International Journal of Engineering Education Vol. 37, No. 6, pp. 1540–1552, 2021
Printed in Great Britain

0949-149X/91 $3.00+0.00

CQpy: A Handy Code Quality Inspector for Online Python Programming Courses*

XIAO LIU AND GYUN WOO**
Dept. of Electrical and Computer Engineering, Pusan National University, Busan, Republic of Korea.
E-mail: {liuxiao, woogyun}@pusan.ac.kr

Most programming courses teach students how to write programs that produce the correct output without focusing on code quality. Although teaching about code quality is important, more emphasis is placed on code correctness. The reason is not its conceptual complexity but a lack of efficient methods for teaching good programming practices. This paper presents a code quality inspector named CQpy as a subsystem of an online judge for assessing both the correctness and the quality of programs submitted by students. Once the submitted code is judged to be correct, a student can further inspect code quality issues (CQIs) using CQpy. The timely feedback and automatic suggestions provided by CQpy help students improve their code quality through self-study. A controlled experiment revealed that there were thousands of unsolved CQIs in the programs submitted in a Python course in 2019 without CQpy. However, when CQpy was used, 91% of the detected CQIs were addressed by students who took the same Python course online during the COVID-19 pandemic in 2020. According to a student survey conducted at the end of the course, 90.12% of the students were satisfied with using CQpy in the online course, and 81.7% affirmed that they had improved their skills related to code quality after taking the course. Based on the experimental results, we identified a set of common CQIs representing the most frequent mistakes made by programming students. The results of this study could enhance the code quality education in future online Python programming courses.

Keywords: code quality inspection; automatic programming assignment evaluation; online course

1. Introduction

Teaching code quality is difficult, not because of complex concepts, but because of the lack of an appropriate teaching method. Code quality should be included in the evaluation criteria of programming assignments. However, many universities adopt online judges to assess the correctness of students' programs, but none of them has the functionality for inspecting code quality sufficiently [1]. The need for code-quality-related education drives researchers to analyze code quality issues (CQIs) in students' programs [2] and investigate approaches to improve the students' code quality [3]. Although there are existing tools to evaluate code quality [4–6], most of these tools either miss evaluation criteria or require significant human intervention. As a result, they are inefficient and inconvenient in practical programming courses.

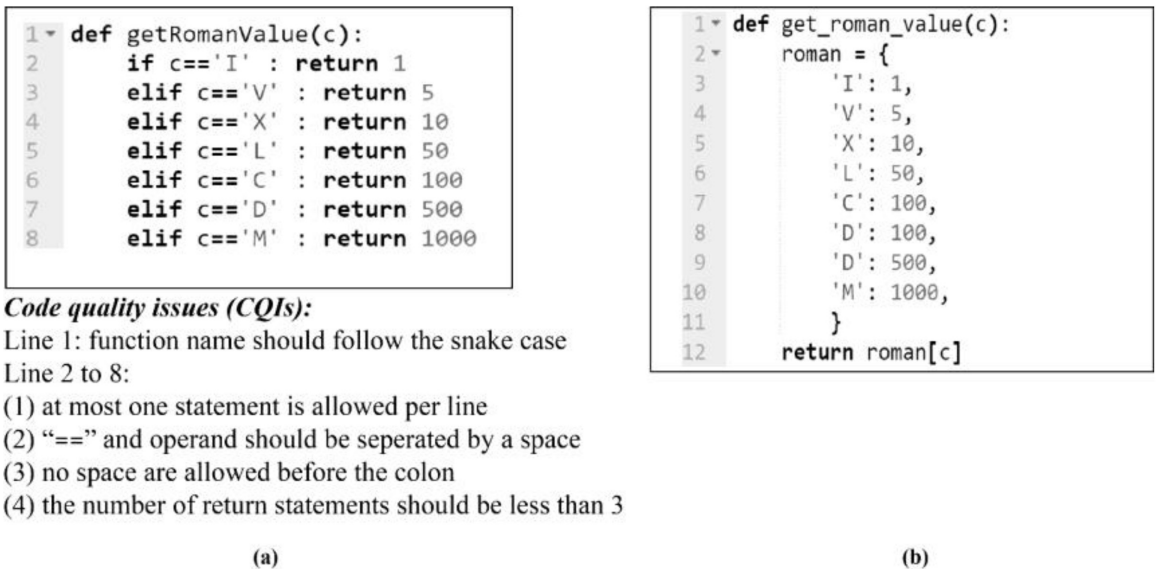
Teaching students to produce high-quality code is important in software engineering education because the industry values and requires guaranteeing the software quality more than ever [7]. Based on our past teaching experience, most students usually pay attention to the output correctness but fail to maintain code quality when writing programs. Fig. 1(a) shows an example of a student's poor-quality code with several CQIs. In contrast, Fig. 1(b) shows an example of implementing the same function with good code quality. Although the two functions generate the same result, the function in Fig. 1(b) has better readability and a more concise structure than that in Fig. 1(a).

Researchers have noted the lack of teaching and measuring code quality [8], and studies show that it is essential to enhance code quality in programming education [9–11]. However, most of the state-of-the-art studies related to improving code quality in education focus on face-to-face courses, where the instructors can apply their approach in person and interact with students in the classroom. During the COVID-19 pandemic, many universities decided to convert face-to-face courses to online courses. Online courses require more student autonomy than face-to-face courses. Specifically, the manual feedback for code quality can hardly be performed effectively in online courses.

Moreover, although current studies related to improving the code quality of students' programs are indeed measuring code quality, the correctness of the program output needs to be checked either by a human grader or by another online judge. It is inconvenient to evaluate students' programs for both correctness and quality because the instructor and students will have to keep switching tools for checking the correctness and for receiving the code-quality-related feedback. To the best of our knowledge, it is difficult to find a system that not only

** Corresponding author.

* Accepted 14 July 2021.



```
def getRomanValue(c):
    if c=='I' : return 1
    elif c=='V' : return 5
    elif c=='X' : return 10
    elif c=='L' : return 50
    elif c=='C' : return 100
    elif c=='D' : return 500
    elif c=='M' : return 1000
```

Code quality issues (CQIs):

Line 1: function name should follow the snake case

Line 2 to 8:

(1) at most one statement is allowed per line

(2) "==" and operand should be seperated by a space

(3) no space are allowed before the colon

(4) the number of return statements should be less than 3

(a)

```
def get_roman_value(c):
    roman = {
        'I': 1,
        'V': 5,
        'X': 10,
        'L': 50,
        'C': 100,
        'D': 100,
        'D': 500,
        'M': 1000,
        }
    return roman[c]
```

(b)

Fig. 1. Examples of bad and good code quality.

considers the correctness of a student's program but also inspects its code quality fully.

This paper introduces a handy tool, CQpy (Code Quality inspector for Python), and describes our experience with teaching code quality using CQpy during the COVID-19 pandemic. CQpy is implemented as a subsystem of an online judge. Once the submitted code is judged to be correct in terms of its output, a student can further inspect any existing CQI using CQpy. The detecting rules used in CQpy sufficiently cover various aspects of code quality that are wildly adopted in the industry, which are not only feasible for identifying students' code-quality-related mistakes but also help students maintain a good programming paradigm when they become professional developers.

