MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Detecting code quality defects in students’ solutions

Master’s Thesis

ANNA ŘECHTÁČKOVÁ

Advisor: RNDr. Tomáš Effenberger

Department of Computer Science

Brno, Spring 2023

MUNI
FI

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Anna Řechtáčková

Advisor: RNDr. Tomáš Effenberger

Acknowledgements

First and foremost, I thank my supervisor, for always being ready to discuss any questions I had, no matter how big or small, for pointing out my many oversights in section headings and for offering me this topic in the first place. Thank you Tom, it was a pleasure to work with you :)

I am also ever so grateful to Ondra for guiding me through all the industry best practices and for all the food and tea.

Abstract

Code quality is critical for easy code maintenance, but teaching it to novice programmers through manual code reviews scales poorly. Several automated tools exist, but their feedback is only sometimes relevant, or they are hard to use by novices. The focus of this thesis is to create a tool that overcomes the shortcomings of the existing tools.

To achieve that, I developed EduLint, a Python linter adapting Pylint and Flake8, tailoring them to deliver only feedback suitable to novice programmers. The tool can detect over 200 code quality defects, some of which are not detected by any other tool. It can be configured to suit various programming tasks and target groups. Students can use the tool through a web interface, which also displays explanations for the detected defects.

I compare EduLint with other Python linters on a dataset of 100,000 submissions by novice programmers, showing EduLint provides code quality feedback that is more relevant for novices.

Keywords

automation, program analysis, Python, source code reviews, artificial intelligence, data analysis

Contents

1 Introduction **1**

2 Aims and Scope **3**
- 2.1 Defect definition . . . 3
- 2.2 Target audience . . . 4
  - 2.2.1 Novice programmers . . . 4
  - 2.2.2 Educators . . . 4
- 2.3 Requirements . . . 5
  - 2.3.1 Defects relevant to novice programmers . . . 5
  - 2.3.2 Precise detection of defects . . . 6
  - 2.3.3 Configurability . . . 8
  - 2.3.4 Clear descriptions of defects and how to fix them 8
  - 2.3.5 Ease of use . . . 8

3 Code quality defects **10**
- 3.1 Categorization . . . 10
  - 3.1.1 Formatting . . . 10
  - 3.1.2 Unsuited construct . . . 11
  - 3.1.3 Simplifiable . . . 11
  - 3.1.4 Unused . . . 11
  - 3.1.5 Erroneous . . . 11
  - 3.1.6 Error prone . . . 12
  - 3.1.7 Poor name . . . 12
  - 3.1.8 Long or overly complex . . . 12
  - 3.1.9 Duplicate . . . 12
  - 3.1.10 Poorly designed . . . 13
- 3.2 Defect examples . . . 13

4 Existing solutions **17**
- 4.1 Industry-grade linters . . . 17
  - 4.1.1 Pylint . . . 17
  - 4.1.2 Pyflakes . . . 18
  - 4.1.3 Pycodestyle . . . 19
  - 4.1.4 Flake8 . . . 19
  - 4.1.5 Wemake python styleguide (WPS) . . . 19
  - 4.1.6 Ruff . . . 20

4.2 Integrated development environments (IDEs) 20
4.2.1 Thonny 20
4.2.2 Industry-grade IDEs 20
4.3 Educational linters 21
4.3.1 PyTA 21
4.3.2 Hyperstyle 22
4.3.3 Others 23

5 EduLint 24
5.1 Design decisions 24
5.2 Architecture and linting process 26
5.3 Configuration 27
5.3.1 Configuration means 28
5.3.2 Default and extended defect groups 28
5.3.3 Other configuration options 29
5.3.4 Documentation 29
5.4 Custom detectors 30
5.5 Postprocessing 31
5.6 Explanations 31
5.7 Website 33
5.7.1 Design decisions 33
5.7.2 Interfaces 34

6 Comparison with other tools 36
6.1 Experiments setup 36
6.1.1 Code datasets 36
6.1.2 Defects dataset 39
6.1.3 Running the tools 39
6.1.4 Cleaning the linting data 40
6.2 Comparison by frequent defects 43
6.2.1 Finding frequent defects 43
6.2.2 Relevance reasoning 45
6.2.3 Summary 48
6.3 Comparison by frequent defect categories 49
6.3.1 Relevance reasoning 49
6.3.2 Detectable defects 51
6.3.3 Detected defects 52
6.3.4 Summary 54

6.4 Comparison by requirements 55
6.5 Threats to validity 57

7 Adherence to requirements 59
7.1 Evaluation of adherence 59
7.1.1 Defects relevant to novice programmers 59
7.1.2 Precise detection of defects 61
7.1.3 Configurability 62
7.1.4 Clear descriptions of defects and how to fix them 63
7.1.5 Ease of use 63
7.2 Future work 63
7.2.1 Different configurations for different target groups 63
7.2.2 More defect detectors 64
7.2.3 Prioritization and presentation 65
7.2.4 Thonny plugin 65

8 Conclusion 66

A EduLint installation and running 68

B Archive organization and contributions 69

C EduLint's custom detectors 70
C.1 Detectors 70
C.1.1 Unsuited construct 70
C.1.2 Simplifiable 74
C.1.3 Unused 79
C.1.4 Error prone 79
C.1.5 Poor name 81
C.2 Custom defect frequencies 82

D The complete list of defects 86

Bibliography 94

List of Tables

3.1 Examples of defects 13

6.1 Dataset sizes and their code lengths 39
6.2 Ignored defects 42
6.3 Frequent defects 44
6.4 Detectable defects by category 52
6.5 Detected defects in files by category 53
6.6 Comparison of tools by requirements 56
6.7 False positives (FP) examples 57

C.1 Custom defects 83

D.1 All defects with at least one occurrence 86

List of Figures

5.1 An overview of the linting pipeline 26
5.2 EduLint web interface 35
5.3 EduLint web interface showing an explanation 35
6.1 Defect report cleaning process 41

1 Introduction

Code quality is vital for easy code modification and maintenance [1], which in turn dominates software cost [2]. In spite of this, the teaching of programming focuses prominently on code correctness [3, 1]. Providing novice programmers with manual code reviews can be resource-intensive or even completely resource-prohibitive [4, 5]. This leads to code quality often being overlooked [1, 3, 6]. Even still, manual reviews come with their own set of issues, like stretching the feedback loop, sometimes to several weeks [7, 8, 9], and different educators often delivering quite different feedback [6, 8, 9].

While there are several tools aimed at novice programmers that provide automatic feedback to code quality (mostly for Java [10, 11, 12, 13, 14, 4, 15] and Python [16, 4, 15]), they come with different caveats: the feedback is sometimes overly demanding of the students' code or the tools are tailored to a specific course. Meanwhile, general-purpose linters are usually hard to use, focus on code formatting, often give feedback that is too advanced, technical, or strict, but also omit a number of defects altogether [6, 3].

EduLint, the tool developed for this thesis, aims to provide plenty of relevant feedback, be easy to use and configurable to allow for differing contexts and educators' opinions. It is a Python linter which wraps Pylint and Flake8, both professional Python linters, and uses a custom configuration to fit novice programmers better. It detects over 200 different code quality defects and formatting inconsistencies. The default configuration can be altered in several ways, either by enabling or disabling individual defects or whole groups of them. The configuration can also be distributed as a part of a task's starter code, which saves the students from managing configuration files.

I created several custom detectors for defects undetected by other tools (e.g. the code uses a while loop instead of a for loop, uses float division and explicit conversion to integer instead of integer division). I also reimplemented several detectors from other tools so that they would generate fewer false positives, detect more variations of the defect or provide more actionable feedback.

To make the tool easier to use for novice programmers, I also developed a web interface for EduLint, where a novice programmer can paste their code and receive feedback, together with an explanation of why they should fix the defect and code examples of how they can fix it.

Both the linter[1] and the web interface[2] are publicly available, open-source projects.

I argue that EduLint focuses on code quality feedback relevant to novice programmers, compared to several industry-grade and educational linter. I detail how I ensured the tool provides well-suited feedback through precise detectors and show what the tool's shortcomings are.

In Chapter 2 of this thesis, I start by defining several terms and concepts I use throughout this thesis, I characterize the target audience of EduLint, and I conclude with the list of requirements set for the resulting tool. In Chapter 3, I present a categorization of different defects, with examples. In Chapter 4, I introduce existing solutions, both educational and general-purpose, and discuss their limitations. In Chapter 5, I describe EduLint itself: the design decisions I made, the tool's architecture, the linting pipeline, and I end it by outlining the web interface. I split the evaluation of the tool between two chapters: In Chapter 6, I show how EduLint compares to other solutions, arguing that it provides feedback that is especially relevant to novice programmers. In Chapter 7, I evaluate the tool's adherence to the requirements I specified. In the same chapter, I also outline the future work. In the last chapter, I conclude the thesis. In the appendix, I describe how to run EduLint (A), what is the organization of the thesis archive (B), I list the custom detectors I developed and explain how they operate (C), and I show a list of defects EduLint can detect (D).

1. `https://github.com/GiraffeReversed/edulint/tree/v2.6.4`
2. `https://github.com/GiraffeReversed/edulint-web/tree/thesis`

2 Aims and Scope

In this chapter, I first describe what is a defect. Then I introduce the target audience (students and educators): who are they, what specifics they have and how they are expected to use the tool. Third, I present a defect categorization to illustrate the broadness of different defects and to provide a means of comparing different tools. In the end, I describe the main requirements for the tool and the reasoning behind each of them.

2.1 Defect definition

Defect is defined by Effenberger et al. [17] as "any imperfect part of code for which it is possible to provide some feedback or advice".

Defect report reports a specific defect instance to the user. Such a report contains, among others, the line (and possibly the end line) on which the defect appears and a message describing what is wrong and, in some cases, how it can be fixed. The message can be tailored to include parts of the code where the defect occurs. An explanation of why the defect should be fixed, and code examples of how to fix it can also be a part of a defect report.

A report from EduLint can look as follows. A report in machine-readable format may include more information.

[line] 14: Use augmenting assignment: `'a //= 2'`

Why is it a problem?
Using augmented assign is not only shorter, but also clearer and less prone to errors.

How to solve it?
For example, instead of writing `x = x + 1`, you can write `x += 1`. Works for almost any operator.

Defect detector is an automatic means of discovering a defect.

2.2 Target audience

This section describes the two main groups of users of the tool: the novice programmers that use the tool to get feedback on their coding ability and the educators that assess the novice programmers' skills and potentially provide feedback based on the results from the tool.

2.2.1 Novice programmers

Many groups of people learn to program: computer science students, participants in programming courses and seminars, users of various online learning environments and massive open online courses, and so on. A novice programmer might only learn of a defect in their code if it is pointed out to them and be motivated enough to fix it only if they learn why they should avoid it.

Novice programmers also usually have limited technical knowledge (e.g. using the command line to run programs may present a challenge).

Novice programmers often start learning programming by writing short, self-contained code (tens to lower hundreds of lines) inside a single file. The use of advanced structures in their code is possibly undesirable (as such usage might suggest that the student cannot solve the problem using simpler constructs – e.g. a student might use a closure because they do not know how to pass arguments to a normal function, a student might use a loop's else branch as they might not be aware that a loop does not have to have it).

2.2.2 Educators

I use educators as a broad term for programming task creators, task graders, consultants, course instructors, etc. They share (among others) the goal of delivering quality feedback to novice programmers efficiently and within their budget.

Each educator usually works with a group of students (be it students in a seminar group or all users of a learning environment). Educators usually possess some degree of technical knowledge.

There are different ways in which different educators may regard code quality feedback: a course instructor might decide to grade code quality as a part of students' evaluation, and a task creator might add tips on how to come up with a clean solution. However, many educators or educator groups cannot provide code quality feedback systematically, as providing students with individual code reviews is resource-intensive. Aggregate feedback (while still being costly to prepare) may not completely mitigate all defects in individuals' code (for example, a student may be under the impression that some relevant part of the feedback does not apply to their code or assume their code is all right, even though it contains some defect that was just too rare to be addressed in the aggregate). Also, students may only address feedback if they are required to.

2.3 Requirements

In this section, I detail the major requirements for the developed tool and describe why they are significant, supporting them with related research in several cases. I collected these requirements consulting with other educators and observing novice programmers.

2.3.1 Defects relevant to novice programmers

The tool should detect a wide variety of defects relevant to novice programmers: defects regarding constructs they frequently come into contact with (e.g. loops, functions, not abstract classes) and defects appropriate to their level of experience. It is not required for the tool to report *all* relevant defects, but all reported defects should be relevant.

Several papers present a list of defects with an indication that the authors consider them relevant to novice programmers:

- Effenberger et al. [17] list 32 code quality defects appearing in Python solutions from an online learning environment used by high-school and university students and show their frequencies across different CS1 topics.
- De Ruvo et al. [7] list 16 defects prevalent in Java code by CS1/CS2 students and in project assignments from a software design course. Only one of the defects cannot occur in Python.
- Keuning et al. present three papers containing a list of defects:

  - In 2017 [18], they listed 24 defects found in solutions by novice users of BlueJ IDE, though seven of these are specific to Java or OOP and two regard switch statements, which are not present in Python.
  - In 2019 [6], they presented three solutions to a task in Java and report which defects the samples contain would educators point out to students. The number of defects ranges around 20, but the exact number of defects listed in this paper cannot be determined. They only list some suggestions under general or miscellaneous, and the defects differ in granularity. Most of the defects are not specific to Java or OOP.
  - In 2021 [14], they introduced their tool for teaching code quality feedback. Nevertheless, the tool is slightly tailored to a set of six selected Java tasks that the students improve, assisted by the tool. They list 19 defects, of which approximately 12 seem independent of the selected tasks.

- Groeneveld et al. [19] primarily focus on exploring whether creative novices also write clean code. However, to determine code quality, they present a list of 26 defects, out of which 17 are also relevant to Python.

- X. Liu et al. [15] list the 10 most frequent defects in Python code detected by the tool presented in the paper, which they developed to provide code quality feedback.

The developed tool should often detect defects that some of these papers mention.

2.3.2 Precise detection of defects

The detectors should have high precision. A novice programmer may not be able to determine that a defect report is a false positive, which can be confusing and lead to them lowering the quality of their code by trying to fix the defect. Also, should the tool be used as a part of the grading process, false positives could lead to unfairly lost points for the students. Therefore, the precision of the detectors is more important than their recall.

A false positive occurs when a detector flags some part of code that does contain the construct it searches for but in a context in which a human would judge the use as legitimate. For example, consider cases when a student uses `else: if c` instead of `elif c`. Such a situation may sound like an obvious defect, but consider the following simplified code:

```
if c1:
    if c2:
        # do A
    else:
        # do B
else:
    if c2:
        # do something similar to A
    else:
        # do something similar to B
```

While this code could be contested for possible code duplication, transforming the outer `else` to `elif` does not much improve its quality – and on the contrary, it breaks the current consistency of similar code having the same indentation.

On the other hand, a human might judge every occurrence of a defect as legitimate. I do not consider this situation a false positive because it should be possible for the educator to disable the defect as a whole.

Nevertheless, this does not mean that the tool should emit no false positives on any code – some constructs have their uses in code written by an experienced programmer but which often get misused by novices. An example of this is writing a condition as `c is True` – while an experienced programmer might use this to handle three-valued logic, a novice usually only includes the extra code without knowing they can omit it. Therefore, reporting that defect (and others like it) is desirable since a novice programmer should learn of their misconception. In contrast, a more experienced programmer can find an alternative approach.

2.3.3 Configurability

Each educator might have a different idea of what defects novice programmers should learn about, as shown by several research groups [6, 20]. Some defects may only be relevant to some kinds of novice programmers (high school versus college students). Therefore, the tool should allow each educator to use only those detectors they deem relevant for their students. Also, a different configuration might be appropriate for each task (for example, detectors could be added gradually during the semester). Ideally, novice programmers should never be forced to manage the configuration themselves, as they should focus primarily on learning programming rather than attending to the tools they use.

Since this requirement leads to the tool having plenty of configuration options, they should all be listed and documented.

2.3.4 Clear descriptions of defects and how to fix them

Each reported defect should come with a clear description of what problems the defect can cause and why the student should fix it. Receiving an explanation can help novice programmer understand their misconception and avoid the defect in the future. The report should ideally include examples of how the student can fix the defect.

Nutbrown et al. [9] mention this feature as one of the main advantages of automated assessment tools.

2.3.5 Ease of use

As stated in Section 2.2.1, novice programmers might struggle with installing and running a tool from the command line, especially if they do not know anybody with whom they might consult technical problems. Also, not all novice programmers have the rights to install software on the computer they use (they might only have a school computer available).

Also, novice programmers should ideally not manage configuration files to use the configuration their educator meant for them.

The tool should therefore provide a way for students to use it without struggling with technical problems impeding their learning process.

Most tools providing code quality feedback to novice programmers do not just display their output as plain text. They use a web interface [14], an IDE plugin [11], they generate output in HTML [16] or were integrated into a learning environment [4, 13, 12, 15].

3 Code quality defects

To better understand the scope of defects that (novice) programmers' code contains and to better compare different linters, I present a defect categorization. For each category, I give examples of defects that belong to it.

In the first section, I describe the criteria I used to categorize each defect. In Section 3.2, I give examples of defects in each category and sample codes in which the defect manifests. The table also shows whether EduLint reports this defect.

3.1 Categorization

I distinguish ten categories of defect based on why is code containing it problematic: *formatting*, *unsuited construct*, *simplifiable*, *unused*, *erroneous*, *error prone*, *poor name*, *long or overly complex*, *duplicate*, *poorly designed*.

I started with the six categories presented by Effenberger et al. [17], but I decided to split complex code into *unsuited construct* and *simplifiable*. I also added three categories (*erroneous*, *error prone*, *poorly designed*) for defects that they did not mention but which occur nevertheless.

This categorization can sometimes lead to ambiguous conclusions about a defect's category. For example, a *formatting* defect could also trigger a `SyntaxError`, thus earning a place in the *erroneous* category. Removing *unused* code does *simplify* the code. *Long or overly complex* and *duplicate* are both symptoms of *poorly designed* code. I claim that this categorization is still useful for gaining insight into the capabilities of different linting tools and their comparison.

For each category, I mention several examples of its defects. For more examples and code samples that illustrate the defect, see Table 3.1.

3.1.1 Formatting

A *formatting* defect is related to whitespace, parentheses and import placement. Most of these are PEP8 violations.

