

User-Centric Study and Enhancement of Python Static Code Analysers

Steven Chen
Lund University
UPC - ETSEIB
ohangyu@outlook.com

Emma Söderberg
Lund University
emma.soderberg@cs.lth.se

Alan T. McCabe
Lund University
alan.mccabe@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 developers, has largely overlooked the needs of novice developers and non-static languages like Python. In this paper, we investigate the experiences of novice Python programmers with static code analysis tools. We aim to understand how these novices interact with and perceive these tools, with a focus on identifying usability pain points. To this end, we conducted initial user research with a survey followed by interviews. The insights derived from these studies were used to develop an enhanced version of the Pylint extension to Visual Studio Code, incorporating additional quick-fixes to improve the user experience connected to configuration of static analysis tools. The developed prototype extension was finally evaluated in a user study. The results from the interviews and surveys suggest that false positives, otherwise reported as a dominant cause of usability issues with code analysis, may not be as dominant for novice users who may focus on other aspects and not challenge the code analysis results. In addition, the results from the evaluation give early input on one possible direction for an enhanced Python code analyser interaction focused on novices.

1. Introduction

Code analysers, tools that inspect source code for potential problems like coding standard violations, are often integrated into the development process in different ways (Nachtigall, Nguyen Quang Do, & Bodden, 2019). Some analysers are plugged into integrated development environments (*IDEs*) to provide developers 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, at a time when it 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.

However, while static code analysers can be a powerful tool for developers, they also have limitations. Empirical research over the past decade has looked into programmers' experiences to understand why they need or use static code analysers (Christakis & Bird, 2016; Do, Wright, & Ali, 2022). Johnson et al. (Johnson, Song, Murphy-Hill, & Bowdidge, 2013) found the issue of false positives to be the most impactful reason why professionals stopped using code analysers. They also emphasized the importance of integrating the analysis tool into the coding environment. In a later study by Christakis and Bird (Christakis & Bird, 2016), they also found false positives to be significant barrier and suggested not enabling all code analyser rules by default. Moreover, some tools may require a significant amount of configuration and setup, or may produce messages that are difficult to understand (Nachtigall et al., 2019; Nachtigall, Schlichtig, & Bodden, 2022). These issues can make it challenging for developers to integrate static code analysers into their development process and get benefits from them, especially for novice users.

In our work, we focus on the experiences of novice Python users. As Python has rapidly grown within the computer science industry, particularly in the fields of machine learning and data analysis, there is 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 despite

encountering it within their coding environments, such as *IDEs* or notebooks. With the perspective that a good user experience with code analysers is critical to fully utilise the benefits of these tools (Do et al., 2022), we set out to increase our understanding of novice Python developers' experience of code analysers in order to gain insights into how the design may be adapted to meet their needs.

2. Method

The overall objective of this work is to investigate Python static code analysers and to design and implement improved features that can enhance the efficiency and fluency of the novice programmers' workflow. With this in mind, we focused on the following two research questions:

RQ₁ What experience do novice Python developers have with code analysers?

RQ₂ How can the interaction with Python code analysers be improved for novices?

The work presented in this paper was conducted as part of a M.Sc. thesis project at Lund University (Chen, 2023).

2.1. Data gathering and analysis

To address **RQ₁**, we first gathered empirical data by circulating a questionnaire among novice Python users. This covered topics such as familiarity with code analysers, perceived usefulness, and challenges faced. The questionnaire had three sections: basic information, users' experience with code analysers, and prior knowledge of static code analysers; and included a four screenshots featuring several Python code analysis results as examples (Fig. 4 - Fig. 7). These screenshots were included because we suspected that most users would have previous experience with code analysers, though they may not have had explicit knowledge of the concept and technology. Before distribution, we conducted a pilot survey for feedback, which was used to improve the questionnaire. We collected 39 survey responses from both offline and online sources.

After the survey, we followed up with a semi-structured interview to gather greater insight. The participants for the interviews were recruited from the survey respondents via a final question in the survey. In total, six interviews were conducted and recorded with informed consent, after which they were analysed for relevant content. An overview of participants in the survey and interviews is included in Table 1.