To verify the efficacy of CQpy, we conducted two experiments. In the first experiment, we explored the problems in code quality education retrospectively using CQpy. In this experiment, we explored the CQIs in students' programs from past submissions. Two thousand Python programs were collected from undergraduate students' submissions during the 2019 academic year. The programs were analyzed and examined in CQpy for code quality inspection. The most frequent CQIs detected in these programs were categorized by their effects on the program, with adequate suggestions for addressing them.

In the second experiment, we measured the effectiveness of CQpy in improving code quality in programming education. This experiment was conducted with an actual introductory programming course at Pusan National University (PNU), Busan, Republic of Korea, during the first semester of 2020. The target course was a Python programming course for undergraduate students, which was held online during the selected timeframe because of the COVID-19 pandemic. A quantitative analysis was conducted on the code quality improvement in students' programs with the use of CQpy. The qualitative effect was also analyzed based on a student questionnaire to determine the satisfaction with CQpy's user experience and to identify the shortcomings of the system to improve it.

The remainder of this paper is organized as follows. Section 2 presents the background of code quality education in pre-COVID-19 courses and the technologies for detecting CQIs related to this study. Section 3 describes our methods, including the development of CQpy and the experiment design. Section 4 presents the experimental results – both quantitative and qualitative. Section 5 discusses our findings and the limitations of the current CQpy system. Finally, Section 6 concludes the paper.

2. Background

2.1 Code Quality and its Education pre-COVID-19

There are various aspects of code quality, such as capability, usability, performance, reliability, and maintainability [12]. These aspects affect the characteristics of the program, including efficiency, vulnerability, and security. Therefore, to train students to become skilled developers, it is necessary to teach them to understand the importance of code quality and to maintain a good quality of programs.

To summarize the CQIs in students' code, Stegeman et al. [9] proposed generic rubrics to provide feedback on code quality in programming courses. Nine models covered the coding criteria in terms of naming, control structures, and expression of code

quality. Code quality inspection rules were collected by referring to professional software literature and through interviews with teachers. The resulting models covered complexity, expression collapsibility, code duplication, and coupling levels. Detailed feedback could be used to analyze the teaching of programming courses. However, a human grader was required to evaluate the code quality of students' programs, which could be a burden.

Dietz et al. [10] examined the programming assignments collected from Java programming courses at the University of Bamberg, Germany. They found that students usually focused on functionality rather than code quality. This study used an analysis tool named CodeScene to analyze the code quality in Git commits. However, their study did not provide solutions for improving the code quality.

Kasahara et al. [11] presented a gamification approach for grading the correctness of programming assignments and motivating students to generate high-quality code by ranking the code quality of their programs using lines of code (LOC) and cyclomatic complexity (CC). According to their findings, students aimed to attain higher ranks by optimizing the LOC and CC in their assignments submitted to an online judge. Although LOC and CC are important facets that affect the code quality and efficiency of a program, the vulnerability and security issues cannot be addressed by either of these metrics.

2.2 SonarQube and CQI Detection

Existing tools for CQI detection include SonarQube, Fortify, and Squale, which are widely used to analyze and review program codes [13]. Fortify and Squale are commercial tools, and SonarQube has been deployed in two versions: commercial and community. SonarQube consists of a scanner and a server [14]. The server provides an environment for managing code and inspecting results; the scanner detects the CQIs in the uploaded program and generates an inspection result. SonarQube supports various types of detections, including bug checks, vulnerability checks, and code smell detection. The rules used in SonarQube (such as complexity measurement, use of control flow, and naming convention inspection) are helpful for detecting the weaknesses of programs and providing useful suggestions to improve code quality [15].

The basic quality inspection mechanism in SonarQube is to statically analyze the program code, considering several quality rules (to detect whether the code passes the rules or not), and generate the inspection results. The rules used in SonarQube cover the programming guidelines of each corresponding language, common programming regulations, de facto coding patterns, and other widely used coding conventions. These rules have also been used in the industry to standardize software quality [16]. Therefore, the code quality practice using SonarQube is beneficial for students during their job search after graduation.

3. Materials and Methods

3.1 System Development

We developed CQpy based on SonarQube for code quality inspection. CQpy invokes the scanner of SonarQube to detect CQIs and plugs the results into the server of an online judge called neoESPA, which we developed previously. We only use the open-source portion of SonarQube to develop CQpy as that portion is sufficient for students' programs to identify their novice-like mistakes.

Fig. 2 demonstrates the sequence diagram of using CQpy with neoESPA from the student's perspective. A student can upload his/her source code through a web browser. The uploaded file is stored in the system storage and then sent to the Python interpreter to generate the output for the hidden test input. The correctness checker verifies the output of the program with the criteria output, assesses its correctness score, and stores it in the database. Once the program has been checked for correctness, the system storage sends the code to CQpy for code quality inspection. The detected CQIs are also stored in the database. Both the correctness score and the detected CQIs are sent back to the student's browser as the feedback of the submission.

We developed our own web user interface for CQpy in place of the original interface of SonarQube to plug it into neoESPA seamlessly. Our interface can also be helpful to reduce the students' learning cost of a new system. Fig. 3 presents the web interface to submit and obtain feedback for a student's program. Once the student selected the corresponding homework number (1), the file type (2), and the source code file (3), the file will be sent to the system by clicking the upload button (4). The correctness score of the program output will be displayed (5) after execution. Usually, ten predefined criteria data are tested for output correctness, and each of them assigns ten points if the program output matches the corresponding criteria. The CQIs in the student's code are presented in a popup window (6). For each CQI, the incorrect part of the code is marked by a wavy underline with a brief explanation. The student can click on the brief explanation (7) to obtain a more detailed description (8) of the cause of the CQI and a suggestion of how to fix it.

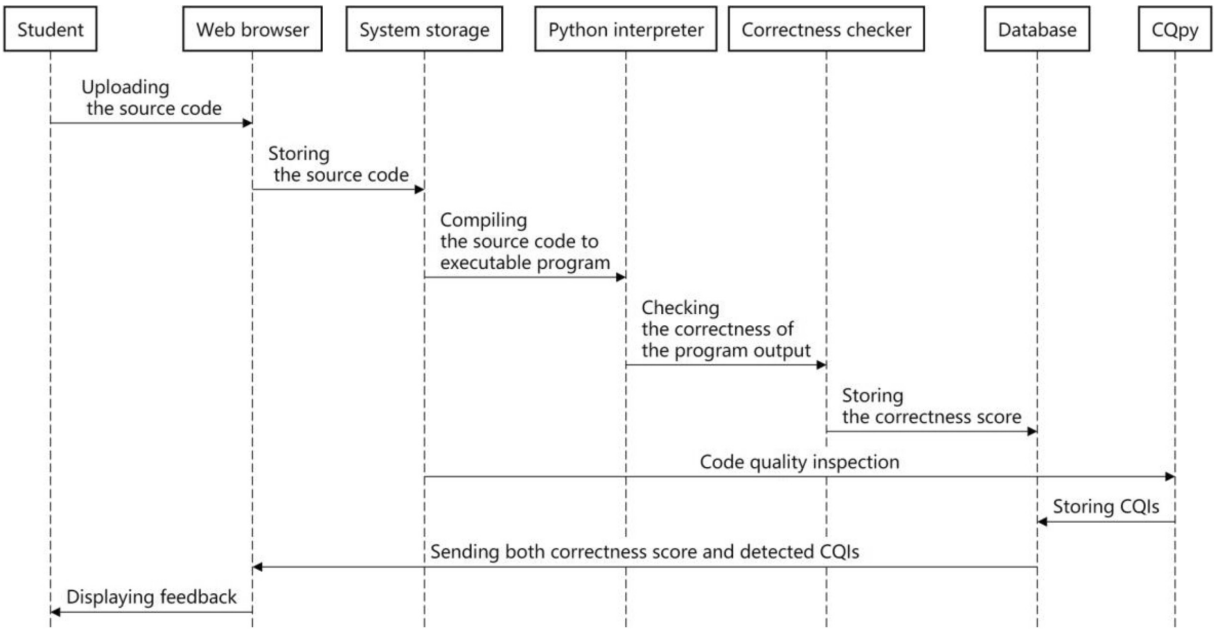


Fig. 2. Student's perspective using neoESPA with CQpy.

