UNIVERSIDADE FEDERAL DE ALAGOAS
INSTITUTO DE COMPUTAÇÃO
PROGRAMA DE PÓS GRADUAÇÃO
INFORMÁTICA

Naelson Douglas Cirilo Oliveira

LINT-BASED WARNINGS IN PYTHON CODE: FREQUENCY, AWARENESS AND REFACTORING

Maceió - AL
2022

NAELSON DOUGLAS CIRILO OLIVEIRA

LINT-BASED WARNINGS IN PYTHON CODE: FREQUENCY, AWARENESS AND REFACTORING

Dissertação apresentada ao Programa de Pós-Graduação em Informática da Universidade Federal de Alagoas, como requisito para obtenção do grau de Mestre em Informática

Orientador: Prof. Dr. Márcio de Medeiros Ribeiro

Maceió - AL
2022

Catalogação na Fonte
Universidade Federal de Alagoas
Biblioteca Central
Divisão de Tratamento Técnico
Bibliotecário: Marcelino de Carvalho Freitas Neto – CRB-4 - 1767

O48l Oliveira, Naelson Douglas Cirilo.
Lint-based warnings in Python code : frequency, awareness and refactoring / Naelson Douglas Cirilo Oliveira. – 2022.
52. : il.

Orientador: Márcio de Medeiros Ribeiro.
Dissertação (mestrado em informática) - Universidade Federal de Alagoas. Instituto de Computação. Maceió, 2022.
Texto em inglês.

Bibliografia: f. 48-52.

1. Python (Linguagem de programação de computador). 2. Análise estática. I. Título.

CDU: 004.43

UNIVERSIDADE FEDERAL DE ALAGOAS/UFAL
Programa de Pós-Graduação em Informática – PPGI
Instituto de Computação/UFAL
Campus A. C. Simões BR 104-Norte Km 14 BL 12 Tabuleiro do Martins
Maceió/AL - Brasil CEP: 57.072-970 | Telefone: (082) 3214-1401

Folha de Aprovação

NAELSON DOUGLAS CIRILO OLIVEIRA

LINT-BASED WARNINGS IN PYTHON CODE: FREQUENCY, AWARENESS, AND REFACTORING

Dissertação submetida ao corpo docente do Programa de Pós-Graduação em Informática da Universidade Federal de Alagoas e aprovada em 29 de agosto de 2022.

Banca Examinadora:

gov.br Documento assinado digitalmente
MARCIO DE MEDEIROS RIBEIRO
Data: 05/09/2022 11:20:11-0300
Verifique em https://verificador.iti.br

Prof. Dr. MARCIO DE MEDEIROS RIBEIRO
UFAL – Instituto de Computação
Orientador

gov.br Documento assinado digitalmente
BALDOINO FONSECA DOS SANTOS NETO
Data: 04/09/2022 09:21:27-0300
Verifique em https://verificador.iti.br

Prof. Dr. BALDOINO FONSECA DOS SANTOS NETO
UFAL – Instituto de Computação
Examinador Interno

Prof. Dr. ROHIT GHEYI
Universidade Federal de Campina Grande-UFCG
Examinador Externo

RESUMO

Python é uma linguagem de programação caracterizada por sua sintaxe simples e curva de aprendizagem fácil. Como muitas linguagens, Python tem um conjunto de boas práticas que devem ser seguidas com o objetivo de evitar *bugs* e melhorar outros atributos qualitativos como manutenibilidade e legibilidade. Neste contexto, a não observação destas práticas pode ser detectada usando ferramentas de *linting*. Trabalhos anteriores conduziram estudos de modo a melhor entender a frequência de uma classe de problemas que podem ser encontrados com linters de Python: warnings, aqui chamados de warnings detectado pror *linting* (lint-based warning). De toda forma, estes trabalhos ou dependem de pequenos conjuntos de dados ou focam em apenas poucos domínios como aprendizagem de máquina ou projetos de desenvolvimento de sistemas web. Neste trabalho, nós fornecemos um estudo de métodos mistos onde analizamos a frequência de seis warnings detectados em linting em mil cento e dezenove diferente projetos Python de código aberto. Além disto, nós conduzimos uma pesquisa para conferir se os desenvolvedores conseguem notar os warnings que estudamos aqui. Em particular, queremos checar se eles são conseguem identificar tais warnings. E finalmente, nós avaliamos as sugestões ao submeter *pull requests* para remover os warnings em projetos de código aberto. Nossos resultados mostram que 38% de 1119 projetos têm ao menos um warning detectado por linting. Após analisarmos os resultados da pesquisa, nós também mostramos que desenvolvedores Python preferem código sem os warnings detectados por linters. Sobre os pull requests, nós conseguimos uma taxa de 71.8% de aceitação. Nós então utilizamos o conhecimento obtido nos estudos iniciais e implementamos um plugin grátis e de código aberto para Visual Studio Code que é capaz de não apenas detectar, mas também corrigir os warnings detectados por linting que estudamos aqui.

Key-words: Python, Linters, Análise Estática.

ABSTRACT

Python is a popular programming language characterized by its simple syntax and easy learning curve. Like many languages, Python has a set of best practices that should be followed to avoid bugs and improve other quality attributes (such as maintenance and readability). In this context, non-compliance to these practices can be detected by using linting tools. Previous work conducted studies to better understand the frequency of a class of problems that can be found using Python linters: warnings, here named as lint-based warnings. However, they either rely on small datasets or focus on few domains, such as machine learning or web-systems projects. In this work, we provide a mixed-method study where we analyze the frequency of six lint-based warnings in 1,119 different open-source general-purpose Python projects. To go further, we also conduct a survey to check whether developers are aware of the lint-based warnings we study here. In particular, we intend to check whether they are able to identify the six lint-based warnings. To remove the lint-based warnings, we suggest the application of simple refactorings. Last but not least, we evaluate the suggestions by submitting pull requests to remove lint-based warnings from open-source projects. Our results show that 39% of the 1,119 projects have at least one lint-based warning. After analyzing the survey data, we also show that although the perception of inexperienced developers might differ, most of the developers prefer Python code without lint-based warnings. Regarding the pull requests, we achieve a 71.8% of acceptance rate and part of the denied pull requests were not caused by a rejection of the refactoring, but due to internal policies of the project. We then used the knowledge gathered throughout the three initial studies to implement a free and open source Visual Studio Code plugin able to not only detect, but also fix the lint-based warnings we study.

Key-words: Python, Linting, Static Analysis.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

CUE	Consider using enumerate;
CUI	Consider using in;
CUW	Consider using with;
DDV	Dangerous default value;
RBI	Redefined built-in;
SC	Singleton comparison;

CONTENTS

1 INTRODUCTION

Python is a popular programming language and it is well known for its fast learning curve, simplicity, and flexibility (SAABITH; FAREEZ; VINOTHRAJ, 2019). These characteristics make it a viable option for both new and experienced developers (RAMALHO, 2015; BASSI, 2007). As many languages, Python has a set of best practices that should be followed to avoid bugs and improve the overall quality of the code (VAVROVÁ; ZAYTSEV, 2017; BRONSHTEYN, 2013). In this context, non-compliance to these practices can be detected by using linting tools (CHEN et al., 2016; CHEN et al., 2018; DASGUPTA; HOOSHANGI, 2017), thus easing the further improvement of the source code quality. For example, developers may redefine built-in Python functions, such as `dict` and `list`—even though this practice is considered error-prone.[1] Although the Python interpreter does not raise a potential problem for this situation, linting tools are able to identify and raise a warning.

Previous works (OORT et al., 2021; ZHANG; CRUZ; DEURSEN, 2022; RADU; NADI, 2019; LIAWATIMENA et al., 2018; RAHMAN; RAHMAN; WILLIAMS, 2019; VAVROVÁ; ZAYTSEV, 2017) conducted studies to better understand the frequency of a class of problems that can be found by Python linters: warnings, here named as *lint-based warnings*. Examples of these warnings include Consider Using With, Redefined Built-in, Consider Using Enumerate, and Dangerous Default Value. We detail these warnings in Chapter 2. However, these studies rely on small datasets or focus on a few domains, such as machine learning, web-systems projects, or projects written by a restricted group of developers. Also, these studies neither focus on suggestions to remove such warnings nor on understanding how developers perceive them.

1.1 MOTIVATING EXAMPLE

To better motivate our work, we show an example of lint-based warning. To better illustrate this warning and a potential threat it might bring to the table, we rely on an issue[2] from the famous Django-Rest web development framework. The issue exemplifies a case where the built-in function `list` is being redefined by a class within the framework and, according to the report, "*this can have some unintended side effects in the rest of the class definition.*"

An excerpt from the code of the issue is shown in Figure 1. The original version contains the built-in function `list` being redefined within the `MyViewSet` class (see line 2 on the Original panel). Due to how the Python scope logic works, the default-value assignment in line 4 (`callable=list`) assigns the redefined `MyViewSet.list` instead of the built-in

[1] <https://github.com/ash16/TYTS/issues/19>
[2] <https://github.com/encode/django-rest-framework/issues/5884>

function `list`. In line 9, the `convert_collection` method is called with the `list` parameter, which happens to be the built-in function `list`. This call yields the following list: `[1, 2, 3]`. On the other hand, line 11 relies on the default-value assignment (`callable=list`), which is the `MyViewSet.list` method. In this situation, the call yields the "*woopsie*" string. In addition to the redefinition itself, notice that developers might get confused due to different methods with the same name.

Original

```
 1| class MyViewSet():
 2|     def list(self):
 3|         return 'woopsie'
 4|     def convert_collection(self, collection, callable=list):
 5|         return(callable(collection))
 6| m = MyViewSet()
 7| m.convert_collection([1,2,3], tuple)
 8| (1, 2, 3)
 9| m.convert_collection((1,2,3), list)
10| [1, 2, 3]
11| m.convert_collection((1,2,3))
12| 'woopsie'
```

Refactored

```
 1| class MyViewSet():
 2|     def as_list(self):
 3|         return 'woopsie'
 4|     def convert_collection(self, collection, callable=list):
 5|         return(callable(collection))
 6| m = MyViewSet()
 7| m.convert_collection([1,2,3], tuple)
 8| (1, 2, 3)
 9| m.convert_collection((1,2,3), list)
10| [1, 2, 3]
11| m.convert_collection((1,2,3))
12| [1, 2, 3]
```

Figure 1 – Redefined Built-in warning.

To identify lint-based warnings automatically, we might rely on tools such as Pylint.[3] In this context, Pylint is one of the most prominent tools, which is built upon static analysis of the code with the goal of detecting several lint-based warnings (OORT et al., 2021). In particular, given the code presented at the top of Figure 1, Pylint would raise a "Redefined Built-in" warning.

To remove this warning, we show a refactored version at the bottom of Figure 1 where we just change the name of the method from `list` to `as_list` in line 2. Given this scenario, we argue that it is quite important to better understand lint-based warnings. In particular, cataloguing, identifying, and refactoring lint-based warnings in Python code represent important tasks to avoid issues like this one from the Django-Rest framework. In this work, we study six different lint-based warnings, including the one detailed in this section, *i.e.*, the Redefined Built-in warning (ALEXANDRU et al., 2018; OORT et al., 2021). In Chapter 2 we show possible refactoring solutions to remove these six warnings. Most of these solutions were gathered from the Pylint manual.

[3] <https://pylint.org>

1.2 OUR WORK

In this work, we aim to highlight studies on lint-based warnings by providing not only data about their frequency on Python projects, but also the perceptions of Python developers about such warnings. To do so, we report the results of a mixed-method study where we initially analyze the frequency of six lint-based warnings in 1,119 different open-source general-purpose Python projects. Then, we conduct a survey to check whether developers are aware of the six lint-based warnings we study in this work. This survey allows us to gather information on to what extent developers are able to identify these lint-based warnings in Python code. To remove the lint-based warnings, we also suggest the application of refactorings. Here, we rely on the idea of floss-refactoring (MURPHY-HILL; PARNIN; BLACK, 2011), where "*the programmer uses refactoring as a means to reach a specific end, such as adding a feature or fixing a bug.*" In our case, we are fixing a lint-based warning. Last but not least, based on these refactorings, we submit 55 suggestions in the format of pull requests to remove lint-based warnings from open-source projects from GitHub. These pull requests are then listed to the developers which maintain the projects, where they have the options to react by accepting, or rejecting them. Afterwards we evaluate the opinion of these developers regarding the lint-based warnings. Finally we propose a tool able to automatically remove lint-based warnings from Python source code.

