type of problematic code can be changed and customised by the user through configurations. Regarding naming standards in Pylint, they belong to the Convention category, with the reference `'invalid-name/C0103'` [26].

6.1.2 Message Control

Pylint provides various options to control messages [28]. The easiest way to do this is by adding annotations to the source code, disabling or enabling specific rules. We can disable violations in a specific line, from a line, or within a single scope or block.

In Listing 6.1, there are different methods of applying these annotations. On line 13, we disable the unused-argument issue, which will not raise the issue message from `meth1`. On line 20, we disable the no-member error that would be raised since we are trying to call a non-existing attribute of the class. We can always enable and disable as needed throughout the lines, as shown in the different blocks and scopes in the `meth5` (line 34) and `meth6` (line 49) functions.

```
1 class Foo(object):
2     """block-disable test"""
3
4     def __init__(self):
5         pass
6
7     def meth1(self, arg):
8         """this issues a message"""
9         print(self)
10
11    def meth2(self, arg):
12        """and this one not"""
13        # pylint: disable=unused-argument
14        print(self\
15              + "foo")
16
17    def meth3(self):
18        """test one line disabling"""
19        # no error
20        print(self.bla) # pylint: disable=no-member
21        # error
22        print(self.blop)
23
24    def meth4(self):
25        """test re-enabling"""
26        # pylint: disable=no-member
27        # no error
28        print(self.bla)
29        print(self.blop)
30        # pylint: enable=no-member
31        # error
32        print(self.blip)
33
34    def meth5(self):
35        """test IF sub-block re-enabling"""
36        # pylint: disable=no-member
37        # no error
38        print(self.bla)
39        if self.blop:
40            # pylint: enable=no-member
41            # error
42            print(self.blip)
43        else:
44            # no error
45            print(self.blip)
46        # no error
```

```
47     print(self.blip)
48
49     def meth6(self):
50         """test TRY/EXCEPT sub-block re-enabling"""
51         # pylint: disable=no-member
52         # no error
53         print(self.bla)
54         try:
55             # pylint: enable=no-member
56             # error
57             print(self.blip)
58         except UndefinedName: # pylint: disable=undefined
59             -variable
60             # no error
61             print(self.blip)
62         # no error
63         print(self.blip)
```

Listing 6.1: Example of how to disable violations

The other alternative to configure Pylint is by editing its `pylintrc` file [30]. When Pylint analyses a Python file in VS Code, it searches for configuration files at different levels and with varying priorities, with the `pylintrc` file being the highest priority and the first one to look for.

The `pylintrc` file can be generated using the command line, which creates a file with the default configuration of the tool. In this file, there are two sections of interest. The first one is the section under `MESSAGE CONTROL`; in the `disable` variable, we can list the rules that we want to deactivate. The other section is under `BASIC`, in the `good-names` variable, where we can specify the variable names that should always be accepted. We will discuss these options in detail later in Section 6.5.2.

6.2 Language Server Protocol

The Language Server Protocol (LSP) is a standardised communication protocol designed by Microsoft [17] to enable communication between a language server and a client, typically an *IDE* or a code editor. LSP defines a common set of messages and conventions for the client and server to exchange, enabling the language server to provide intelligent code editing features, such as autocomplete, error-checking, and jump-to-definition, among others.

LSP allows for the development of language servers that can be reused across multiple editors and platforms. This promotes consistency and reduces duplication of effort, as programmers can focus on implementing language-specific features in a single language server, instead of implementing them separately for each editor.

In the context of this project, the Python Language Server serves as the analysis tool that is implemented in Python and runs in a separate process. This approach offers benefits such as avoiding heavy CPU and memory usage [15]. On the other hand, the Language Client

```

pylintrc x
pylintrc
1  [MAIN]
2
3  # Analyse import fallback blocks. This can be used to support both Python 2 and
4  # 3 compatible code, which means that the block might have code that exists
5  # only in one or another interpreter, leading to false positives when analysed.
6  analyse-fallback-blocks=no
7
8  # Load and enable all available extensions. Use --list-extensions to see a list
9  # all available extensions.
10 #enable-all-extensions=
11
12 # In error mode, messages with a category besides ERROR or FATAL are
13 # suppressed, and no reports are done by default. Error mode is compatible with
14 # disabling specific errors.
15 #errors-only=
16
17 # Always return a 0 (non-error) status code, even if lint errors are found.
18 # This is primarily useful in continuous integration scripts.
19 #exit-zero=
20
21 # A comma-separated list of package or module names from where C extensions may
22 # be loaded. Extensions are loading into the active Python interpreter and may
23 # run arbitrary code.
24 extension-pkg-allow-list=
25
26 # A comma-separated list of package or module names from where C extensions may
27 # be loaded. Extensions are loading into the active Python interpreter and may
28 # run arbitrary code. (This is an alternative name to extension-pkg-allow-list
29 # for backward compatibility.)
30 extension-pkg-whitelist=
31
32 # Return non-zero exit code if any of these messages/categories are detected,
33 # even if score is above --fail-under value. Syntax same as enable. Messages
34 # specified are enabled, while categories only check already-enabled messages.
35 fail-on=
36
37 # Specify a score threshold to be exceeded before program exits with error.
38 fail-under=10

```

Figure 6.2: Default pylintrc file.

operates as a special extension written in TypeScript/JavaScript that communicates with the server through the protocol over JSON-RPC.

JSON-RPC is a remote procedure call (RPC) protocol encoded in JSON (JavaScript Object Notation). RPC is a communication protocol that enables a client to request a server to execute a specific procedure or function with supplied parameters. The server then processes the request, performs the requested operation, and returns the result to the client. In Figure 6.3, we can see an illustration of how the language server works in conjunction with a development tool.

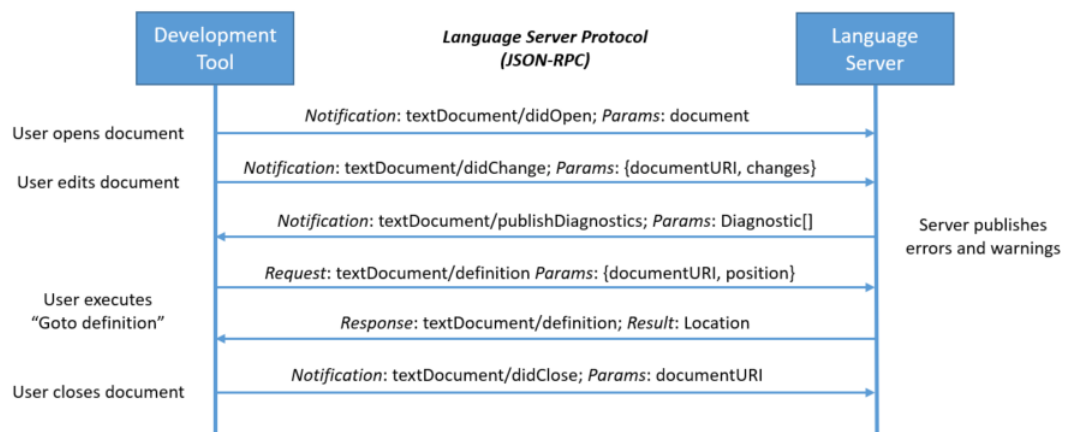


Figure 6.3: Communication protocol between a LSP and a development tool.

6.3 Pylint Extension on VS Code

The integration of Pylint as a linter into VS Code is accomplished through the editor's extension feature [18]. The extension leverages the LSP to handle information and communicates with the editor.

The typical workflow of the Pylint tool integrated into VS Code is as follows:

1. **User opens a Python file in VS Code:** When a user opens a Python file in VS Code, the editor initialises the necessary extensions and services for that file type, including the Pylint extension.
2. **LSP initialisation:** The LSP client (VS Code) establishes a connection with the LSP server (a separate process running Pylint). This connection enables them to exchange information about the source code and its analysis.
3. **LSP configuration:** The LSP server receives configuration information from the LSP client, such as Pylint settings specified by the user in their VS Code settings or working directory. This allows the language server to customise its analysis based on user preferences and project requirements.
4. **User edits code:** As the user edits the Python code in the editor, VS Code sends the changes to the LSP server through a series of incremental updates or by sending the full document text.
5. **LSP server analyses code:** The LSP server, which has Pylint integrated, analyses the code according to the given configuration. Pylint parses the code, checks for issues based on its rules and heuristics, and generates diagnostic messages.
6. **LSP server sends diagnostics to the client:** Once Pylint completes its analysis, the LSP server generates (based on Pylint's analysis) and sends the diagnostic messages back to the LSP client (VS Code) using the established connection.
7. **Displaying diagnostics and linting:** VS Code receives the diagnostic messages and displays them in the editor. The user can then review these messages and make corrections as needed.
8. **Quick fixes and code actions:** When possible, the Pylint extension in VS Code can offer quick fixes or code actions to help the user address the reported issues. This can include automatic fixes or suggested changes that the user can accept or reject.

The primary focus of development in this project has been on the last aspect of quick-fix and code actions. In the following section, we will discuss in detail how the Pylint LSP handles diagnostics to generate and provide quick-fix features for the VS Code client editor.

6.4 Current Quick-Fix Features

Among all the code issue messages available from Pylint, only a few of them have a quick-fix feature implemented in the LSP server of the extension. The lack of options and functionalities to assist programmers in addressing linted problems emerged as one of the primary

concerns from the semi-structured interview research presented in Section 5.2. As it stands, the Pylint extension for VS Code offers a limited set of 14 code messages which doesn't include the invalid-name issue, as shown in Figure 6.4.

The code issues that have associated a quick-fix option are grouped into three categories. Each category provides a distinct solution for the various types of code issues. The first two categories methods are shown in Listing 6.2. When the Pylint linter identifies a diagnostic that returns a code issue listed in the *quick_fix* method, it yields a corresponding function (using Python decorators) that generates a CodeAction object from the *lsprotocol* package. This CodeAction is then transmitted to the client via JSON-RPC and provided to the user. The formatting action involves using an external document formatting providers like *pep8*, *black* or *yapf* [14]. The organise import command consolidates specific imports from the same module into a single import statement, and organises import statements in alphabetical order.

```

1 @QUICK_FIXES.quick_fix(
2     codes=[
3         "C0301:line-too-long",
4         "C0303:trailing-whitespace",
5         "C0304:missing-final-newline",
6         "C0305:trailing-newlines",
7         "C0321:multiple-statements",
8     ]
9 )
10
11 def fix_format(
12     _document: workspace.Document, diagnostics: List[lsp.
13         Diagnostic]
14 ) -> List[lsp.CodeAction]:
15     """Provides quick fixes which involve formatting
16         document."""
17     return [
18         _command_quick_fix(
19             diagnostics=diagnostics,
20             title=f"{TOOL_DISPLAY}: Run document
21                 formatting",
22             command="editor.action.formatDocument",
23         )
24     ]
25
26 @QUICK_FIXES.quick_fix(
27     codes=[
28         "C0410:multiple-imports",
29         "C0411:wrong-import-order",
30         "C0412:ungrouped-imports",
31     ]
32 )

```

```
30
31 def organize_imports(
32     _document: workspace.Document, diagnostics: List[lsp.
        Diagnostic]
33 ) -> List[lsp.CodeAction]:
34     """Provides quick fixes which involve organising
        imports."""
35     return [
36         _command_quick_fix(
37             diagnostics=diagnostics,
38             title=f"{TOOL_DISPLAY}: Run organize imports"
39             ,
40             command="editor.action.organizeImports",
41         ]
```