This category includes defects like mixing tabs and spaces, over-indented code, whitespace before `)` or after `(`, import not placed at the top of the module, and similar.

3.1.2 Unsuited construct

Defects in this category deal with code that can be rewritten using another construct (or another way to use the same construct) into a cleaner, more efficient or more readable version. This may include suggesting using a for loop instead of a while loop, using integer division `a // b` instead of `int(a / b)`, iterating directly like `for val in lst` instead of `for i in range(len(lst))`, testing if a value is in a set instead of comparing to several variables, and so on.

3.1.3 Simplifiable

Code with a *simplifiable* defect can have some part deleted or tweaked without affecting the behaviour. This can include dropping `is True` from a condition or simplifying `if c: return True else: return False` to `return c`, dropping `else`/`elif` after `return`/`break`/`continue`, removing redundant operations like adding 0 or empty string, removing loops that make at most one iteration and more.

3.1.4 Unused

Similar to *simplifiable*, *unused* deals with code that can be removed, but in this case, it is unused variables or arguments, or entire lines that can be removed (useless `pass`, unreachable code, variable assigning to itself, etc.).

3.1.5 Erroneous

Code containing a defect in the *erroneous* category throws an unexpected error when run. This may include passing too many or too few arguments to a function call, using an undefined name, type errors, using return outside a function, explicit division by zero and many more defects.

3.1.6 Error prone

While not causing an error, defects in this category mark code that may cause an error or an error could be easily introduced into it during modifications. Defects in this category include using global variables or wildcard imports, comparing floats for equality or adding or removing elements from a structure being iterated over.

3.1.7 Poor name

This category contains defects regarding variable, argument, class or module names. This category contains defects regarding names breaking naming conventions (e.g. a variable name in PascalCase instead of snake_case), names with confusing characters or character combinations (e.g. `l` vs `1`, `O` vs `0`), names shadowing built-ins or names defined in an outer scope, ill-suited names (like `foo`, names with spelling errors, possibly single-character variable names), or names that are outright misleading or hindering readability (like `i` in `for i in text`).

3.1.8 Long or overly complex

This category contains all defects regarding too long functions, functions with too many arguments, nested blocks, methods or attributes in a class, elif branches, or local variables in a code block, and logical operators in a condition etc.

All these defects depend heavily on how many "too many" is. The defect "function is too long" will manifest differently if "too many" is 100 or 10 lines.

3.1.9 Duplicate

There are multiple defects related to duplicate code: duplicate sequences or blocks, repeated expressions or the same code in both branches of an if statement.

3.1.10 Poorly designed

This category contains fundamental defects like code that is poorly decomposed, does not use an appropriate algorithm, or is generally hard to follow or maintain.

Some of these defects can be easily detected by proxy like *long or overly complex* or *duplicate*, but those do not provide advice on how to fix it. Many defects in this category can be challenging to detect and suggest fixes, even for humans.

3.2 Defect examples

Table 3.1 shows examples of defects in each category, together with a code excerpt in which the defect manifests. The table also shows whether EduLint is (to the best of my knowledge) the only tool that detects the defect (★), if EduLint detects the defect via a custom detector I developed to improve over detectors in other tools (↑), if I adopted a detector from another tool (✓), or if EduLint does not detect the defect at all (×). If an asterisk follows after the symbol (*), EduLint does not detect the defect in its default configuration, but in some extension (for the difference, see Section 5.3.2).

Table 3.1: Examples of defects

Defect	Example	EduLint
Formatting		
inappropriate whitespace	`( x+y )`	✓

missing newline	`def fun1():`
	`    # body`
	`def fun2():`
	`    # body`
		✓

no empty lines in function to separate logical parts		×

Defect	Example	EduLint
Unsuited construct		

using a `while` loop instead of a `for` loop	`while i < n:`
	`    # body`
	`    i += 1`
		★

using float division instead of integer division	`int(x / y)`	★

iterating using indices instead of iterating directly	`for i in range(len(lst)):`
	`    # code only reading`
	`    # from lst[i]`
		↑

comparing to several variables instead of testing if the value is in a container	`x == 1 or x == 2 or x == 3`	↑
using a magical constant instead of applying `ord` to a letter	`ord(char) - 65`	★
Simplifiable		
`is` with a boolean value	`val is False`	↑

simplifiable `if return`	`if c:`
	`    return True`
	`return False`
		↑

nested `if`s	`if c1:`
	`    if c2:`
	`        # body`
		★

unnecessary `else`/`elif` after `return`, `break` or `continue`	`if c:`
	`    return val1`
	`else:`
	`    return val2`
		✓

redundant arithmetic	`val + []`	↑

A loop making at most one iteration	`for i in range(1):`
	`    # body`
		↑

Unused		
unused variable		✓

useless `pass`	`def fun():`
	`    pass`
	`    # body`
		✓

unreachable code	`def fun():`
	`    # body`
	`    return`
	`    # unreachable code`
		✓

self-assigning variable	`val = val`	✓

changing control variable in a `for` loop	`for i in range(n):`
	`    # body`
	`    i += 1`
		★

Defect	Example	EduLint
Erroneous		
passing too many arguments	`len(lst1, lst2)`	×
undefined variable		✓
type error	"result: " + 10	×
using `return` outside a function		✓
explicit division by zero	`val / 0`	×
Error prone		
global variables		↑
wildcard imports	`from module import *`	✓
comparing floats for equality	`x == 0.3`	×

adding elements to a structure that is being iterated over or removing elements from it	`for val in lst:`
	`    # body`
	`    lst.remove(x)`
		↑

Poor name		
name does not follow naming conventions	`localVariable`, `global_constant`, `CLASSNAME`	↑
name can be easily mistaken	`l`, `I`, `1`, `O`, `0`	✓
name shadows built-in function	`list`, `sum`	✓
name containing a spelling error	`lenght`	×
name is misleading	`for i in text`	×
single-variable name outside of usual context	list named `x` containing measured temperatures	↑[1]
using variable whose name starts with an underscore	`_ += 1`	×
Long or overly complex		
line too long		✓
too many statements in a function/method		✓*
too many arguments		✓*
nesting too deep		✓*
too many public methods		×
call chain too long	`fun3(fun2(fun1(arg)))`	×
overly complex augmented assignment	`a += b ** c - d`	×

1. EduLint enables configuring which single-character variable names are allowed to use in a given task. See C.1.5 for more information.

Defect	Example	EduLint
Duplicate		[2]
duplicate sequence[3]	`f(1); f(2); f(3); f(4)`	×

duplicate block[3]	`if c: f(1); g(1); f(1)`
	`else: f(2); g(2); f(2)`
		×

duplicate expression[3]	`if a[n//2] % 2 == 0:`
	`    print(a[n//2] % 2)`
		×

same statement in the positive and the negative branch of an `if` statement	`if c: f(1); i += 1`
	`else: g(1); i += 1`
		×

Poorly designed		[4]
missing decomposition	code probably contains duplicate code blocks	×
illogical decomposition	code is decomposed into functions that do not have clear purpose	×
duplicating standard library	writing custom sorting function	×
inappropriate algorithm	merging two sorted lists by concatenating them and then sorting	×
inappropriate data structure	repeatedly testing for presence of an item in a list	×

2. EduLint currently detects none of these defects and it is a crucial space for improvement, discussed in Section 7.2.2.
3. Example is adapted from Effenberger et al. [17].
4. EduLint detects none of these defects as there are other defects it still misses which are easier to detect than the defects in this category.

4 Existing solutions

In this chapter, I describe existing solutions, starting with industry-grade solutions, be it various stand-alone linters and style checkers or tools and capabilities built directly into IDEs, ending with linters developed specifically for novice programmers.

I only considered the tools (or parts of them) dedicated to providing code quality feedback to any Python code on demand.

I discuss my reasons for choosing or disregarding the reuse of any particular tool during the development of EduLint in Section 5.1.

For a comparison of selected tools (Pylint, PyTA, Hyperstyle) and EduLint with regards to the requirements set in Section 2.3, see Section 6.4.

4.1 Industry-grade linters

This section lists widely used, real-world, industry-grade linters and style checkers, their capabilities and limitations and how they can be configured and extended.

All of the tools discussed here are available as pip packages. Some are also available on a web page (primarily those focusing on code formatting).

4.1.1 Pylint

Pylint [21] is a linter that "checks for errors, enforces a coding standard, looks for code smells, and can make suggestions about how the code could be refactored". Over half of all defects it can detect falls is either *erroneous* or *error prone* (though this says nothing about how often they occur in code).

It is highly configurable (each defect can be disabled or enabled individually), and several defects come with their own configuration options (e.g. bad variable names that Pylint should always report).

Pylint can be extended by creating plugins – Python modules containing classes that register with Pylint when it is being run. Each class can detect several defects.

Out of the box, it comes with short descriptions of the reported defects and, for some, examples of code that contain the defect and its fixed version, but these are not presented to the user when they use the tool to lint their code. They are either available through a special command or in the documentation.

It provides a wide variety of detectors for many defects but does not always detect them consistently. For example, in the following code segment, Pylint reports (among others) `R1703: The if statement can be replaced with 'return bool(test)'` as expected.

```
if c:
    return True
else:
    return False
```

However, in the following code segment, Pylint does not report the `simplifiable-if-statement` defect at all.

```
if c:
    return False
else:
    return True
```

Also, since Pylint targets more experienced programmers, several detectors have a high number of false positives, counting on the programmer to be able to evaluate whether it is relevant or not.

All this makes Pylint unsuitable to be used by novice programmers out of the box. At the same time, Pylint can form a basis for another tool that would provide an interface more suited to novice programmers and their educators.

4.1.2 Pyflakes

Pyflakes [22] is a simple Python error checker. It is very conservative, trying to avoid false positives as much as possible. While it detects many defects that Pylint also does, it prides itself on being more efficient.

Pyflakes is not documented by itself, apart from how to use it, and is not extensible. It is meant to be used as a part of Flake8 (more in Section 4.1.4).

4.1.3 Pycodestyle

Pycodestyle [23] is a tool that simply reports PEP8 violations, by default only those the creators deemed "unanimously accepted".

black, autoflake and isort are also tools focused on formatting, but they automatically change the code to match some recommendations from PEP8.

4.1.4 Flake8

Flake8 [24] is a tool that combines Pyflakes and Pycodestyle. It also wraps mccabe, a tool to measure McCabe's cyclomatic complexity and report if it exceeds a preset threshold. Therefore, it focuses prominently on *formatting* and *erroneous* defects.

Like Pylint, Flake8 is highly configurable, allowing each defect to be enabled or disabled individually. Flake8 is also extensible by a long list of available plugins. These plugins are independent pip packages that, once installed, automatically run whenever Flake8 is run (unless disabled).

4.1.5 Wemake python styleguide (WPS)

WPS [25] is one of the most extensive Flake8 plugins (at least by the number of detected defects). It presents itself as "the strictest and most opinionated Python linter ever". It aims to provide feedback to all kinds of defects, most notably to reduce complexity (detecting a significant number of defects in the *long or overly complex* category), enforcing consistency by detecting *formatting* and *poor name* defects and suggesting improvements to *unsuited constructs*, and helping to avoid bugs by pointing out *erroneous* and *error prone* defects.

It is available in two flavours: the full `wemake-python-styleguide`, depending on Flake8 and several of its plugins, and `wps-light`, which only depends on Flake8, without the extra dependencies.

Though it detects a high number of defects, many of these can be controversial (forbidding the use of f-strings and explicit string concatenation in favour of the `format` method, forbidding static methods in favour of class methods or functions and more) or focusing on details like forbidding uppercase string modifiers. Also, almost all of the defect messages simply state that they found a forbidden construct without indicating how to fix it. This information can only be found in the documentation.

4.1.6 Ruff

Ruff [26] is a fast reimplementation of Flake8 and some of its plugins and several other tools, though it does not report those code style defects that an autoformatter can fix. It provides autofix support.

The development of Ruff started after I started developing the tool for this thesis. As Ruff is installable through pip and integrates well with Python projects, it might be worth considering transitioning to it from the currently used Flake8.

4.2 Integrated development environments (IDEs)

Programmers often get assistance directly through their IDEs. They can often obtain additional aid by installing a plugin.

This section describes some solutions to providing code quality feedback directly through the IDEs.

4.2.1 Thonny

Thonny [27] is an IDE developed specifically for novice programmers. It has a simple GUI so as not to overwhelm the novices. It provides a debugger which allows for stepping through expression evaluation rather than debugging the code line by line. It also assists with finding syntax errors and several more features tailored to the needs of novice programmers.

Thonny also wraps Pylint and mypy and employs custom dynamic analysis to warn its users of possible issues in their code. The warning may contain a description of what is happening or questions which may help the user to fix the problem. It also rewords some of the Pylint's messages to make them easier to understand.

4.2.2 Industry-grade IDEs

In this section, I discuss the capabilities of IDEs used in industry practice, like PyCharm, Visual Studio Code and similar.

Many IDEs provide some hints or feedback on code quality and possible errors. This advice is tailored to the needs of professional programmers, advising advanced structures or libraries that might not be available or allowed to use by novices.

So while creating an IDE plugin is a convenient way to deliver feedback to a programmer, current industry-grade IDEs often do not provide feedback suited to novice programmers.

4.3 Educational linters

This section details linters developed for providing code quality feedback and evaluation specifically to novice programmers.

4.3.1 PyTA

PyTA, short for Python Teaching Assistant, is a set of tools presented by D. Liu et al. [16]. They developed it primarily to assist novice programmers with finding and fixing common bugs (so many detected defects fall into the *erroneous* category). Nevertheless, it also checks for *formatting* defects and some others.

It was developed for CS1 courses at the University of Toronto. It wraps around Pylint and Pycodestyle (described in sections 4.1.1 and 4.1.3, respectively). It extends Pylint by several custom defect detectors.

D. Liu et al. [16] have shown that making the tool available to their students reduced the number of attempts needed to solve an assignment, arguing that static analysis may be useful in developing tools for novice programmers.

PyTA is distributed to students as a pip package. The students import functions from the package and use them to get feedback on their code. The tool can also be used to check pre- and postconditions of a function at runtime or to assist with debugging a loop by providing debug prints for each iteration.

By default, students receive the feedback in the form of an HTML page that contains all detected defects, their severity, description with possible hints on how to fix the defect and, if relevant, excerpt from their code with relevant passages highlighted or with a hint of a solution. The tool also changes some defect messages to make them more understandable to a novice programmer.

Using the HTML page, the student can be presented with much more information than just through CLI, allowing them to understand the defect better and making it easier to fix.

It is possible to configure the tool by configuring the wrapped Pylint directly, using its `.pylintrc` file. It is also possible to alter the wording of Pylint's messages through a dedicated configuration file.

The tool expects particular workflow (e.g. it suggests fixing errors "before submission") or specific course rules (e.g. there is a notion of "forbidden usage" or messages like "Global variables must be constants in CSC108/CSC148"). Upon encountering an error, the user is asked to "report this to [their] instructor". This makes the tool less usable out of the box. (The text in quotation marks is taken directly from the tool's output.)

4.3.2 Hyperstyle

Hyperstyle is a tool for code quality evaluation, presented by Birillo et al. [4]. It allows linting code in five different languages (Python, Java, Kotlin, JavaScript, Go) [28] by employing real-world linters and adapting them to the educational process.

The tool was integrated into JetBrains Academy and Stepik learning platforms to provide code quality feedback to their users.

To lint Python specifically, it wraps Pylint and Flake8 with the `wps-light` plugin and several others (described in sections 4.1.1, 4.1.4 and 4.1.5 respectively) and radon, a tool for measuring various complexity metrics. It does not alter any of the messages, provides no extra detectors, and provides no explanations or examples of the defects.

Apart from producing reports of individual defects, it grades the code quality with one of four levels (EXCELLENT, GOOD, MODERATE, BAD).

The configuration files of the specific linters are the only way to configure which detectors the tool runs.

Away from the platforms that use it, Hyperstyle is available in a public docker image.

4.3.3 Others

This section briefly describes several other solutions focused on Java code.

AutoStyle [13] is a tool for providing feedback on Python code (among others). It works by clustering solutions to a given task (works only with solutions containing a single function) and having a teaching assistant provide feedback for each cluster rather than each student. Together with the TA, the system gives incremental improvement tips to the student based on similar but slightly better solutions. This requires a human in the loop and bootstrapping the system with existing solutions to each task.

FrenchPress [11] is an Eclipse plugin that gives students automated feedback on Java defects related to object-oriented programming.

WebTA [12] is a system that gives automated feedback on Java defects, though there is an effort to extend it to other languages. It allows educators to implement their own detectors.

Refactor Tutor [14] is a system that provides several examples of low-quality Java code and guides the student to improve the code using hints. I found no study on whether this helped them improve their code.

5 EduLint

To fulfil the requirements set in Section 2.3, I developed a tool I call EduLint. It wraps around Pylint and Flake8 to use their current functionality and extensibility through plugins. Some of their messages are altered to be more understandable to novice programmers. I also implemented several custom plugins to extend the list of detectable defects. EduLint exposes the command line configuration options of the underlying linters to benefit from their configurability.

EduLint is available as a pip package. See Appendix A to learn how to install and run it. Since I argued that installing and running a package can be troublesome for novice programmers (Section 2.3.5), I also developed a web page serving as an interface, which I describe in the last section.

In this chapter, I comment on why I chose the technologies I did. Then I describe the tool's architecture and the process a piece of code goes through to be linted. After that, I detail how to configure the tool and its options. I mention the reasons for developing custom detectors. Then, I outline the purpose of the three postprocessing steps: filtering, rewording and overriding. In the second-to-last section, I describe how explanations for the defects are obtained and used. I describe the web interface in the last section.

It is important to note that while EduLint aims to provide relevant feedback to many kinds of novice programmers, the tool was calibrated for IB111, a CS1 course taught at the Faculty of Informatics, Masaryk University: I used codes from the students of the course to create the default configuration of detected defects, I also used them for testing the tool's precision during development, and the custom detectors I developed were partly determined by the mistakes that students of IB111 often make.

5.1 Design decisions

In this section, I list the decisions behind the tool's development and the reasoning behind them.

I decided to lint each file individually rather than use information from other solutions to the same task, as such an approach would rely on the existence of other solutions and would not work well with brand-new tasks.