The results of our studies show that 39% of the 1,119 projects have at least one lint-based warning where the Consider Using With and Redefined Built-in were the lint-based warnings with most cases among the projects. We also show that, in general, the majority of developers are able to identify Python code with lint-based warnings and prefer an equivalent version of the code without them. We also find that both perception and preference increase alongside the developer experience, where the more experienced the developer, the more they can identify these lint-based warning as a practice to be avoided. The study using pull requests achieved 71.8% of acceptance rate, showing that the majority of developers maintaining open-source projects agree on the importance of fixing lint-based warning on their projects. Among the accepted pull requests, some of the repositories are maintained by Microsoft, Apache, and Facebook. Most of the denied pull requests were not actually rejected due to the developers' disagreement, but for secondary factors such as internal policies on their projects.

1.3 SUMMARY OF CONTRIBUTIONS

In summary, this work provides the following research contributions:

- An empirical study on the frequency of six lint-based warnings in 1,119 open-source Python projects (Chapter 3);

- A survey with 50 participants to evaluate how developers perceive Python lint-based warnings (Chapter 4);
- An empirical study accessing the opinion of project-maintainer developers on how they perceive Python lint-based warnings (Chapter 5);
- An IDE compatible plugin able to automatically fix lint-based warnings on Python applications (Chapter 6).

1.4 WORK ORGANIZATION

We report the results of three empirical studies to better understand them. First, we study their frequency in 1,119 open-source Python projects (Chapter 3); second, we conduct a survey with 50 participants to check whether developers are aware of code versions without lint-based warnings (Chapter 4); and third, we submit 55 pull requests to open-source projects to check whether developers accept refactored code versions without lint-based warnings (Chapter 5). All data, scripts, and results of the three studies we execute in this work are available online.[4]

[4] <https://github.com/publicationdata/scam2022>

2 REFACTORING CATALOG

In this chapter we present a catalog defining and exemplifying the six lint-based warnings studied in this work. We also point possible problems these lint-based warnings may cause if left ignored on existing Python code. Besides the presentation of the warnings, we also suggest refactorings to remove each of them. Here, we use the floss-refactoring (MURPHY-HILL; PARNIN; BLACK, 2011) definition, where "*the programmer uses refactoring as a means to reach a specific end, such as adding a feature or fixing a bug.*" In our work, notice that our refactorings are intended to fix lint-based warnings.

2.1 CONSIDER USING WITH (CUW)

This warning happens when an I/O stream is opened in a manner (see the `Original` side in Figure 2) where the close operation must be explicitly present in the code. When a stream is opened and the responsibility of closing is left to an explicit method call, there are two main potential risks. The developer might forget to call the `close` method or the line with it might not be executed due to an unexpected control-flow change (*e.g.*, an exception).

Original

```
f = open('text.txt', 'r')
text = f.readlines()
print(text)
f.close()
```

Refactored

```
with open('text.txt', 'r') as f:
    text = f.readlines()
    print(text)
```

Figure 2 – Consider Using With example alongside with its refactoring.

In Figure 2 an example of the Consider Using With is refactored by removing the `close` statement. This refactoring leverages the keyword `with`, which acts by creating a code block where after its execution, the names linked to this block, the stream `f` in this case, automatically executes an "on close" operation. For I/O streams this operation is the closing of the stream itself. Therefore, this refactoring removes the developer's obligation of manually freeing resources.

2.2 REDEFINED BUILT-IN (RBI)

This lint-based warning occurs when the code redefines the value of a built-in Python function.[1] For example, at the top of Figure 3, the developer is redefining the original `dict` built-in Python function. This is potentially problematic because if the code tries to execute the original `dict` function with `dict(obj)`, an exception will be raised since `dict` is no longer a function, but a dictionary object instead.

[1] <https://docs.python.org/3/library/functions.html>

Original

```
dict = {1: 'a', 2: 'b', 3: 'c'}
print(dict.get(1))
```

Refactored

```
dictionary = {1: 'a', 2: 'b', 3: 'c'}
print(dictionary.get(1))
```

Figure 3 – Redefined Built-in example alongside with its refactoring.

To remove the lint-based warning, we simply change the name of the newly created `dict` variable to a name that is not used by any built-in function.

2.3 SINGLETON COMPARISON (SC)

This lint-based warning occurs when a variable is compared to a singleton value, *i.e.*, `None`, `True` or `False`, using the equality operator (==). For example, at the top of Figure 4, the `value` variable is compared to the singleton `None` using the == operator. It is possible for that comparison to evaluate as `True` even when `value` is not a `None` object.

Original

```
if value == None:
    print('Hello world')
```

Refactored

```
if value is None:
    print('Hello world')
```

Figure 4 – Singleton Comparison lint-based warning example alongside with its refactoring.

The refactoring for this scenario is shown in Figure 4 where the equality operator is replaced by the identity operator (`is`). The identity operator compares if both objects point to the same memory address without relying on any custom logic built into any of the objects. This kind of comparison is better suited for singleton objects since they are allowed to have a single copy in-memory. Therefore if two objects are instances of the same singleton class, it is expected for them to have the same memory address.

2.4 CONSIDER USING ENUMERATE (CUE)

This lint-based warning occurs when an iterable, like a list, is accessed on a loop using a direct indexer variable the same way it is commonly used in languages like C or Java. This format of code needlessly leaves the responsibility of accessing the exact data cell on the iterable to the developer via the `iterable[index]` entity.

Original

```
values = [2, 4, 8, 16]
for i in range(len(values)):
    print('index: ', i)
    print('value: ', values[i])
```

Refactored

```
values = [2, 4, 8, 16]
for i, value in enumerate(values):
    print('index: ', i)
    print('value: ', value)
```

Figure 5 – Consider Using Enumerate lint-based warning and its respective refactoring.

The refactoring presented for the Consider Using Enumerate lint-based warning shown in Figure 5 takes advantage of the `enumerate` function. This native function takes an iterable and unpacks its indexes alongside to their associated values. By doing so, the manual indexer entity `values[i]` is avoided on the code and it is kept written in a more idiomatic format.

2.5 CONSIDER USING IN (CUI)

This lint-based warning happens when a variable is checked against several different values using the equality operator (==). At the top of Figure 6, we present an example of this situation. Notice that this may produce a long chain of comparisons or a confusing comparison within a loop, therefore potentially hindering code readability.

Original

```
if value == 1 or value == '1' or value == 'one':
    print(value)
```

Refactored

```
if value in [1, '1', 'one']:
    print(value)
```

Figure 6 – Consider Using In example alongside with its refactoring.

To remove this warning, we create a temporary list with the values to be checked against the variable. To do so, we rely on the Python built-in operator `in`.

2.6 DANGEROUS DEFAULT VALUE (DDV)

This lint-based warning happens when a callable, *i.e.*, a function or method, is defined using a mutable object as the default value of any of its arguments, as shown at the left-side of Figure 7 on the `collection` parameter. Python deals with default values of callables by creating an object and sharing it with all executions of the callable. In the case where the object is a mutable, the callable will be able to change its value, therefore affecting future executions of the callable.

At the right-side of Figure 7, the Dangerous Default Value is fixed using the proposed refactoring example. The idea behind this refactoring is to bind the mutable object at run

time instead of during the definition of the callable. We therefore prevent it from being shared between executions of the callable.

Original

```
def append(value, collection = []):
    collection.append(value)
    return collection
```

Refactored

```
def append(value, collection = None):
    if collection is None:
        collection = []
    collection.append(value)
    return collection
```

Figure 7 – Dangerous Default Value lint-based warning and its respective refactoring.

3 FREQUENCY

This chapter describes the first empirical study we conduct in this work. Here, we intend to investigate the frequency of the lint-based warnings in Python projects.

3.1 PLANNING

In this study, we intend to answer the following research question. **RQ_1: To what extent the selected Python lint-based warnings are frequent in open-source projects?** To do so, we rely on 1,119 public Python projects. We collect statistics and information with respect to the frequency of the six lint-based warning we study here.

We select these six lint-based warnings because they are detectable by a public available tool, Pylint. In addition, they have been explored in previous researches (RADU; NADI, 2019; OORT et al., 2021; LIAWATIMENA et al., 2018).

3.2 SETTINGS

The data for this study was gathered using the official GitHub API[1] by listing all the available Python projects and activating the search filter to remove from the results projects that are marked as forks. With the list of repositories available, the next step was to iterate over it by cloning each project one at time. After cloning each project we executed the analysis using the Pylint 2.9.3 tool. To improve performance, we configured the tool to detect only the six selected warnings.

The results of the analysis of each project were joined into a single table where each row of the table represents the incidence of a single warning into the project. The table was built using the following columns:

- The Pylint id code of the warning.
- The name of the warning.
- The name of the project.
- The path to the file where the warning was found.
- The line number where the warning was found.
- The commit hash of the project during the analysis time.

[1] <https://docs.github.com/pt/rest>

With the collected data available, we extract statistical information of the warnings to answer **RQ**$_1$.

3.3 SUBJECTS

To perform our evaluation, we rely on Python projects with different sizes, architectures, and domains. To better illustrate the diversity of the projects, Figure 8 presents four different metrics from all 1,119 projects we analyze: age, stars, contributors, and lines of code.

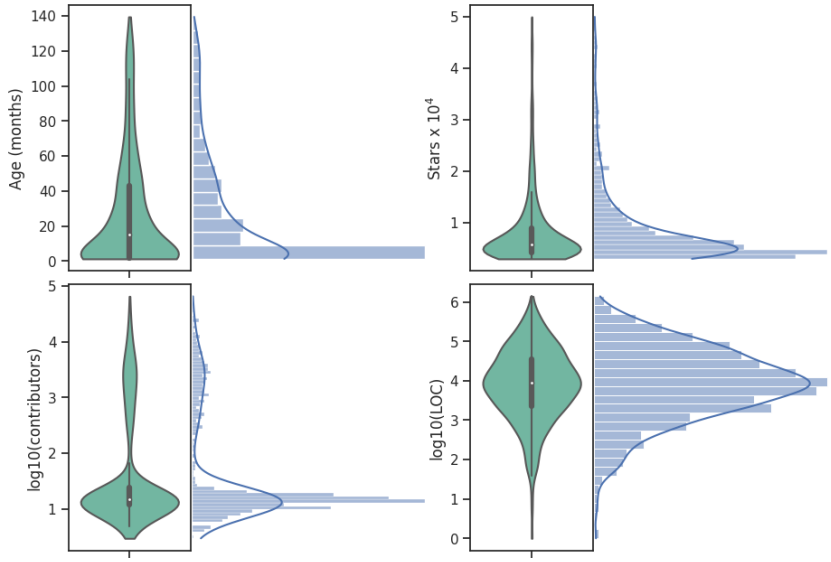

Figure 8 – Aggregate plots showing the metrics of the projects we analyze.

Examples of the projects we analyze in this work include the famous deep learning framework `huggingface/transformers`, the machine learning library `scikit-learn/scikit-learn`, `googleapis/google-api-python-client` and orthers like `explosion/spaCy`, `edgedb/edgedb`, `numpy/numpy` and `apache/airflow`.

Since we intend to focus on projects in a broad extent way (unlikely previous works (CHEN et al., 2018; OORT et al., 2021), we did not exclude projects based on its popularity or application.

3.4 RESULTS AND DISCUSSION

Our results show that 39% of the projects had one case of at least one of the six lint-based warnings. Exploring further this incidence, we show that Consider Using With was the most frequent lint-based warning: it appears 513 times in all projects, followed by the Redefined Built-in, which appeared 343 times in all projects. Figure 9 shows the frequency of all six warnings across all projects.

