Code Quality Defects Across Introductory Programming Topics

Tomáš Effenberger
tomas.effenberger@mail.muni.cz
Masaryk University
Brno, Czech Republic

Radek Pelánek
xpelanek@fi.muni.cz
Masaryk University
Brno, Czech Republic

ABSTRACT

Research on feedback in introductory programming focuses mostly on incomplete and incorrect programs. However, most of the functionally correct programs also contain defects that call for feedback. We analyzed 114,000 solutions to 161 short coding problems in Python and compiled a catalog of 32 defects in code quality. We found that most correct solutions contain some defects and that students do not stop making them if they do not receive targeted feedback. The catalog of defects, together with their prevalence across common topics like expressions, loops, and lists, informs educators which defects to address in which lectures and guides the development of exercises on code quality. Additionally, we describe defect detectors, which can be used to generate valuable feedback to students automatically.

CCS CONCEPTS

• **Applied computing** → **Education**; • **Social and professional topics** → **CS1**.

KEYWORDS

introductory programming; Python; code quality; feedback

ACM Reference Format:
Tomáš Effenberger and Radek Pelánek. 2022. Code Quality Defects Across Introductory Programming Topics. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2022), March 3–5, 2022, Providence, RI, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3478431.3499415

1 INTRODUCTION

The ability to solve programming problems is not the same as the ability to solve them well [8, 14]. In our experience from teaching a CS1 course and developing an online learning environment, most students solve the assigned problems—but the code quality is poor. Consider the following problem: "Write a function `impose_fine(age, beer)` that returns `True` if a person under 18 drinks beer." The problem is easy to solve; the success rate in our learning environment is over 90%. It is not, however, easy to solve *well*; a typical solution contains redundant if-else and redundant comparison to `True`:

```
def impose_fine(age, beer):
    if age < 18 and beer == True:
        return True
    else:
        return False
```

Even after solving many programming problems, students still commit these two defects. Without feedback on code quality, the students might not be aware of the mistakes they make. Solving more problems improves their ability to solve programming problems, but the code quality can remain poor. The students learn to write code that is functionally correct but difficult to read and extend. We thus focus on the code quality of functionally correct solutions and use the term *defect* to describe any imperfect parts of code for which it is possible to provide some feedback or advice.

Professional code analysis tools do not provide helpful feedback to novices [8, 15]. Despite the importance of feedback on code quality, research on automated feedback in introductory programming exercises has been predominantly focused on incorrect programs [18]. Many studies investigated automatic error detection [2, 5, 10], providing hints [12, 20, 22], and identifying misconceptions related to incorrect solutions [13, 23, 24].

Some studies did explore the code quality of correct solutions in introductory programming. Typically, they analyzed just a few problems [15, 17, 19, 25], a few general issues like duplicate and unused code [1], or measures of code quality like the number of lines and cyclomatic complexity [3, 4]. Only two studies analyzed the prevalence of specific defects in large datasets, both from introductory programming courses in Java [8, 14]. All these studies reported the prevalence either in the individual problems or in the whole dataset; unknown is the prevalence in individual topics.

The aim of this study is to broaden our knowledge about code quality defects in introductory programming by answering the following research questions: *Which defects appear in correct solutions to short coding problems in Python? How are they distributed across problems and topics?* To answer these questions, we analyzed 114,000 correct solutions to short programming problems and compiled a catalog of 32 defects. We describe automatic detectors and the prevalence of these defects across diverse problem sets covering common topics taught in CS1.

The catalog can be directly used by educators as an inspiration for what to discuss and highlight in lectures. It also facilitates the development of refactoring exercises and programming problems focused on prevalent defects. Developers of learning environments can use the catalog and detectors to automatically generate feedback to students or to measure performance beyond binary correctness [21]. Researchers may use our catalog as a starting point for building an even broader, more general and systematic taxonomy.

SIGCSE 2022, March 3–5, 2022, Providence, RI, USA

ACM ISBN 978-1-4503-9070-5/22/03…$15.00
https://doi.org/10.1145/3478431.3499415

2 METHOD

To compile the catalog of defects, we used an extensive dataset of student solutions and processed it in two phases (exploratory and confirmatory).