In several tools developed by other researchers [16, 13, 12], defect detection and their presentation are coupled into a single tool. I developed a pip package independent; it takes a source file and presents a defect report for each defect it contains (either in plaintext or as a JSON for further processing). As a separate project, I created a web interface which receives the JSON output and displays it to the user. This way, the package can potentially be used by other tools that display the output of a linter to the users (and similarly, the website could show results from other tools, provided they are in the expected format).

I decided against implementing autofixing capabilities, encouraging the students to fix the defects themselves and learning not to create them in the first place.

Since the beginning of the development, I intended to avoid reimplementing detectors already available in other tools. I chose to base the tool on Flake8 and Pylint (both described in Section 4.1), as together they cover a wide range of defects, and both come with an ecosystem of plugins, with a possibility to add more.

Currently, EduLint does not use any of the Flake8's plugins, as for each of them, their interaction with already existing detectors would have to be checked so that the users do not receive a report of one defect more than once. Also, as noted in Section 4.1.5, most WPS messages could be more helpful for novice programmers and should be altered, which is also true for some other plugins. Each newly added detector needs to be checked for false positives too. Still, adding them is a part of future work, as discussed in Section 7.2.2.

I decided against using any other tools: I intentionally disregarded any autoformatters and did not use the whole of any educational linters described in Section 4.3. Out of those usable for Python code, only Hyperstyle and PyTA are open-source.

Hyperstyle for Python only wraps Pylint and Flake8 with its plugins, so I decided to use those directly rather than through another tool. That also gives me a direct way to alter their configuration on each run.

PyTA has several disadvantages: it has limited usage in a general context, as argued in 4.3.1. It also requires the user to install a package, with no alternative way to run the tool, and should some changes to the default configuration be made, the user would have to place the configuration files in the correct locations by themselves, which is deemed unsuitable by the requirements set in Section 2.3. I also decided against wrapping the tool directly, as significant parts do not relate to code quality feedback (runtime contract checking, debugging assistance). At the same time, the tool is open-source, and its licence allowed me to adopt some of their detectors (top-level code, a `for` using an attribute or an indexed array as a control variable: `for lst[i] in range(n)`).

I considered integrating mypy, a static type checker, but decided against it to keep the initial version simpler to develop. Adding it should take no more than a few hours.

I chose to develop my custom detectors as Pylint plugins as those consist only of a Python module, while Flake8 plugins require their own package and all the handling that comes with it.

5.2 Architecture and linting process

In this section, I describe the tool's architecture on a high level and outline the phases of linting a file. The following sections describe the phases in greater detail.

I based the architecture of the tool on the pipe-and-filter architectural pattern. The pipeline can be seen in Figure 5.1.

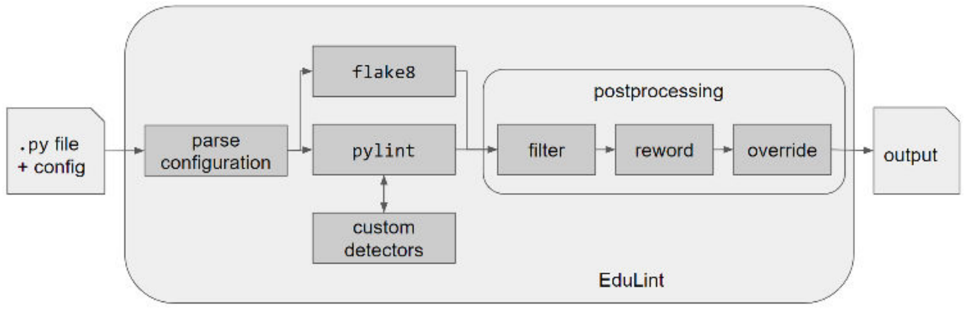


Figure 5.1: An overview of the linting pipeline

The linting process inputs one or more files and configuration options passed through the command line interface. Additionally, each of these files may contain individual configuration in comments.

parse configuration The configuration is extracted from the files in the first step and combined with the configuration from the command line. Some configuration options may be translated to options for Flake8 and Pylint; others affect EduLint directly.

Flake8, Pylint; own detectors In the second step, the two used linters are run with the configuration extracted in the previous step. Pylint is run with the custom plugins I created. Each linter produces output in a slightly different format, so it gets converted into a unified structure. The linters run independently of each other but sequentially rather than in parallel (though this improvement would be easy to implement).

filter While some detectors are completely disabled in the configuration, only some emitted messages are let through for others.

reword At this step, the text of a message may be changed to be more comprehensible to a novice.

override For some combinations of messages, it makes sense to display only one, for example, when a programmer might fix both in fixing one. Only the one that would solve the other is kept in this step. For an example, see Section 5.5.

In the end, the resulting defects are passed to the standard output as plain text or JSON.

Presently, the filter and reword steps are used only lightly but can be extended easily.

5.3 Configuration

For any given set of defects to warn novice programmers about, there would be discord over whether some should be included. For that reason, I have built EduLint so that every message can be enabled or disabled. At the same time, through discussion with other educators, I created what I believe to be a reasonable default set of defects for college students and others seriously interested in learning programming. The default configuration can be extended in several ways.

In this section, I first describe how an educator can configure EduLint. Then I describe the reasoning behind the default configuration and its extensions. After that, I list other available options and conclude with a description of the documentation and how it is generated.

5.3.1 Configuration means

Currently, there are two ways to configure EduLint: through its command line interface or by including a comment in a specific format directly in the linted code.

The intention behind exposing the configuration option through the command line is that an educator could lint a set of files without altering them first.

The intention behind setting configuration in the files directly is to allow for easy configuration sharing with students. The configuration can be a part of the starter code a student receives alongside the assignment. This way, novice programmers do not have to think about the configuration; they simply lint a file whose starter code they received. As long as they do not delete the line with the configuration, they receive the feedback their instructor intended for them.

Distributing the configuration as a part of the file also naturally allows for different tasks having different configurations.

5.3.2 Default and extended defect groups

The default configuration is used when a file is linted without additional options. It contains a list of manually selected defects that would be problematic in any programming language.

Flake8 is used as-is since PEP8 is a set of conventions mostly accepted among Python programmers, and it is useful for novice programmers to learn to adhere to it. It is already required to pass cleanly through its detectors to get points for any homework in IB111.

For Pylint, I created a curated list of relevant defects, selected out of all defects Pylint can detect. To prepare the list, I first used Pylint to check all submissions from IB111 in 2020 and 2021 to see which ones even appeared. I checked the defect occurrences to determine the number of false positives and disregarded the detectors that generated too many. I also scanned the defects with no occurrences to see if any might be relevant, even if they are rare.

This way, I comprised a preselection of defects. Then I went through this shorter list, and for each defect, I decided if it should always be reported or if it makes sense to report it in a group of similar defects. As a last step, I consulted the resulting list with two other educators and adjusted it accordingly.

The list of defects that EduLint should always report became the default configuration for Pylint.

Three other groups of defects emerged during the process: defects whose removal can improve the code but are not necessarily bad, defects specific to Python, and defects related to code complexity. Each of these groups got a configuration option that can enable all defects in the group at once. The goal of these extensions is to allow for a straightforward augmentation of the default configuration, should a student themself seek improvement or should an instructor have stricter requirements of their students than default EduLint.

This result comes with several caveats, which I discuss in Section 7.2.1.

5.3.3 Other configuration options

EduLint also allows for passing configuration options directly to the underlying Pylint and Flake8 or disabling a whole tool. There is an option to specify which single-character variable names are to be allowed: if it is set, all others will trigger a defect (see C.1.5 for more details). The last option is specific to the needs of IB111.

5.3.4 Documentation

All configuration options, as well as all custom detectors, are documented[1]. Wherever possible, the documentation is automatically generated directly from EduLint's source code and configuration files to ensure it is up to date.

1. `https://edulint.readthedocs.io/en/v2.6.4/`

The documentation lists which defects are detected in the default configuration and which are detected in some of its extensions.

Each custom defect has a template of its message together with a short description of when it is emitted.

5.4 Custom detectors

For the tool, I implemented 33 custom detectors. Seventeen of these detect defects that are, to the best of my knowledge, not detected by any other available tool. Compared to detectors in other tools, the remaining sixteen detectors are either more precise (the global variable detector does not emit false positives if the code contains type aliases), broader in scope (for example, "redundant arithmetic" also detects adding an empty string or dividing a value by itself), or detect a more specific situation to provide actionable feedback (`while True` is only detected if the first statement of its body is an `if` ending with a `break` – in which situation the `if`'s negated condition can be used as the condition of the loop).

Some notable examples of these defects can be found in Table 3.1. See Table C.1 for the complete list. It includes defects like using a `while` loop instead of a `for` loop, float division instead of integer division, and a negated condition instead of an `else` branch.

I had several sources of inspiration for the new custom detectors: the lists of defects presented in Section 2.3.1, most notably the list by Effenberger et al. [17], the list of defects to point out in code review that novice teaching assistants in IB111 receive, consultation with other educators at FI MUNI and my own experience with novice code.

I list the detectors I implemented in Appendix C. For each defect, I describe the defect, often with code samples, clarify why I decided to detect it and explain how it is detected. Section C.2 shows frequencies of these defects in novice code (together with a sample code exhibiting the defect and information on whether it is a newly detected defect or an improved detector).

5.5 Postprocessing

This section explains the purpose and the workings of the postprocessing steps of the linting process.

The first two steps, filtering and rewording, use a manually prepared regular expression that matches the given message emitted by Pylint. Thus parsed, it can be filtered based on specific parts of the message or reworded, placing relevant parts (like code excerpts) into the new wording.

Currently, EduLint uses these two steps just lightly: the filter helps to go around a feature of the used Pylint version, where Pylint reports any variable name shorter than three characters, saying "[variable name] doesn't conform to snake_case naming style", even if it does, which can be misleading for any programmer. The rewording capabilities are used very little. The understandability of the messages currently relies more on explanations, as described in Section 5.6, as they can provide more information.

The overriding possibility exists because sometimes Pylint emits two messages that refer to two defects on one line, each with a different fix, or fixing one can also fix the other, but not vice versa. An example of this can be found in the following code:

```
if c:
    return True
else:
    return False
```

There is `else` after the first `return`, but the `if` statement can be replaced by `return c`.

Getting two defect reports with conflicting advice can confuse novice programmers, who may not know which action to take. To avoid this issue, it is possible to specify which messages can be suppressed by others if they relate to the same line.

5.6 Explanations

Novice programmers may struggle with understanding or fixing a defect, or they may not see a reason to fix it. To address this, Edulint provides an explanation for each defect. The explanation clarifies the unwanted effect of the used construction and why it should be altered. The explanation also contains examples of the defective code and its corrected variant.

Let us consider a defect in which multiple comparisons are used instead of testing for the presence of an element in a container. An example of how to fix this could be:

Comparison of a variable to two values can be simplified using the `in` operator. This is more readable and also safer against copy-paste errors.

```python
def problematic(text: str):
    if text == 'a' or text == 'b':
        return

def good(text: str):
    if text in ('a', 'b'):
        return
```

If you are comparing a single character, you can also do this:

```python
def good(char: str):
    if char in 'ab':
        return
```

This concludes the example of how to fix the defect. An explanation for why a student should fix the defect could be that shorter conditions are usually easier to read.

Currently, most of these explanations are autogenerated from data from various sources: Pylint's documentation, which contains longer descriptions of the defects and code examples, and Thonny's source code, which also builds on Pylint's message descriptions with some slight alterations.

Any automatically generated explanation can be overridden by a manually written one. At present, manual explanations are written mainly for the custom defects.

The goal for the explanation is for each defect to have its own manually written explanation tailored specifically to novice programmers.

The explanations are a part of the EduLint package though currently, they are not accessible through the command line interface and can only be viewed through the website I describe in Section 5.7.

This is because the package is primarily intended to be used by educators to check on frequent defects in a set of submissions. Novice programmers are expected to use the website.

5.7 Website

In this section, I describe the website developed to serve as a web interface for EduLint. In the first subsection, I describe the design decisions that affected the development of the website. In the second subsection, I describe the application programming interface (API) I provide and the web interface.

A public instance of the website is available[2].

5.7.1 Design decisions

I present EduLint to novice programmers through a website. This way, it is available to all those capable of using the internet, which should cover all novice programmers. The website also does not require the installation of any new software. The drawback is that it requires the novice programmer to leave the environment they use to get the code quality feedback. I plan to mitigate this by developing a Thonny plugin (see 7.2.4 for more information.

I did also consider developing a plugin for an IDE. However, I decided against it as it would have higher maintenance cost and as (novice) programmers use different IDEs, there would be an incentive to develop multiple plugins, which would up the development time since the original intention was to make the tool available to as many users as possible.

I built the backend using `flask` as it is a lightweight web framework, and I only developed a small site.

For similar reasons, I created the front end without any framework in native JavaScript. While this was enough to develop the basic version of the site, it proved insufficient for any more modifications, and I am rewriting the site to use React, though this is not a part of the thesis.

2. `https://edulint.com`

5.7.2 Interfaces

The website provides two separate interfaces: the API for exchanging data and the graphical web interface for displaying them. This section describes them in turn.

API

Most importantly, the programming interface provides endpoints to upload and lint a file. It also provides endpoints to get the explanations and the available EduLint versions.

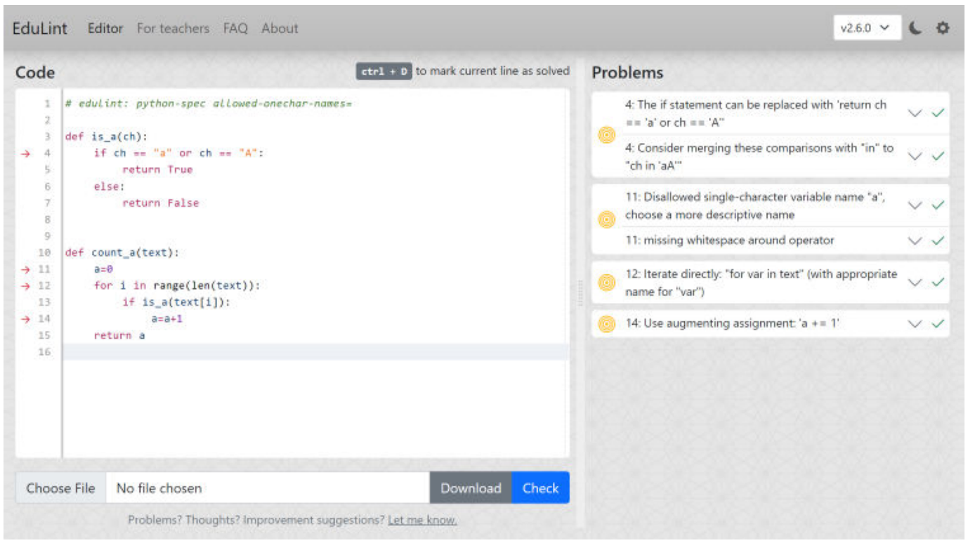
Most available endpoints are documented using Swagger[3].

Web interface

I developed the web interface to be as simple as possible: the main site only allows users to paste or upload the code and check it to show its defects, see Figure 5.2. For each reported defect, it shows its line and message. The user can jump to the line with the problem (this can take the user to the line with the problem even if they edited the file, though the displayed number does not change due to technical complications). The user can also mark individual defects as solved (the front end cannot determine this on its own, as the linter is written in Python, and while it is possible to run it in-browser, it has not been a priority). The user can also display the explanation for the defect (as described in Section 5.6). A defect with an explanation can be seen in Figure 5.3.

The website also provides a quick overview of where to find more information aimed at educators.

3. `https://edulint.com/api/`


EduLint
Editor
For teachers
FAQ
About
v2.6.0
Code
ctrl + D to mark current line as solved
Problems
Choose File
No file chosen
Download
Check
Problems? Thoughts? Improvement suggestions? Let me know.

```
1   # edulint: python-spec allowed-onechar-names=
2
3   def is_a(ch):
4       if ch == "a" or ch == "A":
5           return True
6       else:
7           return False
8
9
10  def count_a(text):
11      a=0
12      for i in range(len(text)):
13          if is_a(text[i]):
14              a=a+1
15      return a
16
```

4: The if statement can be replaced with 'return ch == 'a' or ch == 'A''

4: Consider merging these comparisons with "in" to "ch in 'aA'"

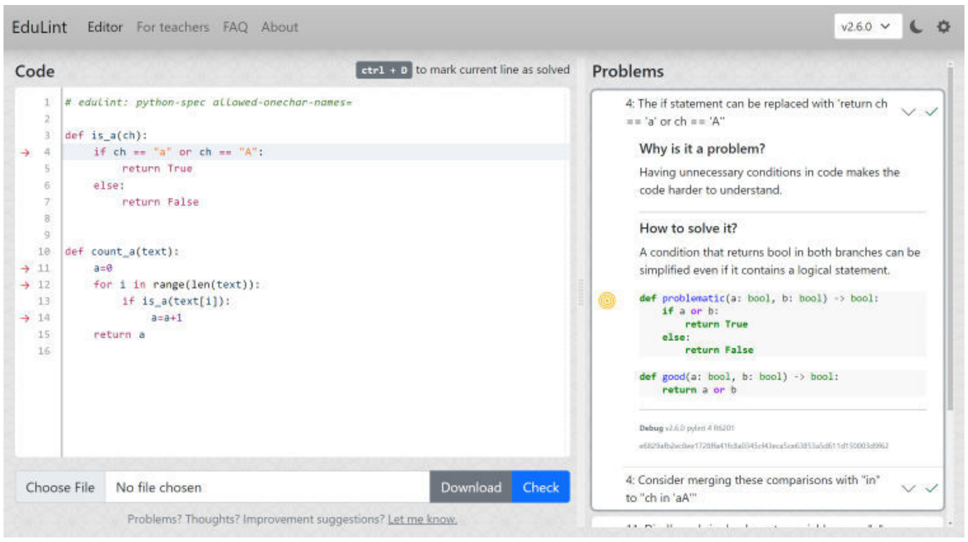
11: Disallowed single-character variable name "a", choose a more descriptive name

11: missing whitespace around operator

12: Iterate directly: "for var in text" (with appropriate name for "var")

14: Use augmenting assignment: 'a += 1'

Figure 5.2: EduLint web interface


EduLint
Editor
For teachers
FAQ
About
v2.6.0
Code
ctrl + D to mark current line as solved
Problems
Choose File
No file chosen
Download
Check
Problems? Thoughts? Improvement suggestions? Let me know.

```
1   # edulint: python-spec allowed-onechar-names=
2
3   def is_a(ch):
4       if ch == "a" or ch == "A":
5           return True
6       else:
7           return False
8
9
10  def count_a(text):
11      a=0
12      for i in range(len(text)):
13          if is_a(text[i]):
14              a=a+1
15      return a
16
```

4: The if statement can be replaced with 'return ch == 'a' or ch == 'A''

Why is it a problem?