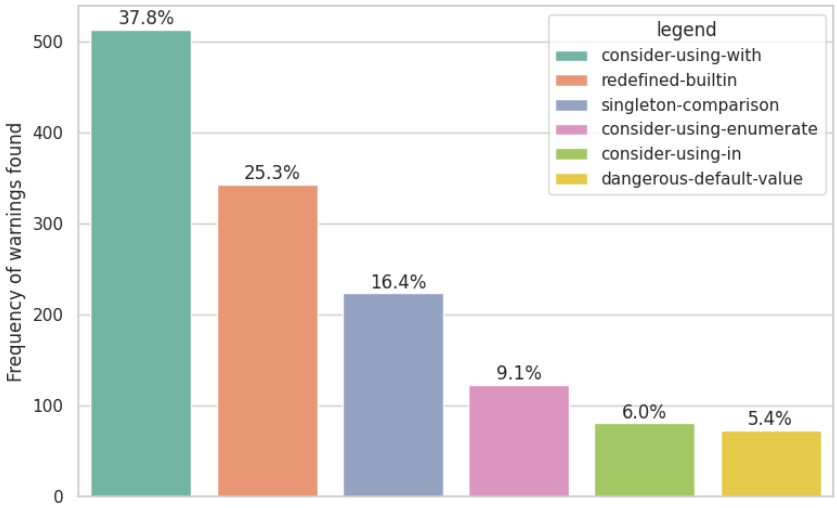

Figure 9 – Frequency of all six warnings across all projects.

In Figure 10, we describe in how many projects each individual lint-based warning is present. For example, we show that, among the projects with at least one lint-based warning, 36.2% of them contain the Consider Using With lint-based warning.

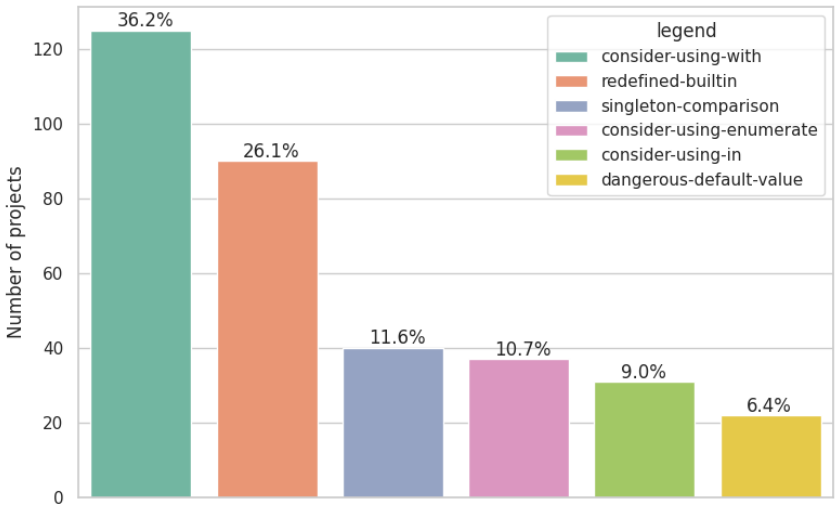

Figure 10 – Frequency of the projects with the given warning.

Similar results were found in (OORT et al., 2021) where all warnings, except the Consider Using With lint-based warning, were broadly found among their 74 projects composed only by machine-learning projects. Although (OORT et al., 2021) also searches for the warning Consider Using With, it was not found as a frequent case in their smaller, with 74 projects, dataset.

To better explore our results, we calculated some correlations using the Pearson correlation coefficient (BENESTY et al., 2009). In particular, we performed the following correlation tests:

- Number of stars of the project and number of lint-based warnings;
- Number of developers of the project and number of lint-based warnings;
- Number of LOC of the project and number of lint-based warnings.

Number of stars and LOC indicated little to no correlation to the number of lint-based warnings, with respectively $\rho = -0.02$ and $\rho = 0.03$ and the number of developers indicated a weak level of correlation, i.e., $\rho = 0.13$.

Answer to RQ_1: Our results show that 39% of the projects contain at least one of the six lint-based warnings we study in this work. Some warnings are more present than others. In particular, Consider Using With and Redefined Built-in are the most frequent ones among the projects.

3.5 THREATS TO VALIDITY

The use of Pylint represents a threat to internal validity due to the fact it is prone to false positive warning detection according to its documentation.[2] To minimize this threat we manually analyzed a random sample of 400 warnings detected using Pylint in order to check whether they were false positives, but all the cases pointed to true positives.

The set of projects used in this empirical study may represent a threat to external validity since it may not fully represent the universe of Python projects, mainly because we target open-source projects while Jupyter notebooks and closed-source projects were not included. To minimize this we aimed to analyze hundreds the projects available on GitHub, focusing on projects of different sizes, architectures, and domains. Also the projects have a great diversity of ages, number of stars, and number of contributors.

Another threat is that we focused only on six lint-based warnings. Whilst we cannot increase such a number, it is important to mention that the set of lint-based warnings we selected has been studied and explored by previous works.

[2] <https://pylint.pycqa.org/en/latest/intro.html>

4 SURVEY

In this chapter we describe the second empirical study we executed on this work.

4.1 PLANNING

The objective of this survey is to unveil the preference of developers between code without lint-based warning and code with lint-based warning. Here we intend to answer the following research question: **RQ_2: To what extent do Python developers prefer Python code written without lint-based warnings?**.

4.2 SETTINGS

Prior to publishing the survey, we elaborated six simple code examples (one per lint-based warning). Thus, for a given lint-based warning, we present two code versions, one with the lint-based warning and the other without it. To remove the lint-based warning, we rely on the corresponding refactoring presented in Chapter 2. Notice that having simple code examples was intentional. Here we intended to minimize the participants distraction on the domain of the code, on specific algorithms etc. Also, this allowed us to have a survey that could be answered quickly (WRIGHT; SCHWAGER, 2008; DEUTSKENS et al., 2004).

We designed the survey using the Survey Monkey tool.[1] We published the survey link on Reddit, via the subreddits r/Python[2] and r/pythontips.[3] The survey accepted answers during a period of 30 days.

The first question on the survey asked the country of origin of the participants. The second question asked how many years the participants have been programming using Python. The possible options were: *< 1 year; 1-3 years; 3.1-5 years; 5.1-10 years; > 10 years.*

4.3 PROCEDURE

The survey presented six others questions, each addressing a single lint-based warning. As mentioned, we have two code versions per lint-based warning. To avoid bias (i.e., participants choosing only one side), we randomly placed the code versions: sometimes the code without lint-based warnings appeared at the left-hand side and sometimes at the right-hand side. We labelled the left-hand side as (a) and the right-hand side as (b). We left

[1] surveymonkey.com
[2] www.reddit.com/r/Python
[3] www.reddit.com/r/pythontips

the participant to pick one of the possible answers: *Strongly prefer (a)*; *Prefer (a)*; *Indifferent*; *Strongly prefer (b)*; *Prefer (b)*; *I do not know*.

An example of how the questions where presented to the participants is shown in Figure 11.

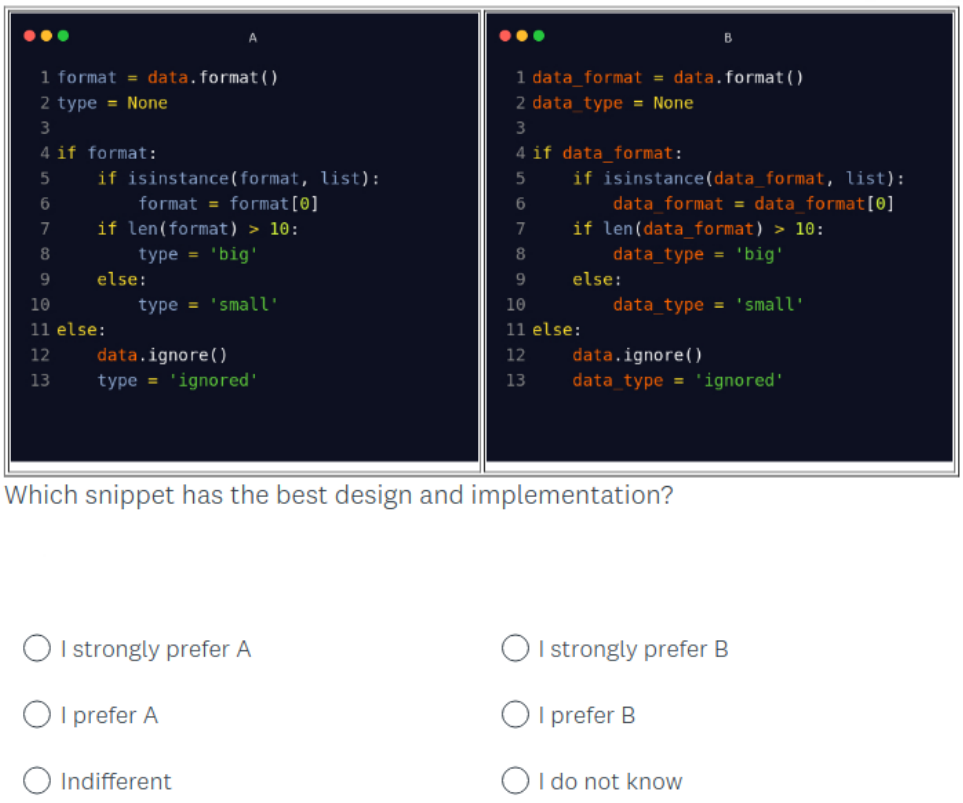

Figure 11 – Survey question format.

4.4 RESULTS AND DISCUSSION

As results, we received 50 answers coming from developers located in 18 different countries. The distribution of the experience time of the participants is shown in Figure 12.

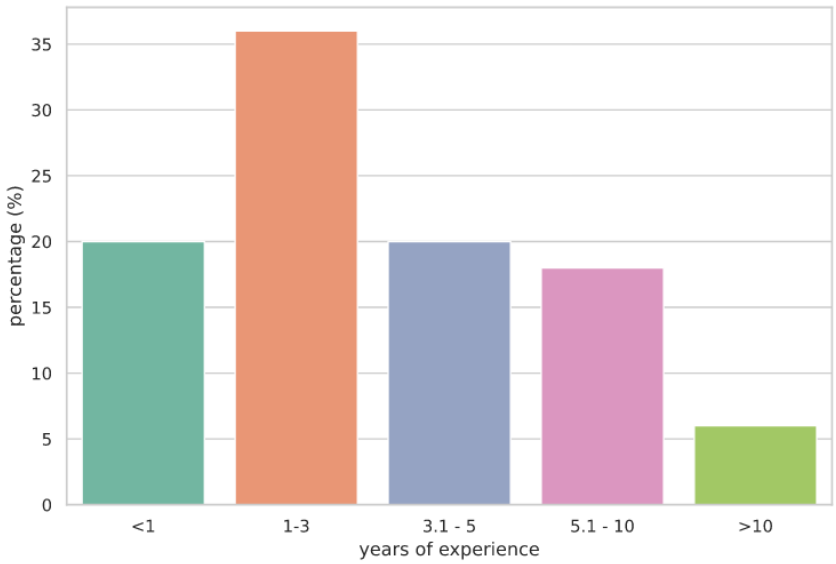

Figure 12 – Experience distribution among the participants.

We now present the results of the six lint-based warnings we study in this work.

4.4.1 Consider Using With

For this lint-based warning, 82.35% of the participants marked the refactored version of the example, where 17.64% marked the version with the lint-based warning. Unlike the refactored version, the version with the lint-based warning has a sequence of an `open` statement followed by an operation and then a `close` statement.

a

```
def update_file(path, new_content):
    with open(path, 'a') as f:
        f.write(new_content)
```

b

```
def update_file(path, new_content):
    f = open(path, 'a')
    f.write(new_content)
    f.close()
```

Figure 13 – Consider Using With alternatives.