2.1 Data Collection

We used data from an online learning environment for introductory programming (`umimeprogramovat.cz`), which is used by both high school and university students. The environment provides a standard interface for solving short coding problems directly in the browser. Students can run their code to compare its output to the expected results. They get feedback only on the correctness, not code quality.

The environment offers 161 problems divided into 17 problem sets, which cover most topics typically included in first university programming courses (CS1). There are two types of exercises: Classic and Turtle. In Python Classic, students implement a function that computes or prints something and which is tested on hidden data. In Turtle Graphics, students write a program that draws a specified picture using the turtle graphics concept [7]. All the problems can be solved with 2–20 lines of code; median times range from 1 to 20 minutes. Many problems contain some scaffolding code, often just a function header, but sometimes also a more extensive code to modify or extend.

In this study, we consider only correct solutions. Together, there are 114,000 solutions from 11,000 students collected during a period of 2.5 years. The distribution across problems is highly skewed towards the easier ones, ranging from several thousand to just a few tens of solutions per problem.

2.2 Defects Detection

The analysis had two phases: exploratory and confirmatory. In the exploratory phase, we used generic, imprecise detectors to learn which defects exist in the data and collect their examples. In the confirmatory phase, we developed specific, accurate detectors to capture these defects.

Exploratory detectors can be indiscriminate, i.e., not distinguishing between different defects, and overgenerating, i.e., reporting false positives. We employed existing code style checkers designed for real-world software development [11], custom detectors of duplicate code, and an interpretable clustering technique [9] that uses all submitted solutions to reveal common inappropriate approaches and missing programming constructs. In addition, we have considered defects described in existing studies of code quality [6, 8, 14, 17], but only some were applicable to short Python programs.

In the second phase, we implemented confirmatory detectors. In contrast to the exploratory detectors, confirmatory detectors should not report any false positives to avoid overestimation of the prevalence and misleading feedback if used in a learning environment. The confirmatory detectors are functions that operate over abstract syntax trees. For example, to detect *augmentable assignment* (`x = x + 1`), we inspect all non-augmented assignment nodes with binary operation on the right-hand side. The assignment is augmentable in two cases: (1) the left child is equivalent to the target variable (`x = x - 2`), (2) the right child is equivalent to the target variable and the operation is commutative, i.e., either multiplication or addition in case of numbers (`x = 2 + x`).

All defects are listed in Table 1. For most of them, the implementation is a straightforward translation from the general description in Table 1. Table 2 provides the details needed to replicate our detectors.

3 RESULTS

Table 1 lists the 32 defects that we detected, grouped by the main related programming concept. The last column indicates possible feedback to students, formulated for brevity as an instruction to follow in order to remove the defect. In practice, the feedback could be longer, explaining why is the current code undesirable and how is the suggested modification better.

3.1 Defects in Specific Problems

For any problem, we can apply the automatic detectors to find out what are the common defects in code quality. In the Introduction, we have mentioned the problem *Impose Fine*. From over 1000 correct solutions, 90% contain a defect. The most common defects are *redundant if-else* (58%), *redundant comparison* (56%), and *inappropriate whitespace* (42%). In total, we have detected 12 different defects, although half of them appear just once or twice.

As another example, consider a problem to write a function that counts the occurrences of letters 'a' and 'A' in a text. From over 400 solutions, 74% contain a defect. The three most common defects are *inappropriate whitespace* (35%), *mergeable equal* (33%), and *for loop with redundant indexing* (25%), all present in the following solution:

```
def count_a(text):
    a=0
    for i in range(len(text)):
        if text[i]=="a" or text[i]=="A":
            a+=1
    return a
```

Other detected defects include *misleading name* (18%, `for i in text`), *augmentable assignment* (16%, `a = a + 1`), *one-letter name* (5%), *duplicate if* (4%), and 6 rare ones.

We could continue analyzing the 159 other problems one by one. Instead, we leverage a large number of problems and climb one level of abstraction up by grouping similar problems practicing the same topic. Doing this gives us more generally applicable insights.

3.2 Prevalence across Topics

Table 3 shows the prevalence of each defect across 17 problem sets. The most prevalent defect is *inappropriate whitespace*, which is arguably also the least serious one. More serious and still rather common in both Classic and Turtle exercises are *duplicate blocks* and *one-letter names*, but even these are not universally prevalent in all problem sets.