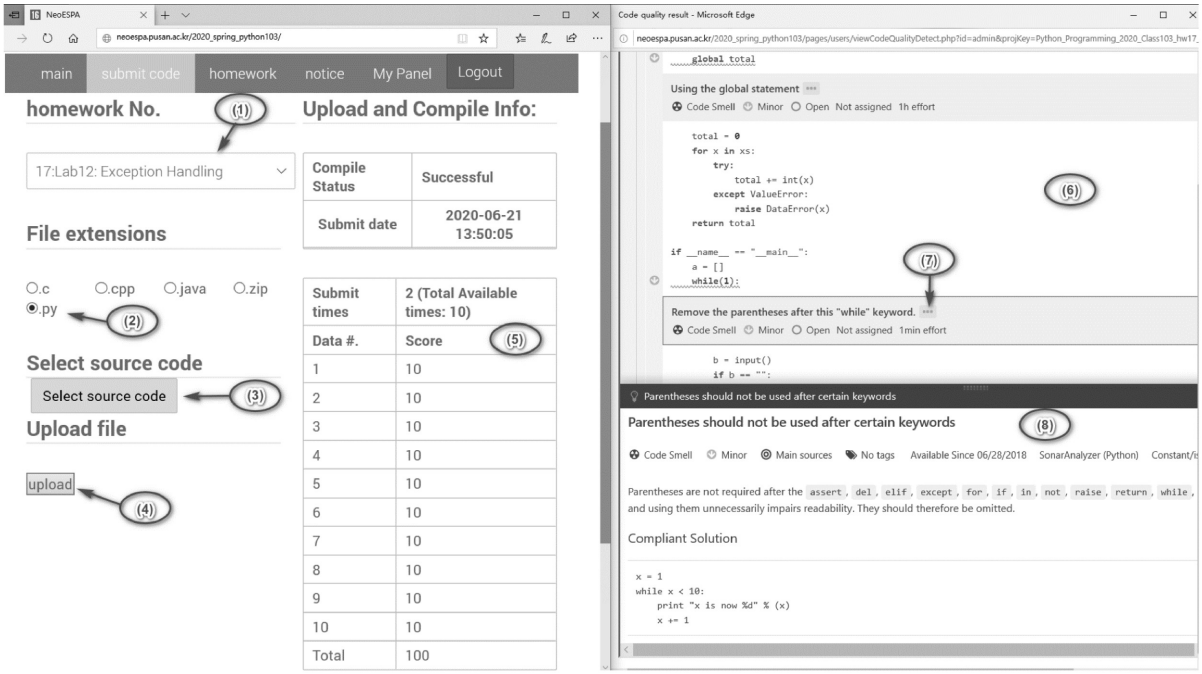


Fig. 3. Web interface of a programming assignment submission.

3.2 Data Collection

We collected students' programs as experimental data in two parts. The first part contained the students' programs in the Python programming course in the 2019 academic year. The second part contained the programs submitted in the first semester of 2020 by students taking the online Python programming course and used neoESPA with CQpy.

3.2.1 Programs Submitted in the 2019 in-person Course

According to our department's curriculum, Python is taught in the first semester of the first year. Two thousand programs were collected from a total of 60 students in the 2019 academic year. These assignments covered various aspects of programming such as data input/output, conditional statements, type handling, and object-oriented programming.

After identifying the correctness of the collected programs, we prepared 900 correct programs to inspect their code quality using CQpy. The programs were used to quantify the CQIs in students' programs caused by the lack of code-quality-related education in the 2019 programming course. By analyzing the CQIs of these source codes, we can determine a set of common CQIs that can be used for educational purposes in future programming courses.

3.2.2 Programs Submitted in the 2020 Online Course

Owing to the COVID-19 pandemic, all programming courses in PNU were conducted as online courses during the first semester of 2020. The online Python programming course was taken by 90 students. We used neoESPA with CQpy in this online course to evaluate the correctness and quality of students' programs. The students submitted more than three thousand programs during this semester.

Although this is an introductory programming course, the students were from different grades and majors. To allow every student to have enough time to learn how to write a working program, CQpy was applied to five assignments after the midterm exam. Students were asked to undergo the code quality inspection if their programs have perfectly passed the correctness check in neoESPA. There were 62 students whose programs met this requirement, and they contributed 920 submissions for code quality inspection, at least for one of their assignments. These programs were used to analyze the improvement in students' code quality during their continuous submissions to CQpy.

3.3 Experiment Design

We conducted two experiments using the two sets of collected data. For the first experiment, we submitted all the 2019 students' programs to CQpy to inspect the code quality and classify the detected CQIs. The classified data can help us understand the most frequent CQIs in students' programs and how to improve students' code quality efficiently.

The second experiment was conducted during the first semester of 2020 because the students' submissions were continuous throughout the online course. We asked the students whose programs were 100% correct in terms of output to use CQpy for the code quality inspection. They were given bonus points for reducing their CQIs to three after they reviewed the CQpy feedback, followed the suggestions, and resubmitted the revised programs. At the end of the semester, we offered a questionnaire to students to understand their perspectives on using CQpy for improving their code quality. A quantitative analysis measured the improvement achieved by students in their code quality using CQpy. A qualitative analysis helped us understand the students' opinions on using this approach in the online course.

The Python programming course is held once a year at PNU, and it is a core course for disciplines such as computer science and engineering (CSE), engineering, and science. Approximately half of this course is designated for CSE students and the other half for students with other majors and is available for all grade levels. To compare the effects of using and not using CQpy, we chose the same distribution of students' grades and majors in both 2019 and 2020.

We use the experimental results to answer the following three research questions (RQs):

RQ1: What types of and how many CQIs in the students' programs were caused by the lack of corresponding education in the 2019 programming course?

RQ2: To what extent has CQpy helped students improve their code quality in the 2020 online programming course?

RQ3: What are the positive and negative opinions regarding the use of CQpy to improve programming skills and code quality from the students' perspective?

4. Results

4.1 RQ1: Frequent CQIs in the 2019 Programming Course

There are 1,728 CQIs detected in students' programs from the 2019 programming course. The CQIs are classified by their effects on the programs: readability (RDB), security (SEC), and performance (PFM). Table 1 presents the seven most frequent CQIs classes incurred from the source codes submitted in the 2019 course. For each effect class, CQI details, such as the violated reason, the number of detections (Freq.), typical example codes, and the suggestion produced by CQpy, are presented.