Strongly prefer (a)	74.51%
Prefer (a)	07.84%
Indifferent	00.00%
Strongly prefer (b)	05.88%
Prefer (b)	11.76%
I Do not know	00.00%

4.4.2 Redefined Built-in

As results, 68.63% of the participants favored the option without the lint-based warning, against 13.72% whom preferred the version with it. A parcel of 11.76% of the participants stated that they are indifferent between both versions. A possible explanation for this result is that these participants might not be aware of the built-in `type` function.

a

```
format = data.format()
type = None

if format:
    if isinstance(format, list):
        format = format[0]
    if len(format) > 10:
        type = 'big'
    else:
        type = 'small'
else:
    data.ignore()
    type = 'ignored'
```

b

```
data_format = data.format()
data_type = None

if data_format:
    if isinstance(data_format, list):
        data_format = data_format[0]
    if len(data_format) > 10:
        data_type = 'big'
    else:
        data_type = 'small'
else:
    data.ignore()
    data_type = 'ignored'
```

Figure 14 – Redefined Built-in alternatives.

Strongly prefer (a)	05.88%
Prefer (a)	07.84%
Indifferent	11.76%
Strongly prefer (b)	50.98%
Prefer (b)	17.65%
I Do not know	05.88%

4.4.3 Singleton Comparison

For this question the majority of the participants favored the option without the lint-based warning, a total of 72.54% against 22.81% who marked the non-refactored version. The parcel of participants favouring the version with the warning may be a consequence from some peculiarities of the Python API: unlike other programming languages (RAMALHO, 2015), we should avoid comparing singletons, e.g., `None`, by using ==.

a

```
def set_rank(self):
    if self.rank is None:
        self.rank = Rank()
```

b

```
def set_rank(self):
    if self.rank == None:
        self.rank = Rank()
```

Figure 15 – Singleton Comparison alternatives.

Strongly prefer (a)	47.06%
Prefer (a)	25.48%
Indifferent	03.92%
Strongly prefer (b)	13.73%
Prefer (b)	09.08%
I Do not know	00.00%

4.4.4 Consider Using Enumerate

The results show that 76.47% of the participants preferred the version of the code with the refactoring applied to it. Other 23.52% parcel favored the version with the lint-based warning. Some programming languages (like C) does not have a native `enumerate` functionality. Thus, developers might tend to use explicitly the index based on previous experience (before Python).

a

```
def get_index(target, values):
    for i in range(len(values)):
        if values[i] == target:
            return i
    return -1
```

b

```
def get_index(target, values):
    for i, value in enumerate(values):
        if value == target:
            return i
    return -1
```

Figure 16 – Consider Using Enumerate alternatives.

Strongly prefer (a)	11.76%
Prefer (a)	11.76%
Indifferent	00.00%
Strongly prefer (b)	58.82%
Prefer (b)	17.65%
I Do not know	00.00%

4.4.5 Consider Using In

The vast majority of the participants selected the refactored version, counting a total of 84.31% against 9.8% who preferred the non-refactored version of the code.

a

```
if value in [1,2,4]:
    print('Ok')
```

b

```
if value == 1 or value == 2 or value == 4:
    print('Ok')
```

Figure 17 – Consider Using In alternatives.

Strongly prefer (a)	72.55%
Prefer (a)	11.76%
Indifferent	00.00%
Strongly prefer (b)	05.88%
Prefer (b)	03.92%
I Do not know	05.88%

4.4.6 Dangerous Default Value

The result for this question pointed that 49.02% declared a favorable opinion upon the refactored version of the code and 43.14% of them marked the version with the lint-based warning as more appealing. Unlikely the other questions, this one did not present a vast preference towards the refactored version. At a first glance, the version with the lint-based warning may look simpler and easier to read due to less lines of code and smaller cyclomatic complexity. Nonetheless it presents the risks described in Section 2.6.

a

```
def set_actions(self, actions = []):
    self.actions = actions
```

b

```
def set_actions(self, actions = None):
    if actions:
        self.actions = actions
    else:
        self.actions = []
```

Figure 18 – Dangerous Default Value alternatives.

Strongly prefer (a)	27.45%
Prefer (a)	15.69%
Indifferent	03.92%
Strongly prefer (b)	39.22%
Prefer (b)	09.80%
I Do not know	03.92%

4.5 SUMMARY

In general, the survey results show that most developers tend to choose the code versions without the lint-based warnings, i.e., the refactored version. The only exception is the Dangerous Default Value (DDV) lint-based warning, where we could not identify a general consensus.

To better understand our results, we also crossed the survey results and the participants experience. Figure 19 presents this result. Darker colors represent that many developers selected the option. For example, many participants with 1-3 years of experience preferred the refactored version of the lint-based warnings Consider Using In (CUI) and Consider Using With (CUW). As results, notice that developers with less experience time using the Python language are more prone to prefer the versions with lint-based warnings than experienced developers.

Answer to RQ_2: The majority of the participants prefer Python code without the given lint-based warnings and this preference increases alongside the experience time of the developer.

4.6 THREATS TO VALIDITY

The amount of participants on the survey may not represent all the groups of Python developers (e.g., machine learning developers, web-based systems developers, mathematicians developers etc). This represents a threat to external validity. To minimize

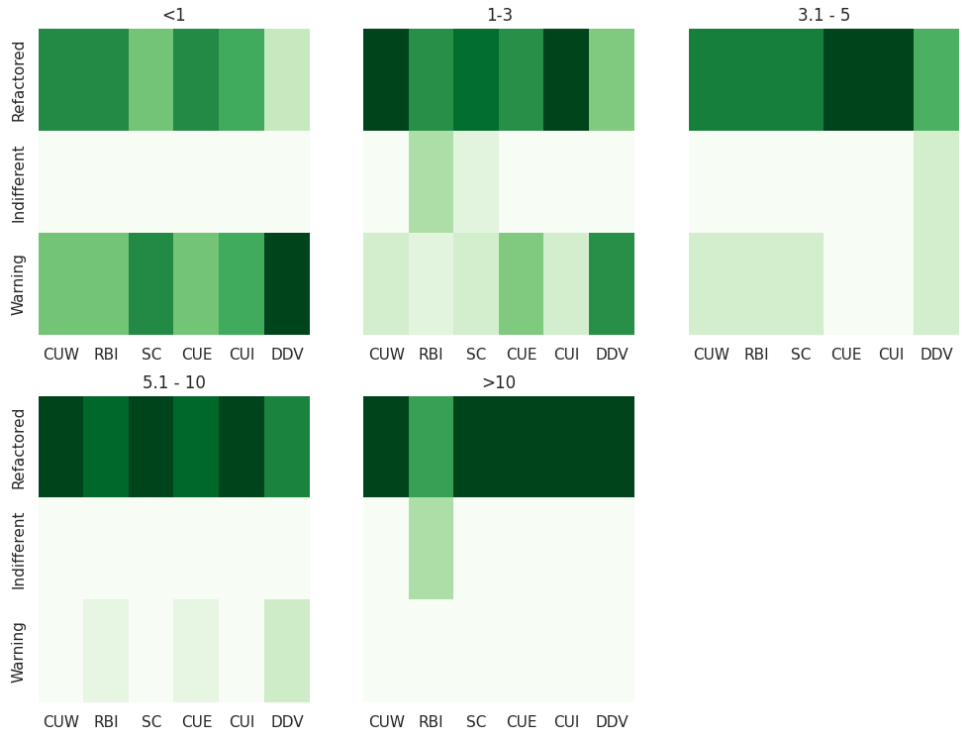

Figure 19 – Survey answers categorized by the experience time of the participants.

this threat we published the survey in a globally public platform and asked for participants without imposing any restriction regarding on their relation with Python.

5 PULL REQUESTS

In this chapter we describe the third empirical study we execute in this work.

5.1 PLANNING

In this study, we intend to investigate whether developers accept contributions to remove lint-based warnings in Python code. To do so, we identify lint-based warnings in public projects, refactor them by using the source code transformations we present in Chapter 2, and submit pull requests. Afterwards, we evaluate how the developers of such projects perceive the lint-based warnings we focus on this work. With this procedure we aim to answer the following research question: **RQ_3: To what extent do open-source developers accept submissions to refactor lint-based warnings in Python Code?**

5.2 SETTINGS

The initial step is to build a subset of the lint-based warnings collected in the first experiment (Chapter 3). To select the lint-based warnings, we rely on the following procedures:

1. Projects without any contribution on the last twelve months are excluded from the targets, otherwise our pull requests would be at the imminent risk of being ignored;
2. We should only submit one pull request to a project, in order to prevent the results to be influenced by a specific project developer. This is important to minimize bias, such as having several decisions (accepting or rejecting the pull requests) from the same developer;
3. For each of the six lint-based warnings, we decided to select 10 code examples to submit the pull requests, which would sum up to 60 pull requests. This decision was taken just to make our analysis and human effort feasible. However, after applying the two previous procedures, we could not find 10 code examples for some of the lint-based warnings. This way, we end up with 55 pull requests, instead of 60.

After selecting the lint-based warnings, the next step is to manually fix the code using the refactorings we suggest in Chapter 2. In order to decrease the latency (YU et al., 2015) of the pull request review, our pull requests contain a small number of changes, with a maximum of three lines of code for each of them.

5.3 SUBJECTS

To perform the pull requests submission, we rely on 55 projects. The projects include `facebookresearch/ParlAI` (Miller et al., 2017), `microsoft/cascadia-code`, and `apache/tvm`. Table 1 presents all the projects we used in this study. We also show the number of pull requests of the six lint-based warnings. In particular, we submitted pull requests to refactor 10 Singleton Comparison SC, 10 Consider Using Enumerate (CUE), 10 Consider Using In (CUI), 6 Consider Using With (CUW), 10 Dangerous Default Value (DDV), and 9 Redefined Built-in RBI.

Table 1 – Submitted Pull Requests

Warning	Project Name	Result	Comments
CUW	microsoft/cascadia-code	Accepted	Thanks!
CUW	bokeh/bokeh	Accepted	Thanks!
CUW	seatgeek/fuzzywuzzy	Open	
CUW	gelstudios/gitfiti	Open	
CUW	apache/tvm	Accepted	Thanks!
CUW	sympy/sympy	Accepted	Thanks!
RBI	lauris/awesome-scala	Accepted	
RBI	brightmart/text_classification	Open	
RBI	Linzaer/Ultra-Light-Fast-Generic-Face-Detector-1MB	Open	
RBI	ipython/ipython	Accepted	Thanks!
RBI	python-poetry/poetry	Denied. Internal policy.	The project does not take pull requests.
RBI	ultralytics/yolov5	Accepted	Merged!
RBI	lukemelas/EfficientNet-PyTorch	Open	
RBI	scipy/scipy	Accepted	Thanks!
RBI	wistbean/learn_python3_spider	Accepted	
SC	NVIDIA/FastPhotoStyle	Open	
SC	PaddlePaddle/PaddleOCR	Open	
SC	corpnewt/gibMacOS	Open	
SC	nuno-faria/tiler	Accepted	Thanks!
SC	trustedsec/unicorn	Open	
SC	wb14123/seq2seq-couplet	Open	
SC	anishathalye/neural-style	Accepted	Thanks!
SC	jadore801120/attention-is-all-you-need-pytorch	Open	
SC	a1studmuffin/SpaceshipGenerator	Open	
CUE	thunil/TecoGAN	Open	
CUE	Jack-Cherish/python-spider	Accepted	
CUE	lengstrom/fast-style-transfer	Open	
CUE	facebookresearch/detr	Accepted	Thanks!
CUE	luyishisi/Anti-Anti-Spider	Open	
CUE	wbt5/real-url	Rejected	
CUE	kennethreitz/records	Open	
CUE	stewartmcgown/uds	Open	
CUE	StevenBlack/hosts	Accepted	Thanks!
CUE	xuebinqin/U-2-Net	Open	
CUI	PySimpleGUI/PySimpleGUI	Denied. Internal policy.	The project does not take pull requests.
CUI	facebookresearch/ParlAI	Accepted	Thanks!
CUI	aristocratos/bpytop	Accepted	Thanks!
CUI	persepolisdm/persepolis	Open	
CUI	Morizeyao/GPT2-Chinese	Open	
CUI	EleutherAI/gpt-neo	Accepted	Thanks!
CUI	kovidgoyal/kitty	Accepted	
CUI	coleifer/peewee	Rejected	I'll pass.
CUI	chineseocr/chineseocr	Open	
CUI	skywind3000/ECDICT	Accepted	Thanks!
DDV	geekcomputers/Python	Accepted	
DDV	cython/cython	Accepted	Thanks!
DDV	n1nj4sec/pupy	Open	
DDV	smicallef/spiderfoot	Accepted	Thanks! The project has other warning cases like this to be fixed.
DDV	tzutalin/labelImg	Accepted	
DDV	espressif/esptool	Denied. Intended warning.	This warning behavior is intended on the code.
DDV	bottlepy/bottle	Rejected	The change does not fix anything.
DDV	elceef/dnstwist	Rejected	The change does not fix anything.
DDV	aaPanel/BaoTa	Open	
DDV	s3tools/s3cmd	Denied. Internal policy.	The change does not fix anything.