Listing 6.2: Available quick fix options acting on the editor

As seen in Listing 6.2, depending on the problematic code type, different commands will be executed in the editor through the corresponding CodeAction. For example, when the code type belongs to the C03 group, the Pylint extension generates a CodeAction that triggers the *formatDocument* action (on line 19). Conversely, if the issue corresponds to a code type in the C04 group, the *organizeImports* command (on line 39) is executed when the user applies the quick-fix. These two commands are built-in features available within the VS Code system.

```
1 @QUICK_FIXES.quick_fix(
2     codes=list(REPLACEMENTS.keys()),
3 )
4 def fix_with_replacement(
5     document: workspace.Document, diagnostics: List[lsp.
        Diagnostic]
6 ) -> List[lsp.CodeAction]:
7     """Provides quick fixes which basic string
        replacements."""
8     return [
9         lsp.CodeAction(
10             title=f"{TOOL_DISPLAY}: Run autofix code
                action",
11             kind=lsp.CodeActionKind.QuickFix,
12             diagnostics=diagnostics,
13             edit=_create_workspace_edits(
14                 document,
15                 [
16                     _get_replacement_edit(d diagnostic,
                        document.lines)
```

```

17         for diagnostic in diagnostics
18         if diagnostic.code in REPLACEMENTS
19             ],
20         ),
21     )
22 ]
23
24 REPLACEMENTS = {
25     "C0113:unneeded-not": [
26         {
27             "pattern": r"\snot\s+not",
28             "repl": r"",
29         }
30     ],
31     ...
32
33     "R1721:unnecessary-comprehension": [
34         {
35             "pattern": r"\{([\w\s,]+) for [\w\s,]+ in ([\w\s,]+)\}",
36             "repl": r"set(\2)",
37         }
38     ],
39     "E1141:dict-iter-missing-items": [
40         {
41             "pattern": r"for\s+(\w+),\s+(\w+)\s+in\s+(\w+)\s*:",
42             "repl": r"for \1, \2 in \3.items():",
43         }
44     ],
45 ]
46 }

```

Listing 6.3: `_get_replacement` function

The third group of errors with quick-fix features requires a different type of CodeAction. Instead of sending an action with a command to execute, it sends the workspace edit results to the current working document. The CodeAction generation process is similar, but now a WorkspaceEdit object from the *lsprotocol* package is used to create the CodeAction. This action will use a list of TextEdit objects as arguments, generated from addressing all the diagnoses of the document identified in the group of code issues. The message categories that belong to this group can be found in the *REPLACEMENT* dictionary of Listing 6.3.

The text edition consists of finding the exact line of code, identifying, and replacing the corresponding pattern of string with the correct format. One example is the C0113/unneeded-not issue. The program finds the location of the problematic code and looks for the pattern of "not not", and replaces it to an empty string.

```
1 def _command_quick_fix(  
2     diagnostics: List[lsp.Diagnostic],  
3     title: str,  
4     command: str,  
5     args: Optional[List[Any]] = None,  
6 ) -> lsp.CodeAction:  
7     return lsp.CodeAction(  
8         title=title,  
9         kind=lsp.CodeActionKind.QuickFix,  
10        diagnostics=diagnostics,  
11        command=lsp.Command(title=title, command=command,  
12           arguments=args),  
13    )  
14 def _create_workspace_edits(  
15     document: workspace.Document, results: Optional[List[  
16         lsp.TextEdit]]  
17 ):  
18     return lsp.WorkspaceEdit(  
19         document_changes=[  
20             lsp.TextDocumentEdit(  
21                 text_document=lsp.  
22                 OptionalVersionedTextDocumentIdentifier(  
23                     uri=document.uri,  
24                     version=document.version if document.  
25                     version else 0,  
26                 ),  
27                 edits=results,  
28             )  
29         ],  
30     )  
31 def _get_replacement_edit(diagnostic: lsp.Diagnostic,  
32     lines: List[str]) -> lsp.TextEdit:  
33     new_line = lines[diagnostic.range.start.line]  
34     for replacement in REPLACEMENTS[diagnostic.code]:  
35         new_line = re.sub(  
36             replacement["pattern"],  
37             replacement["repl"],  
38             new_line,  
39         )  
40     return lsp.TextEdit(  
41         lsp.Range(  
42             start=lsp.Position(line=diagnostic.range.  
43                 start.line, character=0),
```

```

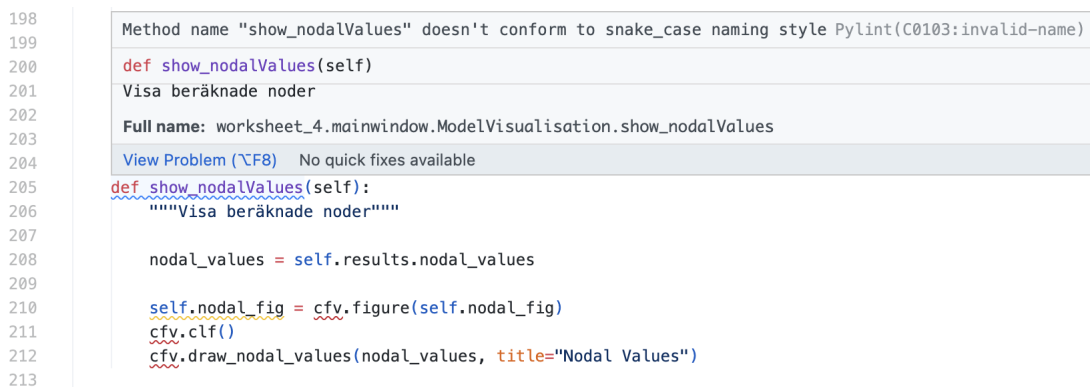
40         end=lsp.Position(line=diagnostic.range.start.
41             line + 1, character=0),
42     ),
43     new_line,
44 )

```

Listing 6.4: Utility functions used on generating CodeActions

An interesting characteristic of the current quick-fix options is that when multiple code issues belong to the same group, they will all be affected when a corresponding CodeAction is executed. This is because the commands sent from the first and second groups, as well as the text edits, run on the entire document and impact problems of the same type.

This configuration may not be suitable for our design, as we want to treat each naming case differently. There might be situations where we only want to ignore or make an exception for a specific variable or line, so we aim to provide CodeActions that only affect the current diagnosis of code, rather than addressing the entire document.



The screenshot shows a Pylint error message in a tooltip: "Method name 'show_nodalValues' doesn't conform to snake_case naming style Pylint(C0103:invalid-name)". Below the message is a button "View Problem (^F8)" and the text "No quick fixes available". The code snippet below the message is:

```

198
199
200 def show_nodalValues(self)
201     Visa beräknade noder
202
203     Full name: worksheet_4.mainwindow.ModelVisualisation.show_nodalValues
204     View Problem (^F8) No quick fixes available
205 def show_nodalValues(self):
206     """Visa beräknade noder"""
207
208     nodal_values = self.results.nodal_values
209
210     self.nodal_fig = cfv.figure(self.nodal_fig)
211     cfv.clf()
212     cfv.draw_nodal_values(nodal_values, title="Nodal Values")
213

```

Figure 6.4: Current Pylint message with invalid-name.

6.5 Adding Ignore Features to C0103

After understanding how the current language server provides quick-fix options, we can now implement similar solutions to address the invalid-name code issue. As mentioned in Section 6.1.2, we have two approaches to help users ignore the rule: by adding annotations to the source code or by configuring working directory's `pylintrc` file.

In designing the various options, we had to consider the different usage scenarios that a developer might encounter. Providing different levels of rule-ignoring control could be beneficial, allowing users to adapt based on their requirements. We use the different levels of control presented in Section 6.1.2 as a starting point for our implementation.

In the following sections, we will discuss the various CodeActions developed to address the invalid-name code issue, as shown in Listing 6.5, 6.6, 6.7. In total, we have created four CodeActions that add annotations to the document and two CodeActions that involve new commands to create and modify the `pylintrc` file.

```
1 @QUICK_FIXES.quick_fix(  
2     codes=[  
3         "C0103:invalid-name",  
4     ]  
5 )  
6 def ignore_naming(  
7     _document: workspace.Document, diagnostics: List[lsp.  
8         Diagnostic]  
9 ) -> List[lsp.CodeAction]:  
10     """Provides quick fixes which involve ignoring the  
11         naming standard at differenet levels."""  
12     diagnostic = diagnostics[0] # Extract the single  
13         diagnostic from the list  
14     diag_line = _document.lines[diagnostic.range.start.  
15         line]  
16     match = re.search(r"\b([a-zA-Z_]\w*)\b", diag_line)  
17     return [  
18         ...  
19     ]
```

Listing 6.5: New ignore naming standard quick-fix

6.5.1 Disabling Through Annotations

To add annotations in the working Python file is a similar process to the existing solution in Listing 6.3. Our goal is to create a `WorkspaceEdit` objects, using `TextEdit` objects that identifies the location of the underlined variable, and add the disabling annotation to the corresponding location depending on user's needs. The `CodeActions` related to annotations, which correspond to the collapsed line 14 in Listing 6.5, are shown in Listing 6.6. Once we have the `WorkspaceEdit` generated, it serves as *edit* argument to create the editing type `CodeAction`.

```
1 [lsp.CodeAction(  
2     title=f"Ignore: at this line ({TOOL_DISPLAY})",  
3     kind=lsp.CodeActionKind.QuickFix,  
4     diagnostics=diagnostics,  
5     edit=_create_workspace_edits(  
6         _document, [_ignore_naming_line(diagnostic,  
7             diag_line)]  
8     ),  
9 ),  
10 lsp.CodeAction(  
11     title=f"Ignore: from this line ({TOOL_DISPLAY})",  
12     kind=lsp.CodeActionKind.QuickFix,  
13     diagnostics=diagnostics,
```

```

13     edit=_create_workspace_edits(_document, [
14         _ignore_naming_next(diagnostic)]),
15 ),
16 lsp.CodeAction(
17     title=f"Ignore: in current block ({TOOL_DISPLAY})",
18     kind=lsp.CodeActionKind.QuickFix,
19     diagnostics=diagnostics,
20     edit=_create_workspace_edits(
21         _document, [_ignore_naming_block(_document,
22             diagnostic)]
23     ),
24 ),
25 lsp.CodeAction(
26     title=f"Ignore: at document level ({TOOL_DISPLAY})",
27     kind=lsp.CodeActionKind.QuickFix,
28     diagnostics=diagnostics,
29     edit=_create_workspace_edits(_document, [
30         _ignore_naming_document()])],

```

Listing 6.6: Ignoring naming rule CodeActions that adds annotation to the source code

There will be four CodeActions available for the user that add annotations. Adding the new annotation text to the same line will ignore the naming rule at the same line of code. Similarly, we can give the user the option to ignore the rule starting from the variable they are dealing with by simply adding the text one line above the current one. On the other hand, if the user wants to disable the invalid-name check for the entire document, we can easily achieve that by adding the annotation at the beginning of the Python file.

Finally, if the user wants to disable the underline within the same block or iteration, we will need to find the upper level of the block and add the note just below the definition. Some utility functions used to help create TextEdits objects are shown in Listing 6.7.

Until then, all of the implementation had taken place on the language server side and in Python. To create custom commands to be executed by the VS Code editor, we needed to work with the client side, which was written in TypeScript and later built into JavaScript once the project was compiled. Both commands were constructed in the commands.ts file, as shown in Listing C.1.

```

1 def _ignore_naming_line(diagnostic, line_text) -> lsp.
   TextEdit:
2     comment_text = " # pylint: disable=invalid-name"
3     line_length = len(line_text)
4     return lsp.TextEdit(
5         lsp.Range(
6             start=lsp.Position(line=diagnostic.range.
7                 start.line, character=line_length),

```

```

            line, character=line_length),
8         ),
9         comment_text,
10    )
11
12 def _ignore_naming_block(document: workspace.Document,
13 diagnostic) -> lsp.TextEdit:
14     comment_text = "# pylint: disable=invalid-name\n"
15     start_of_block_line = _find_beginning_of_block(
16         document, diagnostic.range.start.line
17     )
18     return lsp.TextEdit(
19         lsp.Range(
20             start=lsp.Position(line=start_of_block_line,
21                               character=0),
22             end=lsp.Position(line=start_of_block_line,
23                               character=0),
24         ),
25         comment_text,
26     )
27
28 def _ignore_naming_document() -> lsp.TextEdit:
29     comment_text = "# pylint: disable=invalid-name\n"
30     return lsp.TextEdit(
31         lsp.Range(
32             start=lsp.Position(line=0, character=0),
33             end=lsp.Position(line=0, character=0),
34         ),
35         comment_text,
36     )
37
```

Listing 6.7: Ignoring naming rule CodeActions that adds annotation to the source code

6.5.2 Disabling Through Pylintrc

Unfortunately, unlike adding annotations to the source code, dealing with the `pylintrc` file has some limitations when it comes to customising rules behavior between lines of a document.