Having unnecessary conditions in code makes the code harder to understand.

How to solve it?

A condition that returns bool in both branches can be simplified even if it contains a logical statement.

```
def problematic(a: bool, b: bool) -> bool:
    if a or b:
        return True
    else:
        return False
```

```
def good(a: bool, b: bool) -> bool:
    return a or b
```

Debug v2.6.0 pylint 4 R6201

[illegible]

4: Consider merging these comparisons with "in" to "ch in 'aA'"

Figure 5.3: EduLint web interface showing an explanation

6 Comparison with other tools

In this chapter, I compare EduLint to Pylint, PyTA and Hyperstyle. This comparison aims to argue that the feedback EduLint provides is relevant to novice programmers more often than feedback provided by other tools and that no tool other than EduLint fulfils the requirements set in Section 2.3

In the first section, I introduce the dataset of defect reports I used for the evaluation: where from were the original novice codes, how I obtained the reports and how I cleaned them. Then, I compare the tools, first by frequently reported defects, to provide insight into which defects reports the students would receive most often had they used the given tool. Then, I compare the tools by frequently reported defect categories to determine which categories the tools focus on and provide a more comprehensive overview of the tools' capabilities. Lastly, I compare the tools by the requirements set in Section 2.3.

6.1 Experiments setup

I used a dataset of more than a hundred thousand student submissions for the experiments. From it, I obtained a dataset of 3.2 million defect reports.

I first describe the three sources of novice code I used, then I describe how I categorized the defects the tools report, and lastly, I detail how I obtained the defect reports and how I cleaned the data for further analysis.

6.1.1 Code datasets

I used three sources of Python code for the experiments: submissions from the last three years of IB111 (the course is held one semester a year), the last eight years of the Correspondence Seminar of Informatics (KSI) organized by students of FI MUNI and submissions from users of Stepik and JetBrains Academy education platforms, made available by Birillo et al. [29]. I refer to this last dataset as the Hyperstyle dataset because it was used to evaluate this tool introduced in the paper. I introduce each of these datasets in more detail in turn.

These datasets contain mostly correct single-file solutions to various tasks written by novice programmers.

In the experiments, I used them all joined without filtering or balancing.

Neither the IB111 nor the KSI dataset can be publicly disclosed, as they contain solutions to tasks used repeatedly each year. The solutions are also the students' intellectual property and are available only for research purposes within FI MUNI. The Hyperstyle dataset is freely available [29].

IB111 dataset

This dataset contains 80,049 mostly correct solutions from about 1,500 students to various programming tasks on introductory programming topics: use of control structures (conditions, loops, functions), basic algorithms (sorting, binary search), basic data structures (numbers, lists, strings, sets, dictionaries, classes as structures), recursion.

The dataset contains several completely incorrect solutions, as though students have some tests available, they sometimes submit a solution that does not pass them.

In the course, the code must pass cleanly through Flake8 checks to earn any points for the task.

The students usually receive a starter code that contains the assignment text, headers of functions they are to implement, and several tests for their implementation (asserts inside the `main()` function).

As the other datasets are significantly smaller, this dataset influences the experimental results the most.

KSI dataset

This dataset contains 19,256 submissions made by about 2,700 users in the past eight years of the seminar. The seminar covers a broad range of topics, from doubly linked lists and deterministic finite automata to data analysis and web servers.

The seminar's participants are not required to adhere to any formatting rules or conventions.

The participants often receive a starter code containing the headers of the functions they are to implement and several tests (prints at the top level of the code).

Hyperstyle dataset

The dataset contains 24,250 Python submissions by 300 users of Stepik and JetBrains Academy education platforms. I only used 9,842 submissions by 150 users, as the rest were codes collected after they introduced Hyperstyle to the users.

The nature of the tasks and the environment in which the submissions were collected are not discussed. By observation, the tasks range from computing the circumference of a circle whose radius they receive from the standard input to implementing functions for Huffman encoding and decoding.

It also seems the users were not required to adhere to any formatting rules or other conventions.

The users seem required to write a utility that inputs information from the standard input, a function or a class. The code sometimes seems to be written to be inserted into another piece of code (for example, it uses undefined variables).

Dataset size overview

Table 6.1 shows an overview of the dataset sizes and lengths of the codes they contain. To lower the differences between the lengths of the codes (due to the presence of tests and assignment text directly inside the submitted file), I removed empty lines, lines that only contain a comment (start with `#`), for the IB111 dataset any code that is inside the `main` function or after it (tests) and for the KSI dataset any print statements at the top level (also tests). I did this only to show the lengths of the codes, not during the analyses (as it would be challenging to remove only the irrelevant parts, leaving the code in working order).

Table 6.1: Dataset sizes and their code lengths

		code lines percentile		
	files	**25%**	**50%**	**75%**
IB111	80,049	16	24	38
KSI	19,256	9	18	31
Hyperstyle	9,842	3	6	10

6.1.2 Defects dataset

I also comprised a list of all defects the tools are capable of detecting and labelled them with the categories from Chapter 3. I used three extra categories for the labelling: *internal* for reports of a tool's error or warning (e.g. error while parsing the code, reports that a file was ignored), *deprecated* for warnings about Python features novices are not likely to encounter (e.g. which are emitted only for some pre-Python 2.5 syntax) and *advanced*, to differentiate defects regarding more advanced topics (the defects in this category could still be categorized in terms of the original categories from Chapter 3).

To determine which topics are advanced, I examined whether they are being taught or required in IB111 (if not, it is advanced). Some topics or constructs that ended up *advanced* are docstrings, exceptions, lambda functions, star expressions, generator functions, decorators, class and static methods, double underscore methods, inheritance, access levels, abstract classes, metaclasses, properties, slots, threading, async/await etc.

6.1.3 Running the tools

I ran the experiments using Python 3.8.10, Pylint 2.14.5, Flake8 3.9.0, and EduLint 2.6.4. The used versions of Flake8 plugins can be found in Hyperstyle's requirements[1]

I configured each tool to report all defects it is capable of detecting. To obtain the defects any one tool would provide, I filtered all the reports for those the tool detects in its default configuration.

1. `https://github.com/hyperskill/hyperstyle/blob/0a1eafb51ed20d5e92b65656ca25aae6aa4dc494/requirements.txt`

I did not manage to employ several PyTA plugins (detectors for undefined variables, redundant assignment and type error), as they required advanced static analysis that I was not able to run. I also did not use Flake8's cohesion plugin, as it emitted errors I could not fix. For several other detectors (for Pylint, Flake8 and PyTA), I had to fix several minor bugs (e.g. the analysis terminated with an error when the analyzed code used an undefined variable as a default argument value).

6.1.4 Cleaning the linting data

Overview of the cleaning process with defect counts after each phase can be found in Figure 6.1.

I received 11 million defect reports of 505 different defects on the dataset of just over a hundred thousand files.

More than half of these reports were relics of how I ran the analysis, so I removed them. These were defects regarding spelling errors in comments (I did not install a Czech dictionary for the plugin), incorrect module names and names that are too long (I named the modules containing the novice codes for convenience, not to adhere to PEP8 requirements or length limitations), unable to import, no such name in module (some submissions relied on a file being present in the same folder or on some library being installed), and Pylint's duplicate code (the detector detects only similarities between different files; all the submissions were stand-alone, so it detected similarities, like the same set of tests, between different submissions). After filtering out these, I had just over 5 million defect reports.

Furthermore, I decided to remove several other defects, as most of their occurrences were caused by the starter code already (e.g. PyTA reports the use of input/output functions as a defect, but the KSI dataset often provides tests as top-level prints) or their occurrence was an unavoidable part of the solution (some tasks in the Hyperstyle dataset require using `print` and `input`). I list the defects I ignored in Table 6.2 and why I chose to ignore them. This left a dataset of around 3.9 million defect reports.

I also aggregated several defects from Pylint and PyTA into one: Pylint's missing module docstring, missing function or method docstring and missing class docstring became one defect; similarly for

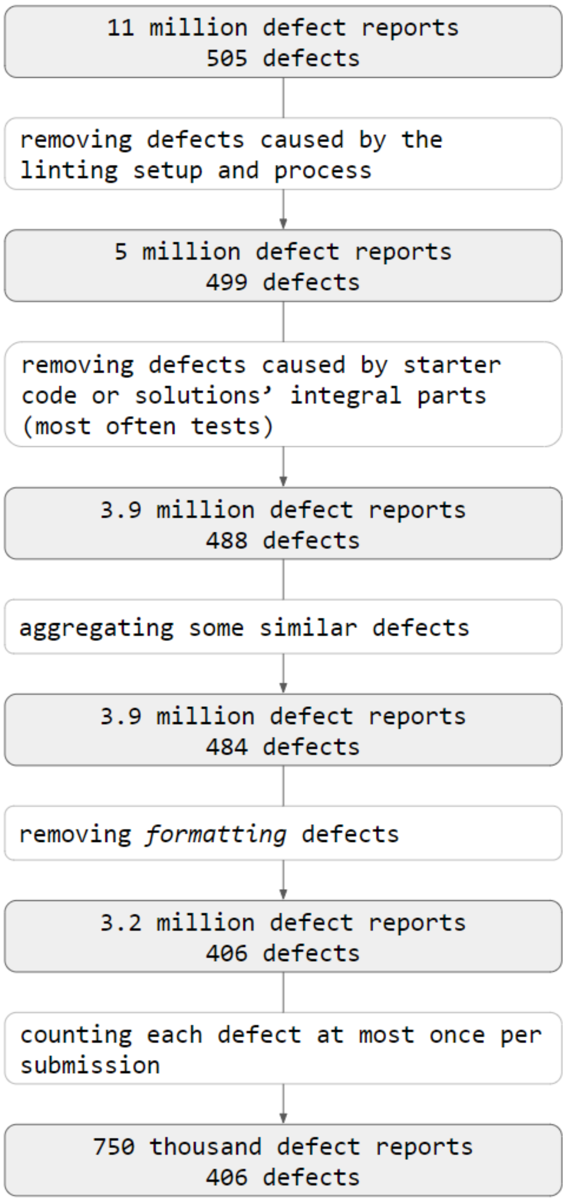

11 million defect reports
505 defects
removing defects caused by the linting setup and process
5 million defect reports
499 defects
removing defects caused by starter code or solutions' integral parts (most often tests)
3.9 million defect reports
488 defects
aggregating some similar defects
3.9 million defect reports
484 defects
removing *formatting* defects
3.2 million defect reports
406 defects
counting each defect at most once per submission
750 thousand defect reports
406 defects

Figure 6.1: Defect report cleaning process

Table 6.2: Ignored defects

Defect	Additional information	Reason to ignore
Pylint		
too few public methods	by default less than 2	several IB111 assignments use classes as structures, so they only have the initializer method
some unused imports	`from ib111 import week_01`, which is not used later	IB111 uses importing a special unused value from a module to limit constructs which students have available
PyTA		
use of global variables	also flags type aliases	several Hyperstyle tasks do not require a function, but a short script working through standard input/output
top-level code	flags any top-level code that is not an assignment to a constant; also flags type aliases	KSI dataset has test prints on top-level
use of input/output function	`input`, `open`, `print`	KSI and Hyperstyle datasets often endorse or even require the use of these
forbidden imports	any import is flagged as forbidden unless configured otherwise	solutions often use imports, most often for `typing`
Hyperstyle		
found magic number	any number that is not between -10 and 10 (and several others)	IB111 uses "magic constants" in tests
found string constant overused	any constant string that is used more than three times	IB111 uses constant strings in tests
too many asserts		IB111 uses asserts in tests, sometimes several of them
wrong magic comment	`noqa` or type annotation in a comment	IB111 uses `noqa` to suppress Flake8 errors in non-student code
overused expression	like `len(lst)` or `[1, 2, 5, 66]` used more than four times	IB111 repeats expressions in tests
EduLint		
global variables		same reason as for PyTA's global variables detector
top-level code	flags any top-level code that is not an assignment to a variable	same reason as for PyTA's top-level code detector

PyTA, which otherwise reports missing return, argument and attribute type annotation separately. I did this so that the analysis of the most frequent defects in Section 6.2 would show a broader range of defects.

PyTA, Hyperstyle and EduLint all report *formatting* defects detected by Pycodestyle. Hence, I decided to ignore all *formatting* defects, as they are not particularly interesting for comparison. Also any dataset where PEP8 is not enforced (which KSI and Hyperstyle datasets are) is bound to contain plenty of its violations. Removing them left 3.2 million defect reports.

Throughout the comparisons, I do not use the total occurrence counts of the defects, but I count each defect at most once per file. I do this so that the defects that are more likely to occur multiple times would not so easily overwhelm other defects. (For example, Pylint reports using CRLF line endings. It would be one of its five most frequent defects, even though it appears in only 2% of the files.) The dataset contains 750 thousand of thus counted defect reports of 406 defects.

6.2 Comparison by frequent defects

In this section, I show frequent defects reported by the selected tools and argue their (ir)relevance. I summarize the results at the end of the section.

I aim to show that the defects frequently reported by EduLint are more relevant to novice programmers than defects frequently reported by the other tools.

6.2.1 Finding frequent defects

Table 6.3 shows the five most frequently occurring defects for each of the selected tools.

The table also shows how relevant the defect is to novice programmers for a better overview. The arguments for the results can be found later in the section.

Table 6.3: Frequent defects

Defect	Additional information	Files with the defect	Percent of detected[a]	Relevant
Pylint				
missing docstring		98.7%	40.5%	×
variable name does not conform to naming style	many false positives, see Table 6.7	33.8%	13.8%	∼
compare to a falsey value	code uses `if lst == []` instead of `if not lst` and similar	11.6%	4.8%	×
unused variable	most are control variables of a for loop	9.2%	3.8%	∼
unnecessary `else` after `return`		7.5%	3.1%	∼
PyTA				
missing docstring		98.7%	41.2%	×
missing type annotation		60.4%	25.2%	∼
unused variable	most are control variables of a loop	9.2%	3.8%	∼
line too long	over 100 characters	5.4%	2.3%	∼
redefining built-in function	`sum = 0`	4.2%	1.8%	∼
Hyperstyle				
possibly misspelled name	many false positives, see Table 6.7	52.0%	13.4%	∼
function with too much cognitive complexity	the specific measure is not well documented	23.0%	5.9%	∼
compare to a falsey value	code uses `if lst == []` instead of `if not lst` and similar	15.4%	3.9%	×
block variables overlap	two consecutive or nested for loops use the same control variable name	14.9%	3.8%	∼
explicit string concatenation	code uses `"str: " + val` instead of `"str: {}".format(val)`	12.2%	3.1%	×
EduLint				
use augmented assign	code uses `x = x + 1` instead of `x += 1`	7.6%	13.2%	✓
unnecessary `else` after `return`		7.5%	13.0%	∼
iterate directly instead of using indices	code uses `for i in range(len(lst))` instead of `for val in lst`	5.6%	9.7%	✓
redefining built-in function	`sum = 0`	4.2%	7.3%	∼
simplifiable if statement	code uses `if c: return True else: return False` instead of `return c`	4.1%	7.2%	✓

[a] Shows the number of occurrences of this defect over the total number of all defects detected by the tool (each defect is counted at most once per file).

6.2.2 Relevance reasoning

In this section, I argue the (ir)relevance of defects found in Table 6.3. To support the reasoning, I use the papers presenting a list of defects I introduced in Section 2.3.1.

missing docstring Out of the considered papers, Groeneveld et al. [19] and Keuning et al. [18] mention docstrings, but not as an item on their list of defects, but as a defect they *did not* include. Other papers also mention docstrings, though: Börstler et al. [3] explicitly question the relevance of docstrings for novice programmers. Edwards et al. [30] also mention docstrings, but only as a defect that frequently occurs in novice code, without any arguments for their relevance.

I argue that for a programmer who struggles to write their first function, it is not all that important if they document it as well. Also, since novice programmers are often tasked with writing a single function, any documentation would probably be just a compression of the assignment.

variable name does not conform to naming style As in the previous case, Groeneveld et al. [19] did not include this defect. However, complying with standard conventions, like Python's naming conventions, helps make the code readable to other programmers. All of the considered tools enforce keeping to naming conventions in their default configuration. Nevertheless, Pylint combines this check with discouraging the use of short variable names, which leads to many seeming false positives and confusing defect messages.

Pylint's behaviour does not change in the latest version (2.17.3) nor the upcoming Pylint 3.0.0. The behaviour can be at least reconfigured, but primarily through configuration files.

compare to a falsey value None of the papers mentions this defect. I argue that using `not lst` to test for emptiness can be confusing to novice programmers, especially if they simultaneously learn a language with a stronger type system. While getting used to this idiom is useful for programmers who intend to focus on Python, it is less relevant to novices who learn programming in general since many other languages do not support this construct.

unused variable Most of the papers mention unused variable as a defect [17, 7, 19, 15], although Keuning et al. [18] decided to disregard all defects related to unused code, though they do not state why. However, most of the cases reported by this detector are unused control variables of a loop. The expected solution is to start the variable name with an underscore. Novice programmers are usually unfamiliar with this convention, so any feedback should contain this advice (these report messages do not).

Still, it is debatable whether novice programmers should be made to follow this convention. Since underscore is not a usual variable name, they might be tempted to treat the loop differently from others, or they might start using the variable named underscore once they realize they need it, which can only lead to code that is even more confusing than if the original variable just stayed unused. (Effenberger et al. [17] specifically disregard cases when the unused variable is a loop's control variable.)

unnecessary `else` after `return` None of the papers mentions this defect. In some cases, fixing it is only a minor change. However, if the `else`'s body is a long block of code, this leads to a lower code indentation, which is usually easier to read.

missing type annotation None of the papers mentions this defect, but several of them deal with Java code, where they are compulsory.