5.4 EXAMPLES OF SUBMITTED PULL REQUESTS

In this section, we present some examples of pull requests we submitted to the repositories presented in Table 1. Here, we show one example of each lint-based warning we focus on this work.

Figure 20 presents the change we submitted via pull request to the `cascadia-code` repository, maintained by Microsoft. It fixes a Consider Using With case. In the original version of the code there is an `open` statement which is never closed during the life cycle of the application. We fix this scenario by applying the refactoring proposed in Chapter 2, where we enclose the `open` operation within a `with` block.

Original

```
config = yaml.load(open(INPUT_DIR/"stat.yaml"), Loader=yaml.SafeLoader)
```

Refactored

```
with open(INPUT_DIR/"stat.yaml") as f:
    config = yaml.load(f, Loader=yaml.SafeLoader)
```

Figure 20 – Consider Using With refactoring on the microsoft/cascadia-code repository.

Figure 21 shows a case of the Singleton Comparison. This lint-based warning was found on the `FastPhotoStyle` repository, maintained by NVIDIA.

Original

```
if cont_seg.size == False or styl_seg.size == False:
    ...
```

Refactored

```
if cont_seg.size is False or styl_seg.size is False:
    ...
```

Figure 21 – Singleton Comparison pull request example.

Figure 22 shows how the refactoring was applied to the `scipy/scipy` repository. For that we changed the `dir` name from the loop, which is the name a a built-in function, to `directory`.

Original

```
for dir, subdirs, files in os.walk('IPython'):
    package = dir.replace(os.path.sep, '.')
    ...
```

Refactored

```
for directory, subdirs, files in os.walk('IPython'):
    package = directory.replace(os.path.sep, '.')
    ...
```

Figure 22 – Redefined Built-in refactoring on the scipy/scipy repository.

Figure 23 illustrates the submitted pull request that we submitted to the `detr` repository, which is maintained by Facebook. In this pull request we fix a Consider Using Enumerate scenario by removing the `range(0, len(outputs))` iterator and replacing it by `enumerate(outputs)`, as well changing the direct indexer `outputs[i]` from the body of the loop.

Original

```
for i in range(0, len(outputs)):
    try:
        torch.testing.assert_allclose(outputs[i], ort_outs[i], ...)
```

Refactored

```
for i, element in enumerate(outputs):
    try
        torch.testing.assert_allclose(element, ort_outs[i], ...)
```

Figure 23 – Consider Using Enumerate refactoring on the `facebook/detr` repository.

In Figure 24 we show how a pull request was submitted to the `gpt-neo` repository, which is maintained by EleutherAI. The pull request addresses a scenario of the Consider Using In lint-based warning.

Original

```
assert (mode == tf.estimator.ModeKeys.TRAIN or mode == tf.estimator.ModeKeys.EVAL)
```

Refactored

```
assert mode in [tf.estimator.ModeKeys.TRAIN, tf.estimator.ModeKeys.EVAL]
```

Figure 24 – Consider Using In refactoring on the EleutherAI/gpt-neo repository.

Figure 25 shows the content of the pull request we submitted to the `LabelImg` repository, maintained by Tzutalin.This pull request address on fixing a case of the Dangerous Default Value lint-based warning. To remove the warning we changed the default empty list value from the `argv` argument from the function `get_main_app` following the refactoring show in Chapter 2.

Original

```
def get_main_app(argv=[]):
    ...
```

Refactored

```
def get_main_app(argv=None):
    if not argv:
        argv = []
    ...
```

Figure 25 – Dangerous Default Value refactoring on the labelImg repository.

5.5 RESULTS AND DISCUSSION

As results, among the 55 submitted pull requests, 32 were analyzed by the developers, giving us a response rate of 58.1%. Figure 26 illustrates that a total of 71.8% of the analyzed

submissions (i.e., 32) were accepted by the maintainers and 12.5% were rejected. One specific developer accepted the pull request and mentioned that the Dangerous Default Value lint-based warning seems to be spread throughout the code of the project:

> "*This change is reasonable. There are many of these which should be addressed.*"

Two developers rejected our pull requests to remove the Dangerous Default Value lint-based warning:

> "*This issue does not apply, as none of the functions modify or change the data referenced by its arguments.*"
> "*The globals variable is not mutated and does not leave the local scope of the function. This is safe to do.*"

Notice that the aforementioned two reasons seem to be similar: Although this code structure opens the possibility for future bugs, the pull request was denied because there are no bugs currently.

Another 12.5% were also rejected, but the rejection was caused not from the lack of interest by the developers, but from internal project policies, meaning that at least part of the developers in this group could agree on the relevance of the refactorings. For example, one of the developers stated the following:

> "*The project does not take pull requests.*"

In this last category we have scenarios of projects that do not accept pull requests from external members of the project core-team. There was also a case of the Dangerous Default Value where the pull request was denied because the lint-based warning was intentionally written on the code in order to be used as a low-cost memory buffer.

Considering that only 12.5% of the maintainers denied the refactorings based on their opinion about it, we can conclude that the percentage of maintainers prone to accept these refactorings ought fall between 71.8% and 87.5%.

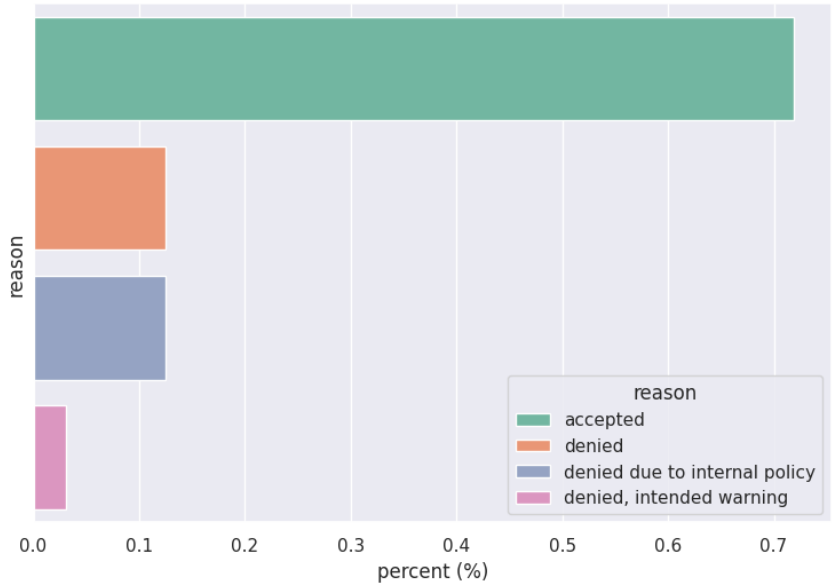

Figure 26 – Distribution of the acceptance of the pull requests.

Answer to RQ_3: Our results show that, from the set pull requests analyzed, at least 71.8% up to a maximum of 87.5% of the Python projects developers care to work on applying refactorings in order to remove lint-based warning from the code.

5.6 THREATS TO VALIDITY

Previous research works (LENARDUZZI et al., 2021; SOARES et al., 2015) describe that the acceptance of pull requests on open-source projects are strongly influenced by the reputation of the developer within the project. Also, pull requests aiming only at fixing design defects are usually less accepted. These situations might bring threats to the internal validity of our study. To minimize this, we followed other described metrics which maximize the acceptance of pull requests, like keeping the changes small and changing only a single file for each pull request.

The selected subset of targeted projects threats to external validity. Therefore our results may not generalize to other projects. We try to minimize this by selecting projects of different sizes, architectures, and domains. Also, they are maintained by companies (e.g., Microsoft, Apache, and Facebook) and by open-source software contributors. Thus, we could improve the diversity of the projects we analyzed.

6 TOOL SUPPORT

In this chapter we describe the implementation and usage of Yapfa (*Yet Another Python Formatter Again*).[1] A tool we developed by refining the Google's Yapf [2] (*Yet Another Python Formater*) implementation in a manner to extend its formatting capabilities allowing it not to only detect lint-based warnings, but also apply refactorings to them. It is executable as a stand alone application as well a Visual Studio plugin able to fix Python code during the development phase using only a simple IDE command. The tool is extendable, meaning that developers can register new lint-based warnings and craft their custom refactorings.

6.1 DETECTION APPROACH

Each lint-based warning leaves a well defined pattern on specific nodes of the abstract syntax tree (AST) of the application. For instance, a Singleton Comparison warning will only be present on top of a `Compare` node [3] and a Dangerous Default Value warning can be found possibly only on a `FunctionDef` node.[4] For this matter we can say that the `Compare` node is the origin of the Singleton Comparison as well the `FunctionDef` node is the origin of the Dangerous Default Value warning. In Python, all node types inherit from the generic `AST`[5] node type, therefore every lint-based warning is naturally originated from it. Nonetheless a node type may originate more than one type of lint-based warning.

Taking the Singleton Comparison warning as an example, we have that when it comes on the format of `foo == None`, it creates a `Compare` node on the AST with the configuration shown in Figure 27.

```
Compare(
    left=Name(id='foo', ctx=Load()),
    ops=[Eq()],
    comparators=[Constant(value=None)])
```

Figure 27 – Compare node with Singleton on right-side

Most of the lint-based warnings have variations on its patterns. In our Singleton Comparison example, if the operands were inverted to `None == foo` the node with the singleton constant would appear on the `left` subnode instead of in `comparators`. However the origin node structure, *i.e.* `Compare`, is kept the same.

[1] <https://github.com/NaelsonDouglas/yapf>
[2] <https://github.com/google/yapf>
[3] <https://docs.python.org/3/library/ast.html#ast.Compare>
[4] <https://docs.python.org/3/library/ast.html#ast.FunctionDef>
[5] <https://docs.python.org/3/library/ast.html#ast.AST>

We then leveraged on this *a priori* info of which type of node originates which lint-based warning to build an AST parsing approach using the Visitor Design Pattern (GAMMA et al., 1993). Our algorithm visits the nodes on the AST and if the visited node is the possible origin of one or more lint-based warnings, then we execute a matcher to confirm if the possible lint-based warnings exists on the visited node.

The matcher for each lint-based warning is based on the Structural Pattern Matching[6] feature introduced in Python 3.10 (released in October 2021). Where for each lint-based warning we implement a detection matcher taking as input the origin node for that lint-based warning and using the Structural Pattern Matching approach check whether the node fits to the lint-based warning pattern.