2.2. Prototype development and evaluation

To answer **RQ₂**, we analysed the gathered empirical data for issues that may be addressed with an intervention that could be implemented within the scope of a Master's thesis project. We elected to focus on the Python code analyser Pylint (Microsoft, n.d.; Pylint Contributors, n.d.-b) and the invalid-name (Pylint Contributors, n.d.-a) issue. Over the course of the project, we modified an existing Pylint extension to the Visual Studio Code editor¹, to provide an alternative interaction connected to the invalid name issue. We described the enhanced extension in more detail in Section 3.3.

To evaluate the enhanced Pylint extension, we designed two simple Python exercises containing errors detectable by Pylint. 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, the users were instructed to name every term (constants, variable, class...) using 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 exercises were presented in a random order, one using the existing Pylint extension from the marketplace, and the other using our enhanced Pylint. The exercise was supervised by the first author, and participants were encouraged to think aloud, 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

¹<https://code.visualstudio.com/>

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 file of the Pylint analyser.

Finally, we openly asked for suggestions and feedback to gain insights for future studies and improvements. An overview of participants in the evaluation is included in Table 1 and the prototype extension is available as open-source².

Table 1 – Overview of study participants and their background

Phase	Participant ID	Participant’s Background
Survey	S1	The experience with Python ranged from less than 6 months (31%), 6m-2y (36%), 2-5y (26%), >5y (8%). Python had been encountered in academic courses (95%), personal projects (46%), and work (26%).
	S2	
	⋮	
	S39	
Interview	P1 (also S38)	Industrial/Informatics Engineering
	P2 (also S11)	Informatics Engineering
	P3 (also S35)	Computer Science
	P4 (also S39)	Industrial Engineering
	P5 (also S37)	Computer Science
	P6 (also S36)	Computer Science
Prototype Evaluation	C1	Civil Engineering
	C2	Molecular Biology
	C3 (also P3)	Computer Science
	C4	Computer Science

2.3. Threats to Validity

The **sample size** in this study, especially in the prototype evaluation, 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. For gathering of empirical data and recruitment of participants, there may be a bias in the **sample selection**. Specifically, for the interviews we used convenient 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.

Additionally, it is important to consider the potential influence of experimenter bias on our findings. Participants may have been aware of their involvement in an experiment focused on code analysis. This awareness could have influenced their behavior and responses, leading to experimenter bias. While we cannot confirm this hypothesis definitively, it is a factor worth considering when evaluating the outcomes of our study.

In conducting the survey, we employed both online and offline procedures to collect data, introducing **inconsistent procedure and conditions** as the degree of interaction varied between online and offline

²<https://gitlab.com/lund-university/vscode-pylint-invalid-name>

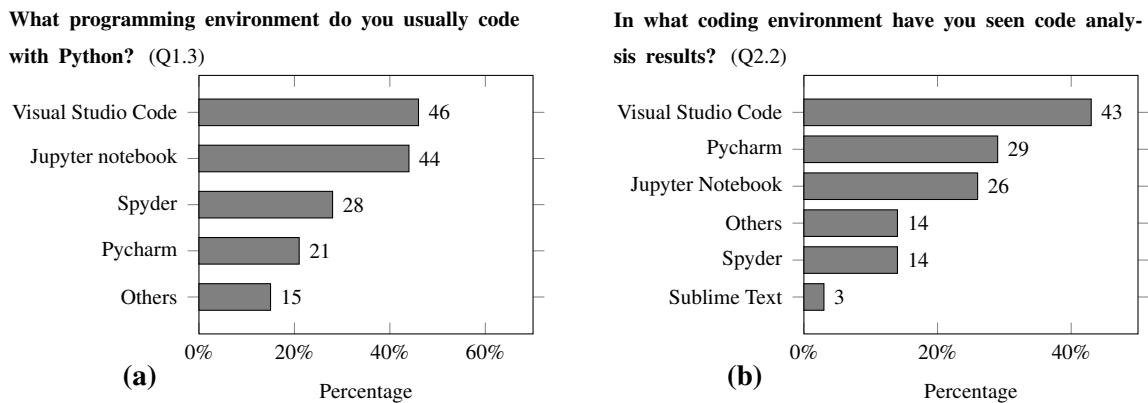


Figure 1 – Summary of the programming environment experience of the participants in the survey.

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. It is 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.

3. Results

In this section, we present the results of the study split into results from different activities; survey, interviews, prototype development, and prototype evaluation.