Using type annotations and type checking helps to avoid many errors. However, type systems may not be obvious to a novice programmer and require a proper explanation which a linting tool is probably not created to provide. While encouraging novice programmers to use type annotations and type checking is definitely of relevance, it is debatable whether a tool should enforce type annotations in a general context.

line too long Out of the considered papers, only Effenberger et al. [17] list too long line as a defect, though Edwards et al. [30] also mention it as a defect that occurs frequently.

Overly long lines are problematic for two reasons: they can be a symptom of behaviour that is too complex, which is definitely problematic, and they also pose the practical challenge of fitting the code on a computer screen. Even though the defect can be solved by breaking the line up, this may not address the over-complication. A report of this defect provides no actionable advice on solving the underlying issue. Attempts at fixing it can lead to a line broken up illogically and thus even harder to read.

redefining built-in function Only Effenberger et al. [17] advocate against shadowing built-in functions. However, this defect cannot occur in Java, as it does not have functions as such.

Shadowing built-ins can hinder readability and introduce bugs. It is uncertain whether even functions that novice programmers do not usually come into contact with but which block useful variable names (like `next` or `id`) should be reported. At the same time, some IDEs highlight built-in functions, so a novice could be confused about why are some local variables highlighted differently from others.

possibly misspelled name None of the considered papers mentions this defect. A misspelled name is more of a stylistic issue since most IDEs highlight undefined variables or methods.

function with too much cognitive complexity Only X. Liu et al. [15] mention cognitive complexity specifically, but Keuning et al. [18] and Groeneveld et al. [19] list cyclomatic and NPath complexity. Other research groups also suggest some complexity metric as a proxy for measuring code quality [31, 32]. The popularity of these measures is possibly partly because they are easy to compute.

The defect is similar to "line too long": it is a symptom of poorly readable code but does not provide any actionable advice on improving it.

block variables overlap None of the papers mentions this defect. Using the same name for the control variable of two nested `for` loops can be problematic, as using the variable would probably lead to hard-to-fix bugs for the novice programmer. Nevertheless, the case this detector reports most frequently are two consecutive loops that use the same control variable name, which can be a valid decision. The detector is, therefore, not very precise.

explicit string concatenation None of the papers mentions this as a defect. The reports come from WPS, whose documentation [33] states that `.format` is preferred over explicit concatenation, `%` operator and f-strings for consistency reasons. However, consistency may not be the main concern for novice programmers. Also, the use of `+` to join two strings is far easier to grasp than the `format` method.

use augmented assign Effenberger et al. [17] and Keuning et al. [6, 14] mention endorsing the use of augmented assign.

The augmented statement is shorter and less error-prone, as the programmer cannot write the name of a wrong variable. (In Python, `+` and `+=` do not have the same semantics on all types; EduLint takes care to suggest the change only if it is sound.)

iterate directly instead of using indices Effenberger et al. [17], Keuning et al., Keuning et al. [6, 14] and Groeneveld et al. [19] all prefer using foreach-style loop over iterating using indices. Using a foreach-syle loop enables naming the container's elements, improving readability and saving the mental effort of evaluating the subscript expression.

simplifiable if statement Effenberger et al. [17], De Ruvo et al. [7], Keuning et al. [18, 14] and Groeneveld et al. [19] all agree that simplifiable if statements should be simplified. Otherwise, the code is unnecessarily complex and cluttered.

6.2.3 Summary

When examining frequently detected defects, EduLint reports more relevant defects than other considered tools. The relevance of the reported defects is supported by several research groups that listed defects in novice code that should be pointed out, listed in Section 2.3.1.

The other tools focus on more advanced topics and specific Python idioms and provide feedback that can be confusing without much more explanation or is not actionable. Some of the frequent defect reports are also riddled with false positives.

Also, only two defects EduLint reports got filtered out during the evaluation process (use of global variables, top-level code), and it did not provide false positives, showing it to be possibly more robust than the other tools.

6.3 Comparison by frequent defect categories

In this section, I show into which categories fall the defects the tools detect, as this provides a more general overview of the tools' scope than the frequent defects discussed in the previous section.

I aim to show that EduLint is more focused on categories *unsuited construct* and *simplifiable* and to argue that these are of particular relevance to novice programmers when regarding code quality.

In this comparison, I also compare the tools to EduLint in its extended configuration, not only the default, to showcase how the extensions alter EduLint's behaviour.

In this section, I first evaluate the relevance of different categories of defects. Then I show which categories of defects the tools are most focused on detecting and which they actually detect in the dataset, after which I summarize the results.

6.3.1 Relevance reasoning

Out of the categories introduced in Chapter 3, I do not discuss *formatting*, as it is not particularly interesting, *duplicate*, as after cleaning the defect reports, none of the tools detects any defect in this category, and *poorly designed*, as again none of the tools reports any defects in this category. I defined the *advanced* category in Section 6.1.2.

I use the same set of papers I used to argue the relevance of frequent defects in Section 6.2.2, presented in Section 2.3.1. However, the often-mentioned defect categories are also heavily influenced by which categories contain defects that are easy to detect (e.g. computing cyclomatic complexity vs finding misleading variable names).

unsuited construct* and *simplifiable These categories can be considered especially relevant to novice programmers' code quality, as addressing them can correct misconceptions or fill gaps in the knowledge of the students. Effenberger et al. [17] list 17 defects falling into these two categories out of the total 32. Out of 15 defects listed by De Ruvo et al. [7], 10 fall into these two categories. About half of the defects listed by Keuning et al. [18, 6, 14] fall into these categories, though the precise number is difficult to determine, as the defects are not described clearly enough. Six out of the seventeen Python-relevant defects given Groeneveld et al. [19] and two out of ten defects by X. Liu et al. [15] are *unsuited construct* or *simplifiable*. However, they also list several defects on the border between *simplifiable* and *unused*.

unused Defects in this category can also be a symptom of an error (e.g. unreachable code), but often they can be just an omission (like an unused import). Their relevance partly depends on the expected level of tidiness. Effenberger et al. [17] mentions five *unused* defects, De Ruvo et al. [7] mention two, Groeneveld et al. [19] give three and X. Liu et al. [15] list five.

erroneous* and *error-prone Reporting defects in these categories can help students discover bugs and fix their misconceptions faster, as was shown by D. Liu et al. [16]. However, they do not directly relate to code readability.

poor name The most interesting class of defects in this category are names that are ill-fitting or outright misleading (e.g., using `i` for something that is not an index, naming a container after what it contains, but using singular count), but none of the tools detects defects in this class (since they rely on the semantics of the code, which is hard to derive automatically, these are challenging to detect in general). Therefore this category effectively focuses on names that are easy to confuse (`l` vs `1` or `length` vs `lenght`) and which break naming conventions. Even though these variations do not address the most fundamental issues in naming variables, they can still improve the code's readability. Effenberger et al. [17] list three defects from this category.

long or overly complex While the defects in this category can be a symptom of *poorly designed* code, which is possibly the most crucial category of defects, the defects themselves do not provide any actionable advice. Beginners' attempts to fix them can lead to confusing line breaking, illogically broken down functionality or overuse of local variables. Such fixes can silence the detector but may lead to code that is of even poorer quality. Therefore, I believe these defects should be treated carefully if reported to novice programmers. Effenberger et al. [17] mention two defects that would fall into this category, Keuning et al. give six in the 2017 paper [18], but the 2019 and 2021 papers do not mention them [6, 14]. Groeneveld et al. [19] list seven and X. Liu et al. [15] list one.

advanced The relevance of the defects in this category can be arguable at best and heavily depends on topics covered in a specific course.

6.3.2 Detectable defects

Table 6.4 shows what proportion of all defects each tool can detect falls into a given category. I only consider those defects that appeared at least once in the dataset and were not filtered out later.

Since each defect corresponds to a specific emittable message, the way the messages are formulated affects the total count. That is, some detectors have one message template, which they fill in with additional information; other detectors use a specific message for each eventuality. This makes the table less representative, as not all defects have the same granularity, but it can still provide an overview of what kind of defects the tool was probably built to report.

About a fifth of defects that Pylint, PyTA and Hyperstyle detect fall into the *advanced* category. The high ratio suggests they were developed for programmers dealing with a broader range of topics. Apart from missing docstrings, PyTA also reports defects dealing with access levels in classes, lambdas and exceptions. Additionally, Hyperstyle reports defects regarding getter and setter methods, inheritance, context managers, and more. Both EduLint's default and the extended configuration detect only two advanced defects, regarding bare `except` and unnecessary `lambda`.

Table 6.4: Detectable defects by category

	unsuited construct	simplifiable	unused	erroneous	error prone	poor name	long or complex	advanced	total
pylint	19%	7%	8%	26%	8%	4%	7%	21%	159
PyTA	17%	5%	8%	32%	8%	7%	7%	17%	132
Hyperstyle	22%	7%	7%	18%	7%	10%	7%	23%	335
EduLint (default)	28%	18%	14%	14%	15%	8%	0%	3%	65
EduLint (extended)	36%	17%	9%	9%	12%	7%	7%	2%	98

The table shows what portion of defects the tool is capable of detecting falls into the category. Only the defects that were detected at least once are counted. The rows sum to 100%, which corresponds to the number in column *total*. The table is colored linearly between 0 and the highest value.

Apart from the *advanced* category, Pylint, PyTA, and Hyperstyle focus prominently on the *erroneous* category; almost a third of defects detectable by PyTA fall into this category, which corresponds to the fact that the tool was developed primarily to help novices with debugging. EduLint somewhat lacks in this category, as it was built to address defects which relate to code quality more closely. Still, EduLint could become more useful during implementation by adopting more of the *erroneous* and *error prone* defects.

Around a fifth of defects that Pylint, PyTA, and Hyperstyle detect fall into *unsuited construct* category, with less attention given to *simplifiable*. Also, Hyperstyle does detect some defects through several detectors, so sometimes it reports the same defect occurrence multiple times.

EduLint focuses most heavily on *unsuited construct* and *simplifiable* categories, and it does detect several defects that other tools do not (as demonstrate in Appendix C).

Still, the overall number of defects detected by EduLint is much lower than by the other tools. This is partly due to EduLint's focus on relevance *and* precision. Still, many detectors from the other tools could be incorporated to improve EduLint's abilities even further (for example, if the code creates a variable only to return it instead of returning the value directly).

6.3.3 Detected defects

Table 6.5 shows what fraction of submissions contains at least one defect from a given category detected by a given tool. The results in

Table 6.5: Detected defects in files by category

	unsuited construct	simplifiable	unused	erroneous	error prone	poor name	long or complex	advanced	total
pylint	11%	22%	17%	6%	13%	37%	16%	99%	266,192
PyTA	9%	2%	16%	6%	62%	11%	15%	99%	261,226
Hyperstyle	40%	37%	19%	8%	18%	64%	37%	8%	424,821
EduLint (default)	19%	15%	5%	2%	1%	8%	0%	0%	62,820
EduLint (extended)	31%	20%	5%	2%	10%	8%	11%	0%	120,613

The table shows what portion of all files contains a defect in the category. Neither the rows nor the columns sum to 100%: a file may contain multiple defects and multiple tools can detect defects in the same category. The number of detected defects in some of the considered categories (each defect counted at most once per file) is in column *total*. The table is colored logarithmically between 0 and the 100%.

this table are greatly affected by the most frequent defects I already discussed in Section 6.2.

For Pylint, the most frequently detected category, present in almost all files, is *advanced*. However, only 7% of files contain a defect that is not a missing docstring. The second most frequent category *poor name* but again, only 4% files contain a defect other than a name not conforming to naming style. The third category is *simplifiable*, though only 12% of files contain a defect which does not discourage comparing with falsey value. After that, Pylint detects *unused* and *long or overly complex* in less than a fifth of all files, and *error prone* and *unsuited construct* in more than a tenth.

For PyTA, the most frequently detected category is again *advanced*, but just 4% of files contain a defect other than missing docstring. The second most frequent category is *error prone*, but just 7% contain a defect unrelated to the presence of type annotations. Then come *unused* and *long or overly complex* with around 15%, and *poor name* and *unsuited construct* present in around 10% files. In only 2% of files, PyTA detects a *simplifiable* defect.

Hyperstyle detects the most defects of all the tools: almost twice as many as the second most prolific tool (Pylint). Its most frequent category is *poor name* in almost two thirds of all files; however only a third contains a defect other than a spelling error in a name. Out of the 40% of files with *unsuited construct*, only 35% contain another defect from using explicit string concatenation and out of the 37% *simplifiable* files, just 27% contain another defect from comparing to a falsey value. While this still leaves a significant percentage of files with a defect in either of these categories, the relevance of some of these defects is still questionable (e.g. advocating for the use of `sum` with a generator instead of a `for` loop, or forbidding testing for emptiness by comparing if the size of a structure is zero, the sixth and seventh most frequent defect in the *unsuited construct* category). A third of files contains *long or overly complex* code. *Unused* or *error prone* defect is each present in a fifth of submissions.

Default EduLint focuses almost exclusively on the *unsuited construct* (a fifth of files) and *simplifiable* (15%) categories, with some attention also given to *poor name* (mostly redefining builtin and name not conforming to naming style – it avoids the false positives generated by Pylint).

Extended EduLint favours *unsuited construct* (a third of files) and *simplifiable* (a fifth of files) even more prominently. It also detects defects from *long or overly complex* and *error prone* categories in around a tenth of the files.

EduLint significantly lacks behind the other tools in the number of detected defects. However, many of the defects reported by the other tools are either arguably irrelevant or duplicates of other reported defects. Also, already in the requirement of precision set in Section 2.3.2, I stated that the precision of the detectors is more important than their recall, which is also reflected in EduLint's defect count.

Even though a significant fraction of defects which Pylint, PyTA and Hyperstyle detect falls into the *erroneous* category, only a lower number of files contains a defect in this category. This is probably caused by the fact that the dataset contained mostly correct solutions. Still, the dataset did also contain solutions which were not correct at all, passed only a part of their tests, or were meant to be run as a part of some large code, so the tools do detect some *erroneous* and *error prone* defects.

6.3.4 Summary

Both when considering which defects EduLint can detect and which get reported, EduLint provides plenty of relevant code quality feedback since most of the reported defects are in the *unsuited construct* and *simplifiable* categories.

The frequent defects which are problematic (as discussed in Section 6.2.2) and get reported by the other linters almost overwhelm the other defects the tools detect. When disregarding these defects, the focus of the tools shifts to reporting *long or overly complex* defects, which can be relevant, but usually lack actionable feedback that would ensure that the code quality improves. Defects from the *unused* category also get reported frequently by Pylint, PyTA and Hyperstyle. However, for 10 % of the files, the only *unused* defect they contain is either unused control variable of a for loop (which I already discussed and marked questionable in Section 6.2.2) or an unused argument (which is often just an unfinished implementation of a function with provided header).

Hyperstyle does detect many defects, both overall and in the *unsuited construct* and *simplifiable* categories. While some of them are potentially relevant (though not the most frequent ones), adopting them would require careful selection, improving the messages and providing explanations.

6.4 Comparison by requirements

In this section, I compare EduLint to the other existing tools with regard to how they fulfil the requirements set in Section 2.3. Table 6.6 shows the comparison and the criteria I used to determine the level of adherence. It can be seen that EduLint is the only tool that fulfills all of the requirements.

The criterion for the tool's precision uses false positives in frequent defects – examples of these are in Table 6.7. The relevance of detected defects was discussed extensively in sections 6.2 (relevance of frequent defects) and 6.3 (relevance of frequent categories).

I did not set a fail criterion for precision, as the tools were mostly precise (with notable exceptions mentioned in Table 6.7), at least to the degree I could determine. I also did not set a fail criterion for relevance to novice programmers, as I find it impossible to argue irrelevance en masse objectively.

I elaborate on EduLint's adherence to the requirements further in Chapter 7.

Table 6.6: Comparison of tools by requirements

	pylint	PyTA	Hyperstyle	EduLint
Precision	∼	∼	∼	✓
Relevance	∼	∼	∼	✓
Configurability	✓	×	∼	✓
Explanations	∼	✓	×	✓
Ease of use	×	∼	×	✓

Criteria

Precision

∼	At least one detector frequently generates false positives (examples in Table 6.7).
✓	Detectors do not generate any known false positives.

Relevance (for more arguments see sections 6.2 and 6.3)

∼	Frequently reports defects which can be argued of little relevance.
✓	Frequently reports defects which can be argued to be relevant to novice programmers.

Configurability

×	The tool does not allow enabling and disabling individual defects on each run.
∼	The tool allows enabling and disabling whole tools it wraps, but not individual defects, on each run.
✓	The tool allows enabling and disabling individual defects on each run.

Explanations

×	The tool provides no explanations or examples, except for the defects' messages.
∼	The tool provides explanations and examples, but only in its documentation.
✓	The tool provides explanations and examples to most defects.

Ease of use

×	The tool is only available from command line (or through a paid service).
∼	The tool provides a graphical output, but still requires installing a package.
✓	The tool can be run through a graphical interface, which does not require installing any special software.

Table 6.7: False positives (FP) examples

Tool	Defect	Files with the defect	FP [a]	FP case
pylint	name does not conform to naming style	34%	cca 90%	the detector flags any variable name with less than three characters as not conforming to snake_case, even if it does
PyTA	use of global variables	26%	cca 40%	the detector flags any definition of a type alias as a global variable, even if it is not modified
Hyperstyle	possibly misspelled name	52%	cca 60%	the detector flags names containing a number, like `p1` or `word1`, common abbreviations like `ipv4`, and actual Python methods like `isdecimal`

a. Probable false positives over the total number of occurrences of the defect.

6.5 Threats to validity

There are multiple limitations to the evaluation methods, especially when comparing tools regarding the frequency of defects and defect categories.

I attempted to use data from several sources of novice code to examine the tools' behaviour in a broader range of settings; however, the sizes of the individual datasets are unbalanced. The results are therefore skewed towards defects appearing in the first-year university course IB111. Also, I used parts of the IB111 dataset for fine-tuning during the tool's development (though I did not examine all 80,000 files).

Out of the compared tools, Pylint nor PyTA were built specifically to provide code quality feedback to novice programmers. Pylint focuses on providing feedback to professional programmers, and PyTA concentrates on helping novices write functionally correct code. I also only considered some existing tools; I did not search for solutions that could work with any programming language, nor did I pursue tools that were not, to the best of my knowledge, already publicly available.

In the analyses, I focused heavily on the most frequent defects and categories, not examining potentially highly relevant but less frequent ones.