The prevalence of all defects is highly non-uniform and depends on the type of exercise, problem set, and even problem within the problem set. First, let us examine the differences between the two exercises (Classic and Turtle). In Turtle Graphics, problems do not use conditional statements, so the related defects, such as *duplicate if*, cannot occur. *Augmentable assignments* and *duplicate expressions*, which are common in Classic problems, also do not appear in most

Table 1: Overview of defects.

Expressions			
augmentable assignment	Variable update expressible as augmented assignment.	`x = x + 4`	Simplify to `x += 4`.
duplicate expression	Repeated occurrences of a complex expression.	`if a[n//2] % 2 == 0:` `  print(a[n//2] % 2)`	Avoid duplicate code by using a variable.
absolute value	Compound expression equivalent to absolute value.	`x < 4 and x > -4`	Simplify condition using `abs`.
repeated multiplication	Repeated multiplication of the same expression.	`y = x*x*x`	Simplify using power (`x ** 3`).
repeated addition	Repeated addition of the same expression.	`y = x+x+x`	Simplify using multiplication (`3 * x`).
redundant arithmetic	Identity operation or constant-valued expression.	`1*x, x/x`	Simplify to `x`. Simplify to `1`.
Conditions			
redundant if-else	If-else statement directly returning a boolean.	`if c: return True` `else: return False`	Simplify to `return c`.
mergeable equal	Comparing the same expression to multiple values.	`c == 'a' or c == 'A'`	Simplify to `c in 'aA'`.
redundant comparison	Comparing boolean expression to a boolean constant.	`expr == True`	Remove redundant comparison to `True`.
redundant not	Boolean expression with negation that can be applied.	`not a <= b`	Simplify to `a > b`.
If			
duplicate if	Consecutive if statements with the same body.	`if c1: print(x)` `elif c2: print(x)`	Avoid duplicate code by joining conditions with `or`.
else if	Else clause containing just single if-else block.	`else:` `  if c: print(x)`	Simplify by using elif instead of else if.
nested if	Nested if-statements avoidable using conjunction.	`if c1:` `  if c2: ...`	Avoid unnecessary nesting by joining conditions with `and`.
redundant elif	If clause with a condition that is always true.	`if x <= y: print(1)` `elif x > y: print(2)`	Simplify by using `else` instead of `elif`.
empty if	Redundant branch containing just `pass`.	`if c: pass` `else: ...`	Simplify by inverting the condition.
Loops			
duplicate sequence	Sequence of statements that are same or differ in a single value.	`print(1); print(2);` `print(3); print(4);`	Avoid duplicate code by using `for` loop.
for with redundant indexing	Iterating through indices, accessing (not modifying) items at the current index only.	`for i in range(len(s)):` `  print(s[i])`	Simplify by iterating directly through elements (`for char in s`).
while as for	While loop with the number of iterations known before its start.	`while i <= n:` `  print(i); i += 1`	Use `for` loop when the number of iterations is known.
redundant for	For loop with zero or single iteration.	`for i in range(1): ...`	Remove the unnecessary `for` loop.
Functions			
duplicate block	Two very similar blocks of code.	`if c: f(1); g(1); f(1)` `else: f(2); g(2); f(2)`	Avoid duplicate code by using a function.
long script	Program with more than 20 lines outside functions.		Use functions to increase readability.
long function	Function with more than 20 lines.		Decompose into multiple shorter functions.
unused parameter	Function parameter is never used in the body.	`def f(x, y):` `  print(x)`	Remove unused parameter (or use it to generalize the behavior).
unused function	Function that is defined but never called.		Remove unused function.
Variables			
one-letter name	One-letter variable name (with exceptions like `i`).	`l = 4`	Use more descriptive variable name.
misleading name	Variable `i`/`j` used to traverse items, not indices.	`for i in text: ...`	Avoid using name `i` for anything but index.
built-in name	Variable name that is name of a built-in function.	`list = [3, 4]`	Avoid using built-in names for variables.
unused variable	Variable that is defined but never read.		Remove unused variable.
Miscellaneous			
inappropriate whitespace	Violating PEP 8 conventions about whitespace (only the universally accepted rules).	`print( x, y )`	Remove the space after `(` and before `)` to adhere to conventions.
long line	Line with more than 100 characters.		Avoid too long lines; they are difficult to read.
unreachable code	Part of source code that is never executed.	`return x; print(x)`	Remove unreachable code after `return`.
unused import	Module is imported but unused.		Remove unused module import.