3.1. Novice Experience with Python Code Analysers: Survey

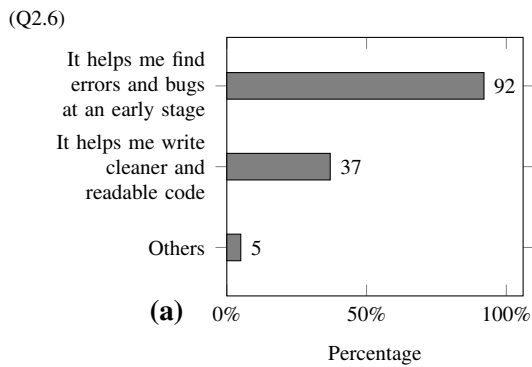
We found that the majority of users (85%) had seen code analysis results for Python before, while a minority (33%) had seen code analysis results for other programming languages. When considering the interaction with code analysis results, a majority of participants (59%) usually check the code analysis results and the majority (69%) typically checks warnings as they appear during coding, as opposed to after finishing the script (36%).

We also asked participants about their usual Python programming environment and found VS Code to be the most commonly used platform (46%), followed by Jupyter Notebook (44%) (see Fig. 1 for details). In connection to the environment, we asked participants if they had ever manually installed code analysers in their coding environment and found that the majority had not done so (77%). We further found that the vast majority of participants (97%) had never configured a code analyser.

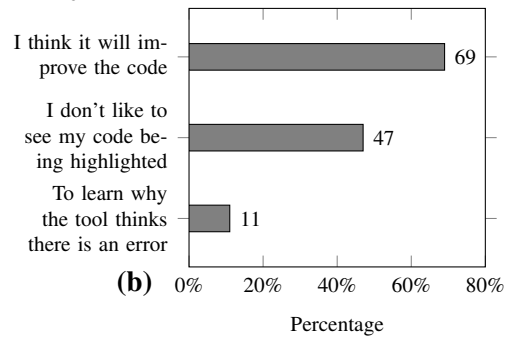
While a majority of the participants (69%) viewed code analysis as beneficial, believing that it has a positive effect on code quality, almost half of the participants (47%) expressed frustration upon seeing their code highlighted (see Fig. 2 (b)). Two reported reasons for ignoring warnings, as opposed to investigating highlighted code, were that the warning looks useless, relating to the concept of false positives, and because of indifference to warnings or messages (Fig. 2 (c)). Despite these frustrations, a high percentage (92%) of users who interact with code analysis agreed that the analyser positively influences their coding process by detecting errors and bugs early (Fig. 2 (a)). When it comes to the reason for ignoring warning messages *after* reading them, 62% of users felt the warnings didn't apply (false positives), and 47% considered the problem difficult to fix (Fig. 2 (d)). In some cases, users mentioned that the problem looked hard to fix, meaning that the tool did not provide an immediate or adequate solution.

For cases where a participant might encounter a code analysis result they do not understand, the majority

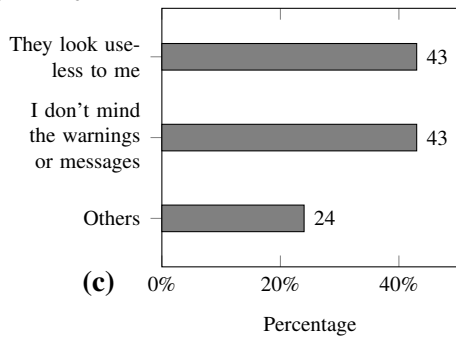
How has the tool helped you improve your code? (Q2.6)



What are the reasons for interacting with the highlighted objects? (Q2.4)



What are the reasons for not interacting with the highlighted objects? (Q2.5)



If you ignore the warnings or messages that the tool displays, what is usually the reason? (Q2.7)

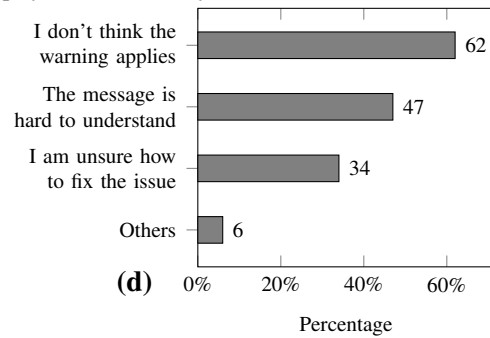


Figure 2 – Summary of survey participants' reported interaction with code analysis.

look to the internet or other programmers (74%) for assistance, while others attempt to fix the issue directly (49%), dig deeper into the code analysis tool (31%), and a fifth of the participants ignore the results altogether (21%).