PyTA does report several defects that I argued are irrelevant in general settings (missing docstrings, missing type annotations) or which were false positives in the context of the dataset (global variables, top-level code). These defects seem to arise from a set of course rules, and they would be highly relevant in any other course with the same rules.

Formatting defects are not particularly interesting, and also all Pycodestyle defects are reported by all of the tools, except for Pylint, so they are not that useful for comparison. However, *formatting* defects still comprise a portion of detected defects, and for some, their relevance is also debatable.

I only compare EduLint's performance on historical data without examining how actual students interact with it.

I strive to support my conclusions through research in the arguments for defects' and categories' relevance. However, the results are still biased in favour of EduLint since I would not have it detect defects I do not consider relevant. There is a similar issue with the overall set of requirements and the criteria for their fulfilment: I developed EduLint so that it would pass this set of requirements with these criteria, though I offer justification for the requirements and I attempted to set the criteria fairly.

7 Adherence to requirements

In this chapter, I first show how EduLint adheres to the requirements set in Section 2.3. I claim it succeeds at employing precise detectors of relevant defects, is highly configurable and that the web interface provides an easy way to interact with the tool, supplying the users with explanations of the defects and examples of how they can be fixed. In the last section, I outline the future work. Most notably, I suggest creating different configurations for different kinds of novice programmers (like high school who are just discovering programming vs university computer science students).

7.1 Evaluation of adherence

In this section, I evaluate how EduLint holds up to the requirements set in Section 2.3 and describe how I ensured the requirements are met.

7.1.1 Defects relevant to novice programmers

As established in Section 2.3.1, the tool should defect a wide variety of defects, all potentially relevant to novice programmers.

EduLint can detect many defects, most of which occur in novice code. Their selection was based on articles dealing with common novice defects, listed in Section 2.3.1, guidelines to novice teaching assistants, consultation with other educators and lastly, personal experience with tutoring novice programmers and reviewing their code. These defects are split between the default and several extension configurations (presented in Section 5.3.2.)

In this section, I show how many defects EduLint detects in code by novice programmers. I describe the reasons behind developing custom detectors. I state that the idea of a default configuration is problematic and suggest mitigation.

I argued for the relevance of the defects EduLint detects extensively in sections 6.2 (shows frequently detected defects are relevant) and 6.3 (shows frequently detected defect categories are relevant).

The number of defects

EduLint is capable of detecting 185 different defects in the default configuration, 222 if the extension groups of Python-specific, code complexity-related and enhancing defects are included.

When EduLint is run on the dataset I introduced in Section 6.1.1, it detects 124 different defects in the default configuration at least once, 159 in the extended configuration. Out of the 61, resp. 63 undetected defects, 57 are Flake8 defects in both cases (15 of those 57 are *advanced* or *deprecated* and 14 regard the `format` method).

Custom detectors

Out of the 33 defects for which EduLint has custom detectors, 17 are not present in any of the examined tools (listed in Section 4), nor any other, to the best of my knowledge. The remaining 16 detectors have either higher precision or higher recall or are narrower in scope but only to provide actionable feedback to a more specific situation (for concrete examples, see Section 5.4).

Table C.1 shows the complete list of these custom defects. For each defect, it shows a code containing it, the information on whether EduLint is the only tool that detects the defect and the frequency of the defect on novice code.

While EduLint does detect many defects that probably would not be addressed otherwise, there are still plenty of defects that go unnoticed. The defect category in which EduLint lacks the most is *duplicate* code, though it shares this weakness with all other discussed tools. See 7.2.2 for more information.

Relevance of the default configuration

The default configuration is relevant primarily to first-year computer science college students but not so much to different groups of novice programmers (like high school students). Even more defects would have to be moved to some extension group to make the default configuration relevant to a more general audience. Creating more extension groups would make using the tool impractical for educators with higher expectations of their students. They would need to enable many of these groups for each task (and study the options more extensively beforehand). I discuss the issue further and outline the mitigation steps I plan to take in Section 7.2.1.

7.1.2 Precise detection of defects

EduLint detectors were chosen or developed with precision on novice code as the first priority.

As stated in Section 2.3.2, any report of a piece of code containing the detected construction but in a context where it could be judged legitimate by a human is considered a false positive (for an example, see the section mentioned above). Despite all efforts, EduLint's detectors still sometimes provide false positives, but these are treated as bugs and should be fixed as soon as possible.

In this section, I describe how I went about ensuring the high precision of the detectors, how I handled false positives I encountered and how the code is tested.

Ensuring precision

For each detector I adopted or developed, I linted submissions from the last three years of IB111 and manually examined 10–20 submissions to see which parts of the code were flagged as defective.

For some detectors I developed, I used an iterative approach. I first wrote a simple detector for the most straightforward variant of the defect and then incrementally added detection for more specific cases, examining which instances appeared and which disappeared in each iteration. This process helped me examine even more specific variants of the defects.

I also monitored the frequency of defects overall. It helped me discover several detectors generating false positives because their defect occurred more often than expected.

Lastly, I used the instructors' solutions for the IB111 tasks as a benchmark, assuming that the code written by instructors should be exemplary. While this assumption has not proven entirely accurate, this process also helped discover several false positives.

False positives remediation

If I encountered any false positives in a detector I wanted to adopt, I either tried to filter out the unwanted messages, reimplemented the detector, or marked the defect as needing more attention.

For each encountered false positive in the detectors I developed, I altered the implementation. Sometimes, this forced me to reconsider the scope of the defect, as there were too many false positives for its broader definition. In some cases, I even discard the defect as a whole (for example, in the case of `else: if` to `elif` described in Section 2.3.2).

Testing

I also wrote a regression test for each false positive in my detectors. I wrote tests for true positives or true negatives as well.

Currently, the project has 151 tests for the 33 custom detectors (some tests regard multiple defects, especially those on real-life code). The project currently runs 296 tests in total. Unfortunately, I cannot provide any coverage information because both the linters are being run as separate subprocesses (to my knowledge, neither provides a stable and documented API for linting files directly).

A new version of EduLint is published only if it passes the tests.

7.1.3 Configurability

EduLint allows each detector to be enabled or disabled on each run. It also allows putting configuration directly into the source code, allowing for easy distribution to students (which is an option that no other tool provides, they usually only allow disabling specific detectors for specific parts of code).

Out of the commonly available means of configuration, EduLint does not implement configuration files. This was initially on purpose, as I wanted to avoid forcing the students to manage configuration files. However, I plan to add support for them to mitigate an issue with the default configuration, which I discuss further in Section 7.2.1.

7.1.4 Clear descriptions of defects and how to fix them

EduLint scrapes explanations and examples of fixing Pylint's defects from Pylint's documentation. It also provides manually written ones for some custom defects. It does not yet provide hints for Flake8 defects because they are mostly straightforward to understand and fix.

While the scraped explanations are better than no explanations, they were not written for novice programmers and can miss important information or opportunities to fix students' misconceptions. In this aspect, the explanations can still be significantly improved.

The explanations are displayed alongside the detected defects on EduLint's web page, though not when the package is used through the command line directly. This behaviour follows the assumption that students will use the web interface while only educators will interact directly with the command line interface.

7.1.5 Ease of use

Although the package itself does not fulfil this requirement (as it still requires installing the package locally), EduLint comes with an accompanying web page (described in Section 5.7) that a student can paste their code into and receive the results there.

As stated in Section 5.7.1, this solution has the advantage of high accessibility and low barrier to use, but it forces the student to leave their IDE. I describe the arguments against developing a plugin in the section mentioned above. Nevertheless, I plan to adapt EduLint as a Thonny plugin; I discuss this in greater detail in Section 7.2.4.

7.2 Future work

In this section, I discuss the improvements I plan for the tool.

7.2.1 Different configurations for different target groups

Even though the tool strives to provide a reasonable default, the idea that there could be one default configuration for all different target groups (from high school students who learn programming as a compulsory topic to college students who hope to make their living as a programmer one day) has proven to be too idealistic. At the same time, the expectation that each educator would select extension groups according to their need creates a high threshold for adopting the tool.

Therefore, I plan to develop several configurations for different groups of novice programmers. These would be distributed in the form of configuration files but as a part of the package. The information on which configuration file to use would still be a part of the file itself. Hence, if an educator chooses one of these pre-prepared configurations, their students would not have to handle any configuration files. I am still considering how to handle distributing custom configuration files.

7.2.2 More defect detectors

While EduLint already detects many relevant defects, it can always detect more. The most glaring gap is in the category of *duplicate* code, where it detects no defects, even though the defects in this category are undoubtedly important (Several of the papers with novice defects listed in Section 2.3.1 contain some defects in the *duplicate* category [17, 18, 14, 19]).

From the considered tools, only Pylint and WPS detect some defects related to code duplication: Pylint can check for identical code segments in multiple files and WPS checks for expressions repeated too many times. Both come with their limitations: Pylint's solution cannot find code segments that are very similar but not identical, and it cannot search for duplicate code inside a single file; WPS flags even repeated uses of simple expressions like `len(lst)`, even though extracting them to a variable would not improve the code much. I briefly searched for other tools but did not find any solution that would be able to find similar code blocks inside one file, except for a PyCharm plugin.

I attempted to write my own detector, but I did not create a precise, robust and general solution in the allocated time, so I decided to abandon the effort for the time being. Creating at least detectors for more specific situations should nevertheless be much easier.

Apart from defects from the *duplicate* category, many other defects remain to be detected. Flake8's plugins already detect several interesting ones; these still need to be vetted for false positives, possibly reworded to be clearer and provisioned with an explanation. I also consider adding mypy as one of the tools that EduLint incorporates.

Lastly, there are some other defects undetected by other tools, like checking if a function is pure.

7.2.3 Prioritization and presentation

The number of defects EduLint can detect is already relatively high and is only expected to grow. For around 800 codes linted during the evaluation described in Chapter 6, EduLint emitted over 50 messages. Currently, the web only shows the list of detected defects (which can become overwhelming if there are too many), without any indication of which are critical to avoid and which are only small enhancement proposals. So, EduLint, and most notably its web interface, could be improved by clearly communicating which defects the novices should focus on the most.

Also, allowing the students to easily receive more feedback on their code than their instructor set as the required level might be beneficial if they seek to improve the code even further.

7.2.4 Thonny plugin

While creating a web page as an interface has many advantages (detailed in Section 5.7.1), this solution is still not ideal. A web interface forces the student to leave the IDE they have been working in, encouraging the practice of fixing all errors when they solve the task rather than throughout when the hints of *erroneous* or *error prone* constructions would be more helpful. Also, in the web interface, the students (currently) cannot run their code to see whether their attempts to fix the defects broke the functionality. This setup might force the students to switch between the web page and their IDE, which can be bothersome.

To balance this issue while avoiding the pitfalls of maintaining multiple IDE plugins, I plan to develop a plugin only for the Thonny IDE, which already has several other features tailored to novice programmers and is often recommended to them. (The students who decide against using Thonny can still use the web interface.)

8 Conclusion

Teaching how to write high-quality code is important because quality code is easier to extend and maintain. However, providing feedback on code quality to novice programmers is costly, bordering on infeasible in many cases. Even though there are multiple automatic tools, including several explicitly aimed at novice programmers, they come with different shortcomings.

To create a tool that avoids at least some of those shortcomings, I proposed, developed and evaluated EduLint. This tool automatically provides feedback on many code quality defects found in Python code by novice programmers.

I set and justified requirements for the tool: it should provide precise and relevant feedback to novice programmers, with examples and explanations, and be easy to use. I introduced defect categorization to reveal the scope of defects that can occur in novice code and to ease assessing the capabilities of existing tools and comparing them to EduLint's. I presented several existing solutions for providing code quality feedback, ranging from industry-grade linters to tools developed specifically for novice programmers.

I described how the EduLint linter and its web interface operate. EduLint is available as a pip package. It extends the basic capabilities of Pylint and Flake8, widely used Python linters, by employing several custom detectors I developed. Many of these detect defects that other examined tools miss (e.g. using a `while` loop instead of a better-suited `for` loop). It allows for flexible configuration of detectors and provides a default configuration crafted specifically for novice programmers. EduLint's web interface can lint a file without installing any specialized software. It also displays explanations for many defects and examples of how a student can fix them.

I extensively evaluated the developed tool: I rigorously examined the relevance of defects EduLint reports compared to defects and defect categories frequently reported by other tools. I concluded that out of all examined tools, EduLint provides the highest share of feedback relevant to novice programmers, as it frequently reports defects and defect categories that are often listed as problematic in novice code in related research. Furthermore, I examined how EduLint adheres to the requirements mentioned above, and I suggested steps to improve its capabilities even further.

I plan to improve EduLint's usability for various kinds of novice programmers by creating different configurations. Currently, its configuration is best suited to first-year college students but is probably too strict for high school students. I also plan to investigate ways to detect duplicate code, as duplication is a frequent and serious defect, but it goes undetected in all of the examined tools.

A EduLint installation and running

The easiest way to run EduLint is to visit `https://edulint.com`, where the user can paste the code they intend to lint and click "Check". The site also contains an example defective code[1].

EduLint can also be installed locally using `pip` by running the following command. It might be necessary first to activate a virtual environment if the tool's dependencies clash with versions of some of the installed packages. It requires a Python version of at least 3.8.

```
python3 -m pip install --user edulint==2.6.4
```

The tool can be run from the command line like so:

```
python3 -m edulint FILE [FILE ...]
```

For ways to configure the tool and available configuration options, see the tool's documentation[2]. It might be desirable to run only Pylint and the custom detectors. This can be achieved by using the `no-flake8` option, running the tool like so:

```
python3 -m edulint -o no-flake8 FILE
```

EduLint can also be run directly from the source code by running the previous command in the root folder of its GitHub project[3] (or in the `edulint` folder of the archive). However, when run this way, it is necessay to set `PYTHONPATH`, to correctly load the custom detectors. To run the tool directly from source, I recommend using the enclosed `Makefile`.

```
make run ARGS="FILE"
make run ARGS="-o no-flake8 FILE"
```

EduLint does report defects in EduLint's source code. Most of them regard lines longer than 79 characters, which is the limit recommended by PEP8.

1. `https://edulint.com/editor/code/example`
2. `https://edulint.readthedocs.io/en/v2.6.4/#configuration`
3. `https://github.com/GiraffeReversed/edulint/tree/v2.6.4`

B Archive organization and contributions

The archive contains three folders:

- Folder `edulint` contains the GitHub project for the linter package. The same version can be found online[1].
- Folder `edulint-web` contains the GitHub project for EduLint's web interface. The same version can be found online[2].
- Folder `analysis` contains the Jupyter notebook with code I used to compare and evaluate the tools and its accompanying files. The notebook cannot be run as I am not at liberty to disclose the data I used. There is also an HTML version of the notebook, which shows at least the output the notebook produced (the notebook itself does not display several tables). The folder also contains the categorized list of defects as a CSV.

I received some minor code contributions to the open-source repositories for EduLint and its web interface. These consisted of improvements of the continuous integration and deployment, the README files, the wrapper for handling sub-processes and some explanations. The relevant commits are identifiable by the author field in GIT history.

1. `https://github.com/GiraffeReversed/edulint/tree/v2.6.4`
2. `https://github.com/GiraffeReversed/edulint-web/tree/thesis`

C EduLint's custom detectors

In this chapter, I first describe how the individual custom detectors operate, and then I show how frequently these custom defects appear in novice code.

C.1 Detectors

In this section, I list the detectors I implemented, grouped by categories I present in Chapter 3. For each, I detail why I implemented it and describe roughly how it works, with examples where relevant. I also mention known issues with these detectors, if there are any.

The detectors walk the abstract syntax tree (AST), inspecting the node types and their values. Since I have built the detectors as pylint plugins, they use the AST representation from the library `astroid`.

C.1.1 Unsuited construct

This section describes newly detected defects that can be fixed using a more appropriate construct.

Use a `for` loop instead of a `while` loop

Novice programmers sometimes forget to use a for loop when the number of iterations is known in advance.

```
i = 0
while i < n:
    # do something
    i += 1
```

This defect is detected when there is a while loop with a condition that comprises a comparison between two values or variables. The detector emits the suggestion if precisely one of these is modified in the body of the while loop, and this modification happens as the last statement of the block. The modification must be incrementing or decrementing the variable by one.

Iterate through values rather than through indices

Novice programmers often prefer the more general approach of iterating through `range(len(list))` to iterate over list elements, even when iterating directly over `list` would suffice.

```
for i in range(len(lst)):
    # do something with value at lst[i]
    # (other than assigning to it)
```

To detect this situation, upon encountering a for loop such as `for i in range(len(list))`, the detector checks the body for all occurrences of `i` and `list[i]`. If there are no occurrences of `i` (except in `list[i]`) and no occurrence of `list[i]` is being assigned to, the detector emits the suggestion to iterate the list directly.

Iterate using enumerate rather than through indices

In some cases, novice programmers might not be aware that they may use enumerate if they need to use both the index and the corresponding value.

```
for i in range(len(lst)):
    # do something with i and lst[i]
```

The detector for this defect is a less restrictive variant of the detector in the previous section – it just detects if both the index and the value at it are used.

Use a while loop with a condition rather than `while True`

Novice programmers sometimes overuse the `while True` loop, making their code less readable as it is unclear under which condition (if ever) the loop terminates.

```
while True:
    if c:
        break
    # do something
```

If a detector finds a while loop with its condition being just `True`, it checks if the first statement of the loop's body is an if statement ending in a break, and if yes, then it suggests using the if's condition negated as the condition of the loop.

Use tighter range boundaries

A novice programmer might use `range(n)` out of habit, even though they want to skip the first/last iteration (for example, when working with consecutive pairs of elements in a list).

```
for i in range(n):
    if i == 0:
        continue
    # do something
```

The detector seeks `for` loops over a range. For each such loop, it checks whether the first statement in the loop is an if statement that tests whether the value of the control variable is equal to the starting value of the range (like in the code above) or to the last value the range would generate. If the if statement tests this and immediately skips the iteration with `continue`, the detector suggests using tighter boundaries for the range instead (e.g. `range(1, n)`).

Use `append`

A novice programmer might create an unnecessary list of size one to append an item to a list: `list += [val]` or `list.extend([val])`.

The detector checks if a single item list is added to a value or if the `extend` method is called with a list of size one. If it finds such a case, it suggests using `append` instead.

Use integer division

A novice programmer may use float division and immediately convert the result to an integer (i.e., `int(p / q)`), even though this may cause numeric instabilities when working with large numbers.