Table 2: Technical details of the used defect detectors.

Duplicate expression: expressions with at least 8 weighted tokens, giving arithmetic operators weight 2.
Absolute value: reference solution contains more absolute values than the student solution and the student solution contains a compound condition where the conditions are both equalities, strict inequalities, or non-strict inequalities.
Repeated multiplication: at least 3rd power (i.e., allowing `x * x`).
Mergeable equal: expression at least 20 characters long.
Nested if: there are two cases where the if statements can be unnested using compound condition: (1) The last clause in the outer if contains just single if without elif or else. (2) Penultimate clause in the outer if contains single if-else and the body of the inner else is same as the body of the outer else.
Redundant elif: complementary comparisons, e.g., `x < y` followed by `x >= y` or `n % 2 == 0` followed by `n % 2 == 1`.
Duplicate sequence: at least 3 statements if it can be rewritten using for-in-range, or at least 4 if it requires iterating through a collection of values. We exclude problems from problem sets before introducing loops.
For with redundant indexing: for-in-range-len statements where each occurrence of the loop variable is reading (not writing to) item from the collection on this current index.
While as for: while loop can be replaced by a for loop if the test is a comparison that contains exactly one variable that is updated by a constant in each iteration, and all the other variables that appear in the test are not updated in the body. We exclude loops containing `break` or `return`.
Duplicate block: at least 3 lines long, have the same number of tokens, and differ in at most 3 tokens.
One-letter name: excluding names that are universally accepted as appropriate for some use cases and also all that appear in at least one scaffolding (`_abcijkmnpxyz`).
Unused variable: excluding loop counters.
Unreachable code: lines after `return`, `break`, or `continue`, if statements with conditions that are always false, and also subsequent branches after a condition that is always true.

Turtle problems. On the other hand, *duplicate sequences* and *long scripts* are much more common in Turtle than in Classic problems.

Even within each type of exercise, the variability between problem sets is enormous. *Redundant if-else* occurs in every other solution in the Logic problem set but much less frequently elsewhere. The Logic problem set is also abundant with *redundant comparisons* and *duplicate if statements*. As one more example, consider *misleading names* and *for loops with redundant indexing*. These defects are rare in most problem sets but prevalent when practicing strings and lists.

One could expect loop-related defects to occur dominantly in the loop-related problem sets, function-related defects in function-related problem sets, etc. However, few defects meet this expectation. Without feedback on code quality, students do not stop making the defect just after some practice. The prevalence fluctuates as there might be more or fewer opportunities for the particular defect, but for most defects, their prevalence tend to increase, not decrease. Many students still use *augmentable assignment*, *redundant comparison*, and *while loops instead of for loops* in the last problem set.

Table 3 hides the variability between problems in a single problem set. Some defects are highly localized, i.e., they only occur in a few specific problems, but in these problems, the prevalence is high. A typical example is an *expression simplifiable to absolute value*: in the Logic problem set, it appears in just 2 from the 10 problems, with the prevalence of 91% and 65%.

3.3 Aggregated Defects

We can climb one more level of abstraction up by grouping similar defects into more general types of defects, such as duplicate and unused code. Table 4 shows the prevalence of several aggregated groups of defects. *Poor formatting* comprises *inappropriate whitespace* and *long line*; *long code* is either *long function* or *long script*; *duplicate* can be *block, sequence, if*, or *expression*; *poor names* can be *one-letter, misleading*, or *built-in*; *unused code* can be *function, parameter, variable, import*, or *unreachable code*; and, finally, unnecessarily *complex code* that can be simplified by a local modification includes the rest of the defects.

In an average problem, every other solution contains a defect. However, many of those defects are just formatting issues. Excluding the formatting issues, about a third of solutions have a defect. Locally *complex code* is prevalent in Classic problems, *long code* in Turtle Graphics, and *duplicate code* in both exercises. *Poor variable names* are not frequent initially but become frequent in more difficult problems. *Unused code* is rather rare in our data.

Intriguingly, *duplicate code* is rare in Classic problem set on for loops, although it is prevalent in the similar problem set in Turtle Graphics. This phenomenon can be partially attributed to a major difference between the two exercises: Classic problems require writing a general function, while Turtle Graphics asks for one specific picture, so it is possible to avoid the loop by a (possibly long) sequence of commands. There are, for sure, other factors at play, such as the choice of specific problems and their scaffolding.