The full text of the questionnaire can be found in appendix A, and a detailed breakdown of the results can be found in appendix B.

3.2. Novice Experience with Python Code Analysers: Interviews

All the participants in the interview part of the study expressed a positive attitude towards code analysers in general. They conveyed the belief that they are powerful tools that improve their coding by detecting errors at an early stage. (e.g., "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).

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, which can inform the design of features to address issues (e.g., "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, "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, and "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).

We asked the respondents if they felt whether the warning messages given by analysis tools were clear enough, and what they typically do if they don't understand them. Based on their responses, we found that messages are usually understandable when the issue is simple, but when it becomes more complex the messages can become difficult to understand and do not provide enough options for clarification. Most users tended to turn to Google to find a solution (e.g., *"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, and *"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).

False positives are a major usability problem with code analysers. To better understand the frequency of this issue among students and novice Python developers, we asked the respondents about their experience with false positives (e.g., *"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, and *"I don't think that's ever happened. My code doesn't usually get too complicated."* – P4).

We asked the interview participants 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, and the attitude towards suggestions was mixed (e.g., *"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, and *"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).

Finally, 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 (e.g., *"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, and *"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).

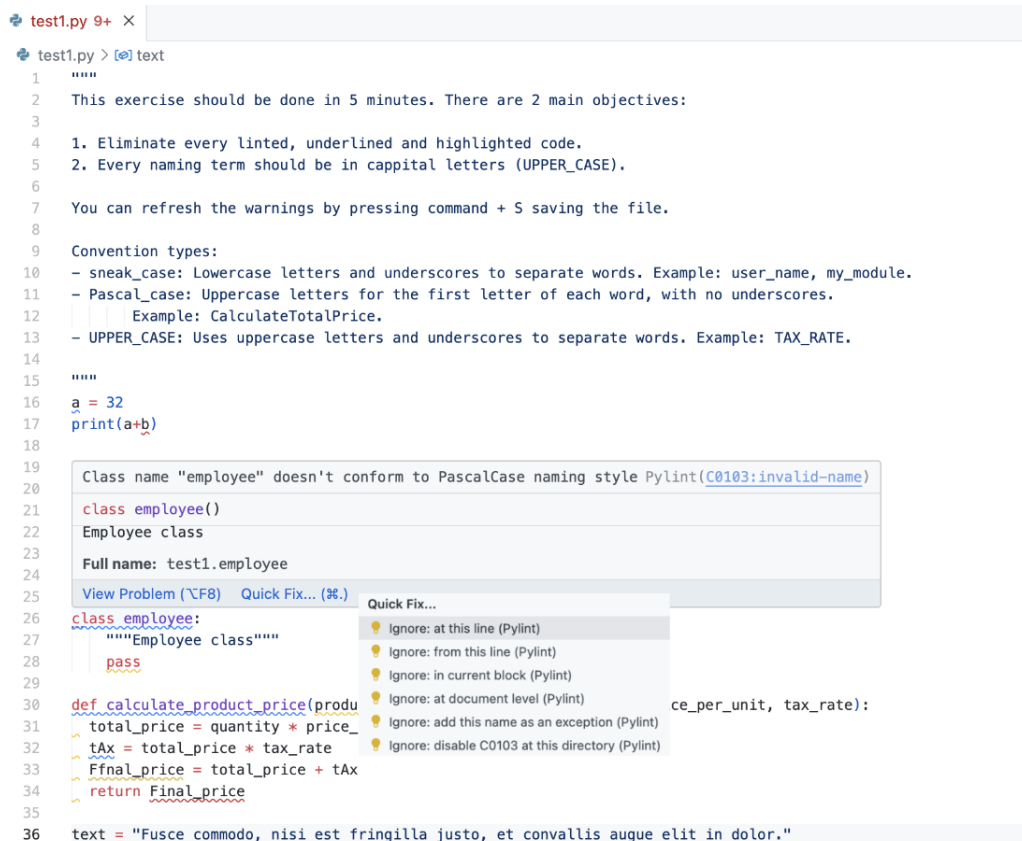
3.3. Prototype Development: Invalid Name Config Support

When analysing the interview results, one specific issue that caught our attention was 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, with a term that does not follow Python's official PEP8 (Guido van Rossum and Barry Warsaw and Nick Coghlan, 2001) style standard, 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 formatting, capitalization, or case. The fact that the tool provide neither a solution nor a fast and intuitive way to ignore the warning can be frustrating for users and may discourage them from using the tool. We found this to be an interesting problem to study and address.