The CQIs in Table 1 indicate that the students were not familiar with the fundamental concepts of computer programming (such as the use of variables and the design of control flow). These issues are common for novice programmers [17]. These CQIs do not affect the correctness but reduce the quality of the program. The readability issues in PyRDB1 and PyRDB2 are the most frequent, which would increase maintenance costs if the students continue to introduce such CQIs in a larger software project in the future. The second group of frequent CQIs lies in security issues

Table 1. CQIs in the 2019 students' programs

Effect	Violated reason	Freq.	Example	CQpy Suggestion
PyRDB1	Naming variables, functions, and classes without following naming conventions	812	Fig. 4(a)	Rename the element under the regulations
PyRDB2	Declaring variables or functions that are not used in the code	378	Fig. 4(b)	Remove the unused code
PySEC1	Using keywords such as return, break, and pass improperly	212	Fig. 5(a)	Modify or remove the corresponding code
PySEC2	Leaving an empty code block	106	Fig. 5(b)	Remove or fill the empty code block
PyPFM1	Conducting too many statements into one function or conditional expression	89	Fig. 6(a)	Split the code into separate functions
PyPFM2	Using the same condition in an if-else statement	91	Fig. 6(b)	Merge or change one of the consequents
PyPFM3	Splitting a collapsible conditional statement into two expressions	40	Fig. 6(c)	Merge the statement with the enclosing one

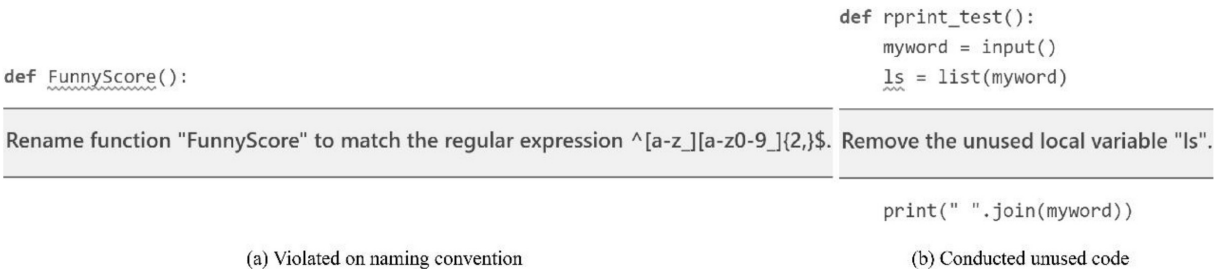


Fig. 4. Examples of PyRDB CQIs.

(PySEC1 and PySEC2), incurring vulnerable code that can cause a crash or memory leak. The third group of frequent CQIs (PyPFM1–PyPFM3) indicates that the students were not familiar with the proper use of conditional expressions. These CQIs can negatively affect the performance of the program.

4.2 RQ2: Quantifying what Students have Learned

We calculated the average changes in CQIs during the students' continuous process of submitting the program – fixing CQIs – resubmitting the revised program. Most of the students stopped the process at the sixth submission because they had already either fixed all CQIs or used up all the submission chances; although they had ten attempts for each assignment, they may have used several attempts to obtain 100 points for the correctness check. Fig. 7 shows that the quality of students' code improved regardless of the assignments, grade levels, and majors. For each graph shown in Fig. 7, the *x*-axis represents the submission time on CQpy, and the *y*-axis represents the number of detected CQIs in the submission. The more CQIs decreased, the more the quality of the code improved throughout the submissions. The decreasing trend of detected CQIs indicates that most students were motivated to pursue better code quality.

Fig. 7(a) illustrates the decrease in CQIs with the number of submissions. The numbers of initial CQIs in Assignment 1 to Assignment 3 at the first submission were 17, 10, and 5, respectively. Assignment 1 had the highest number of initial CQIs because it was the first time the students inspected

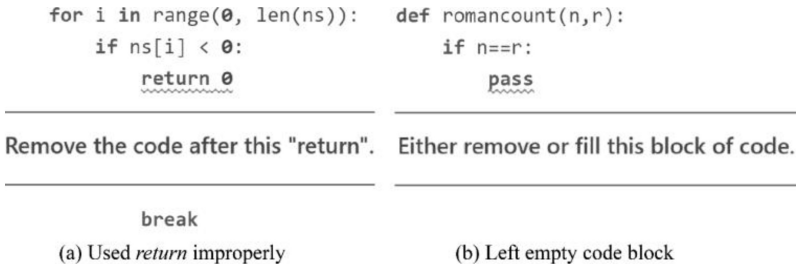


Fig. 5. Examples of PySEC CQIs.

```
def print_c(s):
```

Refactor this function to reduce its Cognitive Complexity from 23 to the 15 allowed.

```
    length = len(s)
1   for i in range(length):
2       if i ==0:
            print_str(s)
3       elif i == length-1:
            reverse(s)
4       else:
5           if length ==4:
                print("{}     {}".format(s[i],s[length-(i+1)]))
6           else:
7               if length == 5:
                    print(s[i], end = "")
8                   for j in range(length+2):
                        print(" ", end = "")
                    print(s[length-(i+1)], end = "")
                    print()
9               else:
                    print(s[i], end = "")
10                  for j in range(length+4):
                        print(" ", end = "")
```

(a) Conducted too many statements in one function

```
if bmi < 15.0:
    print('NOR')
elif 1 15.0 <=  bmi < 16.0:
    print('SUN')
elif 15.0 <= bmi < 16.0:
```

This branch duplicates the one at above.

(b) Conducted same condition in *if elif* statement

```
1   if y < location[1]:
        location[0] = int(x)
        location[1] = int(y)
    elif y == location[1]:
        if x < location[0]:
```

Merge this if statement with the enclosing one.

```
            location[0] = int(x)
            location[1] = int(y)
```

(c) Split collabsible conditions

Fig. 6. Examples of PyPFM CQIs.