The first few problem sets have few defects since there is not much room for them. With more difficult problems, the aggregate prevalence tends to increase. Students might be more skillful, but longer programs offer more opportunities for various defects. There are a few exceptions to this general trend, the most prominent being the Logic problem set, where over 80% of solutions are defected, more than in any other problem set.

3.4 Effect of Scaffolding Intervention

In our learning environment, we have investigated the potential of implicit scaffolding to eliminate defects. Observing the extreme prevalence of *redundant if-else*, we added three problems with scaffolding that demonstrated returning the value of logic expression directly without an if statement. Figure 1 compares the prevalence before and after this intervention. It reduced the prevalence from about 70% to about 40% in the original problems (which we left unchanged). This is a significant decrease, but 40% is still a lot. Even in the scaffolded problems, 10–20% of students delete the provided scaffolding and use if statement.

Table 3: Prevalence (%) of defects across problem sets. Read empty cells as 0%. Abbreviated problem sets: Expr. = Expressions, Modif. = Code Modifications., Num. = Computations with Numbers, ASCII = ASCII art, Func. = Functions, Vars. = Variables.

	Python Classic										Python Turtle							
	Expr.	Logic	If	For	Modif.	Num.	ASCII	Strings	Lists	Bonus	Basics	Loops	Func.	Angles	Patterns	Vars.	Fractals	avg.
augment. assignment	0.2		8.4	10.9	1.8	14.3	2.6	9.9	7.2	17.5			0.2			7.1	2.1	4.8
duplicate expression	0.3	0.1	0.9		1.9	1.8	5.6	6.9	12.3	9.5						2.2	1.8	2.6
absolute value		15.6					6.1			2.7								1.4
repeated multiplication	5.0																	0.3
repeated addition	0.5		0.2	0.2	0.8		0.2	2.2							0.3	0.2		0.3
redundant arithmetic			0.1	0.1	0.4	1.8	0.1	0.4	0.3							0.3		0.2
redundant if-else		47.9			0.1	7.4	0.1	3.8	3.8	1.4								3.8
mergeable equal		1.2	0.2			0.1	9.7	3.6	1.3	4.4								1.2
redundant comparison		5.7			0.1	1.4	0.1	0.8	0.4	2.7								0.7
redundant not		0.2	0.1	0.3	0.1		0.1	0.3	0.6	0.3								0.1
duplicate if		12.1	3.7	0.5	1.0	1.3	3.4	0.8	0.6	7.7								1.8
else if		1.6	4.1	0.2	0.1	0.3	2.0	0.8	0.4	3.5								0.8
nested if		1.3	0.6	0.3	0.1	2.2	0.4	0.5	0.8	3.3								0.6
redundant elif		0.4	1.6	0.4	0.1		0.7	0.9	0.3	0.6								0.3
empty if				0.8				0.2	0.9	0.3								0.1
duplicate sequence					0.1			0.6				15.3	13.6	12.5	18.8	10.5	4.0	4.4
redundant indexing					0.1	1.5		8.6	16.2	4.8								1.8
while as for				1.9		1.8	0.5	1.9	0.7	1.8			0.1			0.4		0.5
redundant for				0.3	0.1		0.6		0.2			1.1	0.8	1.3	3.1	1.3		0.5
duplicate block		2.1	6.2	0.5	8.8	1.3	4.1	6.1	0.1	12.1		11.7	12.7	0.7	14.4	6.5	5.8	5.5
long script											0.1	13.1	3.6	8.1	14.3	8.6	4.7	3.1
long function		0.1	0.9	0.1	0.1	0.1	1.1	0.3	0.4	10.3			0.2		0.1	0.7	8.3	1.3
unused parameter												0.2	3.1	0.3	4.6	1.6		0.6
unused function				0.1				0.4					0.5		0.3	0.1	2.3	0.2
one-letter name	1.3	1.7	0.9	6.7	1.0	5.6	4.5	5.2	6.2	13.7		0.7	3.5	0.8	3.6	5.2	14.3	4.4
misleading name				0.1	0.1	0.7		7.4	8.0	4.3								1.2
built-in name	0.1		0.2	1.2		3.0	0.1	0.3	3.9	3.3					0.1	0.1	2.3	0.9
unused variable	0.1	0.1	0.2	0.4	0.2	0.8	0.7	1.5	2.3	4.7			0.1	0.1		1.5		0.7
inapprop. whitespace	33.9	46.6	49.9	50.8	42.0	56.5	49.6	42.8	35.3	58.9	0.3	2.1	6.6	3.7	8.0	27.4	56.3	33.6
long line		0.1	0.8	0.1		0.1	0.3	0.6	0.3	3.1								0.3
unreachable code		0.5		0.1		0.2		0.2	0.1	0.2								0.1
unused import										0.1						0.1		0.0