We opted to tackle the invalid-name issue in the context of VS Code, as it was the most used programming environment among survey respondents. We found that Pylint, a well known Python code analyser available as an extension to VS Code, offers a limited set of quick-fixes that address 14 code problems which do not include the invalid-name issue. Hence, we set forth to explore a solution for this issue in a

variant of the Pylint extension for the Visual Studio Code editor³.

In our implementation, we developed four quick-fix options that enable users to address the invalid-name issue at different levels: on a single line, from a specific line, within the current block, or at the document level by adding annotations. Additionally, we designed two quick-fix options that handle a `pylintrc` configuration file, allowing users to either disable the rule entirely in the working directory or add the treated naming variable as an always accepted term. We also considered adding support for a quick fix that would adjust the name to the form suggested by the issue, but unfortunately we did not have enough time in the project to also develop this feature. The final view of our implementation is illustrated in Fig. 3.



```
test1.py 9+ X
test1.py > [e] text
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 cappital letters (UPPER_CASE).
6
7 You can refresh the warnings by pressing command + S saving the file.
8
9 Convention types:
10 - sneak_case: Lowercase letters and underscores to separate words. Example: user_name, my_module.
11 - Pascal_case: 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 a = 32
17 print(a+b)
18
19 Class name "employee" doesn't conform to PascalCase naming style Pylint(C0103:invalid-name)
20
21 class employee()
22 Employee class
23
24 Full name: test1.employee
25
26 View Problem (⌘F8) Quick Fix... (⌘.) Quick Fix...
27 class employee:
28     """Employee class"""
29     pass
30
31 def calculate_product_price(produ
32     total_price = quantity * price_
33     tAx = total_price * tax_rate
34     Ffinal_price = total_price + tAx
35     return Final_price
36
37 text = "Fusce commodo, nisi est fringilla justo, et convallis augue elit in dolor."
```

Figure 3 – Implemented quick-fixes.

3.4. Prototype Evaluation: Invalid Name Config Support

The two types of quick-fix options developed in the prototype extension address the invalid name issue 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 can be an 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 desirable 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`

³The extension with more implementation details is available at <https://gitlab.com/lund-university/vscode-pylint-invalid-name>.

file and modify its content, which might not be an intuitive process for developers.

The purpose of this final evaluation was 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 asked the participants to edit two simple Python programs via instructions given in comments in the beginning of each file. The instructions gave the participants the task of naming terms according to a specific naming scheme, edits designed to trigger naming issues generated by Pylint. The participants carried out one of the tasks with the new Pylint extension developed in the project and one task with the default Pylint extension. We aimed 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.

All of the volunteers interviewed in the evaluation 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.

3.4.1. 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.

3.4.2. 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 that of C4: they 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.

3.4.3. 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.

4. Discussion and Conclusions

Interestingly, the mainly novice developers among our participants did not appear to be significantly affected by false positives. Although a high percentage of respondents in Fig. 2 answered that they ignore warning messages because they believe the warning does not apply, none of the users we interviewed could provide an example of frustration with false positives.

Instead, in the suggestion section of the interviews, our participants 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 possible explanation for this contrasting perspective lies in the expertise level of our participants. Given that most of them had limited experience with the Python language, they may have a higher level of trust in the code analyser compared to their own understanding. In contrast, professional developers, with their extensive knowledge and expertise, may exhibit a higher level of confidence in assessing the correctness of the analysis tool. Novices, therefore, might be more inclined to rely on the analyser's results, even when confronted with warning messages that they believe do not apply directly to their code.

Another contributing factor could be the purpose for which novices typically use the Python language. Since many novice users are likely to work on smaller-scale projects or specific tasks, their code is generally less complex compared to those of professional developers. As a result, they may encounter fewer instances of false positives, as the simplicity of their code reduces the probability of provoking such cases.