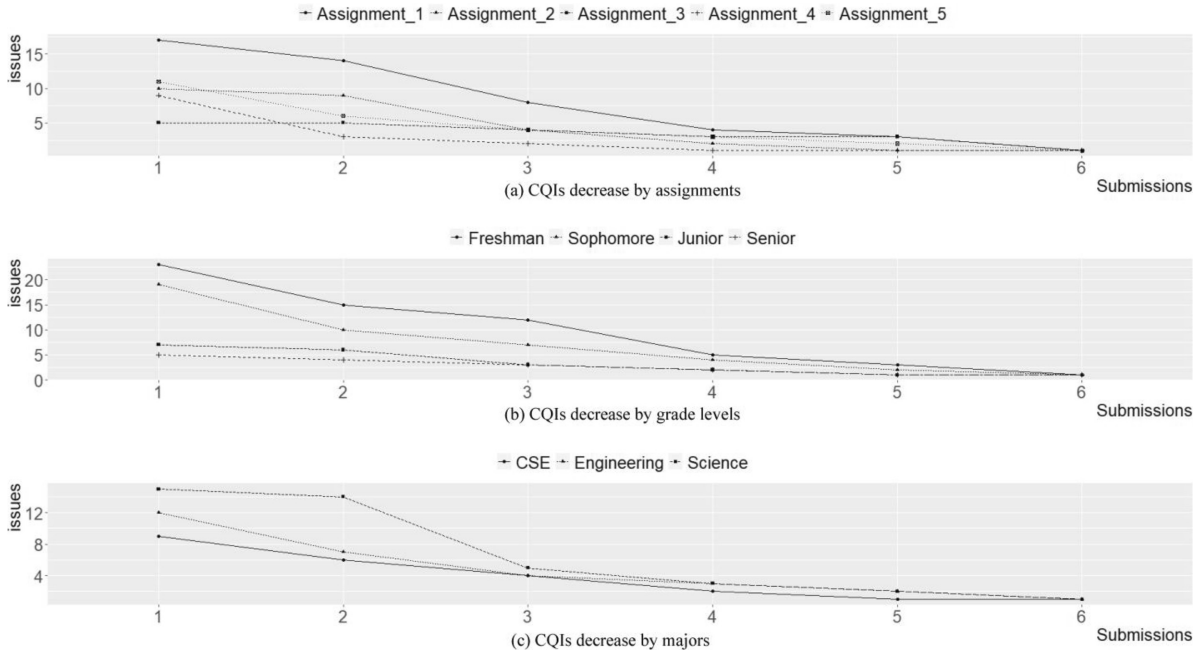
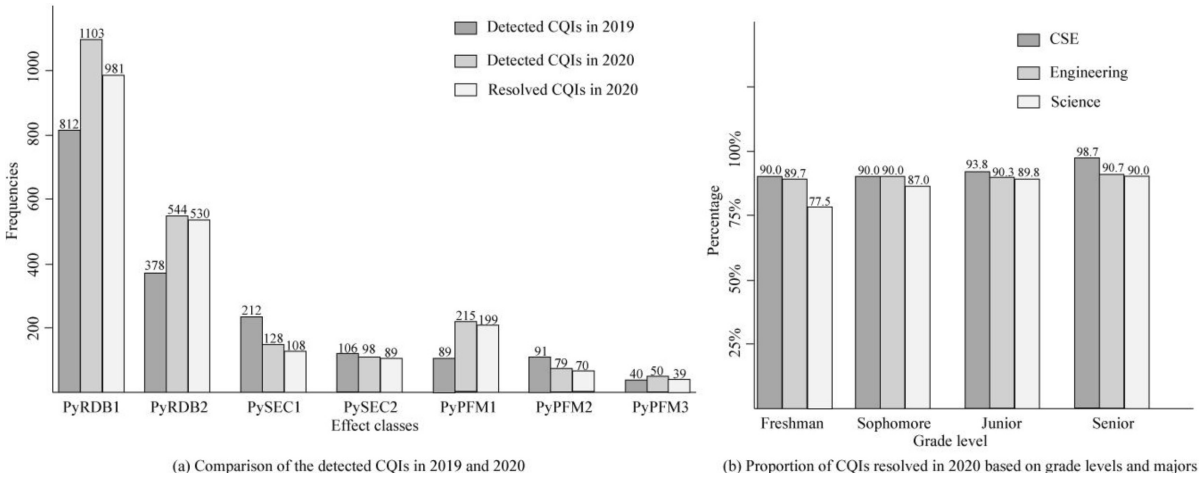


Fig. 7. Decrease in the number of CQIs for different categories.

their code quality. This number was reduced in the following two assignments because CQpy teaches students to pay attention to their code quality. However, the numbers of initial CQIs in Assignments 4 and 5 were 9 and 11, respectively, which were more than the number of initial CQIs in Assignment 3, and do not fit the explanation at first glance. The reason is that these assignments cover more advanced concepts (such as inheritance, subclasses, and exception handling), which cause additional code quality difficulties when novice students develop their programs. With the help of CQpy, students could eventually reduce the number of CQIs to improve their code quality in the advanced assignments.

Fig. 7(b) illustrates the decrease in CQIs by grade



(a) Comparison of the detected CQIs in 2019 and 2020

(b) Proportion of CQIs resolved in 2020 based on grade levels and majors

Fig. 8. Detected and resolved CQIs in both in-person and online courses.

level. At the initial submission, freshman (23 CQIs) and sophomore (19 CQIs) students introduced more CQIs than junior (7 CQIs) and senior students (5 CQIs). Such a finding indicates that after acquiring code-quality-related knowledge, the students who already had programming knowledge were more adept in following the coding conventions than the beginners. Additionally, the decrease in CQIs for the freshman and sophomore students was more obvious than the smooth decrease for the junior and senior students. Such a result indicates that, although the beginners made many mistakes in their earlier submissions, they could eventually fix the CQIs through CQpy.

In the CQI decrease by college majors in Fig. 7(c), the students majoring in CSE introduced 9 CQIs, the fewest at the initial submission, followed by those majoring in other engineering fields (12 CQIs) and by those majoring in science subjects (15 CQIs). This is reasonable since CSE students take more programming-related courses than non-CSE-majoring students. Although engineering majoring students have fewer programming courses than CSE students, they paid particular attention to quality control. The science majoring students seemed to have difficulties in fixing CQIs at their first and second submissions, but they could fix all CQIs over time.

For all data shown in Fig. 7, at the sixth submission, the average of the detected CQIs remained less than one, for some students have fixed all CQIs. We marked them as 1 in the graph because the number of CQIs cannot be a fraction. There are 2,217 CQIs detected in the 2020 online course, with 2,016 of them being resolved by students. The number of detected CQIs in 2020 is considerably larger than that detected in 2019 because students usually could not resolve all the detected CQIs in consecutive resubmissions. Therefore, some remaining CQIs were counted more than once in CQpy with each subsequent submission. The number of CQIs identified in the students' first submission was 1,822 in 2020, which was similar to that in 2019. On average, 91% of CQIs detected in the initial submission were solved by the sixth submission.

Fig. 8(a) presents the comparison of the CQIs detected and resolved in the 2020 online course, along with the CQIs detected from the 2019 Python course. The same classifications of CQIs in the two courses indicate that students committed common mistakes in readability, security, and performance.

Fig. 8(b) depicts the proportions of students in the 2020 online course that resolved the identified CQIs according to the grade level and major. As the grade level increases, the proportion of resolved CQIs also increases. Additionally, the highest proportion of resolved CQIs is that for CSE students for all grade levels, followed by those for engineering and science students.

To determine the level of significance of the effect of the students' grade level and major on the proportion of resolved CQIs, we used two-way analysis of variance to analyze the dependence of student groups using CQpy in terms of resolving the CQIs. For each of the 62 students who used CQpy at least once, we collected the program in which the student had resolved the most CQIs to represent his/her best performance in terms of code quality. Table 2 lists these students according to their grade level and major.

Fig. 9 presents the analysis results based on the distribution of the percentage of resolved CQIs according to the grade level and major. Our null hypothesis (H_0) is that the students can resolve their CQIs regardless of their grade level and major, and the alternative hypothesis (H_1) is that the students' ability to resolve CQIs is affected by their grade

Table 2. Number of students who used CQpy

Grade level	CSE	Engineering	Science	Total
Freshman	10	0	0	10
Sophomore	8	5	6	19
Junior	6	8	5	19
Senior	5	5	4	14

level and major. We considered $\alpha = 0.05$ as the significance level.

As shown in Fig. 9(a), the distribution of the percentage of resolved CQIs based on the grade level carries a significance of $p = 0.015026 < 0.05 = \alpha$, which indicates that the grade level is a significant factor affecting the resolving of CQIs by students. Fig. 9(b) depicts that the significance of the distribution of the percentage of resolved CQIs based on the major is $p = 0.000198 < 0.05 = \alpha$, which indicates that the students' major is also a significant factor that affects their resolution of their CQIs. We can reject H_0 and accept H_1 based on the p-values obtained. Moreover, we can conclude that the students' majors have a greater impact than their grade levels on the resolution of CQIs.