For the Singleton Comparison example, we are aware it is detectable when a `Compare` node is set with the following configurations: The `ops` field is set to an equality operator.[7] And there is a singleton constant set to either the `left` field or the `comparators` field, or even both. This matcher can be implemented using the Structural Pattern Matching defined in Figure 28.

```
def sc_detect(node:ast.Compare) -> bool:
    match node:
        case (Compare(ops=[Eq()],comparators=[Constant(value=True)])|
            Compare(ops=[Eq()],comparators=[Constant(value=False)]) |
            Compare(ops=[Eq()],comparators=[Constant(value=None)])  |
            Compare(ops=[Eq()], left=Constant(value=True))  |
            Compare(ops=[Eq()], left=Constant(value=False)) |
            Compare(ops=[Eq()], left=Constant(value=None))
            ):
                return True
        case _:
                return False
```

Figure 28 – Singleton Comparison Structural Pattern Matcher detector

6.2 AUTOMATIC REFACTORING

When the matcher detects a specific lint-based warning on a node, it then pass this node to the function responsible to apply the specific refactoring for that lint-based warning. This function takes as input the node with the *a priori* known structure, transforms the node applying the refactoring described in our catalog (Chapter 2), then returns the refactored node allowing it to replace its original version with the lint-based warning on the AST. The refactoring for the Singleton Comparison example stands on simply replacing the == operator by `is`. This operation is implemented as shown in Figure 29, which is a simple direct change

[6] <https://peps.python.org/pep-0636/>
[7] <https://docs.python.org/3/library/ast.html#ast.Eq>

on the node.

```
def sc_fix(node:ast.Compare) -> ast.Compare:
    node.ops = [ast.Is()]
    return node
```

Figure 29 – Singleton Comparison Node fix function

Therefore, depending on the lint-based warning complexity, more elaborate `fix` methods ought be implemented. The implementations for the other lint-based warning cases are available on the projects repository on GitHub.[8]

The whole approach process is graphically represented in Figures 30 and 31. Where they show the representation of a generic AST with nodes containing lint-based warnings represented by the color red. Our tool parses the AST and transforms nodes containing lint-based warnings by detecting them using the structural pattern matching on top of the Visitor design pattern.

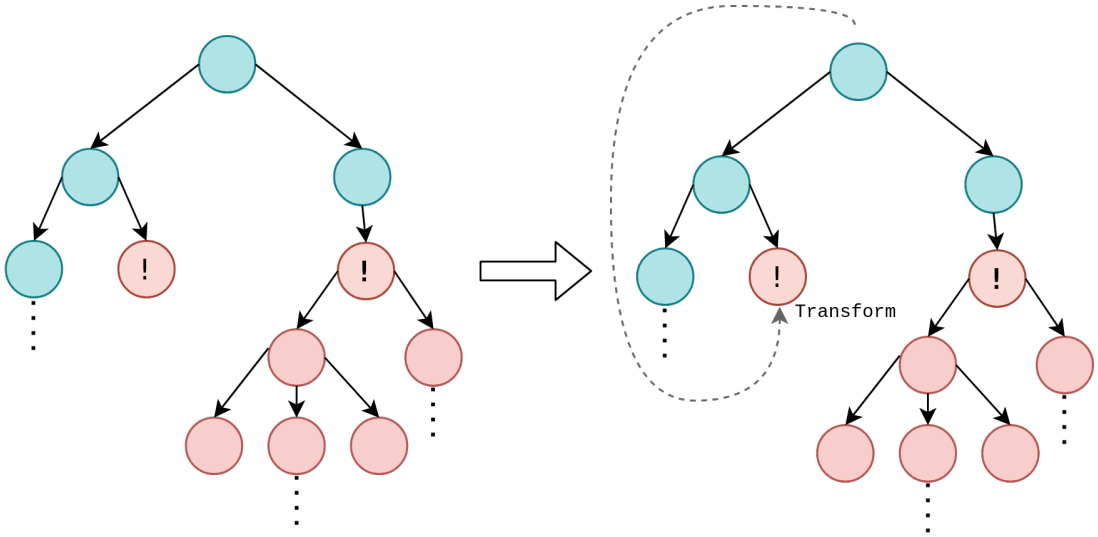

Figure 30 – Code AST. Red nodes represent nodes with lint-based warning.

The transformation may not only modify nodes as shown in Figure 30, but also change the entire sub-tree of a node with a warning, as shown in Figure 31.

[8] <https://github.com/NaelsonDouglas/LintWarningRemover>

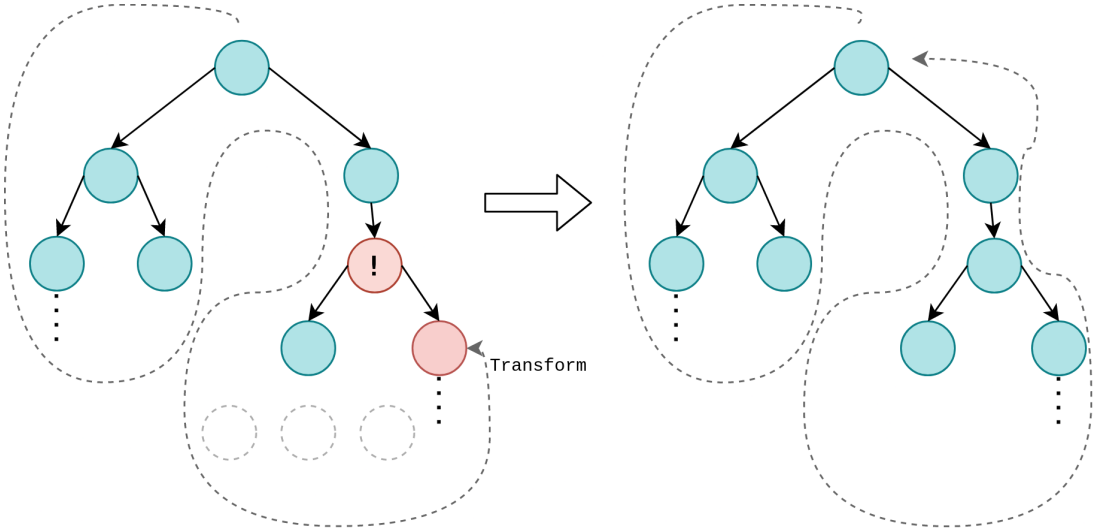

Figure 31 – AST transformation.

6.3 OBSERVATIONS

We intentionally did not implement a `fix` method for the Redefined Built-in lint-based warning. Due to its nature, automatically refactoring this lint-based warning will lead to ambiguity when selecting which version of the redefined name to keep on the references to it across the code.
This problem is better described on the example shown in Figure 32.

```
1| def print(data):
2|     if printer.has_paper():
3|         printer_head.print(data)
4| print('foo')
```

Figure 32 – Which `print` function version should be kept at line four?

In this example the `print` function is redefined and afterwards it is executed as `print('foo')`. A simple fix would be to change the name of the newly created `print` function preventing it from shadowing the built-in `print`. But it is not possible for the tool to decide if the intention of the developer at the last line was to call the built-in `print` or the redefined version of it. Therefore it is not viable to replace the name and keep the consistency for all the references to that name across the code. For such scenario the usage of a warning alerting the developer to manually take care of the situation is better suitable than an automatic refactoring.

The Consider Using With lint-based warning check is actually performed by visiting the generic `AST` node type instead of a more precise node. It happens because the code defining this warning may spread across several sibling nodes in an AST. The example in

Figure 33 shows a scenario where a Consider Using With lint-based warning is resumed into three lines of code: the opening of the resource; a generic and unlimited amount of lines of code and the closing of the resource.

```
f = open('text.txt', 'r')
# Any amount of lines of code here
f.close()
```

Figure 33 – Consider Using With sibling spread example.

Both opening and closing nodes are actually siblings on the AST, as well the entire code between them. And such snippet of code could be present in diverse contexts, be it a function definition (`FuncDef`), class definition (`ClassDef`[9]) or others. Therefore a more efficient method of detecting this lint-based warning is by checking it for every node and checking if the configuration `name = open(resource)`, followed by a random amount of nodes then a `name.close()` is present on the children of the node.

6.4 LINT-BASED WARNINGS

In this section we individually resume the approach used for both the detection and the automatic refactoring of each lint-based warning.

6.4.1 Consider Using With

Detectable in generic `AST` nodes. First we detect if among its children the pattern of a resource stream being opened and linked to a target name, followed by any amount of nodes, then we check if there is a node closing the same target name linked to the opening stream. In order to apply the refactoring to it, first we used the target name described on the `open` node and the node created by the open function call in order to build an empty `with` block: `f = open(resource)` —> `with open(resource) as f:`
Then we copy all nodes between the `open` and the `close` nodes and linked them as a child nodes of the recently created `with` node; We then deleted the `close` node; Then removed the `open` node from the AST and replaced it with the recently created `with` node.

6.4.2 Redefined Built-in

Detectable in `Assign` nodes by checking if the target field contains any `Name` node where the `id` field of if is equals to any built-in name. We did not implement the fix for this lint-based warning due to the implications described in Section 6.3.

[9] <https://docs.python.org/3/library/ast.html#ast.ClassDef>

6.4.3 Singleton Comparison

Detectable in `Compare` nodes by checking if at least one of the `left` or `comparators` fields is assigned to a `Constant` node with its `value` field assigned to a singleton value. To apply the refactoring to it, we just assign the value `[ast.Is()]` to the `ops` field on the `Compare` node.

6.4.4 Consider Using Enumerate

Detectable in `For` nodes by checking if the `iter` field is assigned to a `Call` node where this `Call` has its `func` field pointing to the `range` method and the `args` of it is pointing to second `Call` node linked to the`len` method. To apply the refactoring to it, we first we retrieve from the original loop the iterable variable from the field `target_id` and the original iterable; Them we create a new `For` loop node with the `values` field set as the `(target_id, 'value')` and the iterable is set as `enumerate({target_id})`; Afterwards we replace any instance of `{iterable}[{target_id}]` from the body of the loop with `value`.

6.4.5 Consider Using In

Detectable in `BoolOp` nodes. Select the subset of its children which are of the node type `Compare` and are using the `Eq` operator node. For this subset and group them by their left id name. Check if at least one group has more than one element. The refactoring is provided by retrieving all the nodes belonging to the groups found on the detection phase then them from the original `BoolOp` node;

Each group is a list of comparisons like `[foo==value_1, ..., foo==value_n]`, where within the groups the value of foo is constant. For each group, build a list with `[value_1, ..., value_n]` and create a node with `foo in [value_1, ..., value_n]`. Then add this node to the original `BoolOp` separated by an `Or` operator node.

6.4.6 Dangerous Default Value

This lint-based warning is detectable on `FunctionDef` nodes by checking if there is at least one node from `node.args.defaults` field which is an instance of a mutable type. The refactoring is applied as follows. For each detected value on the format `arg=value`, where `value` is mutable, first replace `arg=value` with `arg=None`. Then add to the beginning of the body of the `FunctionDef` the following check:

```
if not arg:
    arg=value
```

6.5 VISUAL STUDIO CODE PLUGIN

Apart from being executable in a stand-alone script, our tool is also compatible with Visual Studio Code as a formatting plugin. It seamlessly integrates with the IDE by with little to no configuring necessary. In order to use it on the IDE all it takes is to

- Create a new Python virtual environment.
- Install the tool on the virtual environment with:
  `pip install git+https://github.com/NaelsonDouglas/yapf`
- Add the tool as the default formatter on the Visual Studio Code changing `settings.json` adding the registry `"python.formatting.provider": "yapf"`

Then, to use it all what is necessary is to open a Python source code, call the VSCode command palette with `Ctrl+Shift+P`, type `Format Document` and press enter. Or just press the default shortcut `Ctrl+Shift+I`.

In Figure 34 we show a Visual Studio Code screenshot with a simple code and it contains the lint-based warnings Dangerous Default Value, Consider Using Enumerate and Consider Using With respectively on lines 3, 5 and 9.

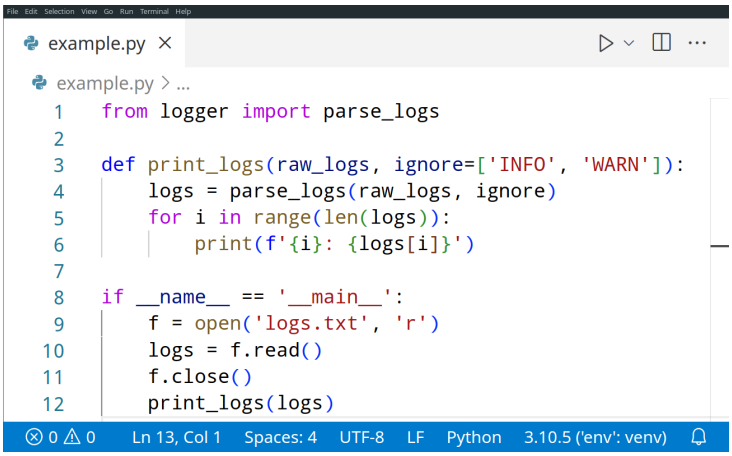

Figure 34 – lint-based warnings on Visual Studio Code.