Despite these limitations, we can provide two beneficial options to programmers. As mentioned previously in Section 6.1.2, we can disable specific rule types under the `MESSAGE CONTROL` section, by adding them to the `disable` variable list. Another section of interest is under the `BASIC` section, where we can add exception variable names to the `good-names` list, so they are always accepted.


```

127 [MESSAGES CONTROL]
128
129 # Only show warnings with the listed confidence levels. Leave empty to show
130 # all. Valid levels: HIGH, CONTROL_FLOW, INFERENCE, INFERENCE_FAILURE,
131 # UNDEFINED.
132 confidence=HIGH,
133     CONTROL_FLOW,
134     INFERENCE,
135     INFERENCE_FAILURE,
136     UNDEFINED
137
138 # Disable the message, report, category or checker with the given id(s). You
139 # can either give multiple identifiers separated by comma (,) or put this
140 # option multiple times (only on the command line, not in the configuration
141 # file where it should appear only once). You can also use "--disable=all" to
142 # disable everything first and then re-enable specific checks. For example, if
143 # you want to run only the similarities checker, you can use "--disable=all
144 # --enable=similarities". If you want to run only the classes checker, but have
145 # no Warning level messages displayed, use "--disable=all --enable=classes
146 # --disable=W".
147 disable=raw-checker-failed,
148     bad-inline-option,
149     locally-disabled,
150     file-ignored,
151     suppressed-message,
152     useless-suppression,
153     deprecated-pragma,
154     use-symbolic-message-instead

```

Figure 6.5: MESSAGE CONTROL section of a pylintrc file.

For this reason, we implemented two CodeActions that execute custom commands to create a pylintrc file (if it does not already exist in the working directory) and modify the *disable* and *good-names* lists. When the user selects the CodeAction of ignoring the invalid-name/C0103 rule entirely in the working directory, the command ensures that there is a pylintrc file with the rule added to the *disable* list. On the other hand, when the user wants to add a variable as an exception to be always accepted, similarly, the corresponding command adds the name to the *good-names* list. The developed CodeActions related to the pylintrc file that go into the collapsed line 14 in Listing 6.5 are shown in Listing 6.8.

```

1 [lsp.CodeAction(
2     title=f"Ignore: add this name as an exception ({
3         TOOL_DISPLAY})",
4     kind=lsp.CodeActionKind.QuickFix,
5     diagnostics=diagnostics,
6     command=lsp.Command(
7         title="Add this name as an exception",
8         command="custom.add_good_name",
9         arguments=[_document, match.group(1)],
10    ),
11 ),
12 lsp.CodeAction(
13     title=f"Ignore: disable C0103 at this directory ({
14         TOOL_DISPLAY})",
15     kind=lsp.CodeActionKind.QuickFix,
16     diagnostics=diagnostics,

```

```
15     command=lsp.Command(  
16         title="Disable naming standard violation for this  
            directory",  
17         command="custom.disable_naming_standard_directory  
            ",  
18         arguments=[_document],  
19     ),  
20 ),]
```

Listing 6.8: Ignoring naming rule CodeActions that create and modifies pylintrc file

For the command that disabled the invalid-name rule, it ensured that a pylintrc file existed before modifying it. If such a file did not exist, it created one by executing `pylint --generate-rcfile > pylintrcPath` in the terminal, where `pylintrcPath` was the current project working directory extracted from the working document argument passed from the server side. Once we knew that the file existed, the command searched in the pylintrc file for the `MESAGE CONTROL` section and the `disable` listing. After identifying the location, it added the invalid-name term at the beginning of the list by replacing the strings.

Similarly, to add a good name term to the `good-name` list, the corresponding command took the working document as an argument but also received the term to add directly from the server side. It checked for the existence of the pylintrc file and used the same methods to search for the positions of the sections of interest. Then, it added the terms to the list.

```
56     // Register the custom command  
57     const disableNaming = vscode.commands.registerCommand('custom.disable_naming_standard_directory', disableNamingStandard);  
58     const addGoodName = vscode.commands.registerCommand('custom.add_good_name', addToGoodNames);  
59     // Add the command to the extension's context so it gets properly disposed when the extension is deactivated  
60     context.subscriptions.push(disableNaming);  
61     context.subscriptions.push(addGoodName);
```

Figure 6.6: Registering new customise commands in the extension.ts file.

After creating these two new commands, it was necessary to add them to the extension.ts file of the project [13], which is the starting point of VS Code extensions that initialises the whole extension server. The registration under the activate function is shown in Figure 6.6.

Chapter 7

Prototype Evaluation

The two types of quick-fix options developed in the previous chapter address the problem in different ways, each presenting unique advantages and disadvantages. For instance, adding annotations directly into the user's source code might be a flexible and visual way of controlling and customising message rules. This is a significant advantage, as users are always aware of which rules are enabled or disabled at different points in their code, and can quickly modify the rules when necessary. However, users might be annoyed by the tool adding extra information and code into their work.

On the other hand, using a configuration file, such as the `pylintrc` file, could be a more discreet solution. The drawback is that it's not as flexible as annotations, as users cannot specify which lines, blocks, or Python files they intend to customise rules for. Although the configuration file might be more acceptable because it remains hidden, users might not be as aware of the available rules as they would be with annotations. Moreover, if users later want to edit or change the settings, they need to locate the `pylintrc` file and modify its content, which might not be an intuitive process for programmers.

The purpose of this final evaluation is to assess users' experiences with Pylint, and the newly developed features, as well as their thoughts on these two distinct methods for addressing code issues. We aim to determine which option is more suitable, if there is one, as well as to collect valuable suggestions and constructive feedback that could guide future work on similar topics.

7.1 Evaluation Setup

To evaluate the enhanced Pylint extension, we designed two simple Python exercises containing errors for Pylint to detect. Both Python test files were similar, each including an introduction explaining the objectives to be achieved, namely, the elimination of all linted and underlined code. In addition to this requirement, each file instructed the user to name every term in a specific format. One exercise required terms to be in upper-case letters (`UPPER_CASE`

standard), while the other required naming everything in lower-case letters (snake_case standard).

The users were asked to complete both exercises in a random order, the one exercise using the existing Pylint extension from the marketplace, and the other using our enhanced Pylint. They were encouraged to verbalize their thoughts, allowing us to guide them if they deviated from the purpose of the exercise.

The aim of this experiment was to observe user interaction with the Pylint analyser and determine if they reacted differently to Pylint's new features. By enforcing a specific naming standard, we simulated a situation where users intended to name variables in a non-standard way. This approach allowed us to observe, in real time, user interactions with the Pylint analyser, providing valuable insights into potential usability issues.

After completing the exercises, we asked users about their experience with Pylint. As each user's experience was unique, we remained flexible in our guidance, providing explanations about the exercise as necessary. During the interview, we made sure to explain how Pylint typically handles naming code issues, the conflict we sought to test, and the enhanced features we developed. Once confident that the interviewee understood the topic, we sought their preference between the two solutions our enhanced tool offered: solving the conflict by adding annotations to their source code, or adjusting the configuration in the `pylintrc` file.

Finally, we openly asked for suggestions and feedback to gain insights for future studies and improvements.

7.2 Results

All of the volunteers interviewed had some level of Python knowledge, but their expertise varied. Some were computer science students with substantial experience with programming languages and coding environments, while others primarily used Python as a tool to complete tasks in specific courses. Given these differences, we will first discuss their performance during the exercise, followed by the topics discussed during the interview. We will refer to participants as C1-C4.

7.2.1 Experience of the Exercise

The participants' responses to underlined codes were diverse. For instance, C2 consistently clicked on the "View problem" option for more information and clicked the light bulb whenever possible. However, the participant quickly lost interest in the message as it wasn't immediately clear and intuitive. On the other hand, C3 quickly understood the error and manually resolved some of the warnings. These differences can be attributed to varying levels of expertise with the Python language and familiarity with the code editor.

Despite these differences, all candidates reacted confused to the naming conflict presented during the experiments. However, they expressed varying levels of annoyance. Some proceeded with the exercise without giving it much thought, while others spent a few minutes trying to find a resolution.

A key observation was that only C1 noticed the difference between the two exercises, where the user could apply the developed ignore options. For example, C2 tried to find

similar options by right-clicking on the linted variables. C3, on the other hand, assumed that the quick-fix option would enforce the standard that the linter was expecting.

7.2.2 Adding Annotation

We asked the candidates how annoyed they would feel if the tool provided an option to ignore issues by adding an annotation to their source code.

Candidates C2 and C4 expressed a low level of annoyance, provided the number of annotations wasn't excessive. In contrast, candidates C1 and C3 expressed a high level of annoyance. C1 stated that having annotations would affect the code's readability, while C3 raised concerns about sharing annotated code with others, as it could be less readable and confusing. This latter opinion contrasts with C4's; C4 expressed a preference for including comments in the code rather than in the `pylintrc` file, arguing that this approach clarifies what is enabled and disabled in the code when it's shared with other collaborators.

7.2.3 Dealing with Pylintrc

We also asked the interviewees how they felt about adding a `pylintrc` file to their directory and ignoring the rule within the configuration.

All interviewees were open to having a configuration file that manages different Pylint code rules. However, after discussing the advantages and disadvantages of this method versus adding annotations to the source code, only C3 insisted that addressing the problem with `pylintrc` would be more suitable. C4 expressed a preference for annotations to enhance collaboration, and C1 suggested that having the annotation at the top of the document would be the best solution. Participant C2 mentioned that the choice between methods could depend on the severity of the rule. They suggested that annotations could be useful for important rules or warnings to consistently remind the user of their status, while for convention problems, the `pylintrc` file might be preferable.

7.2.4 Discussion

Many of the issues that emerged during the interview were related to the general usability of the tool. One of the most significant was that 3 of the 4 candidates didn't recognise the difference between the original and enhanced Pylint tool. The reasons varied depending on the user. Candidate C2 sought functionality by right-clicking the variables; C3 misinterpreted the quick-fix functionality, and C4 didn't notice the option as it was situated at the bottom of the hover message. Furthermore, some problems echoed the conclusions discussed in Chapter 5.1, such as warning messages being difficult to understand. Specifically for the invalid-name issue, C2 mentioned that the naming standard terms like "PascalCase" might be confusing for a user without prior knowledge of the topic, reflecting the warning messages' inadequate explanation of the correct standard or examples. On the other hand, she mentioned that the option to disable the rule might be confusing as it is referred to as "C0103" code, not the name of the code issue.

Chapter 8

Threats to Validity

In this chapter we are going to discuss some potential limitations of our project. We have critically examine factors that might cause biases to the results of this study. We will address different types of validity aiming to provide a comprehensive understanding of the limitations in the study design and methodology.

8.1 Internal Validity

Internal validity refers to the extent to which a research study accurately establishes a causal relationship between the independent variable (such as survey and interview results) and the dependent variable (users' experience with Python code analysers). It is essential to examine potential factors that could introduce bias and influence the results of this project.

Size of sample

In this thesis, we conducted a user-centric research, including an initial survey and several interviews, as well as a final evaluation of the enhanced Pylint tool. In the first data gathering phase, we collected 39 responses and interviewed six participants. For the evaluation of the tool, we interviewed an additional four participants. However, the sample size in this study is relatively small. An investigation with a larger sample size could potentially enhance the internal validity of our findings and the representation of the target population.