4.3 RQ3: Students' Perspectives on using CQpy

Because the questionnaire offered at the end of the 2020 online course was not mandatory, only 71 students (30 CSE students, 22 engineering students, and 19 science students) participated, and each of them answered nine questions. The major and the grade-level distribution of the students that participated in the survey nearly matches the overall pool of students registered in the course. Hence, the results of the questionnaire are adequately equitable and representative for us to understand the students' overall perspective on using CQpy.

The first five questions enquired about the level of agreement in using CQpy to improve their code quality. The levels represented students' attitudes: strongly disagree, disagree, agree, and strongly agree. Fig. 10 shows the students' responses. Importantly, 91.5% of students agreed that CQpy motivated them to pursue better code quality (Q1). CQpy allowed students to submit their programs several times to improve code quality, and 97.2% of students liked this feature (Q2). Moreover, 83.1% of students agreed that the detailed feedback and suggestions from CQpy helped them improve their code quality (Q3). During the online course, 87.3% of students agreed that CQpy helped them learn about code quality (Q4). Finally, 91.5% of students agreed that CQpy was a good system for learning about code quality (Q5). On average, the percentage of agreement on Q1 to Q5 was 90.12%, and the disagreement was 9.88%.

In addition, three closed questions (with yes/no answer) were included to examine the progress in code quality learning and future intentions of using CQpy. Fig. 11 presents the students' responses to these questions. In Q6, 80.3% of students did not have any previous knowledge related to code quality. This result indicates the lack of code quality education in the past programming courses and explains the numerous CQIs described in Section 4.1. After using CQpy, 81.7% of students confirmed that they had learned code-quality-related knowledge (Q7). Furthermore, owing to good user experience with CQpy, 84.5% of students showed that they want to use CQpy in the next programming course (Q8).

Finally, Q9 was an open-ended question asking the students to provide recommendations on improving CQpy. Some students mentioned that they wanted to practice more with CQpy:

> "*It will be more helpful to improve programming quality if the system allows us to practice more, even if the deadline of an assignment has passed or we have exceeded the number of submissions.*"

Some students mentioned the inspection speed of CQpy requesting performance improvement:

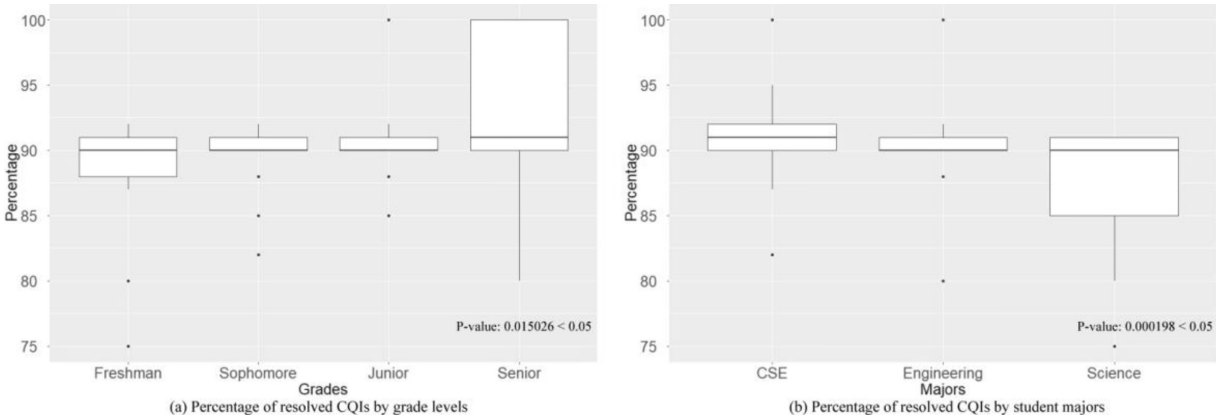


Fig. 9. Statistical significance of grade levels and majors on the percentage of resolved CQIs

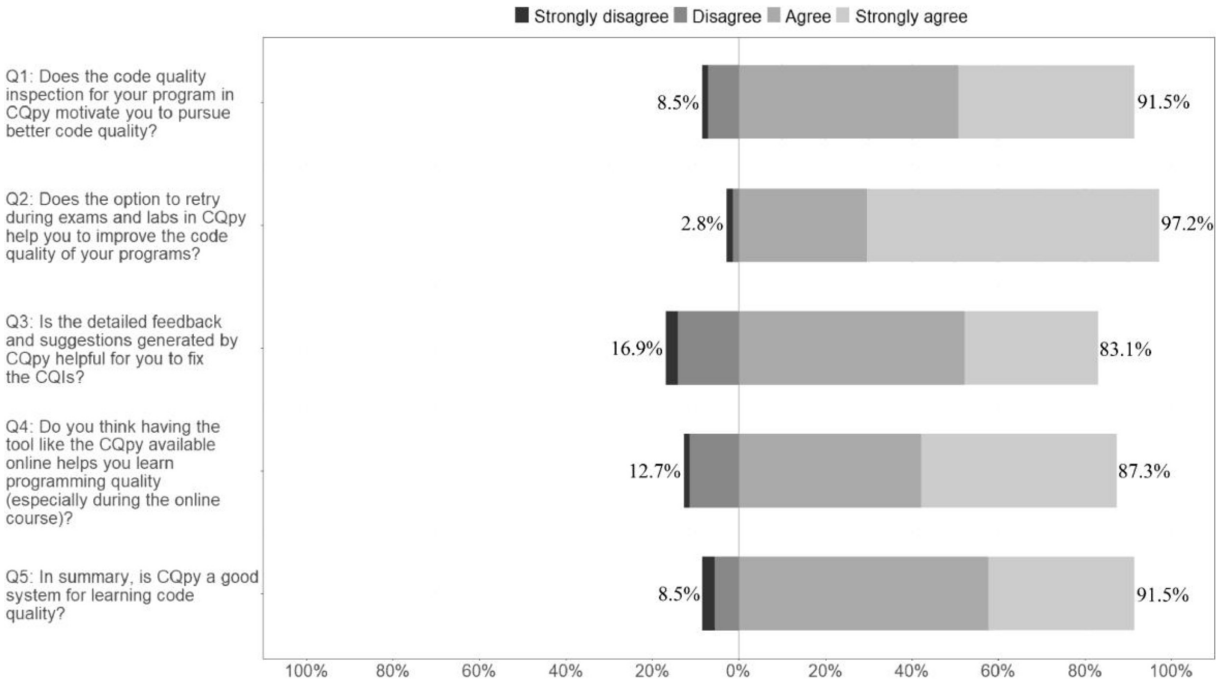


Fig. 10. Level of agreement regarding the user experience of CQpy from the students' perspective.