Table 4: Prevalence (%) of aggregated groups of defects.

	Python Classic										Python Turtle							
	Expr.	Logic	If	For	Modif.	Num.	ASCII	Strings	Lists	Bonus	Basics	Loops	Func.	Angles	Patterns	Vars.	Fractals	avg.
poor formatting	33.9	46.6	50.3	50.8	42.0	56.5	49.9	42.8	35.5	59.5	0.3	2.1	6.6	3.7	8.0	27.4	56.3	33.7
complex code	5.7	60.7	14.9	14.4	3.6	27.9	20.7	26.9	28.9	35.1	0.0	1.1	1.2	1.3	3.4	8.8	2.1	15.1
duplicate code	0.4	14.2	10.7	0.9	11.4	3.9	12.3	13.5	13.0	24.0	0.0	24.5	23.9	13.1	30.5	17.8	11.2	13.3
poor names	1.4	1.7	1.2	8.0	1.1	8.9	4.6	11.6	17.0	20.0	0.0	0.7	3.5	0.8	3.7	5.3	16.5	6.2
long code	0.0	0.1	0.9	0.1	0.1	0.1	1.1	0.3	0.4	10.3	0.1	13.1	3.8	8.1	14.3	9.2	13.0	4.4
unused code	0.1	0.5	0.2	0.6	0.3	1.1	0.8	2.0	2.4	4.9	0.0	0.2	3.7	0.4	4.9	3.2	2.3	1.6
all incl. formatting	39.4	81.9	63.8	58.3	50.9	72.6	63.4	67.7	64.8	80.0	0.5	30.4	35.3	21.6	45.4	50.0	70.4	52.7
all excl. formatting	7.6	62.3	22.8	20.2	15.6	35.7	32.0	46.4	49.6	62.5	0.2	28.6	31.3	18.0	39.5	30.5	32.8	31.5

4 DISCUSSION

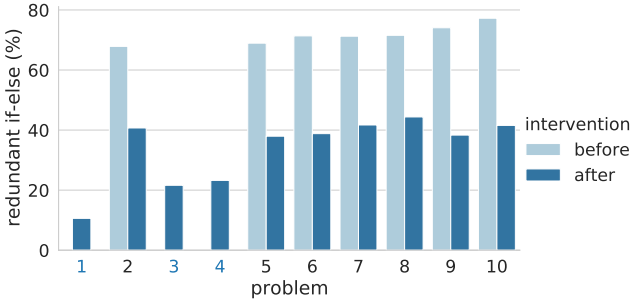
Research on feedback in introductory programming focuses on incorrect programs, but most of the correct programs also contain defects that call for feedback. We created a catalog of 32 defects and analyzed their prevalence in introductory programming problems. In general, defects are prevalent and the prevalence increases with problem difficulty, but there is large variability in prevalence between topics.

4.1 Implications for Educators and Learning Environments

The defects are often symptoms of missing knowledge and do not disappear with practice without targeted feedback or instruction. We should, therefore, teach the students to recognize and fix the defects.

Table 3 indicates when to expect which defects to occur and which are likely to be prevalent enough to be worth whole-class

Figure 1: Prevalence of redundant if-else in the Logic problem set before and after addition of three problems with scaffolding targeting this defect (problems 1, 3, 4).



discussion, attention of teaching assistants during computer labs, and possibly even a targeted exercise. For example, in a lesson on logic expressions and if statements, it is worth highlighting *redundant if-else*, *redundant comparison to* `True`, and *duplicate if*. In a lesson on strings and lists, important defects to highlight are *misleading names* and *for with redundant indexing*. In a computer lab using Turtle Graphics, we should pay special attention to *duplicate code*.