If a detector finds a call to the `int` function whose parameter is the application of / on some values, it suggests using integer division // instead.

Use `isdecimal`

Novice programmers might not be aware of the differences between the functions `isdecimal`, `isnumeric` and `isdigit` and use the wrong one to test if a string is convertible to a number using the `int` function.

If a detector comes across an occurrence of `isnumeric` or `isdigit`, it suggests using `isdecimal`. It flags any use of these functions, as it can be assumed that any use of them in novice programmers' code was meant to test if a string contains a number.

Use augmented assignment

A novice programmer might forget to use an augmented assignment, using a regular assignment instead, though it is longer and possibly more error-prone.

The detector examines assignments, and if it finds one that assigns a binary operator that uses the assigned variable as one of its operands, it suggests using an augmented assignment instead. It does not suggest this if one of the operands is a mutable data structure, as these have different semantics (e.g. `+` creates a copy of both of its operands while `+=` does not). It also does not suggest a change if a list or a string is added to a variable (e.g. `var = "x" + var`). This operation is not commutative, so using the augmented assignment here would lead to a different result.

Compare to a string literal directly

Novice programmers sometimes use magical constants when comparing letters (e.g. `ord(letter) < 65`) as they might not be aware that they can compare characters directly.

The detector searches for comparisons in which at least one operand contains a call to `ord` (to determine if the expression compares letters), and some are constant. If yes and the constant is in the printable range, it suggests removing the call to `ord` and directly comparing the value to a literal string.

As it is usually more readable for a comparison to happen in terms of some border values (specifically `'a'`, `'z'`, `'A'`, `'Z'`, `'0'` and `'9'`), the detector checks if altering the comparison operator (e.g. from `<` to

<=) would allow for comparing to these preferred values instead and if it does, it also suggests changing the operator.

Use `ord` applied to a letter

Similarly to the previous case, novice programmers sometimes use magical constants when doing arithmetic with letters (e.g. `letter - 65`).

The detector checks binary operations to see if one operand contains a call to `ord` (again to determine if the expression deals with letter arithmetic) and if the other is a constant in the printable range. If yes, it suggests replacing it with an `ord` call with the corresponding letter as the argument. Again, the same values are preferred; for example, `ord('A') - 1` is suggested over `ord('@')`.

Use appropriate operation instead of repeating addition/multiplication

A novice programmer may use repeated addition/multiplication instead of multiplication/exponentiation. This can lead to errors if the programmer, for example, multiplies different numbers, and the intention is less clear as well.

The detector checks for cases of addition/multiplication in which both operands are the same (e.g. `x + x`). If one of the operands is again a binary operation (like in, for example, `x * x * x`), it checks whether it is the same type of operation and if so, if both of its parameters are again the same (to find even occurrences where the operation is used multiple times). If it finds such a case, it suggests using multiplication or exponentiation instead.

C.1.2 Simplifiable

This section describes newly detected defects that can be fixed by removing or restructuring some parts of the code.

If statement simplifiable to its condition

Some novice programmers tend to overuse if statements or if expressions in situations where directly manipulating the values used in the condition would suffice. This section shows some examples of code containing variations of this defect and details how they are detected.

Bool values only The basic variation of this defect would be as follows:

```
if c:
    return True
else:
    return False
```

The detector for this defect checks each if statement by first checking if the positive branch contains a single statement: the return of a value. If it does, it then checks if the negative branch immediately returns a value or if the statement right after the if statement is a return statement. The detector extracts the returned values if these two return statements are present. If they are both bool values, it emits the suggestion that the if statement can be simplified either to the if statement's condition or to its negation, depending on which branch returns `True` and which `False`.

The defect might manifest in several other situations: instead of an if statement, the programmer might use an if expression (`True if c else False`) or assign the bool value to a variable rather than returning it. These are detected similarly.

Bool value and an expression A more complicated variation of this defect is when only one of the branches returns a variable containing a bool value:

```
if c:
    return val
else:
    return False
```

This variation is detected similarly to the previous one, with less strict requirements on the returned values: only one must be a bool, and the other can be any expression. If this variation is detected, the detector suggests returning an expression created from the if's condition and the returned expression using logical operators.

Nested if statements The second-to-last variation is when the programmer uses nested if statements instead of joining their conditions with `and`:

```
if c1:
    if c2:
        # do something
```

If the detector for this variation comes across an if statement with no negative branch and the only statement in its positive branch is again an if statement with no negative branch, it suggests merging them into one using logical conjunction.

Consecutive if statements The last variation is when the programmer uses two consecutive if statements returning the same value:

```
if c1:
    return False
if c2:
    return False
# do something
```

This variation is detected when there are two if statements, one immediately after the other, each without an else block, that both directly return the same value. In such a case, the detector suggests merging them into one using logical disjunction.

Known issues There are two known issues with these detectors. The first is that there may be type changes in the suggested solution. For example, should the value `c` in the basic variation not be a bool, then in the suggested `return c`, the returned value would have a different type than in the original solution. It would be possible to suggest returning `bool(c)`, but this would introduce an unnecessary call to the `bool` function when `c` is a bool. Differentiating between these two cases can only be done precisely in type-safe code.

The second known issue is that the resulting condition might be overly complex for all the variations except the first, as it is created by combining two other ones. This issue could be mitigated by specifying, for example, the maximum length of the resulting condition in tokens or in used logical operators.

If statement with an empty positive branch

Sometimes novice programmers write an if statement with only `pass` in the positive branch and all the relevant code in the negative branch, possibly due to refactoring.

```
if c:
    pass
else:
    # do something
```

Suppose the detector finds an if statement with nothing but `pass` in its positive branch. In that case, it suggests negating the condition, moving the code from the negative branch to the positive one and removing the else branch.

A loop making at most one iteration

During refactoring, a novice programmer might create a loop that only makes one iteration without realizing it is no longer needed.

Suppose the detector finds a for loop iterating over a range in which the start plus the step value equals or exceeds the stop value. In that case, it suggests dropping the loop altogether.

Redundant arithmetic

For example, a novice programmer might use some arithmetic operation because they misunderstand the operators' priority.

The full list of operators and their arguments is as follows (`v` can be any expression):

```
v + 0
0 + v
v - 0
0 - v
v * 0
0 * v
v ** 0
v * 1
1 * v
```

```
v / 1
v ** 1
v + ""
"" + v
v / v
v // v
v % v
```

The detector checks binary operations and augmented assignment operators. If it finds one using one of the redundant patterns listed above, it suggests removing the redundant operation.

The detector does not detect `v // 1` and `v % 1`, because some students use it to test whether a variable contains a whole number.

Redundant elif

A novice programmer might write two consecutive if statements with the condition in the second one being a negation of the first one's condition.

```
if n > 0:
    # do something
elif n <= 0:
    # do something else
```

If such a situation is detected, the tool suggests using else instead of the second elif. Whether one condition is a negation of another is checked purely syntactically (by matching operators and subexpressions, taking `not` occurrences into consideration).

If the second if statement is in the `else` branch of the first if statement and has an `else` branch, the detector suggests removing the else branch as it is unreachable.

`is` with a boolean value

A novice programmer might use `is` with a boolean value to test if a condition is true or false (e.g. `c is True`).

If such a use of `is` is discovered, the tool suggests using the condition directly.

C.1.3 Unused

This section describes the newly detected defect that regards lines that do not affect the code's outcome.

Changing control variable has no effect

Novice programmers sometimes change a for loop's control variable in the loop's body even though this does not affect its value in the next iteration. This might result from an oversight during refactoring or a misconception about how the for loop works in Python.

```
for i in range(n):
    # do something
    i += 1
```

If a for loop's control variable is reassigned in the last statement of the loop's body, the detector suggests removing the line.

C.1.4 Error prone

This section describes newly detected defects that regard constructs that can easily introduce a bug into the code.

Using global variables

Novice programmers sometimes introduce global state into their code through global variables, though this is often unnecessary [?]. In some cases, their solution relies on the fact that the tests run some parts of the code only once, which is problematic. The purpose of this checker is to detect any use of global variables.

The detector first collects all variables defined at the global (module) scope. Then, to avoid falsely marking global constants, including type aliases, it checks which variables are being modified anywhere in the code. It considers that a variable defined in another scope does not modify the global variable but also considers scope modification keywords `global` and `nonlocal`.

Modifying structure (by adding or removing elements) that is being iterated over

A novice programmer might modify an iterated-over structure by adding or removing elements, unaware that this may lead to hard-to-find bugs.

```
for elem in lst:
    lst.remove(elem)
```

The detector searches for a for loop that iterates over a structure, then checks what operations are applied to the structure inside the loop. If one of these operations adds elements to the structure or removes some, it reports this, as iterating over a copy is usually preferable.

Using a loop with else

A novice programmer may forget that any code after a loop executes when the loop terminates, not only the one placed in the loop's `else` branch. This misconception seems more prevalent than correctly using the loop's `else` branch. Even the correct use can be hard to read and can usually be reworked.

The detector looks for any loop with an else branch, and if it finds such, it suggests getting rid of the else.

Multiplying a list containing a mutable data structure

A novice programmer might be tempted to use the shortcut of creating a list containing one item several times by multiplying a single-element list with such item by a constant (e.g. `[[]] * n` to create a list of `n` empty lists). If the item is an instance of a mutable data structure, a list containing `n` references to that one instance is created rather than a list of `n` independent instances.

If a detector comes across a list containing a mutable element being multiplied, it suggests using list comprehension instead.

One branch is not returning a value

A novice programmer might write a function that returns a value in one conditional branch but leaves the other one to simply `return` (thus returning implicit None, which is a behaviour they may not even be aware of and which might lead to an error).

```
if n < 0:
    return
return True
```

The detector looks for an if statement that either returns in positive and negative branches or has no else branch but is followed by a `return`. If it finds these two return statements and exactly one of them does not return a value (is a bare `return`), it suggests returning a value in both branches or neither.

C.1.5 Poor name

This section describes newly detected defects related to ill-suited choices of variable names.

Single character variable names

Novice programmers sometimes overuse single-letter variables, be it `i` for purposes other than the control variable of a loop or just using letters of the alphabet for intermediate results.

In general, it cannot be said which single-letter variable names are not appropriate. However, a set of allowed single-letter variable names could be created for a specific task.

I did not write a separate detector for this defect but instead used Pylint's ability to disallow some variable names. On passing an option specifying a list of allowed single-character names, I configure pylint to report usage of any single-character variable and then filter the results, keeping only those reports that mention a name not listed as allowed.

Variable shadowing the control variable of the parent for loop

A novice programmer might habitually name the control variable of every for loop `i`. This may cause them problems if the for loops are nested.

```
for i in range(m):
    for i in range(n):
        # do something
```

If the detector finds out that a `for` loop uses the same name for its control variable as a `for` loop anywhere in its body, it reports this.

C.2 Custom defect frequencies

I developed several custom detectors for EduLint, either to cover a defect that was not detected by any other tool or to create a detector with higher precision or higher recall than available detectors had. In Table C.1, I show how frequently these defects appear in the dataset described in Section 6.1.1. Even though the absolute counts are quite low, each piece of code that does not contain them will be slightly better than the code that does. (Also, they would no longer draw a reviewer's attention should the code receive manual code review.)

Most often, the defects belong to the *unsuited construct*, *simplifiable* and *error prone* categories; in Section 6.3, I argued these are at the intersection of relevant defects (mostly with regards to code quality) and defects detectable with relative ease.

For a description of the detectors themselves, see the previous section.

Same as in Table 3.1, the table also shows whether EduLint is (to the best of my knowledge) the only tool that detects the defect (★), if EduLint detects the defect via a custom detector I developed to improve over detectors in other tools (↑) or if EduLint does not detect the defect at all (×). If the symbol is followed by an asterisk (*), EduLint does not detect the defect in its default configuration, but in some extension (for the difference, see Section 5.3.2).

Table C.1: Custom defects

Defect	Category	Defective code	EduLint	Files with defect
use augmented assign	*unsuited construct*	`x = x + 1`	↑	8336

use enumerate	*unsuited construct*	`for i in range(len(lst)):`
		`    # code using both`
		`    # i and lst[i]`
			↑*	7526

iterate directly	*unsuited construct*	`for i in range(len(lst)):`
		`    # code only reading`
		`    # from lst[i]`
			↑	6145

simplifiable `if return`	*simplifiable*	`if c:`
		`    return True`
		`return False`
			↑	4519

nested `if` statements	*simplifiable*	`if c1:`
		`    if c2:`
		`        # body`
			★*	4414

use `elif`	*unsuited construct*	`else:`
		`    if c:`
		`        # code`
			×[1]	3568

use a `for` loop instead of a `while` loop	*unsuited construct*	`while i < n:`
		`    # body`
		`    i += 1`
			★	2733

consecutive `if` statements	*simplifiable*	`if c1:`
		`    return True`
		`if c2:`
		`    return True`
			★*	2464

`if return` simplifiable using logical operator	*simplifiable*	`if c:`
		`    return True`
		`return x`
			★*	2335

use `isdecimal`	*unsuited construct*	`val.isnumeric()`	★	2091
do not use `is` with bool	*simplifiable*	`val is False`	↑	1887

use `else` instead of `elif`	*simplifiable*	`if x <= y:`
		`    # body`
		`elif x > y:`
		`    # body`
			↑	1276

redundant arithmetic	*simplifiable*	`val + []`	↑	1016
repeated operation	*unsuited construct*	`x * x * x`	★*	1004
use `append`	*unsuited construct*	`lst += [val]`	★	685
use integer division	*unsuited construct*	`int(x / y)`	★	612

1. This defect is currently not reported in any configuration, as it generated too many false positives.

Defect	Category	Defective code	EduLint	Files with defect
global variables	*error prone*		↑	604
use `ord` applied to a letter instead of using a magical constant	*unsuited construct*	`ord(char) - 65`	⋆	539

empty `if` branch	*simplifiable*	`if c:`
		`    pass`
		`else:`
		`    # body`
			⋆	514

compare to a string literal instead of a magical constant	*unsuited construct*	`ord(char) < 65`	⋆	495
simplifiable `if` expression	*simplifiable*	`True if c else False`	↑	467

do not use a loop with `else`	*error prone*	`for val in lst:`
		`    # body`
		`else:`
		`    # body`
			↑	393

do not add elements to iterated structure or remove from it	*error prone*	`for val in lst:`
		`    # body`
		`    lst.remove(x)`
			↑	382

do not use `while True` followed by `break`	*unsuited construct*	`while True:`
		`    if c:`
		`        break`
		`    # body`
			⋆	231

inner loop shadows outer loop's control variable	*poor name*	`for i in range(n1):`
		`    for i in range(n2):`
		`        # body`
			↑*	218

simplifiable `if` with assignment	*simplifiable*	`if c:`
		`    v = True`
		`else:`
		`    v = False`
			↑	195

unnecessary changing control variable	*unused*	`for i in range(n):`
		`    # body`
		`    i += 1`
			⋆	151

use tighter `range` boundaries	*unsuited construct*	`for i in range(n):`
		`    if i == 0:`
		`        continue`
		`    # body`
			⋆	43

`for` loop makes at most one iteration	*simplifiable*	`for i in range(1):`
		`    # body`
			↑	34

one branch is not returning a value	*error prone*	`if n < 0:`
		`    return`
		`return True`
			↑	31

`if` assignment simplifiable using logical operator	*simplifiable*	`if c:`
		`    v = False`
		`else:`
		`    v = x`
			⋆*	25

Defect	Category	Defective code	EduLint	Files with defect

unreachable `else`	*unused*	`if x <= y:`
		`    # body`
		`elif x > y:`
		`    # body`
		`else:`
		`    # unreachable`
			⋆	19

`if` expression simplifiable using logical operator	*simplifiable*	`x if c else True`	$\star^*$	14
multiplying a list containing a mutable data structure	*error prone*	`[[]] * 5`	↑	5

D The complete list of defects

This chapter shows those defects EduLint can detect that had at least one occurrence in the dataset. There are 185 defects with at least one occurrence out of the 222 EduLint can detect. Column Extension says whether the defect is detected in the default configuration or which extension group it belongs to.