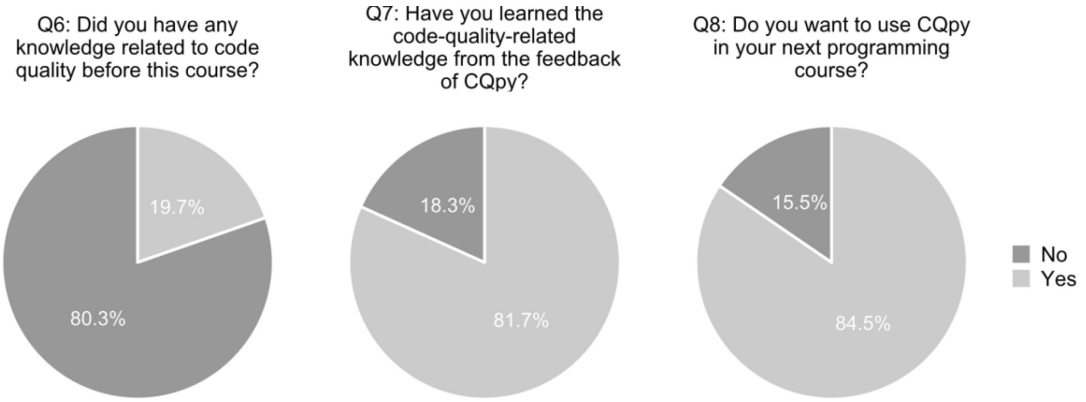


Fig. 11. Students' perspectives on code quality learning and their future intentions of using CQpy.

> "*The code quality inspection helped me to improve my programming quality, although I had to wait for one minute to get the feedback. It will be more efficient if the function can be executed in a few seconds.*"

Those students providing no recommendations indicate that they did not find any inconvenience in using CQpy. A few students expressed that the course was too difficult for them and that they rarely used CQpy for code quality inspection.

5. Discussion

5.1 Benefits of the Online Code Quality Inspection Method

Most of all, the timely feedback with automatic suggestions of CQpy can help students improve code quality in self-study. This feature is particularly effective for online programming courses, where it is difficult for instructors to give face-to-face feedback to students. Students can submit source codes, assess correctness, inspect code quality, and learn-and-fix CQIs in self-study without the intervention of the instructor. Such an advantage provides a favorable learning experience and positive educational effects, especially in online programming courses.

The answer to RQ3 shows that most students have positive perspectives on the user experience of CQpy and confirmed that they gained code-quality-related knowledge after taking the online course. This is a strong recognition of the contribution of our method. Letting students know their mistakes with proper suggestions immediately once they submit their programs can encourage them to

pursue better code quality. It is more efficient than the traditional "knowledge-inculcation" lecturing approaches for teaching code quality.

Furthermore, the detected CQIs in students' programs can be reviewed online in real-time, which helps the instructor to monitor the students' submissions in a timely manner. Assuming that the manual assessment of the code quality of a program by a human grader requires ten minutes on average, it will take more than 153 hours to mark the mistakes, write explanations, and make appropriate suggestions for 920 programs. Using CQpy, the cumbersome task of assessing the code quality of students' programs can be made much easier for the instructors. Students can submit their programs and need to wait just a few minutes to receive detailed feedback automatically. Additionally, the progress of students resolving CQIs can also be monitored by the instructor if the students are willing to participate in pair-programming, which is helpful to timely discover the progress of students learning code quality.

5.2 Comparison of the Pre-, during, and post-COVID-19 Strategies

The comparison of the pre-, during, and post-COVID-19 teaching strategies are presented in Table 3. In pre-COVID-19, the teaching method for code quality mainly depends on human interactions, including the manual evaluation and marking right or wrong issues with a demonstration on typical source codes. During COVID-19, CQpy has helped most of the students to improve their programming skills related to code quality by pinpointing the location of issues with detailed suggestions.

After teaching code quality in the 2020 online course, we realized that there are advantages in both teaching strategies used pre-COVID-19 and during the COVID-19 pandemic. There were a few students who had difficulties in using CQpy (Section 4.3, Q9), and demonstrations are still needed for introducing CQIs and showing how to resolve them. We should also allow the students whose programs are either overdue or partially correct to use CQpy for addressing their unsolved CQIs. In addition, we are going to apply certain practices of pair-programming for students with the instructor/teaching assistant (TA) as supplementary feedback to help students become familiar with using CQpy to resolve CQIs.

5.3 Instructors' Points Of View on CQpy

As expressed by the instructors who taught Python programming in both semesters (2019 in-person and 2020 online), the current CQpy is also effective for programming experts. Even if the naming rule is simple enough, it is easy to violate as some simple predefined names, such as min and max, can be easily used as variable names. Therefore, the use of CQpy is highly recommended, even for the development procedure. Code quality assessment is facilitated because the actual assessment time is significantly reduced from tens of minutes to a few minutes by using CQpy.

To make CQpy viable during the development procedure, the output correctness check should be bypassed. To do this, making a separate sandbox session solely for quality assessment can be a quick solution. Because the syntax and the static semantics of the code should be checked before the quality assessment procedure, the compilation procedure should be performed before CQpy. The compilation module (py_compile) of the standard Python distribution could be used for the compilation step.

We devoted three months to develop the prototype of CQpy and spent another three months testing its functionality and verifying its practicality by inspecting the CQIs in students' programs from their past submissions. Therefore, the construction of CQpy took approximately six months. Considering our familiarity with the automatic judge neoESPA, it may require a few additional months to make a workable code quality inspector like CQpy for other automatic judges.

5.4 Teaching Methods post-COVID-19

The answer of RQ2 shows that students usually have a certain number of CQIs in their first submission, but most of them could eventually solve the CQIs by referring to the feedback from CQpy. As shown in Fig. 7(a), difficult assignments could cause more CQIs than the previous assignments. However, students could eventually resolve these

Table 3. Comparison of Pre-, during, and post-COVID-19 Teaching Strategies

	Pre-COVID-19	**During COVID-19**	**Post-COVID-19**
Teaching method	Demonstration in lectures	Applying CQpy to correctly working code	Demonstration in lectures and applying CQpy to partially correct code
Evaluation approach	Manual evaluation	Automatic evaluation	Automatic evaluation
Feedback form	Marked as correct or wrong with handwritten notes	Detailed descriptions for CQIs and automatic suggestions as solutions	The form of During-COVID-19 and pair-programming with instructor/TA

CQIs as well. In retrospect, the diverse spectrum of difficulty range of problems can open more opportunities for code quality education to a wide range of students.

As shown in Figs. 7(b) and 7(c), the code quality improvements achieved by different grade levels and majors of students prove that the lack of a programming background does not limit students in learning code quality. This finding drives us to consider that it would probably be better if we apply CQpy as early as possible in programming education.

Although online courses may limit the teacher/student face-to-face interaction, it is still practical to teach students about code quality through online conferencing tools such as ZOOM. After the midterm exam of the 2020 online course, we recruited several volunteers to try to solve the exam problems with the instructor in a form of pair-programming. The volunteers were asked to construct programs in a collaborative environment, and the instructor guided them to resolve the CQIs of their programs using CQpy. These practices were posted as demonstrations[1] of the introduction to use CQpy for the rest of the students.

In addition, we analyzed the volunteers' improvement in code quality after the pair-programming exercises. We found that students could resolve more CQIs and obtain more bonus points (20 points) than before (10 points or less). This is an encouraging result for the extension of such pair-programming opportunities during and after the COVID-19 pandemic in additional practice sessions with TAs, given that enough TAs are available.

5.5 Recommendations for Teaching Code Quality

The answer of RQ1 shows that the most frequent CQIs in students' programs were caused by their lack of Python syntax knowledge. Although such CQIs may not affect the program's correctness, students will make similar mistakes when they learn other programming languages if they are not reminded to improve code quality. Furthermore, once the CQIs remain in their programming paradigm, students may write low-quality software when they become developers or even face difficulties in finding jobs after graduating. Therefore, it is necessary to teach students code quality management either through lectures or automatic tools in programming courses.