After pressing the shortcut `Ctrl+Shift+I` in Visual Studio Code, our plugin transforms the code to the refactored version shown in Figure 35.

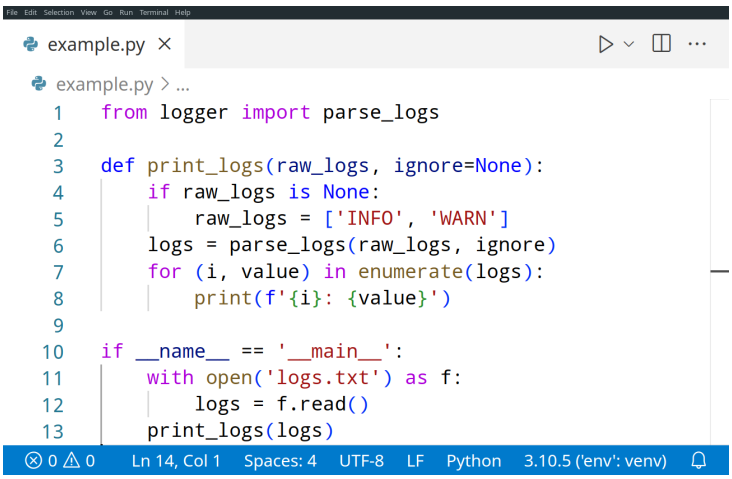

Figure 35 – Automatically refactored version.

7 RELATED WORK

The usage of linters as a mean to evaluate coding practices and code quality was studied and validated by several researchers (FLANAGAN et al., 2002; WAGNER et al., 2005; NAGAPPAN; BALL, 2005; ZHENG et al., 2006; NANDA et al., 2010; BUTLER et al., 2010) and it has been proved as a powerful tool when aiming to describe coding in different domains. Moreover, existing research has explored the use of linters in specific domains like game or web development (BORRELLI et al., 2020; LIAWATIMENA et al., 2018), dependency management (JAFARI et al., 2021), security (GADIENT et al., 2019), and others (PADUA; SHANG, 2017; LI et al., 2017; YU; SU; MA, 2016; ZHAO et al., 2020; HABCHI et al., 2017; VEGI; VALENTE, 2022; NAGY; CLEVE, 2018).

An evaluation of five different JavaScript linters was done by (CHATZIDIMITRIOU et al., 2018) on top of 2000 npm[1] packages and from the results of their study they implement a platform able to constantly monitor the npm registry, thus easing the quality control of JavaScript packages. Similar to it, but on the Python context, Spyse is a proposed solution for a dynamic flexible Python package explorer for the Python Package Indexer[2] which uses lint-based warnings metadata generated by linters in order to offer such flexible search mechanism.

Preliminary studies like (KUMAR; NORI, 2013) and (JOHNSON et al., 2013) revealed that although well accepted, linting-tools were not applied as much as it should be into the software development cycle. More recently, (BELLER et al., 2016) analyze a mixed set of Java, JavaScript, Python and Ruby projects and evaluates how linting tools are used in real applications according to each language. And one of its results shows that the use of linters is not ubiquitous across the languages, but it is specially less common in Python. They also show that projects which apply the usage of linters usually do not deviate the linting tool from its default configurations. Therefore an even more recent study, (VASSALLO et al., 2020), also compares the usage of linters in projects from the same four languages used in (BELLER et al., 2016), and indicates that the usage of linters in Python has been increasing. One of the reasons causing the underuse of linters is the high amount of lint-based warnings they create (BELLER et al., 2016; JOHNSON et al., 2013; MENDONCA et al., 2013) and the inevitable existence of false positive lint-based warnings (RIVAL, 2005; DILLIG; DILLIG; AIKEN, 2012). An approach identifying and cataloguing the incidence of false positive lint-based warnings detection is described by (REYNOLDS et al., 2017). A dataset created using different projects code written by 200 programming students was submitted to lint-based warnings detection in (SRIPADA; REDDY; SUREKA, 2015) as well the application of manual refactorings. On top of that they

[1] <https://www.npmjs.com/>
[2] <https://pypi.org/>

conducted a survey among the students inquiring on their receptiveness about this refactoring approach. The results show that the use of linters as an educational coding tool has resulted in visible learning benefits to the students, similar results are observed by (BREUKER; DERRIKS; BRUNEKREEF, 2011). Börstler *et al.* (BÖRSTLER et al., 2018) describe how developers obtain knowledge about code quality. It concludes that one of the main parameters of code quality used by developers are the default linting patterns which come natively on many modern IDEs. It also concludes that the usage of automatic quality assurance tools are usually applied only by professional developers, but scarcely by students or educators, specially students. The use of linting tools was applied in (KARNALIM; CHIVERS et al., 2022) aiming to uncover quality issues in code developed by undergraduate students. Its conclusions describes a list of usual bad practices within this group.

The linting approach was used by (ZAMPETTI et al., 2017) on a set of 20 different Java projects hosted on GitHub under the objective of describing the general usage of linting tools in continuous integration pipelines. Its conclusions suggest that aiming to better use such tools, developers should customize the linter configurations in a manner to select which warnings are relevant. They also raise awareness on the existence of false positive detection. These suggestions where taken into account when selecting the six to be analyzed on this study.

An study on how often developers apply refactorings when a lint-based warning is generated on Java applications was done in (KIM; ERNST, 2007). They conclude that these manual refactorings are ineffective, since only 10% of them are treated during debug phase. Our automatic refactoring we present (Chapter 6) aims on reducing this ineffectiveness. (KAVALER et al., 2019) explore the usage of linting tools on top of a large set of JavaScript projects while (TÓMASDÓTTIR; ANICHE; DEURSEN, 2018) conduce a survey with JavaScript developers to understand their perceptions of linters adoption in JavaScript projects.

`ESlint`[3] was applied on top of Javascript code snippets found on StackOverflow[4] by (CAMPOS et al., 2019) with the objective of investigating the frequency of JavaScript lint-based warnings on the context of rule violations. They found that on average, each snippet has 11.9 lint-based warnings. The survey conducted by (MUSKE; SEREBRENIK, 2016) over 79 different papers and categorizes different approaches when dealing with lint-based warning.

Previous works (OORT et al., 2021; SIMMONS et al., 2020) are similar to our first study (Chapter 3), since they also use `Pylint` on its default configuration over context-restricted projects. Nevertheless, differently from this work where we analyze projects of several domains, they restrict the studies to machine learning (OORT et al., 2021) and data science (SIMMONS et al., 2020) projects. (OORT et al., 2021) found similar results when compared to ours: five out of the six lint-based warnings we focus in general Python projects

[3] <https://eslint.org/>
[4] <https://stackoverflow.com/>

are also frequent among machine-learning projects. The only exception is the Consider Using With lint-based warning. Simmons *et al.* present a list of 31 lint-based warnings and syntax errors and mention that they are frequent in data science projects. However, only one out of the six lint-based warnings we investigate in this work is in their list: the Consider Using Enumerate lint-based warning. Although we analyze only six lint-based warnings, differently from other works, here we report the results of a further investigation that analyze the developer's perceptions in terms of a survey and pull requests submissions.

A characterization of the code quality metrics on the Django web-framework[5] was done by (LIAWATIMENA et al., 2018). They also used `Pylint` for this purpose. However, the individual results for each lint-based warning collected were not made available. Differently, we were interested in particular results for each of the six lint-based warnings we explore in this work. Chen *et al.* (CHEN et al., 2018; CHEN et al., 2016) describe some challenges involved in detecting code smells (such as long parameter list and long method) in Python. Differently, we do not analyze only code smells, but six lint-based warnings that cover different categories of issues (according to the `Pylint` taxonomy). Rahman *et al.* (RAHMAN; RAHMAN; WILLIAMS, 2019) analyzes Python code stored on Gist[6] in a similar manner to what we accomplished in the first study (Chapter 3), though with the main focus on security smells. The study conducted by (HOOSHANGI; DASGUPTA, 2017) compares the evaluation of Python code using `Pylint` against the evaluation of the same code done by a human specialist. Its conclusions validate the accuracy of `Pylint` as a tool to detect quality flaws in Python code.

A similar, but smaller than ours, dataset was built by (OMARI; MARTINEZ, 2019) where they also used `Pylint` to extract some aggregated code quality metrics to serve as metadata for the dataset. For our study we had the need to keep the individual lint-based warning frequency metrics, instead of their aggregation, therefore the prior dataset did not fit our purposes and we had to build one from scratch covering 1,119 projects. (RUOHONEN; HJERPPE; RINDELL, 2021) described the incidence of security lint-based warnings on Pypi[7] packages using a snapshot of the packages state. The study in (RAI et al., 2020) uses `Pylint` in order to describe a continuous integration approach able to unify coding standards on the code development among heterogeneous teams.

The study conducted by (MONAT; OUADJAOUT; MINÉ, 2021) raise awareness on the topic of linting tools where they are built to analyze code developed to a single programming language, but considering that many modern projects are multi-language, the usage of different linting tools might present non standardized quality metrics. Being that said, they implement a linting tool able to analyze both Python and C. An study on the context of detection and automatic fix of lint-based warnings on the unit testing context in Python was implemented by (WANG et al., 2021) originating from it they implemented `PyNose` a tool

[5] <https://www.djangoproject.com/>
[6] <https://gist.github.com/>
[7] <https://pypi.org/>

focused on automatically refactoring test smells. A `Pylint` extension, named `MLSmellHound`, is proposed by (KANNAN et al., 2022). It detects the warnings in a context-aware approach, considering the purpose of the code under analysis, internal team rules, development life-cycle and whatsoever. It is focused on the detection of the lint-based warnings, but not on the automatic refactoring.

8 CONCLUSIONS

In this work we provide studies towards reducing the knowledge gap on the topic of Python lint-based warnings. For this purpose, it details the results of three different empirical studies (OLIVEIRA MáRCIO RIBEIRO, 2022), each of them regarding a different aspect of the subject: the frequency of lint-based warnings in real applications; the awareness of Python developers about lint-based warnings; and the perception on the relevance to refactor Python code in the presence of lint-based warnings. Initially, we cataloged six different lint-based warnings and we also suggested source code transformation (refactorings) to remove each of them. Then we used these lint-based warnings and refactorings in all the three studies. Specifically, our results bring evidence that the six lint-based warnings are common in the 1,119 we analyzed. Moreover, in general, both developers and project maintainers prefer Python code written without lint-based warnings. However, there are some deviations in this preference when considering different degrees of experience.

8.1 IMPLICATIONS

Now we describe some of the implications that our results might bring to research and to the practice.

We first show that the six lint-based warnings are common in the 1,119 projects we analyzed. In addition, the majority of the developers we interacted with by using the submission of pull requests agree on the relevance of refactoring such warnings. Also, we conclude that the preference for code that does not contain lint-based warnings is more prevalent among experienced developers when compared to inexperienced developers. This sequence of conclusions implies the following:

- First, practitioners and researchers should put more effort to make lint-based warnings and their potential threats to Python code better known, especially among inexperienced developers;
- Second, tools like Pylint identify many warnings. However, additional research should be done in order to create a catalog of refactorings able to remove these several lint-based warnings;

8.2 FUTURE WORK

As future work, we intend to create and validate a catalog of refactorings targeting lint-based warnings. We also intend to expand our study with other lint-based warnings.

We intend to implement a tool able to constantly monitor the Python Package Index. [1] in search of lint-based warnings based quality metrics. Similar to what was implemented for JavaScript by (CHATZIDIMITRIOU et al., 2018).

[1] <https://pypi.org/>

BIBLIOGRAPHY

ALEXANDRU, C. V. et al. On the usage of pythonic idioms. In: **Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software**. [S.l.: s.n.], 2018. p. 1–11.

BASSI, S. A primer on python for life science researchers. **PLoS computational biology**, Public Library of Science San Francisco, USA, v. 3, n. 11, p. e199, 2007.