Table D.1: All defects with at least one occurrence

Message	Extension	Defective code	Files with defect
top-level code	enhancement		23880
line too long	default	over 79 characters	11044
expected 2 blank lines after class or function definition, found 1	default		10447
expected 2 blank lines, found 1	default		9612
no newline at end of file	default		9117
blank line at end of file	default		8643
blank line contains whitespace	default		8414
use augmented assign	default	`x = x + 1`	8336
unnecessary `else` after `return`	default	`if c:` `    return val1` `else:` `    return val2`	8233
use enumerate	Python-specific	`for i in range(len(lst)):` `    # code using both` `    # i and lst[i]`	7526
do not use `open` without specifying encoding	enhancement	`open("/path")`	6819
at least two spaces before inline comment	default		6390
iterate directly	default	`for i in range(len(lst)):` `    # code only reading` `    # from lst[i]`	6145
unnecessary parenthesis after keyword	enhancement	`return(value)`	5988
too many blank lines	default		5731
missing whitespace after `,`	default		5494

Message	Extension	Defective code	Files with defect
redefining built-in	default	`sum = 0`	4631
simplifiable `if return`	default	`if c:` `return True` `return False`	4519
too many arguments	complexity	more than 5	4474
nested `if` statements	enhancement	`if c1:` `if c2:` `# body`	4414
too many branches	complexity	more than 12	4349
trailing whitespace	default		4080
too many statements	complexity	more than 50	3810
name doesn't conform to naming style	default	`localVariable, global_constant, CLASS_NAME`	3608
some `returns` value, some do not	Python-specific	`if c:` `return True` `return`	3203
too many return statements	complexity	more than 6	2929
unused import	default		2785
use a `for` loop instead of a `while` loop	default	`while i < n:` `# body` `i += 1`	2733
missing whitespace around operator	default	`x+y`	2621
consecutive `if` statements	enhancement	`if c1:` `return True` `if c2:` `return True`	2464
`if return` simplifiable using logical operator	enhancement	`if c:` `return True` `return x`	2335
merge comparisons with `in`	Python-specific	`x == 'a' or x == 'b'`	2265
block comment should start with `#`	default	`using """ for block comment`	2214
use `isdecimal`	default	`val.isnumeric()`	2091
use `with` for resource allocation	enhancement	`file = open("/path")`	2050
do not use `is` with bool	default	`val is False`	1887
too many local variables	complexity	more than 15	1817
iterate with `items`	Python-specific	`for key in dct:` `print(key, dct[key])`	1332

Message	Extension	Defective code	Files with defect

Unnecessary `else` after `break`	default	`if c1:`
		`    break`
		`else:`
		`    # body`
			1326

syntax error	default		1322

use `else` instead of `elif`	default	`if x <= y:`
		`    # body`
		`elif x > y:`
		`    # body`
			1276

use f-string for formatting	Python-specific	using `%`, `+`, `join`, `format` or `Template` to format a string	1127
undefined name	default		1090
whitespace before `(`	default		1084
inline comment should start with `#`	default		1079
indentation is not a multiple of 4	default		1059
indentation contains tabs	default		1038
too many boolean expressions in an `if` statement	complexity	more than 5	1036

unnecessary `pass`	default	`def fun():`
		`    pass`
		`    # body`
			1023

redundant arithmetic	default	`val + []`	1016
unreachable code	default	code behind return or raise	1008
missing whitespace around `%`	default		1007
redundant operation	enhancement	`x * x * x`	1004
too many nested blocks	complexity	more than 5	996
unnecessary use of a comprehension, use `list/dict/set`	Python-specific	`{number for number in lst}`	976
use `is` for comparison to `None`	default	`v == None`	916
multiple statements on one line with colon	default	`if c: val = 0`	881
ambiguous variable name (`l`, `O`, `I`)	default		875
unexpected spaces around keyword / parameter equals	default	`def fun(key = val):`	874

import outside top-level	default	`def fun():`
		`    import module`
			824

Message	Extension	Defective code	Files with defect
whitespace before :	default		818
indentation contains mixed spaces and tabs	default		817
use {} instead of dict()	Python-specific		816
whitespace before)	default		780
local variable is assigned to but never used	default		744
whitespace after (	default		733

over-indented code	default	`if c:`
		`     # code indented`
		`     # by five spaces`
			709

use append	default	`lst += [val]`	685
use integer division.	default	`int(x / y)`	612
global variables	default		604

continuation line under-indented	default	`print("Hello",`
		`    "world")`
			542

use ord applied to a letter instead of using a magical constant	default	`ord(char) - 65`	539

empty if branch	default	`if c:`
		`    pass`
		`else:`
		`    # body`
			514

compare to a string literal instead of a magical constant	default	`ord(char) < 65`	495
simplifiable if expression	default	`True if c else False`	467
do not compare to bool	default	`c == True`	464
expected 1 blank line, found 0	default		393

do not use a loop with else	default	`for val in lst:`
		`    # body`
		`else:`
		`    # body`
			393

do not add elements to iterated structure or remove from it	default	`for val in lst:`
		`    # body`
		`    lst.remove(x)`
			382

use set comprehension	Python-specific	`set([fst for fst, _ in pairs])`	356
use [] instead of list()	Python-specific		322
missing whitespace around bitwise or shift operator	default		307

Message	Extension	Defective code	Files with defect
unexpectedly indented comment	default		298
use `max`	enhancement	`if longest < current:` `    longest = current`	290
use `def` instead of assigning `lambda`	default	`val = lambda x: x // 2`	290
multiple spaces before operator	default		288
iterate dictionary directly	Python-specific	`for key in dct.keys()`	256
statement ends with a semicolon	default		251
redefining argument with a local name	default	`def fun(val):` `    val = 0`	249
multiple spaces after keyword	default		248
statement with no effect	default	`val == 0`	234
do not use `while True` followed by `break`	default	`while True:` `    if c:` `        break` `    # body`	231
inner loop shadows the outer loop's control variable	enhancement	`for i in lst1:` `    for i in lst2:` `        # body`	218
multiple spaces after operator	default		197
simplifiable `if` with assignment	default	`if c:` `    v = True` `else:` `    v = False`	195
multiple imports on one line	default		185
assigning variable to itself	default	`val = val`	168
indentation is not a multiple of 4 before a comment	default		166
do not use bare `except`	default		162
boolean expression contains unneeded negation	default	`not val1 == val2`	154
unnecessary changing control variable	default	`for i in range(n):` `    # body` `    i += 1`	151
dangerous default value	default	`def fun(arg=[]):`	147

Message	Extension	Defective code	Files with defect
use `isinstance`	Python-specific	`type(val) == int`	138
invalid escape sequence	default	`"\d"`	137

use tuple unpacking for swapping variables	Python-specific	`tmp = val1`
		`val1 = val2`
		`val2 = tmp`
			129

continuation line over-indented	default	`print("Hello",`
		`           "world")`
			127

use the generator directly for `all`/`any`	Python-specific	`all([val > 3 for val in lst])`	108
too many leading `#` for block comment	default		107

use `join`	Python-specific	`result = ""`
		`for word in lst:`
		`    result += word`
			107

multiple spaces before keyword	default	`val  in lst`	98

unexpected indentation	default	`print("Hello")`
		`  print("world")`
			90

continuation line missing indentation or outdented	default	`print("Hello",`
		`"world")`
			90

expected an indented block	default	`if c:`
		`# body`
			85

test for membership should be `not in`	default	`not val in lst`	82
module level import not at top of file	default		79
wildcard import	default	`from module import *`	78
wildcard import used; unable to detect undefined names	default		78

use `min`	enhancement	`if shortest > current:`
		`    shortest = current`
			75

continuation line with the same indent as the next logical line	default	`if c1 \`
		`    or c2:`
		`    val = 0`
			73

blank lines after function decorator	default		71
method should have `self` as first argument	default	`def method(arg)`	69
undefined name	default		68
redefinition of unused variable	default		63

unnecessary dictionary index lookup	default	`for k, v in dct.items():`
		`    print(k, dct[k])`
			60

Message	Extension	Defective code	Files with defect

expected an indented block before comment	default	`def fun():`
		`# comment`
		`    return 0`
			58

multiple statements on one line with semicolon	default	`x = 0; y = 1`	52
expected 1 blank line before a nested definition	default		51

use tighter `range` boundaries	default	`for i in range(n):`
		`    if i == 0:`
		`        continue`
		`    # body`
			43

using variable before assignment	default		41

a backslash is redundant between brackets	default	`[`
		`    "Hello", \`
		`    "world"`
		`]`
			41

the closing bracket does not match visual indentation	default	`val = ["Hello", "world",`
		`]`
			40

`for` loop makes at most one iteration	default	`for i in range(1):`
		`    # body`
			34

redundant comparison	default	`val == val`	31

one branch is not returning a value	default	`if n < 0:`
		`    return`
		`return True`
			31

disallow trailing comma tuple	default	`val1, val2 = 0, 1,`	29
unused wildcard imports	default		28
function/method already defined	default		27
use `==`/`!=` to compare with constant literals	default	`val is " "`	27

`if` assignment simplifiable using logical operator	enhancement	`if c:`
		`    v = False`
		`else:`
		`    v = x`
			25

reimported module	default		24

continuation line unaligned for hanging indent	default	`[`
		`    "Hello",`
		`      "world"`
		`]`
			21

visually indented line with the same indent as the next logical line	default	`if (c1`
		`    or c2):`
		`    val = 0`
			20

Message	Extension	Defective code	Files with defect

unreachable `else`	default	`if x <= y:`
		`    # body`
		`elif x > y:`
		`    # body`
		`else:`
		`    # unreachable`
			19

`if` expression simplifiable using logical operator	enhancement	`x if c else True`	14
indentation error	default		14
disallowed name	default	`foo`, `bar`	13
f-string with no placeholder	default	`f"Hello world"`	9
`for` loop does not target a name	default	`for lst[i] in range(n):`	6
dictionary key repeated with different values	default	`{0: "Hello, 0: "world"}`	6
`return` outside function	default		5
multiplying a list containing a mutable data structure	default	`[[]] * 5`	5
`<>` is deprecated, use `!=`	default		5
missing whitespace around parameter equals	default	`def fun(val: int=None)`	3
test for object identity should be `is not`	default	`not val is None`	3

a local variable referenced before assignment	default	`val = 0`
		`def fun():`
		`    val += 1`
			1

ambiguous function name (`l`, `O`, `I`)	default		1
do not alias import with same name	default	`import package.sub as sub`	1
`has_key()` is deprecated, use `in`	default		1
do not compare types	default	`type(val) == type([])`	1

Bibliography

1. KIRK, Diana; CROW, Tyne; LUXTON-REILLY, Andrew; TEMPERO, Ewan. On Assuring Learning About Code Quality. In: *Proceedings of the Twenty-Second Australasian Computing Education Conference*. Melbourne, VIC, Australia: Association for Computing Machinery, 2020, pp. 86–94. ACE'20. ISBN 9781450376860. Available from DOI: `10.1145/3373165.3373175`.
2. SCHACH, S. *Object-Oriented and Classical Software Engineering*. McGraw-Hill Publishing, 2010. ISBN 9780077417987.
3. BÖRSTLER, Jürgen; STÖRRLE, Harald; TOLL, Daniel; ASSEMA, Jelle van; DURAN, Rodrigo; HOOSHANGI, Sara; JEURING, Johan; KEUNING, Hieke; KLEINER, Carsten; MACKELLAR, Bonnie. "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*. Bologna, Italy: Association for Computing Machinery, 2018, pp. 70–85. ITiCSE-WGR '17. ISBN 9781450356275. Available from DOI: `10.1145/3174781.3174785`.
4. BIRILLO, Anastasiia; VLASOV, Ilya; BURYLOV, Artyom; SELISHCHEV, Vitalii; GONCHAROV, Artyom; TIKHOMIROVA, Elena; VYAHHI, Nikolay; BRYKSIN, Timofey. Hyperstyle: A Tool for Assessing the Code Quality of Solutions to Programming Assignments. *CoRR*. 2021, vol. abs/2112.02963. Available from arXiv: `2112.02963`.
5. ROY CHOUDHURY, Rohan; YIN, Hezheng; FOX, Armando. Scale-Driven Automatic Hint Generation for Coding Style. In: MICARELLI, Alessandro; STAMPER, John; PANOURGIA, Kitty (eds.). *Intelligent Tutoring Systems*. Cham: Springer International Publishing, 2016, pp. 122–132. ISBN 978-3-319-39583-8.
6. KEUNING, Hieke; HEEREN, Bastiaan; JEURING, Johan. How Teachers Would Help Students to Improve Their Code. In: *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*. Aberdeen, Scotland Uk: Association for Computing Machinery, 2019, pp. 119–125. ITiCSE '19. ISBN 9781450368957. Available from DOI: `10.1145/3304221.3319780`.

7. DE RUVO, Giuseppe; TEMPERO, Ewan; LUXTON-REILLY, Andrew; ROWE, Gerard B.; GIACAMAN, Nasser. Understanding Semantic Style by Analysing Student Code. In: *Proceedings of the 20th Australasian Computing Education Conference*. Brisbane, Queensland, Australia: Association for Computing Machinery, 2018, pp. 73–82. ACE '18. ISBN 9781450363402. Available from DOI: 10.1145/3160489.3160500.

8. STEGEMAN, Martijn; BARENDSEN, Erik; SMETSERS, Sjaak. Designing a Rubric for Feedback on Code Quality in Programming Courses. In: *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*. Koli, Finland: Association for Computing Machinery, 2016, pp. 160–164. Koli Calling '16. ISBN 9781450347709. Available from DOI: 10.1145/2999541.2999555.

9. NUTBROWN, Stephen; HIGGINS, Colin. Static analysis of programming exercises: Fairness, usefulness and a method for application. *Computer Science Education*. 2016, vol. 26, no. 2-3, pp. 104–128. Available from DOI: 10.1080/08993408.2016.1179865.

10. HRISTOVA, Maria; MISRA, Ananya; RUTTER, Megan; MERCURI, Rebecca. Identifying and Correcting Java Programming Errors for Introductory Computer Science Students. In: *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*. Reno, Navada, USA: Association for Computing Machinery, 2003, pp. 153–156. SIGCSE '03. ISBN 158113648X. Available from DOI: 10.1145/611892.611956.

11. BLAU, Hannah; MOSS, J. Eliot B. FrenchPress Gives Students Automated Feedback on Java Program Flaws. In: *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*. Vilnius, Lithuania: Association for Computing Machinery, 2015, pp. 15–20. ITiCSE '15. ISBN 9781450334402. Available from DOI: 10.1145/2729094.2742622.

12. UREEL, Leo C.; WALLACE, Charles. WebTA: Automated iterative critique of student programming assignments. In: *2015 IEEE Frontiers in Education Conference (FIE)*. 2015, pp. 1–9. Available from DOI: 10.1109/FIE.2015.7344225.

13. MOGHADAM, Joseph Bahman; CHOUDHURY, Rohan Roy; YIN, HeZheng; FOX, Armando. AutoStyle: Toward Coding Style Feedback at Scale. In: *Proceedings of the Second (2015) ACM Conference on Learning @ Scale*. Vancouver, BC, Canada: Association for Computing Machinery, 2015, pp. 261–266. L@S '15. ISBN 9781450334112. Available from DOI: `10.1145/2724660.2728672`.

14. KEUNING, Hieke; HEEREN, Bastiaan; JEURING, Johan. A Tutoring System to Learn Code Refactoring. In: *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 562–568. SIGCSE '21. ISBN 9781450380621. Available from DOI: `10.1145/3408877.3432526`.

15. LIU, Xiao; WOO, Gyun. Applying Code Quality Detection in Online Programming Judge. In: *Proceedings of the 2020 5th International Conference on Intelligent Information Technology*. Hanoi, Viet Nam: Association for Computing Machinery, 2020, pp. 56–60. ICIIT 2020. ISBN 9781450376594. Available from DOI: `10.1145/3385209.3385226`.

16. LIU, David; PETERSEN, Andrew. Static Analyses in Python Programming Courses. In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. Minneapolis, MN, USA: Association for Computing Machinery, 2019, pp. 666–671. SIGCSE '19. ISBN 9781450358903. Available from DOI: `10.1145/3287324.3287503`.

17. EFFENBERGER, Tomáš; PELÁNEK, Radek. Code Quality Defects across Introductory Programming Topics. In: *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education - Volume 1*. Providence, RI, USA: Association for Computing Machinery, 2022, pp. 941–947. SIGCSE 2022. ISBN 9781450390705. Available from DOI: `10.1145/3478431.3499415`.

18. KEUNING, Hieke; HEEREN, Bastiaan; JEURING, Johan. Code Quality Issues in Student Programs. In: *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*. Bologna, Italy: Association for Computing Machinery, 2017, pp. 110–115. ITiCSE '17. ISBN 9781450347044. Available from DOI: `10.1145/3059009.3059061`.

19. GROENEVELD, Wouter; MARTIN, Dries; PONCELET, Tibo; AERTS, Kris. Are Undergraduate Creative Coders Clean Coders? A Correlation Study. In: *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education - Volume 1*. Providence, RI, USA: Association for Computing Machinery, 2022, pp. 314–320. SIGCSE 2022. ISBN 9781450390705. Available from DOI: `10.1145/3478431.3499345`.

20. BROWN, Neil C.C.; ALTADMRI, Amjad. Investigating Novice Programming Mistakes: Educator Beliefs vs. Student Data. In: *Proceedings of the Tenth Annual Conference on International Computing Education Research*. Glasgow, Scotland, United Kingdom: Association for Computing Machinery, 2014, pp. 43–50. ICER '14. ISBN 9781450327558. Available from DOI: `10.1145/2632320.2632343`.

21. *Pylint* [online]. [visited on 2023-05-06]. Available from: `https://pypi.org/project/pylint/`.

22. *Pyflakes* [online]. [visited on 2023-05-06]. Available from: `https://pypi.org/project/pyflakes/`.

23. *pycodestyle* [online]. [visited on 2023-05-06]. Available from: `https://pypi.org/project/pycodestyle/`.

24. *Flake8* [online]. [visited on 2023-05-06]. Available from: `https://pypi.org/project/flake8/`.

25. *wemake-python-styleguide* [online]. [visited on 2023-05-06]. Available from: `https://pypi.org/project/wemake-python-styleguide/`.

26. *Ruff* [online]. [visited on 2023-05-06]. Available from: `https://pypi.org/project/ruff/`.

27. *Thonny: Python IDE for beginners* [online]. [visited on 2023-05-06]. Available from: `https://thonny.org/`.

28. *Hyperstyle repository* [online]. [visited on 2023-05-06]. Available from: `https://github.com/hyperskill/hyperstyle`.

29. BIRILLO, Anastasiia; VLASOV, Ilya; BURYLOV, Artyom; SELISHCHEV, Vitalii; GONCHAROV, Artyom; TIKHOMIROVA, Elena; VYAHHI, Nikolay; BRYKSIN, Timofey. *Supplementary materials for the paper "Hyperstyle : A Tool for Assessing the Code Quality of Solutions to Programming Assignments"*. Zenodo, 2021. Version 1.0. Available from DOI: `10.5281/zenodo.5749825`.

30. EDWARDS, Stephen H.; KANDRU, Nischel; RAJAGOPAL, Mukund B.M. Investigating Static Analysis Errors in Student Java Programs. In: *Proceedings of the 2017 ACM Conference on International Computing Education Research*. Tacoma, Washington, USA: Association for Computing Machinery, 2017, pp. 65–73. ICER '17. ISBN 9781450349680. Available from DOI: `10.1145/3105726.3106182`.

31. BREUKER, Dennis M.; DERRIKS, Jan; BRUNEKREEF, Jacob. Measuring Static Quality of Student Code. In: *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*. Darmstadt, Germany: Association for Computing Machinery, 2011, pp. 13–17. ITiCSE '11. ISBN 9781450306973. Available from DOI: `10.1145/1999747.1999754`.

32. PETTIT, Raymond; HOMER, John; GEE, Roger; MENGEL, Susan; STARBUCK, Adam. An Empirical Study of Iterative Improvement in Programming Assignments. In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. Kansas City, Missouri, USA: Association for Computing Machinery, 2015, pp. 410–415. SIGCSE '15. ISBN 9781450329668. Available from DOI: `10.1145/2676723.2677279`.

33. *WPS documentation of ExplicitStringConcatViolation* [online]. [visited on 2023-05-07]. Available from: `https://wemake-python-styleguide.readthedocs.io/en/0.17.0/pages/usage/violations/consistency.html#wemake_python_styleguide.violations.consistency.ExplicitStringConcatViolation`.