Sample selection

The diverse backgrounds of the participants might introduce bias in various ways. Factors such as career background, user expertise, and other individual differences could introduce bias, as participants may come from varied backgrounds and use Python for different purposes. Specifically for the interview sections, we used convenience sampling techniques,

which resulted in some interviewees having prior relationships with the author. This could introduce bias, as these individuals may have tendencies to express positive comments, potentially losing objectivity.

Inconsistency procedure and conditions

In conducting the survey, we employed both online and offline procedures to collect data. This approach could introduce inconsistencies, as the degree of interaction varied between online and offline respondents. In the semi-structured interviews, the majority were conducted virtually via Zoom, while a single session was performed in person. However, the final evaluation took place in person on the author's computer. It's important to note that the location and environmental conditions differed for each interview, which could have also introduced variables affecting the result.

Given that the respondents' backgrounds and expertise varied, the questions posed during the semi-structured interviews and evaluations were not consistent. This inconsistency could have introduced bias. Furthermore, the protocols for the surveys, interviews, and evaluations were generally flexible and open to adjustments, which could also have introduced variability into the results.

Robustness of the implemented code

Although we have tested the enhanced tool with different users, the new features implemented in the Pylint extension lack thorough testing and evaluation from a coding perspective. As such, there may be inconsistencies or bugs that did not surface during this research. To effectively mitigate this, more detailed future work might be needed to ensure the robustness of the implemented code. One possible approach would be to collaborate with the Pylint community, potentially integrating our enhancements into the official repository of the extension.

8.2 External Validity

External validity refers to the extent to which this research findings can be generalised or applied beyond the specific conditions and participants of this study. It examines the degree to which the results hold valid in other populations, settings, or situations.

Pylint evolution

We have been working with an open-source Python code analyser. This tool is constantly being upgraded by the community and contributors of the project. During and after the deployment of this study, the repository for the Pylint extension on Visual Studio Code may have undergone changes and upgrades. These modifications could potentially introduce conflict or bias related to the development processes documented in this thesis.

Chapter 9

Conclusion

Throughout this project, we conducted a user-centric study, with a focus on novice Python programmers. This was achieved by gathering a diverse set of data through the use of surveys and semi-structured interviews.

During the data gathering stage, we were able to gain valuable insights into how beginners in Python, particularly students, perceive and experience Python code analysers. As discussed in Chapter 5, we found that common usability problems experienced in other static programming languages, such as unclear messages and a lack of quick-fix solutions, were also prevalent among our participants. Intriguingly, false positives, often a concern in other contexts, were not a common issue among the students and novice Python programmers we studied.

The feedback from 39 surveys and 6 interviews informed our design process during the project's development stage. This served to guide us in our aim to study and implement new quick-fix features for the Pylint extension in Visual Studio Code, particularly for the invalid-name issue. Our goal was to provide the users with options to ignore this issue at different levels, either through the addition of an annotation or by using a configuration `pylintrc` file in the working directory.

In the final stage of our project, we put our enhanced tool to the test by conducting an evaluation exercise and interview with four students. As discussed in Section 7.2, we found that even when a solution to a specific problem was provided, the presence of other usability issues could potentially hinder the effective use of the feature.

9.1 Future Work

Despite the progress we have made in this project, it is clear that there are still significant barriers that hinder the effective utilisation of code analysers. As such, we have identified several directions for future work:

- Conduct a more thorough investigation of the usability challenges faced by novice

Python programmers. This should involve a larger and more varied sample size to capture a wider range of experiences and challenges.

- Expand the quick-fix options available in the Pylint tool for VS Code or other editor platforms. This could involve developing new features or refining existing ones based on user feedback.
- Consolidate the code implemented in this project and contribute to the wider Pylint community by submitting a merge request to the official GitHub repository. This could provide benefits to a large number of users.
- Refine the warning messages in Pylint to make them more comprehensible, particularly for novice programmers. This could involve rewriting the messages or providing additional context or examples to help users understand the issues identified.

References

- [1] Antoine Miné Aymeric Fromherz, Abdelraouf Ouadjaout. Static type analysis of python programs by abstract interpretation. *Leibniz International Proceedings in Informatics*, 2020.
- [2] Elsevier B.V. Scopus. <https://www.scopus.com/>.
- [3] Maria Christakis and Christian Bird. What developers want and need from program analysis: An empirical study. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 332–343, 2016.
- [4] Descripti. Descripti. <https://www.descript.com/>.
- [5] Lisa Nguyen Quang Do, James R. Wright, and Karim Ali. Why do software developers use static analysis tools? a user-centered study of developer needs and motivations. *IEEE Transactions on Software Engineering*, 48(3):835–847, 2022.
- [6] Aymeric Fromherz, Abdelraouf Ouadjaout, and Antoine Miné. Static value analysis of python programs by abstract interpretation. In *NFM 2018 - 10th International Symposium NASA Formal Methods*, volume 10811 of *Lecture Notes in Computer Science*, pages 185–202. Springer, 2018.
- [7] Guido van Rossum and Barry Warsaw and Nick Coghlan. PEP 8 – Style Guide for Python Code. Python Enhancement Proposal, 2001. Status: Active. Created: 5-Jul-2001.
- [8] Guido van Rossum, Jukka Lehtosalo, Łukasz Langa. Pep484 - type hints. <https://peps.python.org/pep-0484/>.
- [9] Zoltán Porkoláb Hristina Gulabovska. Survey on static analysis tools of python programs. *CEUR Workshop Proceedings*, 2019.
- [10] Brittany Johnson. A study on improving static analysis tools: Why are we not using them? In *Proceedings of the Conference on Static Analysis Tools*, Raleigh, USA, 2023. North Carolina State University.

- [11] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681, 2013.
- [12] Faizan Khan, Boqi Chen, Daniel Varro, and Shane McIntosh. An empirical study of type-related defects in python projects. *IEEE Transactions on Software Engineering*, 48(8):3145–3158, 2022.
- [13] Microsoft. Commands in visual studio. <https://code.visualstudio.com/api/extension-guides/command#creating-new-commands>.
- [14] Microsoft. Editing python in visual studio. https://code.visualstudio.com/docs/python/editing#_formatting.
- [15] Microsoft. Language extension guide. <https://code.visualstudio.com/api/language-extensions/language-server-extension-guide>.
- [16] Microsoft. Language extensions - overview. <https://code.visualstudio.com/api/language-extensions/overview>.
- [17] Microsoft. Language server protocol. <https://microsoft.github.io/language-server-protocol/overviews/lsp/overview/>.
- [18] Microsoft. Vscod-pylint. <https://github.com/microsoft/vscod-pylint>.
- [19] Microsoft. Your first extension. <https://code.visualstudio.com/api/get-started/your-first-extension>.
- [20] Microsoft. Linting python in visual studio code. Visual Studio Code Documentation, 2022.
- [21] Microsoft Corporation. Language server protocol types implementation for python. <https://pypi.org/project/lsprotocol/>.
- [22] Mypy Team. Mypy. <https://mypy-lang.org/>.
- [23] Marcus Nachtigall, Lisa Nguyen Quang Do, and Eric Bodden. Explaining static analysis - a perspective. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pages 29–32, 2019.
- [24] Marcus Nachtigall, Michael Schlichtig, and Eric Bodden. A large-scale study of usability criteria addressed by static analysis tools. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 532–543. ACM, 2022.
- [25] Naelson Oliveira, Márcio Ribeiro, Rodrigo Bonifácio, Rohit Gheyi, Igor Wiese, and Balduino Fonseca. Lint-based warnings in python code: Frequency, awareness and refactoring. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 208–218, 2022.
- [26] Pylint Contributors. Invalid-name / c0103. https://pylint.readthedocs.io/en/latest/user_guide/messages/convention/invalid-name.html.

- [27] Pylint Contributors. Pylint. <https://pylint.readthedocs.io/en/latest/index.html>.
- [28] Pylint Contributors. Pylint message control. https://pylint.readthedocs.io/en/latest/user_guide/messages/message_control.html.
- [29] Pylint Contributors. Pylint message overview. https://pylint.readthedocs.io/en/latest/user_guide/messages/messages_overview.html.
- [30] Pylint Contributors. Running pylint. https://pylint.readthedocs.io/en/latest/user_guide/usage/run.html.
- [31] Sourcegraph. Pygls. <https://pygls.readthedocs.io/en/latest/>.
- [32] Sourcegraph. Pygls github repository. <https://github.com/openlawlibrary/pygls>.
- [33] K. R. Srinath. Python – The Fastest Growing Programming Language. *International Research Journal of Engineering and Technology (IRJET)*, 04(12):354, December 2017. Impact Factor value: 6.171, ISO 9001:2008 Certified Journal.
- [34] Wikidata. Scopus eid. <https://www.wikidata.org/wiki/Property:P1154>.

Appendices

Appendix A

Query Strings, Survey and Interview Questions

Table A.1: Query Strings used to Search on Scopus

Query String	Number of Results
SUBJAREA (comp) AND TITLE-ABS-KEY ((program analysis OR static analysis OR dynamic analysis OR code analysis OR "PYTHON analysis") AND (assess OR measure OR evaluate OR tracking) AND ("developer" OR "user" OR "software engineer" OR "engineer" OR "programmer") AND (experimental study OR user study OR case study OR experiment OR "interviews" OR "survey")) AND (EXCLUDE (DOCTYPE, "cr"))	115
SUBJAREA (comp) AND TITLE-ABS-KEY ((program analysis OR static analysis OR dynamic analysis OR code analysis) AND ("developer experience" OR "usability" OR "programmer experience")) AND (EXCLUDE (DOCTYPE, "cr"))	221
SUBJAREA (comp) AND TITLE-ABS-KEY ((program analysis OR static analysis OR dynamic analysis OR code analysis OR static analyzer OR code analyzer OR dynamic analyzer OR Static Analysis Tools) AND (assess OR measure OR evaluate OR tracking) AND ("developer" OR "user" OR "software engineer" OR "engineer" OR "programmer" OR "usability") AND (experimental study OR user study OR case study OR experiment OR "interviews" OR "survey")) AND (EXCLUDE (DOCTYPE, "cr"))	7
SUBJAREA (COMP) AND TITLE-ABS-KEY ("program analysis" OR "static analysis" OR "dynamic analysis" OR "code analysis" OR "PYTHON analysis" OR "dynamic language analysis") AND ("assess" OR "measure" OR "evaluate" OR tracking) AND ("developer" OR "user" OR "software engineer" OR "engineer" OR "programmer") AND ("experimental study" OR "user study" OR "case study" OR "experiment" OR "interviews" OR "survey")) AND (EXCLUDE (DOCTYPE, "cr"))	276
SUBJAREA (comp) AND TITLE-ABS-KEY ((program analysis OR static analysis OR dynamic analysis OR code analysis OR "Python") AND ("developer experience" OR "usability" OR "programmer experience")) AND (EXCLUDE (DOCTYPE, "cr"))	413
SUBJAREA (comp) AND TITLE-ABS-KEY ((program analysis OR static analysis OR dynamic analysis OR code analysis OR analysis tool) AND (Python) AND (experimental study OR user study OR case study OR experiment OR "interviews" OR "survey"))	31

Section 1

1.1. How much experience do you have working actively with Python? (Select one.)

- Less than 6 months
- Between 6 months and 2 year
- Between 2 and 5 years
- More than 5 years

1.2. What experience do you have with Python? (Select all that apply.)

- University or academy courses
- Personal projects
- Work
- Other (please specify):

1.3. What programming environment do you usually code with Python? (Select all that apply.)

- Pycharm
- Visual Studio
- Sublime Text
- Jupyter Notebook
- Others (please specify):

Section 2

2.1. Figure 1 is an example of static code analysis in Python. Have you ever seen similar highlighted code or warnings before? (Select one.)