The classifications of the common CQIs in Table 1 can help instructors locate common weak points in code quality and enhance the corresponding training in programming education. The examples of bad code quality can be used in the lectures for reminding students to avoid such mistakes when they construct programs. The suggestions of resolving CQIs also can be taught to students for improving the code quality of their programs.

By utilizing CQpy, most of the students can gain code-quality-related knowledge and improve their code quality in self-study (Figs. 7 and 8). The approach of allowing students to continuously submit their programs, fix CQIs, and resubmit the revised programs can improve the efficiency of an online Python programming course. We believe that such a self-study approach will be practical for other programming courses as well.

We investigated the recommendations (Q9) given by the students who had negative perspectives on Q3 (Fig. 10) and Q7 (Fig. 11) and thought that the deadline for each assignment was too tight to submit it multiple times for fixing CQIs. The students who had negative perspectives on Q4 (Fig. 10) and Q8 (Fig. 11) thought it was unfair that only the 100% correct programs could pass the code quality inspection. Therefore, the code quality inspection tools such as CQpy need to be more flexibly applied to partially correct programs.

6. Conclusion

Teaching code-quality-related knowledge is difficult through traditional "knowledge-inculcation" lecturing approaches in programming education because students require continuous practice for learning programming skills. Further, we cannot predict all potential CQIs in students' programs. We implemented an automatic quality inspector, namely CQpy, and conducted a study on applying CQpy as a subsystem of an online judge for checking code quality when students submit their Python programs for assessment.

The improvements in students' programming skills during the COVID-19 pandemic indicate that CQpy played a significant role in educating them on CQIs in programming. Moreover, this empirical study shows that the approach of applying CQpy can help students in different grade levels and majors. Most of the students showed positive responses to using CQpy, and even a few negative responses seem to arise from students' eagerness to improve the quality of their programs, even if they are partially correct. We believe that our self-learning approach facilitated by CQpy is not only effective for online courses but would also be helpful during in-person courses once the COVID-19 pandemic is over.

In the future, we plan to extend the CQpy in both online and in-person programming courses to help

[1] A pair-programming video is posted at https://youtu.be/ewKohpxXh7c, where the CQpy demonstration starts at 18:48.

students learn different programming languages. Once extended, we can publish it as an open-source program with a license compatible with SonarQube. We will enhance code-quality-related education in the programming lectures based on the classified common CQIs. We also intend to analyze the effectiveness of CQpy on a diverse spectrum of difficulties of assignments. Based on this analysis, we could introduce policies of allowing CQpy in complicated programming assignments, especially on deciding the due dates and the policy of applying CQpy to partially correct programs.

References

1. S. Wasik, M. Antczak, J. Badura, A. Laskowski and T. Sternal, A survey on online judge systems and their applications, *ACM Computing Surveys*, **51**(1), pp. 1–34, 2018.
2. H. Keuning, B. Heeren and J. Jeuring, Code quality issues in student programs, in *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, Bologna Italy, July 3–5, pp. 110–115, 2017.
3. L. Jiang, R. Rewcastle, P. Denny and E. Tempero, CompareCFG: Providing Visual Feedback on Code Quality Using Control Flow Graphs, in *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, Virtual, June 17–18, pp. 493–499, 2020.
4. K. K. Zaw, H. W. Hnin, K. Y. Kyaw and N. Funabiki, Software Quality Metrics Calculations for Java Programming Learning Assistant System, in *Proceedings of the 2020 IEEE Conference on Computer Applications*, Yangon, Myanmar, February 27–28, pp. 1–6, 2020.
5. Y. Wang, H. Li, Y. Feng, Y. Jiang and Y. Liu, Assessment of programming language learning based on peer code review model: Implementation and experience report, *Computers & Education*, **59**(2), pp. 412–422, 2012.
6. U. Costantini, V. Lonati and A. Morpurgo, How plans occur in novices' programs: A method to evaluate program-writing skills, in *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, Portland OR USA, March 11–14, pp. 852–858, 2020.
7. D. Silva, I. Nunes and R. Terra, Investigating code quality tools in the context of software engineering education, *Computer Applications in Engineering Education*, **25**(2), pp. 230–241, 2017.
8. D. Kirk, T. Crow, A. Luxton-Reilly and E. Tempero, On Assuring Learning About Code Quality, in *Proceedings of the Twenty-Second Australasian Computing Education Conference*, Melbourne VIC Australia, February 3–7, pp. 86–94, 2020.
9. M. Stegeman, E. Barendsen and S. Smetsers, 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, November 24–27, pp. 160–164, 2016.
10. L. W. Dietz, R. Lichtenthäler, A. Tornhill and S. Harrer, Code Process Metrics in University Programming Education, in *Proceedings of the 2nd Workshop on Innovative Software Engineering Education*, Stuttgart, Germany, February 19, pp. 23–26, 2019.
11. R. Kasahara, K. Sakamoto, H. Washizaki and Y. Fukazawa, Applying Gamification to Motivate Students to Write High-Quality Code in Programming Assignments, in *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, Aberdeen, UK, July 15–17, pp. 92–98, 2019.
12. S. H. Kan, Metrics and models in software quality engineering, Addison-Wesley Publishing Co., 75 Arlington Street, Suite 300 Boston, MA USA, p. 25, 2002.
13. V. Lenarduzzi, A. Sillitti and D. Taibi, A survey on code analysis tools for software maintenance prediction, in Proceedings of the 6th International Conference in Software Engineering for Defence Applications, Rome, Italy, June 7–8, pp. 165–175, 2018.
14. G. Campbell and P. P. Papapetrou, SonarQube in action, Manning Publications Co., 20 Baldwin Road PO Box 761 Shelter Island, NY, p. 7, 2013.
15. D. Falessi and A. Voegele, Validating and Prioritizing Quality Rules for Managing Technical Debt: An Industrial Case Study, in Proceedings of the 2015 IEEE 7th International Workshop on Managing Technical Debt, Bremen, Germany, October 2, pp. 41–48, 2015.
16. M. Foucault, X. Blanc, J. R. Falleri and M. A. Storey, Fostering good coding practices through individual feedback and gamification: an industrial case study, *Empirical Software Engineering*, **24**(6), pp. 3731–3754, 2019.
17. P. K. Sevella, Determining the barriers faced by novice programmers, Doctoral dissertation, Texas A&M University-Kingsville, 2013.

Xiao Liu is a PhD candidate in the Department of Electrical and Computer Engineering at the Pusan National University (PNU), Republic of Korea. He obtained both his Bachelor's and Master's degrees in Computer Science and Engineering from PNU. He developed an online program judging system called neoESPA. He has been maintaining neoESPA for evaluating students' assignments and serving as a Teaching Assistant for several programming courses since 2016. His main research interests include web development, program analysis, and cloud computing.

Gyun Woo is a Professor in the School of Computer Science and Engineering at the Pusan National University (PNU), Republic of Korea. Dr. Woo received his PhD in Electronic Engineering and Computer Science from the Korea Advanced Institute of Science and Technology (KAIST), where he took both his BS and MS in Computer Science. Dr. Woo works on program analysis and language design, especially for beginners. He is also interested in code security, functional reactive programming, and robot programming.