Many of the issues that later emerged during the prototype evaluation interviews were related to the general usability of the tool, suggesting that there is a need for further design iteration. For instance, one of the most significant issues was that 3 of the 4 candidates did not recognise the difference between the original and enhanced Pylint tool. The reason for this 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 survey findings, 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 error code "C0103", not the name of the issue itself.

Based on what we found in this study we see a need for further study of novice interaction with code analysis, both in Python and also other programming languages. As such, we have identified several directions for future work:

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

5. Acknowledgements

The authors would like to thank all the participants in the study. This work has been partially supported by the Swedish Foundation for Strategic Research (grant no. FFL18-0231), the Swedish Research Council (grant no. 2019-05658), ELLIIT - the Swedish Strategic Research Area in IT and Mobile Communications, and the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

6. References

- Chen, S. (2023). *User-centric study and enhancement of python static code analysers* (LU-CS-EX 2023-31). Lund University.
- Christakis, M., & Bird, C. (2016). What developers want and need from program analysis: An empirical study. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)* (p. 332-343).
- Do, L. N. Q., Wright, J. R., & Ali, K. (2022). 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. doi: 10.1109/TSE.2020.3004525
- Guido van Rossum and Barry Warsaw and Nick Coghlan. (2001). *PEP 8 – Style Guide for Python Code*. Python Enhancement Proposal. Retrieved from <https://pep8.org/> (Status: Active. Created: 5-Jul-2001)
- Johnson, B., Song, Y., Murphy-Hill, E., & Bowdidge, R. (2013). Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)* (p. 672-681). doi: 10.1109/ICSE.2013.6606613
- Microsoft. (n.d.). *Vscode-pylint*. (<https://github.com/microsoft/vscode-pylint>)
- Nachtigall, M., Nguyen Quang Do, L., & Bodden, E. (2019). Explaining static analysis - a perspective. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)* (p. 29-32). doi: 10.1109/ASEW.2019.00023
- Nachtigall, M., Schlichtig, M., & Bodden, E. (2022). 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* (pp. 532–543). doi: 10.1145/3533767.3534374
- Pylint Contributors. (n.d.-a). *Invalid-name / c0103*. (https://pylint.readthedocs.io/en/latest/user_guide/messages/convention/invalid-name.html)
- Pylint Contributors. (n.d.-b). *Pylint*. (<https://pylint.readthedocs.io/en/latest/index.html>)

A. Questionnaire

Please see below for questions included in the questionnaire. The screenshots included in the questionnaire are included in Fig. 4, Fig. 5, Fig. 6, and Fig. 7.

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 from 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

```

Figure 4 – Piece of code highlighted by a code analyser.