Virtual learning environments can also be improved by considering the occurrence of defects. Many learning environments give just correctness feedback. This may mislead students into a belief that their solution is appropriate even when it is not, e.g., when they use while loop instead of for loop. Over time, the missing feedback on code quality can reinforce undesirable coding habits.

One remedy is to use the presented defect detectors and give students short feedback on the committed defects (see the right-most column in Table 1). There are also other interventions to consider, such as worked examples, code scaffolding, evaluation based on code quality, and refactoring exercises [16]. Which interventions are effective in which context is yet to be discovered. Preliminary work suggests that reducing defects prevalence is not easy: A targeted tutoring system increased code quality when working inside but not outside the tutoring system [25]. Using professional code analysis tools was not associated with a lower prevalence of defects in students' programs [14]. Our experiment with scaffoldings brought improvement in students' codes, but not as high as we expected.

4.2 Methodological Caveats and Limitations

The defect prevalences need to be interpreted with caution. We have already discussed variability between exercises, topics, and problems. The prevalences thus cannot be expected to transfer to vastly different contexts, as can be confirmed by consulting previous research [1, 14].

In a popular open-ended block-based programming environment Scratch, duplicate code was detected in 26% programs and unused code in 28% [1], much more than in our data. In Java programs from the first four weeks of an academic year, the duplicate code was detected in just 5% programs, while unused code in 27% [14]. These aggregates might be heavily influenced by the precise definitions of duplicate and unused code, but the difference is also present in the prevalence of specific defects; *unused variable* and *unused import* were observed in 16% and 20% of Java programs, while only 0.7% and 0.02% in our data. Even more importantly, most of the individual defects that they detect are not applicable to our context at all since they are specific to Java or longer object-oriented programs.

Even with the same programming problems, the prevalence would change for a different student population, learning environment, or defect definitions. In our case, the environment does not provide any feedback on code quality and the students are often implicitly motivated to finish the problems quickly to have their homework done, which probably inflates the observed prevalences. On the other hand, our list of defects is not yet exhaustive—in particular, we do not detect all instances of poor names and poor formatting—so the prevalence of the aggregated defect groups might be higher.

Even with the same collected data, there are methodological decisions with a large effect on the presentation and interpretation of the results. We report the macro-averaged prevalence, i.e., first computing within-problem prevalences and then averaging them, which gives equal importance to each problem in the problem set. Analogously, the overall prevalence is the macro average of the problem-set prevalences, which gives equal importance to each problem set. Macro-averaging is important because the data are skewed—first few problems have many more attempts than the last ones. The simple average (also called micro average) would put much more importance on the first few problems, considerably biasing the results. When we analyze the prevalence of solutions with at least one defect, the macro average is 53%, but the simple average is just 31%. This reflects the fact that there are not many defects in the simplest problems, which are the ones with the most collected solutions.

Another caveat is that our notion of prevalence is unconditioned, i.e., we report how frequently the defect appears in a randomly selected problem. Since not every problem gives an opportunity to make every defect, low prevalence does not imply that few students would make that defect *if they had an opportunity*. For instance, the overall unconditioned prevalence of *expressions simplifiable to absolute value* is 3.8%, but in the problems whose reference solution contains an absolute value, it is 58%. For the unconditioned prevalence to be high, there must be many opportunities to make that defect *and* the conditional probability of the defect given an opportunity must be high. Conditional prevalence would give another useful view, but it is unclear what exactly counts as an opportunity—a difficulty that is well worth tackling in future research.

Finally, more research is needed to explore the generalizability of results. We have analyzed 161 problems in 17 diverse problem sets. This is a significantly larger sample than what is used in most of the previous research on code quality, which typically analyzed students' solutions to just a few problems. However, our results show that the prevalence of individual defects varies significantly across exercises and topics and the space of introductory programming is still much broader. There certainly are many defects not present in our catalog, which are prevalent in different programming exercises. Moreover, the prevalence depends not only on the specific programming problems but also on their presentation, feedback, and student population. To get a more complete picture, we need to explore prevalence in other contexts.

REFERENCES

[1] Efthimia Aivaloglou and Felienne Hermans. 2016. How kids code and how we know: An exploratory study on the Scratch repository. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*. 53–61.

[2] Amjad Altadmri and Neil CC Brown. 2015. 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. 522–527.