- Yes
- No

2.2. In which coding environments have you seen these warnings? (Select all that apply.)

- Pycharm
- Visual Studio
- Sublime Text
- Jupyter Notebook
- Others (please specify):

2.3. Would you typically interact with the highlighted objects? (Select one.)

- Yes
- No
- Sometimes

2.4. If you interact with the highlighted objects, what are the reasons why? (Select all that apply.)

- I don't like to see my code being highlighted.
- I think it will improve the code.
- Other reasons, please specify why:

2.5. If you don't interact with the highlighted objects, what are the reasons why? (Select all that apply.)

- I don't mind the warnings or messages.
- They look useless to me.
- Other reasons, please specify why:

2.6. Figures 2-4 show more examples of Python code analysis results. From your own experience, how has the tool helped you improve your code? (Select all that apply.)

- It helps me write cleaner, more readable code.
- It helps me find errors and bugs at an early stage.
- Others, please specify:

2.7. If you ignore the warnings or messages that the tool displays, what is usually the reason? (Select all that apply.)

- I don't think the warning applies.
- The message is hard to understand.
- I am unsure how to fix the issue.
- Others, please specify why:

2.8. If you encounter a message that you don't understand, what do you typically do? (Select all that apply.)

- Ignore them
- Try to check the issue by myself.
- Look for external help (e.g., other people, the internet).
- Try to understand the message by digging deeper into the tool.
- Other, please specify:

2.9. When do you typically check the warnings? (Select all that apply.)

- When the warning appears as I code.
- After finishing the script.
- Before committing to Git or similar.
- Other, please specify:

Section 3

3.1. Have you ever used code analysers in other programming languages? (Select one.)

- Yes, please specify:
- I have not used code analysers in other programming languages.

3.2. Have you ever manually installed external code analysers into your coding environment to improve your code quality? (Select one.)

- Yes, please specify:
- No

3.3. Have you ever customized or configured a code analyser? (Select one.)

- Yes, please specify:
- No

Would you be willing to participate in an interview to provide further insight? (Select one.)

- Yes, please leave us a contact email:
- No

Disclaimer: The information you provide in this survey will be used for research purposes only. All responses will be kept anonymous and confidential. **Consent statement:** By proceeding with this survey, you agree to allow us to use the information you provide for research purposes. You understand that all responses will be kept anonymous and confidential. If you choose to withdraw from the survey at any time, you may do so without penalty.

```
1 # 1. Undefined variable
2 def calculate_mpg(milesDriven, gallonsUsed):
3     mpg = milesDriven / gallonsUsed
4     mpg = round(mpg, 1)
5     return mpg
6
7 def main():
8     choice = 'y'
9     while choice.lower() == 'y':
10        # get input form user
11        milesDriven = float(input('enter miles driven: '))
12        gallonsUsed = float(input('enter gallons used: '))
13        # call MPG function
14        calculate_mpg(milesDriven, gallonsUsed)
15        print('miles per gallon:\t',mpg)
16        # determine fate of loop
17        choice = input('do you want to continue: y/n: ')
18
19
20
21 # 2. Too many arguments + Non-existing class attribute
22 class Fruit:
23     def __init__(self, color):
24         self.color = color
25
26
27 apple = Fruit("red", "apple", [1, 2, 3])
28 banana = Fruit("yellow")
29 banana_type = banana.type
30
31
32 # 3. Passing parameter of a different type than intended
33 def say_hi(name: str) -> str:
34     return f'Hi {name}'
35
36
```

Figure 1: Piece of code highlighted by a code analyser.

The screenshot shows the PyCharm IDE with the same code as in Figure 1. The Problems window at the bottom is open, displaying a list of errors:

- undefined_variable.py ~~/Documents/Python 36 problems
- ▲ Pylint: Missing module docstring :1
- ▲ Pylint: Missing function or method docstring :2
- ▲ Pylint: Argument name "milesDriven" doesn't conform to snake_case naming style :2
- ▲ Pylint: Argument name "gallonsUsed" doesn't conform to snake_case naming style :2
- ▲ Pylint: Missing function or method docstring :7
- ▲ Pylint: Variable name "milesDriven" doesn't conform to snake_case naming style :11
- ▲ Pylint: Variable name "gallonsUsed" doesn't conform to snake_case naming style :12
- ▲ Pylint: Undefined variable 'mpg' :15
- ▲ Pylint: Missing class docstring :22
- ▲ Pylint: Too few public methods (0/2) :22
- ▲ Mypy: Too many arguments for "Fruit" [call-arg] :27
- ▲ Pylint: Too many positional arguments for constructor call :27

A tooltip is also visible over the code, showing specific error messages for the 'apple' and 'banana' lines:

- Mypy: "Fruit" has no attribute "type" [attr-defined]
- Pylint: Instance of 'Fruit' has no 'type' member

Figure 2: Code analysis results by Pylint and Mypy in Pycharm.

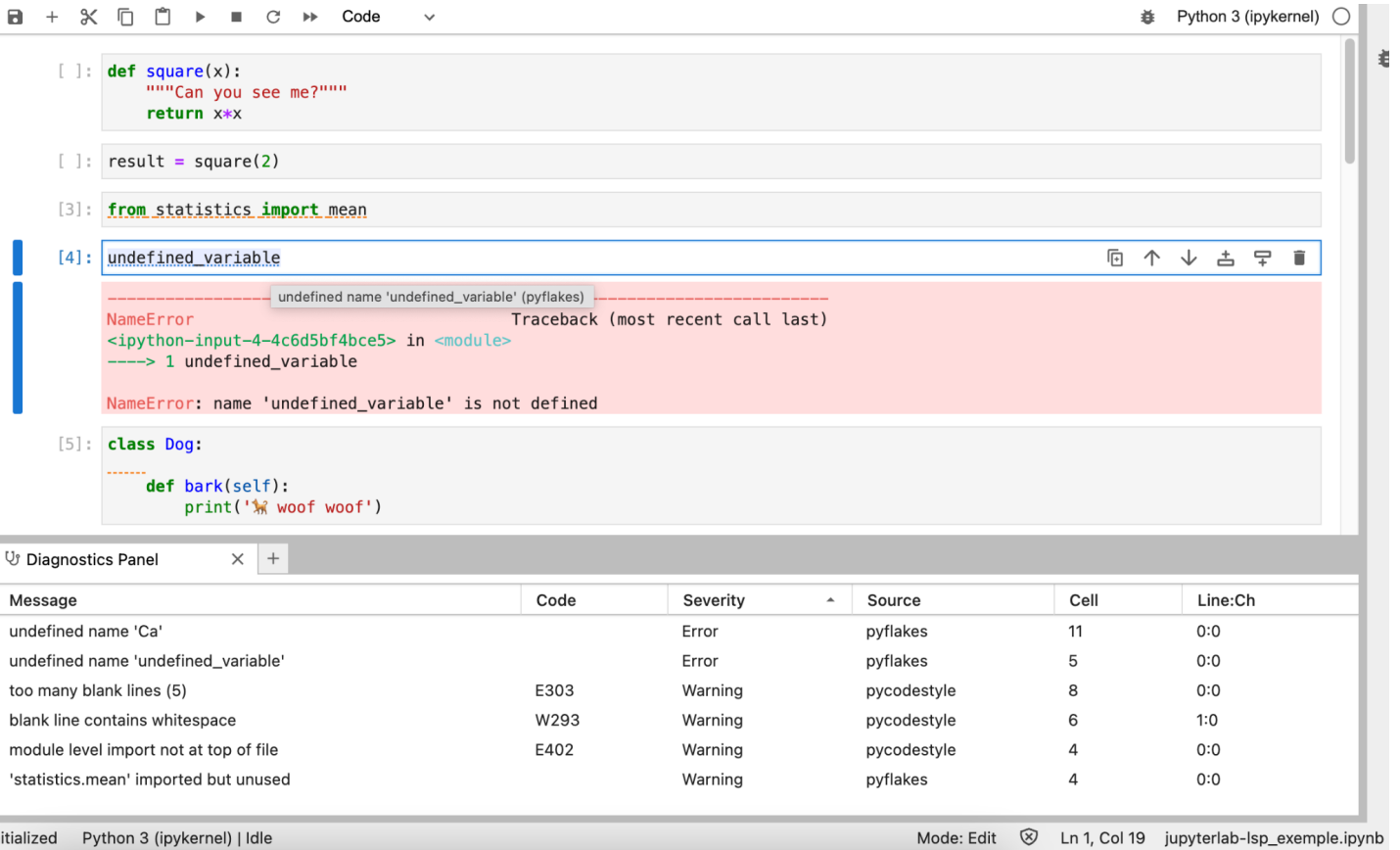


Figure 3: Code analysis results by Jupyter-lsp in Jupyter Notebook.



Figure 4: Code analysis results by Pylint/Flake8 in SublimeText.

Semi-Structured Interview Questions

1. Can you describe your experience with using Python code analysers in the past?
 - 1.1. What do you **like and dislike about the experience**?
2. How do you typically **interact** with the warnings or messages that are displayed by the tool?
 - 2.1. What would be your **coding process**?
3. Can you describe a situation where you found a warning or **message particularly helpful** in improving your code?
4. Can you describe a situation where you found a warning or **message unhelpful or confusing**?
5. How do you typically approach **fixing issues** that are identified by the code analyser?
 - 5.1. Do you use external resources or tools, or rely on your own understanding?
6. Have you ever encountered a situation where you were **unsure how to fix an issue identified by the code analyser**?
 - 6.1. How did you handle this situation?
7. Are there any **features or functionality** that you would like to see **added to Python code analysers** to make them more useful?
8. Can you describe a situation where you feel that the code analyser could have been more helpful in identifying an issue with your code?
 - 8.1. **What could have been done differently to make the tool more useful?**
9. Do you have any suggestions for **how the output** of the code analyser **could be presented in a more user-friendly way**?

Categories:

- Tool experience: what users have experienced with the tool's output (1, 5).
- Workflow: what are the steps that the user takes in the analysis process (2).
- Result Understandability: how the user perceives the information that the analyser provides (3, 4, 6).
- **Tool improvement**: suggestions or opinion from the user about how the tool could be improved (7, 8, 9).

Evaluation Interview Questions

1. Have you encountered similar coding situations where the IDE underlines your code?
2. Which similar code warnings have you encountered in your coding experience?
3. Have you noticed any differences between the two exercises?
4. Addressing the second requirement, have you faced any similar situations where the IDE underlines your variable for not following a naming standard?
5. What do you usually do in those situations?
6. Would you use those ignoring features in your coding routine if they were available?
7. On a scale of 1 to 10, how much does the fact that the tool adds annotations in your source code annoy you?
8. On a scale of 1 to 10, how much does the fact that the tool creates and/or modifies a configuration file in your working directory annoy you?
9. Adding annotations in your code might be more visual and help you be aware of what rules are enabled/disabled, but configuration in a file in your working directory is more discreet. Which one do you prefer?
10. Do you have any suggestions in terms of the new features that would give you options to ignore a rule?

Appendix B

Survey Results

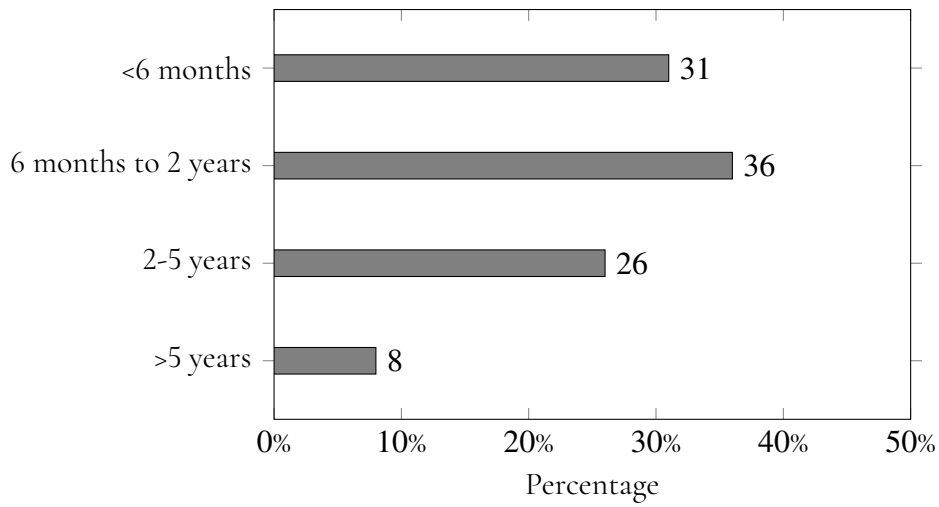


Figure B.1: Question 1.1: How much experience actively working with Python?