```

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 para
33 def say_hi(name: str)
34 main() while choice
35 banana = Fruit("yellow")

```

Problems: Current File 36

- 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

Tooltip for class Fruit:

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

Figure 5 – Code analysis results by Pylint and Mypy in Pycharm.

B. Survey results

Please see Fig. 8 for additional survey results complementing the details included in Section 2.

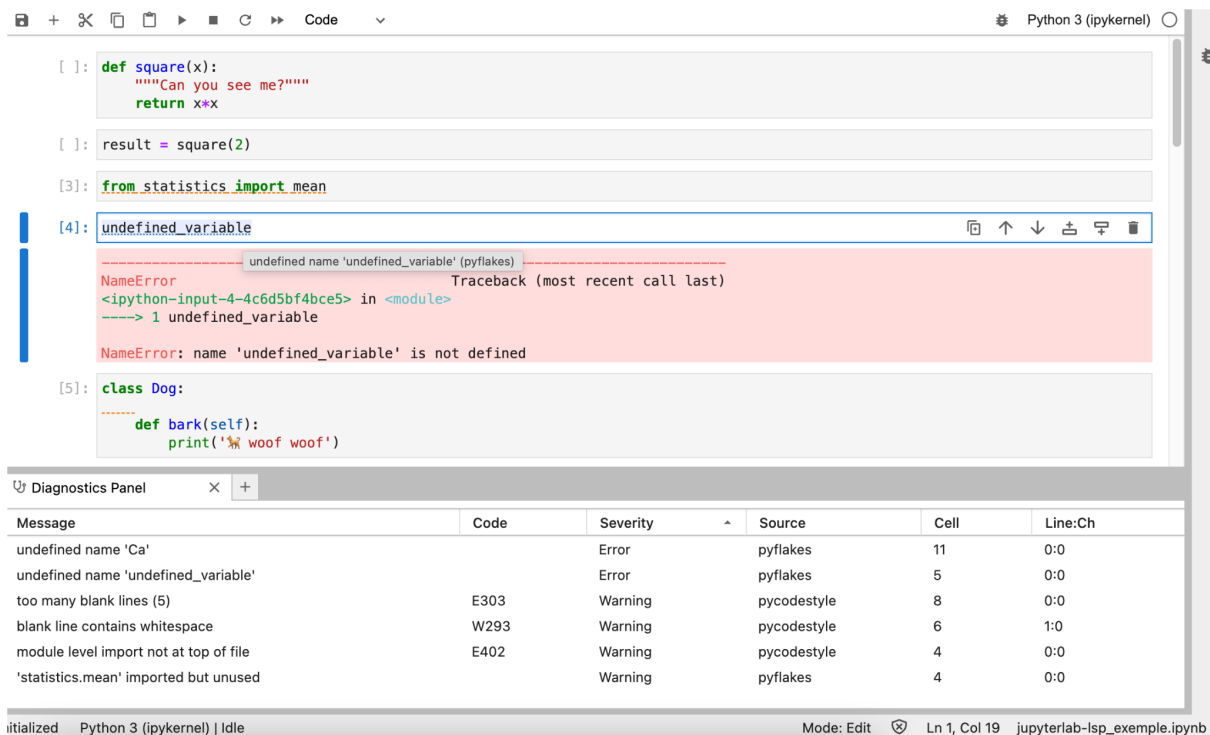


Figure 6 – Code analysis results by Jupyter-lsp in Jupyter Notebook.

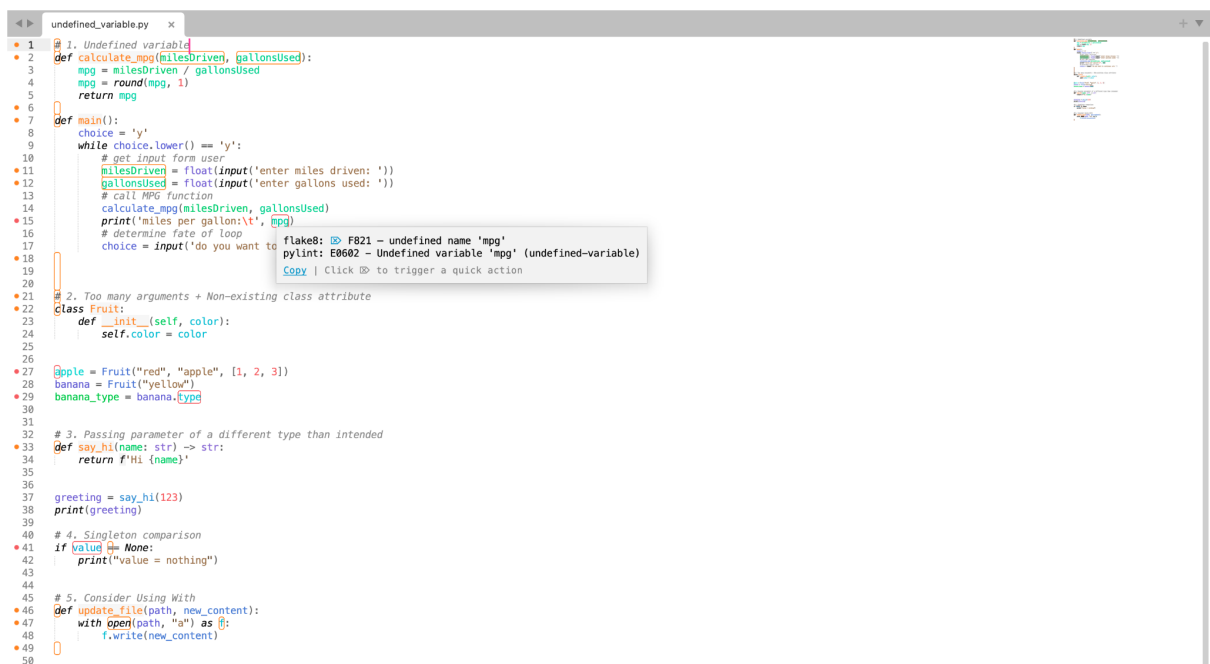
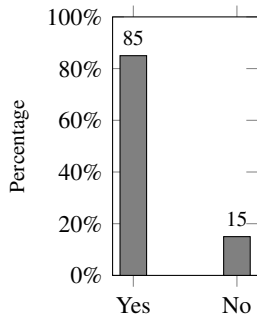
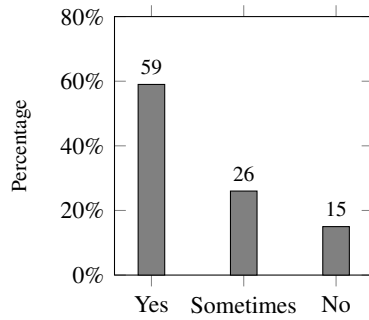


Figure 7 – Code analysis results by Pylint/Flake8 in SublimeText.

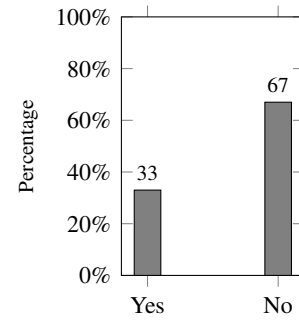