[3] Eliane Araujo, Dalton Serey, and Jorge Figueiredo. 2016. Qualitative aspects of students' programs: Can we make them measurable?. In *2016 IEEE Frontiers in Education Conference (FIE)*. IEEE, 1–8.

[4] Dennis M Breuker, Jan Derriks, and Jacob Brunekreef. 2011. Measuring static quality of student code. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*. 13–17.

[5] Neil CC Brown and Amjad Altadmri. 2017. Novice Java programming mistakes: Large-scale data vs. educator beliefs. *ACM Transactions on Computing Education (TOCE)* 17, 2 (2017), 1–21.

[6] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2009. Relating identifier naming flaws and code quality: An empirical study. In *2009 16th Working Conference on Reverse Engineering*. IEEE, 31–35.

[7] Michael E Caspersen and Henrik Bærbak Christensen. 2000. Here, there and everywhere – on the recurring use of turtle graphics in CS1. In *ACM International Conference Proceeding Series*, Vol. 8. 34–40.

[8] Giuseppe De Ruvo, Ewan Tempero, Andrew Luxton-Reilly, Gerard B Rowe, and Nasser Giacaman. 2018. Understanding semantic style by analysing student code. In *Proceedings of the 20th Australasian Computing Education Conference*. 73–82.

[9] Tomáš Effenberger and Radek Pelánek. 2021. Interpretable Clustering of Students' Solutions in Introductory Programming. In *International Conference on Artificial Intelligence in Education*. Springer, 101–112.

[10] Xinyu Fu, Atsushi Shimada, Hiroaki Ogata, Yuta Taniguchi, and Daiki Suehiro. 2017. Real-time learning analytics for c programming language courses. In *Proceedings of the Seventh International Learning Analytics & Knowledge Conference*. 280–288.

[11] Hristina Gulabovska and Zoltán Porkoláb. 2019. Survey on Static Analysis Tools of Python Programs. In *SQAMIA*.

[12] Jesse Harden, Luke Gusukuma, Austin Cory Bart, and Dennis Kafura. 2021. A Specification Language for Matching Mistake Patterns with Feedback. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. 555–561.

[13] Lisa C Kaczmarczyk, Elizabeth R Petrick, J Philip East, and Geoffrey L Herman. 2010. Identifying student misconceptions of programming. In *Proceedings of the 41st ACM technical symposium on Computer science education*. 107–111.

[14] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2017. Code quality issues in student programs. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*. 110–115.

[15] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2019. How teachers would help students to improve their code. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*. 119–125.

[16] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2020. Student refactoring behaviour in a programming tutor. In *Koli Calling'20: Proceedings of the 20th Koli Calling International Conference on Computing Education Research*. 1–10.

[17] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2021. A tutoring system to learn code refactoring. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. 562–568.

[18] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2018. A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education (TOCE)* 19, 1 (2018), 1–43.

[19] Andrew Luxton-Reilly, Paul Denny, Diana Kirk, Ewan Tempero, and Se-Young Yu. 2013. On the differences between correct student solutions. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*. 177–182.

[20] Jessica McBroom, Kalina Yacef, Irena Koprinska, and James R Curran. 2018. A data-driven method for helping teachers improve feedback in computer programming automated tutors. In *International Conference on Artificial Intelligence in Education*. Springer, 324–337.

[21] Radek Pelánek and Tomáš Effenberger. 2020. Beyond Binary Correctness: Classification of Students' Answers in Learning Systems. *User Modeling and User-Adapted Interaction* 27, 1 (2020), 89–118.

[22] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. 2015. Learning Program Embeddings to Propagate Feedback on Student Code. In *Proceedings of International Conference on Machine Learning (ICML'15, Vol. 37)*. 1093–1102.

[23] Yizhou Qian and James Lehman. 2017. Students' misconceptions and other difficulties in introductory programming: A literature review. *ACM Transactions on Computing Education (TOCE)* 18, 1 (2017), 1–24.

[24] Alaaeddin Swidan, Felienne Hermans, and Marileen Smit. 2018. Programming misconceptions for school students. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. 151–159.

[25] Eliane S Wiese, Michael Yen, Antares Chen, Lucas A Santos, and Armando Fox. 2017. Teaching students to recognize and implement good coding style. In *Proceedings of the Fourth (2017) ACM Conference on Learning@ Scale*. 41–50.