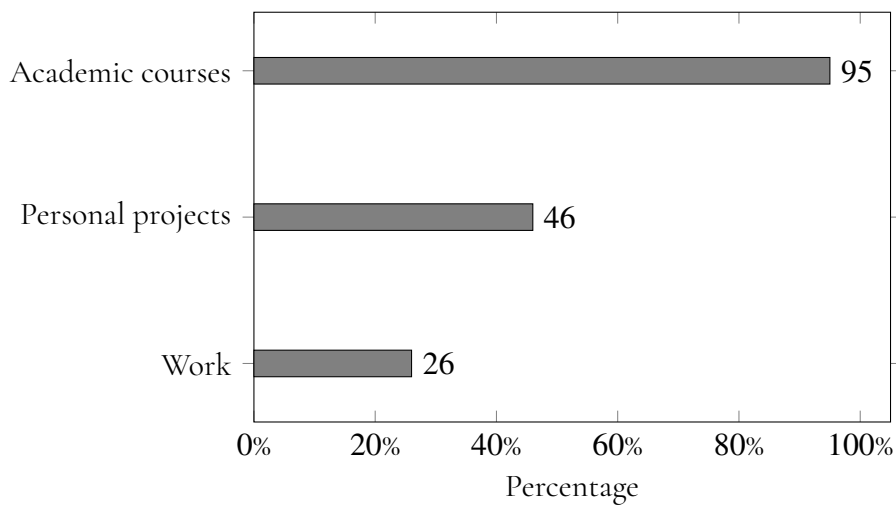


Figure B.2: Question 1.2: What experience do you have with Python?

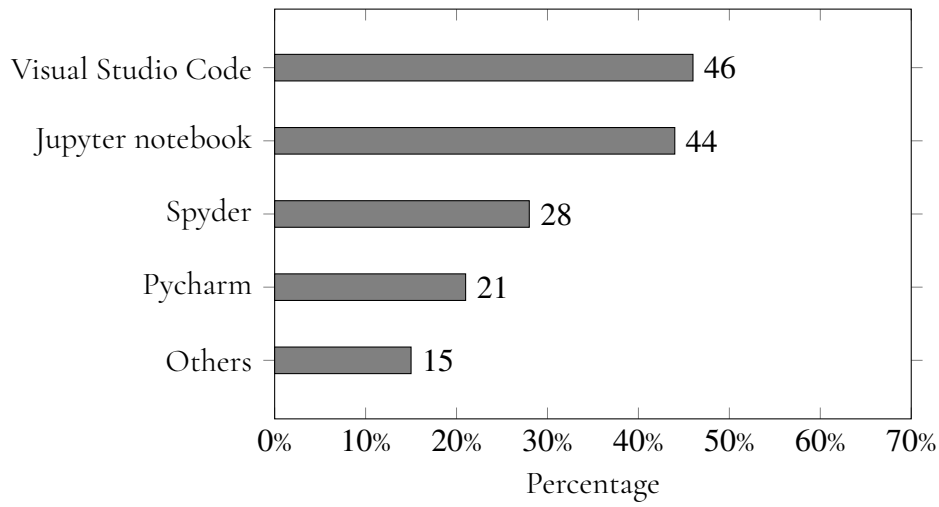


Figure B.3: Question 1.3: What programming environment do you usually code with Python?

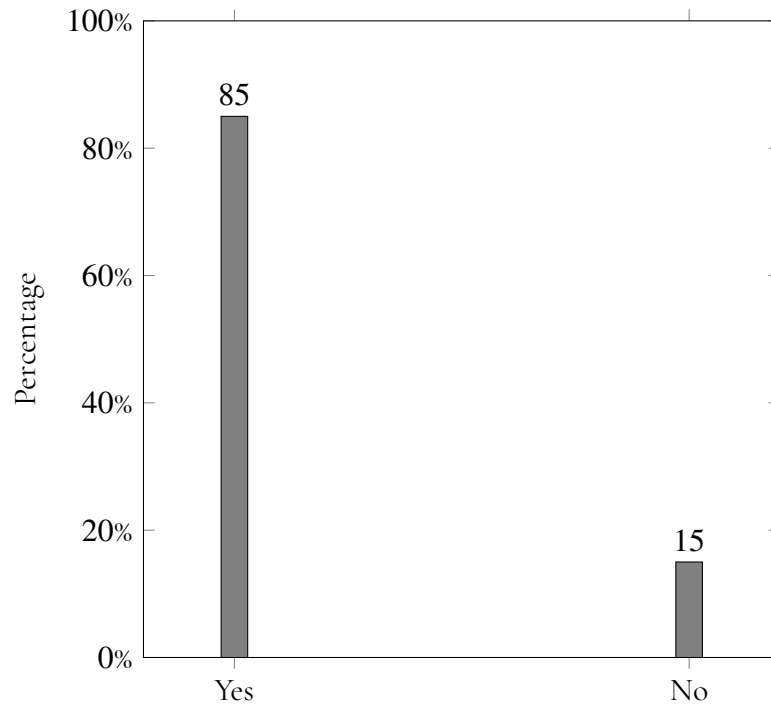


Figure B.4: Question 2.1: Have users seen similar code analysis results?

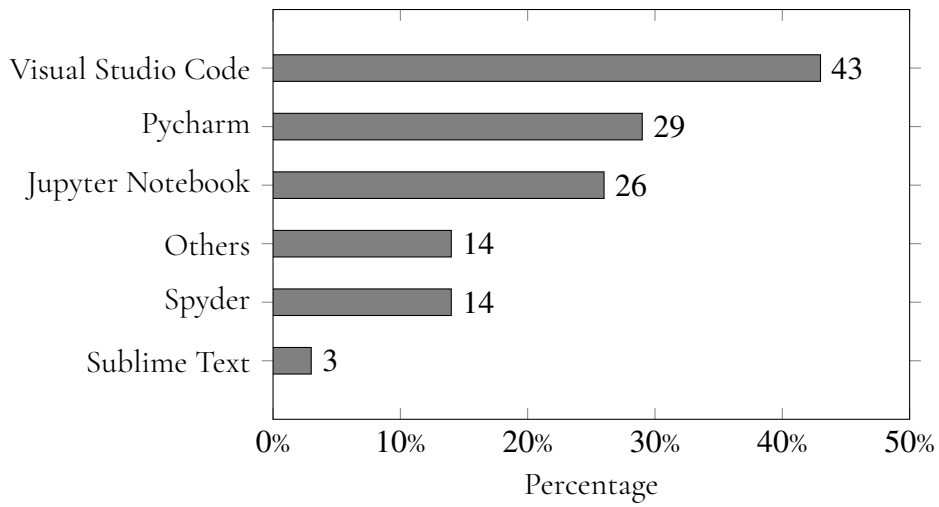


Figure B.5: Question 2.2: In what coding environment have you seen code analysis results?

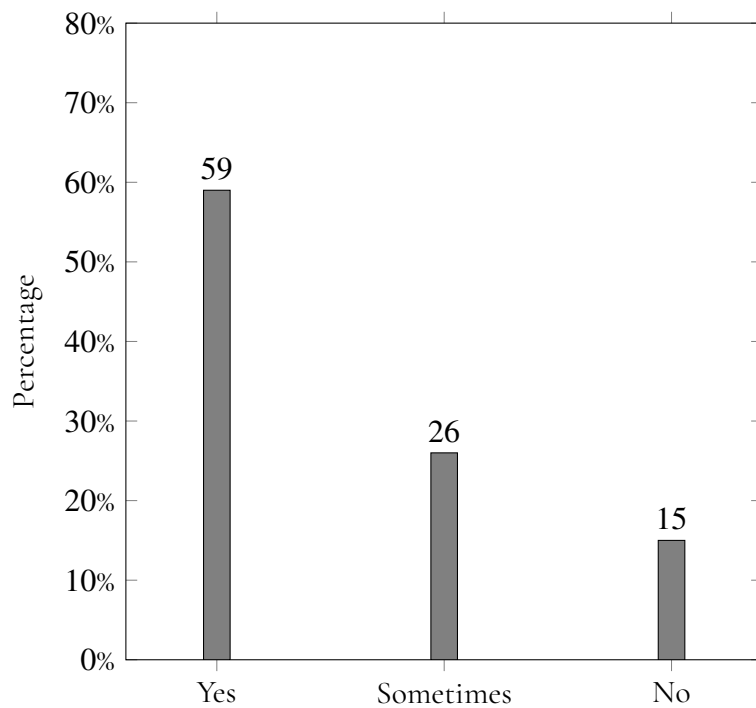


Figure B.6: Question 2.3: Do users check the analysis results?

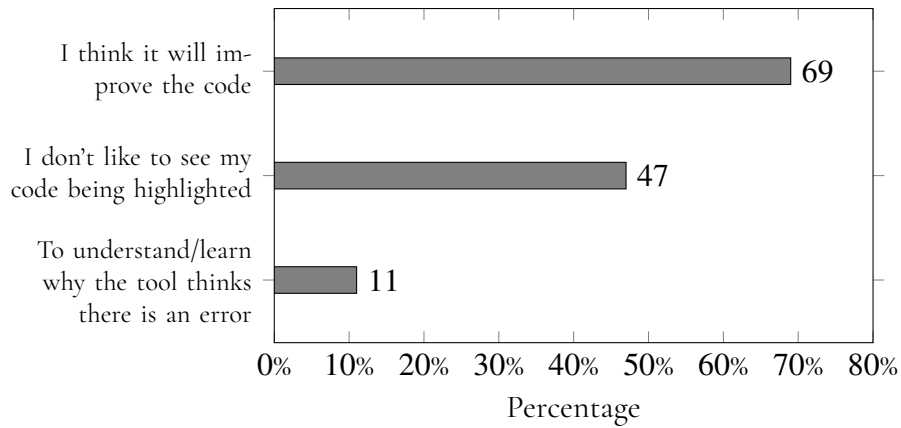


Figure B.7: Question 2.4: What are the reasons of interacting with the highlighted objects?

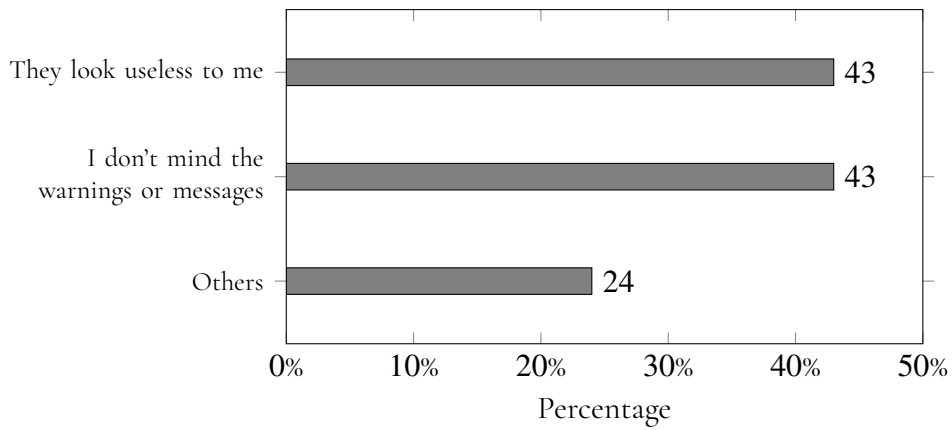


Figure B.8: Question 2.5: What are the reasons of not interacting with the highlighted objects?