Have users seen similar code analysis results? (Q2.1)



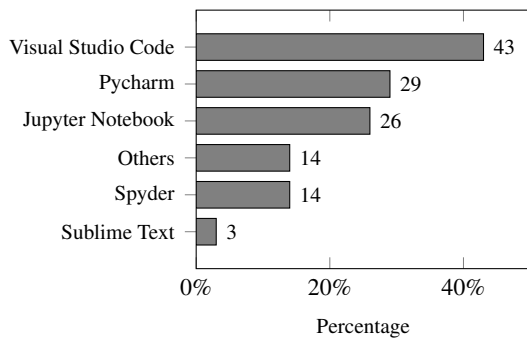
Do users check the analysis results? (Q2.3)



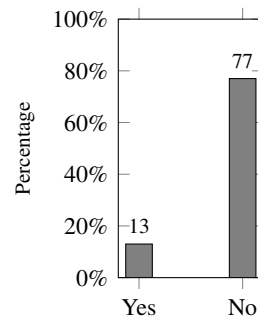
Have you ever used code analysers in other programming languages? (Q3.1)



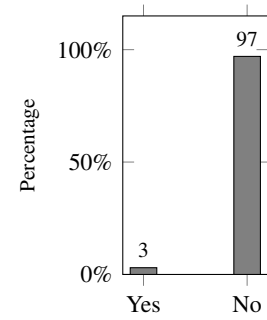
In what coding environment have you seen code analysis results? (Q2.2)



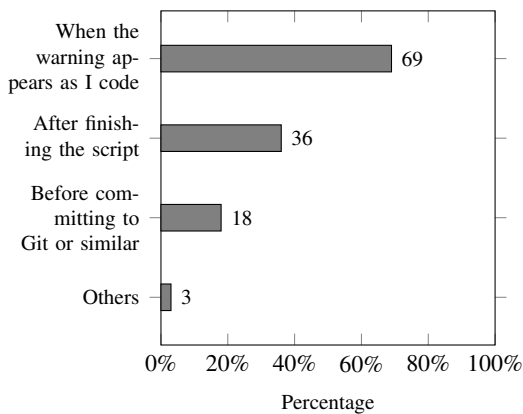
Have you ever manually installed external code analysers into your coding environment to improve your code quality? (Q3.2)



Have you ever customised or configured a code analyser? (Q3.3)



When do you typically check the warnings? (Q2.9)



If you encounter a message that you don't understand, what do you typically do? (Q2.8)

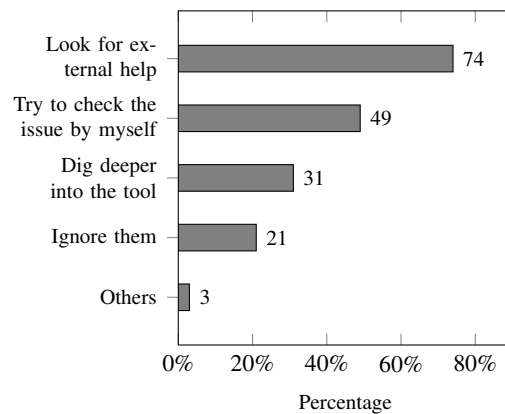


Figure 8 – Summary of survey results.