BELLER, M. et al. Analyzing the state of static analysis: A large-scale evaluation in open source software. In: IEEE. **2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)**. [S.l.], 2016. v. 1, p. 470–481.

BENESTY, J. et al. Pearson correlation coefficient. In: **Noise reduction in speech processing**. [S.l.]: Springer, 2009. p. 1–4.

BORRELLI, A. et al. Detecting video game-specific bad smells in unity projects. In: **Proceedings of the 17th International Conference on Mining Software Repositories**. [S.l.: s.n.], 2020. p. 198–208.

BÖRSTLER, J. et al. " i know it when i see it" perceptions of code quality: Iticse'17 working group report. In: **Proceedings of the 2017 ITiCSE Conference on Working Group Reports**. [S.l.: s.n.], 2018. p. 70–85.

BREUKER, D. M.; DERRIKS, J.; BRUNEKREEF, J. Measuring static quality of student code. In: **Proceedings of the 16th annual joint conference on Innovation and technology in computer science education**. [S.l.: s.n.], 2011. p. 13–17.

BRONSHTEYN, I. Study of defects in a program code in python. **Programming and Computer Software**, Springer, v. 39, n. 6, p. 279–284, 2013.

BUTLER, S. et al. Exploring the influence of identifier names on code quality: An empirical study. In: IEEE. **2010 14th European Conference on Software Maintenance and Reengineering**. [S.l.], 2010. p. 156–165.

CAMPOS, U. F. et al. Mining rule violations in javascript code snippets. In: IEEE. **2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)**. [S.l.], 2019. p. 195–199.

CHATZIDIMITRIOU, K. et al. Npm-miner: An infrastructure for measuring the quality of the npm registry. In: IEEE. **2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)**. [S.l.], 2018. p. 42–45.

CHEN, Z. et al. Detecting code smells in python programs. In: IEEE. **2016 international conference on Software Analysis, Testing and Evolution (SATE)**. [S.l.], 2016. p. 18–23.

CHEN, Z. et al. Understanding metric-based detectable smells in python software: A comparative study. **Information and Software Technology**, Elsevier, v. 94, p. 14–29, 2018.

DASGUPTA, S.; HOOSHANGI, S. Code quality: Examining the efficacy of automated tools. 2017.

DEUTSKENS, E. et al. Response rate and response quality of internet-based surveys: an experimental study. **Marketing letters**, Springer, v. 15, n. 1, p. 21–36, 2004.

DILLIG, I.; DILLIG, T.; AIKEN, A. Automated error diagnosis using abductive inference. **ACM SIGPLAN Notices**, ACM New York, NY, USA, v. 47, n. 6, p. 181–192, 2012.

FLANAGAN, C. et al. Extended static checking for java. In: **Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation**. [S.l.: s.n.], 2002. p. 234–245.

GADIENT, P. et al. Security code smells in android icc. **Empirical software engineering**, Springer, v. 24, n. 5, p. 3046–3076, 2019.

GAMMA, E. et al. Design patterns: Abstraction and reuse of object-oriented design. In: SPRINGER. **European Conference on Object-Oriented Programming**. [S.l.], 1993. p. 406–431.

HABCHI, S. et al. Code smells in ios apps: How do they compare to android? In: IEEE. **2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)**. [S.l.], 2017. p. 110–121.

HOOSHANGI, S.; DASGUPTA, S. Code quality: Examining the efficacy of automated tools. In: ASSOCIATION FOR INFORMATION SYSTEMS. **Americas Conference on Information Systems (AMCIS)**. [S.l.], 2017.

JAFARI, A. J. et al. Dependency smells in javascript projects. **IEEE Transactions on Software Engineering**, IEEE, 2021.

JOHNSON, B. et al. Why don't software developers use static analysis tools to find bugs? In: IEEE. **2013 35th International Conference on Software Engineering (ICSE)**. [S.l.], 2013. p. 672–681.

KANNAN, J. et al. Mlsmellhound: A context-aware code analysis tool. **arXiv preprint arXiv:2205.03790**, 2022.

KARNALIM, O.; CHIVERS, W. et al. Work-in-progress: Code quality issues of computing undergraduates. In: IEEE. **2022 IEEE Global Engineering Education Conference (EDUCON)**. [S.l.], 2022. p. 1734–1736.

KAVALER, D. et al. Tool choice matters: Javascript quality assurance tools and usage outcomes in github projects. In: IEEE. **2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)**. [S.l.], 2019. p. 476–487.

KIM, S.; ERNST, M. D. Which warnings should i fix first? In: **Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering**. [S.l.: s.n.], 2007. p. 45–54.

KUMAR, R.; NORI, A. V. The economics of static analysis tools. In: **Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering**. [S.l.: s.n.], 2013. p. 707–710.

LENARDUZZI, V. et al. Does code quality affect pull request acceptance? an empirical study. **Journal of Systems and Software**, Elsevier, v. 171, p. 110806, 2021.

LI, L. et al. Static analysis of android apps: A systematic literature review. **Information and Software Technology**, Elsevier, v. 88, p. 67–95, 2017.

LIAWATIMENA, S. et al. Django web framework software metrics measurement using radon and pylint. In: IEEE. **2018 Indonesian Association for Pattern Recognition International Conference (INAPR)**. [S.l.], 2018. p. 218–222.

MENDONCA, V. R. L. de et al. Static analysis techniques and tools: A systematic mapping study. **ICSEA**, 2013.

Miller, A. H. et al. Parlai: A dialog research software platform. **arXiv preprint arXiv:1705.06476**, 2017.

MONAT, R.; OUADJAOUT, A.; MINÉ, A. A multilanguage static analysis of python programs with native c extensions. In: SPRINGER. **International Static Analysis Symposium**. [S.l.], 2021. p. 323–345.

MURPHY-HILL, E.; PARNIN, C.; BLACK, A. P. How we refactor, and how we know it. **IEEE Transactions on Software Engineering**, IEEE, v. 38, n. 1, p. 5–18, 2011.

MUSKE, T.; SEREBRENIK, A. Survey of approaches for handling static analysis alarms. In: IEEE. **2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)**. [S.l.], 2016. p. 157–166.

NAGAPPAN, N.; BALL, T. Static analysis tools as early indicators of pre-release defect density. In: IEEE. **Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.** [S.l.], 2005. p. 580–586.

NAGY, C.; CLEVE, A. Sqlinspect: A static analyzer to inspect database usage in java applications. In: **Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings**. [S.l.: s.n.], 2018. p. 93–96.

NANDA, M. G. et al. Making defect-finding tools work for you. In: IEEE. **2010 ACM/IEEE 32nd International Conference on Software Engineering**. [S.l.], 2010. v. 2, p. 99–108.

OLIVEIRA MáRCIO RIBEIRO, R. G. I. W. B. F. N. Lint-based warnings in python code: Frequency, awareness and refactoring. In: IEEE. **2022 IEEE 22st International Working Conference on Source Code Analysis and Manipulation (SCAM)**. [S.l.], 2022.

OMARI, S.; MARTINEZ, G. Enabling empirical research: A corpus of large-scale python systems. In: SPRINGER. **Proceedings of the Future Technologies Conference**. [S.l.], 2019. p. 661–669.

OORT, B. V. et al. The prevalence of code smells in machine learning projects. **arXiv preprint arXiv:2103.04146**, 2021.

PADUA, G. B. D.; SHANG, W. Studying the prevalence of exception handling anti-patterns. In: IEEE. **2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)**. [S.l.], 2017. p. 328–331.

RADU, A.; NADI, S. A dataset of non-functional bugs. In: IEEE. **2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)**. [S.l.], 2019. p. 399–403.

RAHMAN, M. R.; RAHMAN, A.; WILLIAMS, L. Share, but be aware: Security smells in python gists. In: IEEE. **2019 IEEE International conference on software maintenance and evolution (ICSME)**. [S.l.], 2019. p. 536–540.

RAI, I. et al. **CMS-Automation of Unified building and testing**. [S.l.], 2020.

RAMALHO, L. **Fluent Python: Clear, concise, and effective programming**. [S.l.]: " O'Reilly Media, Inc.", 2015.

REYNOLDS, Z. P. et al. Identifying and documenting false positive patterns generated by static code analysis tools. In: IEEE. **2017 IEEE/ACM 4th International Workshop on Software Engineering Research and Industrial Practice (SER&IP)**. [S.l.], 2017. p. 55–61.

RIVAL, X. Understanding the origin of alarms in astrée. In: SPRINGER. **International Static Analysis Symposium**. [S.l.], 2005. p. 303–319.

RUOHONEN, J.; HJERPPE, K.; RINDELL, K. A large-scale security-oriented static analysis of python packages in pypi. In: IEEE. **2021 18th International Conference on Privacy, Security and Trust (PST)**. [S.l.], 2021. p. 1–10.

SAABITH, A. S.; FAREEZ, M.; VINOTHRAJ, T. Python current trend applications-an overview. **International Journal of Advance Engineering and Research Development**, v. 6, n. 10, 2019.

SIMMONS, A. J. et al. A large-scale comparative analysis of coding standard conformance in open-source data science projects. In: **Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)**. [S.l.: s.n.], 2020. p. 1–11.

SOARES, D. M. et al. Acceptance factors of pull requests in open-source projects. In: **Proceedings of the 30th Annual ACM Symposium on Applied Computing**. [S.l.: s.n.], 2015. p. 1541–1546.

SRIPADA, S.; REDDY, Y. R.; SUREKA, A. In support of peer code review and inspection in an undergraduate software engineering course. In: IEEE. **2015 IEEE 28th conference on software engineering education and training**. [S.l.], 2015. p. 3–6.

TÓMASDÓTTIR, K. F.; ANICHE, M.; DEURSEN, A. V. The adoption of javascript linters in practice: A case study on eslint. **IEEE Transactions on Software Engineering**, IEEE, v. 46, n. 8, p. 863–891, 2018.

VASSALLO, C. et al. How developers engage with static analysis tools in different contexts. **Empirical Software Engineering**, Springer, v. 25, n. 2, p. 1419–1457, 2020.

VAVROVÁ, N.; ZAYTSEV, V. Does python smell like java? tool support for design defect discovery in python. **arXiv preprint arXiv:1703.10882**, 2017.

VEGI, L. F. d. M.; VALENTE, M. T. Code smells in elixir: Early results from a grey literature review. **arXiv preprint arXiv:2203.08877**, 2022.

WAGNER, S. et al. Comparing bug finding tools with reviews and tests. In: SPRINGER. **IFIP International Conference on Testing of Communicating Systems**. [S.l.], 2005. p. 40–55.

WANG, T. et al. Pynose: A test smell detector for python. In: IEEE. **2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)**. [S.l.], 2021. p. 593–605.

WRIGHT, B.; SCHWAGER, P. H. Online survey research: can response factors be improved? **Journal of Internet Commerce**, Taylor & Francis, v. 7, n. 2, p. 253–269, 2008.

YU, Y. et al. Wait for it: Determinants of pull request evaluation latency on github. In: IEEE. **2015 IEEE/ACM 12th working conference on mining software repositories**. [S.l.], 2015. p. 367–371.

YU, Z.; SU, X.; MA, P. Mocklinter: Linting mutual exclusive deadlocks with lock allocation graphs. **International Journal of Hybrid Information Technology**, v. 9, n. 3, p. 355–374, 2016.

ZAMPETTI, F. et al. How open source projects use static code analysis tools in continuous integration pipelines. In: IEEE. **2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)**. [S.l.], 2017. p. 334–344.

ZHANG, H.; CRUZ, L.; DEURSEN, A. van. Code smells for machine learning applications. **arXiv preprint arXiv:2203.13746**, 2022.

ZHAO, D. et al. Seenomaly: Vision-based linting of gui animation effects against design-don't guidelines. In: IEEE. **2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)**. [S.l.], 2020. p. 1286–1297.

ZHENG, J. et al. On the value of static analysis for fault detection in software. **IEEE transactions on software engineering**, IEEE, v. 32, n. 4, p. 240–253, 2006.