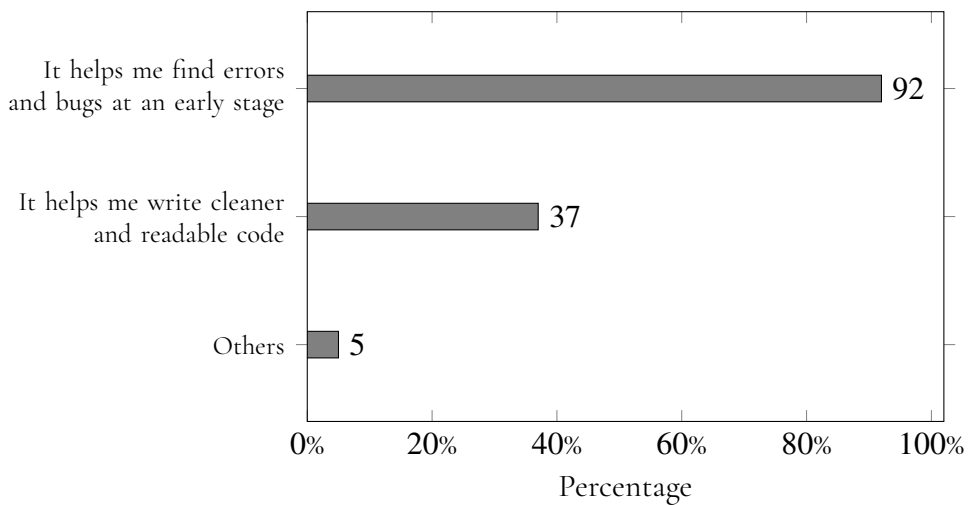


Figure B.9: Question 2.6: How has the tool helped you improve your code?

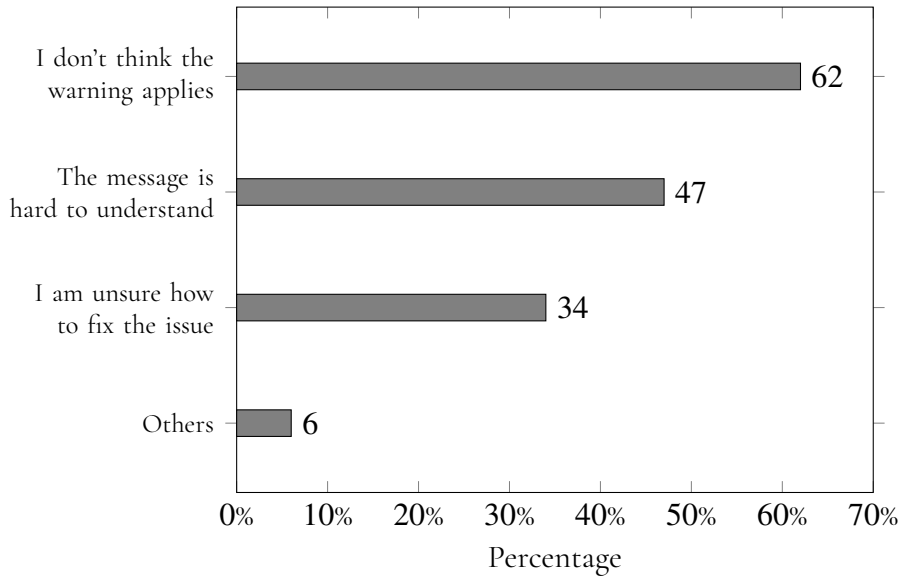


Figure B.10: Question 2.7: If you ignore the warnings or messages that the tool displays, what is usually the reason?

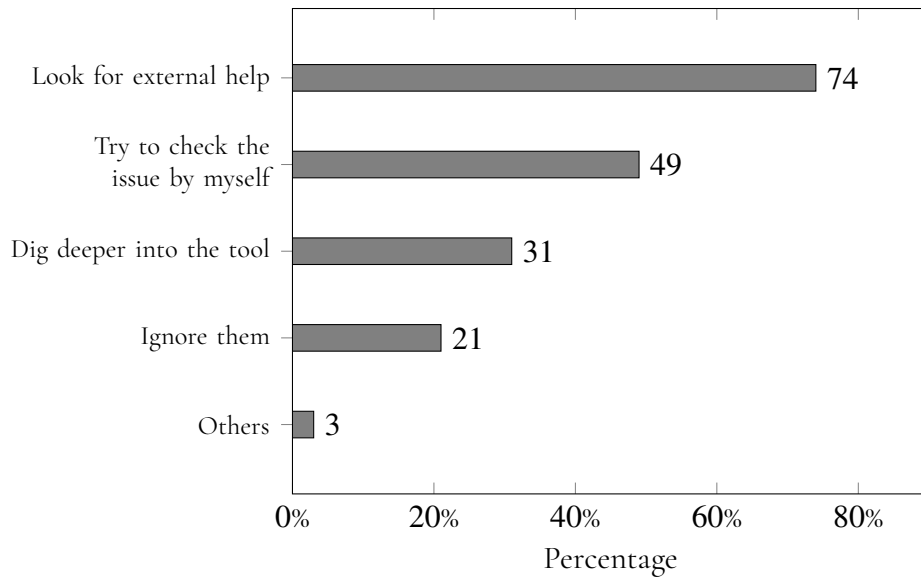


Figure B.11: Question 2.8: If you encounter a message that you don't understand, what do you typically do?

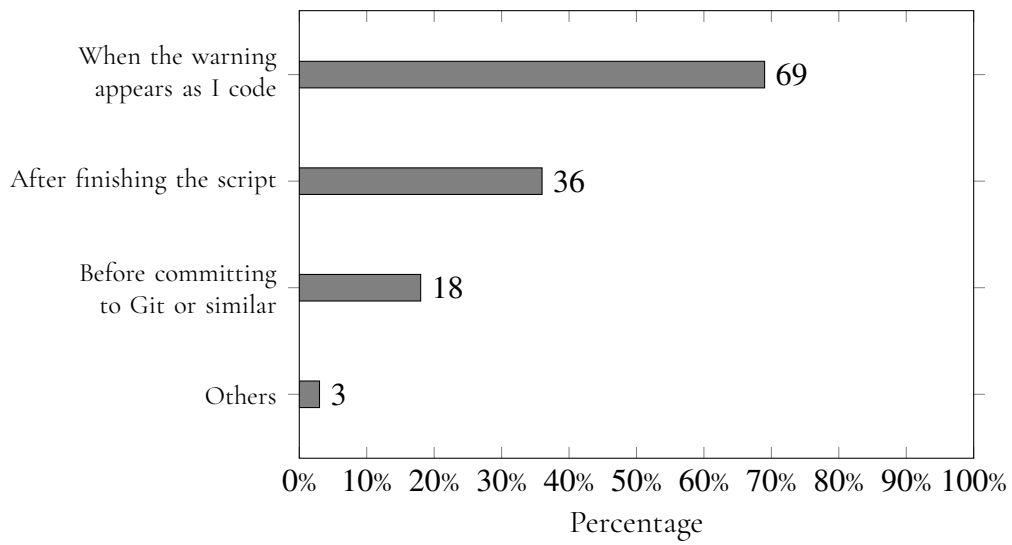


Figure B.12: Question 2.9: When do you typically check the warnings?

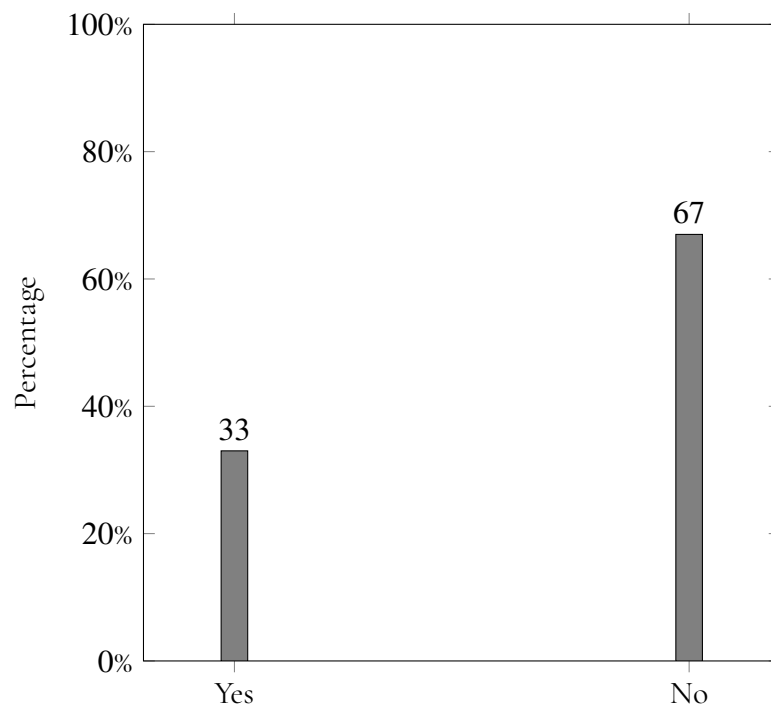


Figure B.13: Question 3.1: Have you ever used code analysers in other programming languages?

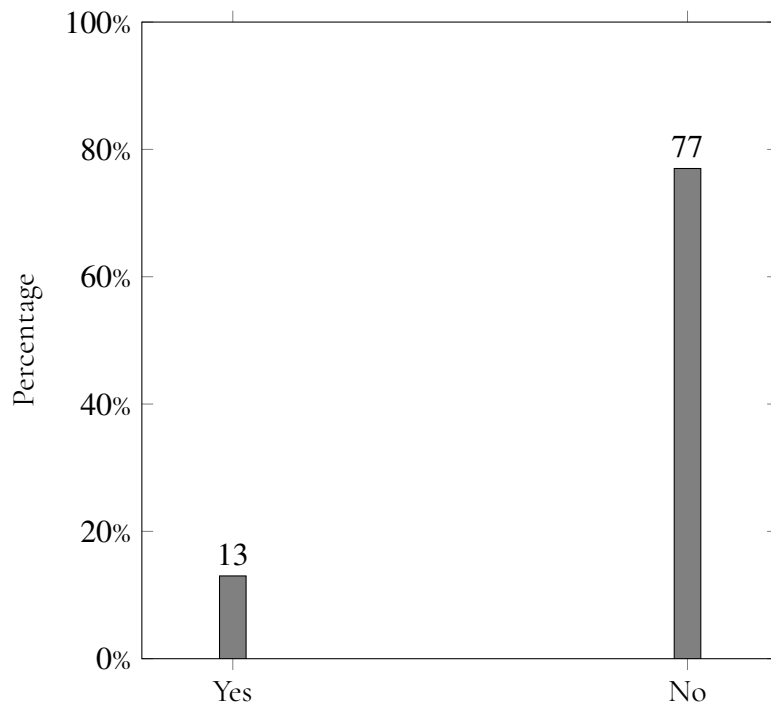


Figure B.14: Question 3.2: Have you ever manually installed external code analysers into your coding environment to improve your code quality?

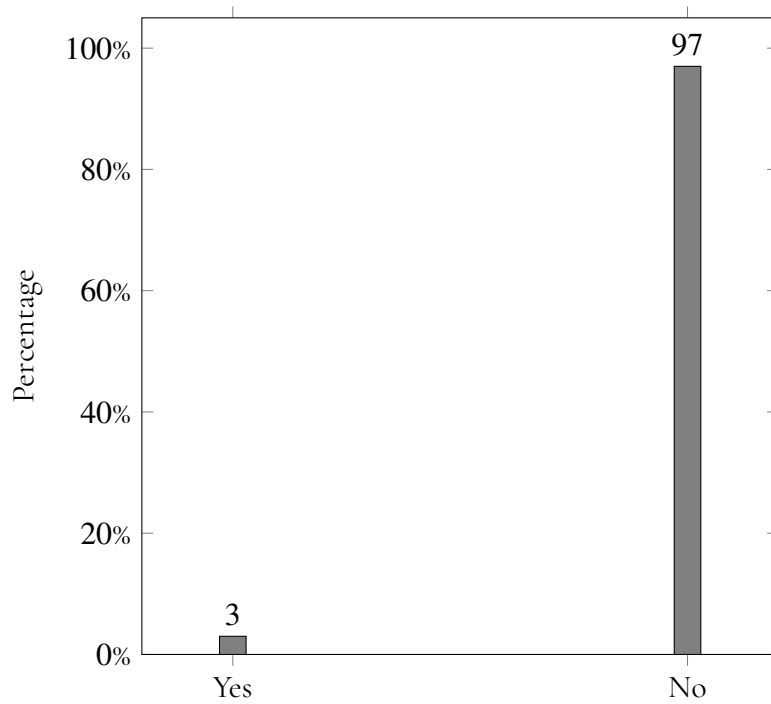


Figure B.15: Question 3.3: Have you ever customised or configured a code analyser?

Appendix C

Code Listing

```
1
2 export async function disableNamingStandard(documentUri:
3   any): Promise<void> {
4   const uri = vscode.Uri.parse(documentUri.path);
5   const workspaceFolder = vscode.workspace.
6     getWorkspaceFolder(uri);
7   const workspacePath = workspaceFolder ?
8     workspaceFolder.uri.fsPath : undefined;
9   const pylintrcPath = workspacePath ? path.join(
10     workspacePath, 'pylintrc') : undefined;
11
12   if (pylintrcPath !== undefined) {
13     try {
14       if (!fs.existsSync(pylintrcPath)) {
15         const generateCmd = `pylint --generate-
16           rcfile > "${pylintrcPath}"`;
17         exec(generateCmd, (error, stdout, stderr)
18           => {
19             if (error) {
20               vscode.window.showErrorMessage(`
21                 Error generating pylintrc: ${
22                   error.message}`);
23             }
24             return;
25           });
26       }
27
28       if (stderr) {
```

```
19         vscode.window.showErrorMessage('
20             Error generating pylintrc: ${
21                 stderr}');
22         return;
23     };
24     });
25     } else {
26         ;
27     }
28 } catch (error: any) {
29     if (error.code === 'EROFS') {
30         vscode.window.showErrorMessage('Cannot
31             update pylintrc: ${pylintrcPath} is a
32             read-only file system.');
```

```
33     } else {
34         vscode.window.showErrorMessage('Error
35             updating pylintrc: ${error.message}');
```

```
36     }
37 }
38 export async function addToGoodNames(documentUri: any,
39     codeName: string): Promise<void> {
40     const uri = vscode.Uri.parse(documentUri.path);
41     const workspaceFolder = vscode.workspace.
42         getWorkspaceFolder(uri);
43     const workspacePath = workspaceFolder ?
44         workspaceFolder.uri.fsPath : undefined;
45     const pylintrcPath = workspacePath ? path.join(
46         workspacePath, 'pylintrc') : undefined;
47
48     if (pylintrcPath !== undefined) {
49         try {
50             if (!fs.existsSync(pylintrcPath)) {
51                 const generateCmd = `pylint --generate-
52                     rcfile > "${pylintrcPath}"`;
53                 exec(generateCmd, (error, stdout, stderr)
54                     => {
55                     if (error) {
56                         vscode.window.showErrorMessage('
57                             Error generating pylintrc: ${
58                                 error.message}');
```

```

51         return;
52     }
53
54     if (stderr) {
55         vscode.window.showErrorMessage('
56             Error generating pylintrc: ${
57                 stderr}');
58         return;
59     };
60     });
61 } else {
62     modifyPylintrcFileToAddName(pylintrcPath,
63         codeName);
64 }
65 } catch (error: any) {
66     if (error.code === 'EROFS') {
67         vscode.window.showErrorMessage('Cannot
68             update pylintrc: ${pylintrcPath} is a
69             read-only file system. ');
70     } else {
71         vscode.window.showErrorMessage('Error
72             updating pylintrc: ${error.message}');
73     }
74 } else {
75     vscode.window.showErrorMessage('Cannot generate
76         or modify pylintrc: no workspace found. ');
77 }
78 }
79
80 function modifyPylintrcFileToAddName(pylintrcPath: string
81     , codeName: string): void {
82     const currentContent = fs.readFileSync(pylintrcPath,
83         'utf8');
84     const lines = currentContent.split('\n');
85     let namingIndex = -1;
86     let goodNamesLineIndex = -1;
87     for (let i = 0; i < lines.length; i++) {
88         if (lines[i].trim() === '[BASIC]') {
89             namingIndex = i;
90         }
91         if (lines[i].startsWith('good-names=')) {
92             goodNamesLineIndex = i;
93         }
94     }
95 }

```

```
88     if (namingIndex >= 0) {
89         if (goodNamesLineIndex >= 0) {
90             if (!lines[goodNamesLineIndex].includes(
91                 codeName)) {
92                 lines[goodNamesLineIndex] = lines[
93                     goodNamesLineIndex].replace('good-
94                         names=', 'good-names=${codeName},');
95             } else {
96                 vscode.window.showInformationMessage('The
97                     code name "${codeName}" is already in
98                     the list of good-names.');
```

```
99             }
100         } else {
101             lines.splice(namingIndex + 1, 0, 'good-names=
102                 ${codeName}');
103         }
104     }
105     fs.writeFileSync(pylintrcPath, lines.join('\n'));
106     vscode.window.showInformationMessage('
107         Successfully updated the pylintrc file to add
108         "${codeName}" to the list of good-names.');
```

```
109 } else {
110     vscode.window.showErrorMessage('The [BASIC]
111         section is not found in the pylintrc file.');
```

```
112 }
113 }
114 }
115 }
116 }
117 }
118 }
119 function modifyPylintrcFile(pylintrcPath: string): void {
120     const currentContent = fs.readFileSync(pylintrcPath,
121         'utf8');
122     const lines = currentContent.split('\n');
123     let messagesControlIndex = -1;
124     let disableLineIndex = -1;
125     for (let i = 0; i < lines.length; i++) {
126         if (lines[i].trim() === '[MESSAGES CONTROL]') {
127             messagesControlIndex = i;
128         }
129         if (lines[i].startsWith('disable=')) {
130             disableLineIndex = i;
131         }
132     }
133     if (messagesControlIndex >= 0) {
134         if (disableLineIndex >= 0) {
135             if (!lines[disableLineIndex].includes('
136                 invalid-name')) {
137                 lines[disableLineIndex] = lines[
```

```

        disableLineIndex].replace('disable=',
        'disable=invalid-name,');
123     } else {
124         vscode.window.showInformationMessage('The
            naming standard violation is already
            disabled in the pylintrc file.');
```

```

    }
125     } else {
126         lines.splice(messagesControlIndex + 1, 0, '
127         disable=invalid-name');
```

```

    }
128     fs.writeFileSync(pylintrcPath, lines.join('\n'));
129     vscode.window.showInformationMessage('
130     Successfully updated the pylintrc file to
        disable the naming standard violation.');
```

```

131 } else {
132     vscode.window.showErrorMessage('The [MESSAGES
        CONTROL] section is not found in the pylintrc
        file.');
```

Listing C.1: Implementation of the commands in TS

```

1  """
2  This exercise should be done in 5 minutes. There are 2
   main objectives:
3
4  1. Eliminate every linted, underlined and highlighted
   code.
5  2. Every naming term should be in capping letters (
   UPPER_CASE).
6
7  You can refresh the warnings by pressing command + S
   saving the file.
8
9  Convention types:
10 - snake_case: Lowercase letters and underscores to
   separate words. Example: user_name, my_module.
11 - PascalCase: Uppercase letters for the first letter of
   each word, with no underscores.
12     Example: CalculateTotalPrice.
13 - UPPER_CASE: Uses uppercase letters and underscores to
   separate words. Example: TAX_RATE.
14
15 """
16
```

```
17 import math
18
19 A= 32
20 a=5
21 print(A+B)
22
23 def create_directory(directory_path):
24     """Create directory"""
25     direct = os.makedirs(directory_path)
26     return direct
27
28
29 class employee:
30     """Employee class"""
31     pass
32
33 def calculate_product_price(product_name, product_type,
34     QUANTITY, PRICEPERUNIT, TAXRATE):
35     total_price = QUANTITY * PRICEPERUNIT
36     product_name='product'
37     product_type='used'
38     tax = total_price * TAXRATE
39     final_price = total_price + tax
40     return final_price
41
42 TEXT = 'Fusce commodo, tellus nec varius bibendum, nisi
43     est fringilla justo, et convallis augue elit in dolor.
44     '
```

Listing C.2: Exercise 1 used to evaluated the prototype

```
1 """
2 This exercise should be done in 5 minutes. There are 2
3     main objectives:
4
5 1. Eliminate every linted, underlined and highlighted
6     code.
7 2. Every naming term should be in lower letters (snake
8     case).
9
10 You can refresh the warnings by pressing command + S
11     saving the file.
12
13 Convention types:
14 - sneak_case: Lowercase letters and underscores to
```

```

    separate words. Example: user_name, my_module.
11 - PascalCase: Uppercase letters for the first letter of
    each word, with no underscores.
12     Example: CalculateTotalPrice.
13 - UPPER_CASE: Uses uppercase letters and underscores to
    separate words. Example: TAX_RATE.
14
15 """
16 import datetime
17
18 name = "John"
19 age = 25
20 print("My name is", name, "and I am", age, "years old.")
21
22 def convert_to_seconds(hours):
23     """Convert minutes to seconds"""
24     seconds = minutes * 60
25     return seconds
26
27 import random
28
29 class animal:
30     """Animal class"""
31     pass
32
33 def calculate_area_of_circle(radius):
34     """Calculate area of circle"""
35     area = math.pi * radius ** 2
36     return area
37
38 sentence = "The quick brown fox jumps over the lazy dog."
39 print("The length of the sentence is:", len(sentence), "
    characters.")

```

Listing C.3: Exercise 2 used to evaluate the prototype

EXAMENSARBETE User-Centric Study and Enhancement of Python Static Code Analysers**STUDENT** Steven Chen**HANDLEDARE** Emma Söderberg, Alan McCabe (LTH)**EXAMINATOR** Martin Höst (LTH)

Decoding Python: Making Code Analysis Tools Friendlier with User-Centered Design

POPULÄRVETENSKAPLIG SAMMANFATTNING **Steven Chen**

Imagine learning Python, but a tool keeps correcting your naming style. This is exactly what novice Python users often face when code analysers flags their variables. Our project combined user-centric research with practical enhancements to Pylint in Visual Studio Code, making Python's naming conventions less daunting.

In an increasingly digital world, programming has become a vital skill. But, for beginners, the experience can be daunting. Static code analysis tools - software that checks code for errors - are invaluable but often intimidating to the uninitiated. This is particularly true for Python, a popular programming language, where the needs of novice users have been largely overlooked.

While Python analysers tools are invaluable for maintaining code quality, they still present many usability challenges that hinder beginners from using them. Addressing this, our degree project adopted a dual approach: insightful user-centric research followed by a practical enhancement of Pylint, a static code analysis tool designed to improve Python code.

First, we dived into the world of novice Python programmers by conducting interviews and surveys. This helped us understand their struggles and frustrations. Armed with these insights, we shifted to the next phase: enhancing Pylint in Visual Studio Code by providing additional quick-fix options for the 'invalid-name' issue.

Instead of forcing users to rectify naming style errors immediately, these quick-fixes offer them

the option to ignore the naming rule at different levels. This introduces an element of flexibility to the learning process, allowing beginners to focus more on understanding Python's basics and less on adhering to strict naming conventions.

The significance of our project lies in its potential to make the early steps of learning Python less intimidating. Code analysis tools like Pylint are integral to writing high-quality code. By making this tool more user-friendly, we create a more inclusive coding environment.

Considering the future implications, our enhancements to Pylint aim to improve the usability of Python analysers overall. By making these tools more accessible and user-friendly, we're not only smoothing the initial journey for beginners but also creating a more efficient coding environment for all users.

In sum, our project exemplifies the fusion of technology and user-centric design. It underscores the importance of user feedback and empathy in creating tools that better meet user needs. Ultimately, we hope our work will serve as a step forward in making Python static code analysers more accessible and user-friendly.