Static Analyses in Python Programming Courses

David Liu
University of Toronto
david@cs.toronto.edu

Andrew Petersen
University of Toronto Mississauga
petersen@cs.toronto.edu

ABSTRACT

Students learning to program often rely on feedback from the compiler and from instructor-provided test cases to help them identify errors in their code. This feedback focuses on functional correctness, and the output, which is often phrased in technical language, may be difficult to for novices to understand or effectively use. Static analyses may be effective as a complementary aid, as they can highlight common errors that may be potential sources of problems. In this paper, we introduce PyTA, a wrapper for pylint that provides custom checks for common novice errors as well as improved messages to help students fix the errors that are found. We report on our experience integrating PyTA into an existing online system used to deliver programming exercises to CS1 students and evaluate it by comparing exercise submissions collected from the integrated system to previously collected data. This analysis demonstrates that, for students who chose to read the PyTA output, we observed a decrease in time to solve errors, occurrences of repeated errors, and submissions to complete a programming problem. This suggests that PyTA, and static analyses in general, may help students identify functional issues in their code not highlighted by compiler feedback and that static analysis output may help students more quickly identify debug their code.

CCS CONCEPTS

• **Social and professional topics** → *Computer science education.*

ACM Reference Format:
David Liu and Andrew Petersen. 2019. Static Analyses in Python Programming Courses. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19), February 27-March 2, 2019, Minneapolis, MN, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3287324.3287503

1 INTRODUCTION

The debugging capabilities of novice programmers is of great importance to their overall success. In a study performed on novice computer science students, Ahmadzadeh et. al. noted that while less than half of the students who were good programmers were good debuggers, the majority of good debuggers were good programmers [1]. Debugging, however, is a difficult task for novices. The lack of experience novices possess regarding debugging strategies results in attempts at abstract debugging techniques which have seemingly little to do with the actual errors present in their code. This can culminate to novices eventually giving up altogether and instead simply trying fixes at random [11]. The difficulty students experience with this have been found to be at least partially attributed to the error messages they are presented with. Students often find these messages confusing due to reasons such as their technicality, brevity, and lack of visual aid [12, 18].

The confusion students face from standard error messages is not their only barrier. Novices could also use assistance with detecting errors involving semantics and logic [1]. For example, in our experience, students frequently write loops that always return in the first iteration. While this error may be obvious to a more experienced programmer or easy to find for a user with training in using a debugger, a novice still struggling with the concept of loops may have difficulty detecting this error. In particular, in data we collected for this report, one student made 15 consecutive submissions containing a loop that would always return on the first iteration. The student never seemed to realize that the cause of the error was an if/else statement within the loop. Instead, the student first made numerous modifications to the condition of the statement, and later proceeded to include an additional if/else statement within the block of the original if/else statement, eventually solving the programming exercise on the 38^{th} submission. Conventional compiler (or language runtime) error messages provided no aid in this situation or with other common novice issues such as maintaining consistent indentation in Python.

While errors in logic and various coding malpractices cannot be detected with standard error messages, they can be detected using static analysis. In this paper, we introduce a static analysis tool for Python called PyTA[1] that has been designed to identify common CS1 student errors and to provide intuitive error messages to aid them in debugging. To demonstrate how easy it is to add static analysis support to a course, we integrated PyTA into our online exercise system. When students submit their code, they are presented with both the standard errors detected by Python and errors reported by PyTA. We then evaluate the impact of the additional feedback on student performance, hoping to see students repeating errors less often and correcting logic errors more efficiently.

2 RELATED WORK

Providing enhanced error messages to assist novices in debugging has been attempted numerous times before. A number of these efforts have focused on improving compiler error messages, and results have been mixed. Rigby and Thompson [14] compare the usability of Eclipse to Gild, a perspective intended to be more appropriate for novices. One of the improvements that Gild made is the incorporation of more novice friendly descriptions of common

[1]PyTA is freely available. Please visit http://www.cs.toronto.edu/~david/pyta/quick_start.html for a quick start guide and https://github.com/pyta-uoft/pyta to access the project source code.

SIGCSE '19, February 27-March 2, 2019, Minneapolis, MN, USA

ACM ISBN 978-1-4503-5890-3/19/02…$15.00
https://doi.org/10.1145/3287324.3287503

compiler errors. In a study of 6 students who had 25 minutes to complete tasks for each IDE it was found that students using Gild had fewer errors and asked fewer questions, but neither of these differences were statistically significant. Becker [3] used Decaf, a Java editor, which provided enhancements for 30 different compiler error messages such as displaying the error in less technical terms, and providing suggestions to resolve it. In addition to the enhanced messages, steps were taken to improve readability such as only displaying the first enhanced error message. The enhanced editor was used for a four-week period by more than 200 students, and they found a reduced number of overall errors, errors per student, and repeated errors. For 8 of these errors there was a statistically significant reduction. Qualitative data also indicated the tool was less frustrating for students and made learning programming easier.

In contrast, both Denny et al. [6] and Pettit et al. [13] found little evidence that compiler error messages improved student outcomes. Denny et al. implemented an enhanced feedback system to users of CodeWrite, a web based tool for students to solve problems in Java [6]. They found that compiler error messages were sometimes ambiguous, so static analysis of code was performed to identify additional errors. The feedback provided by CodeWrite includes the line an identified error occurs on, as well as a more detailed explanation of the cause of the error. However, after analyzing data from 83 students in a CS1 course, it was concluded that enhanced feedback provided no significant benefit. Pettit et al. performed similar work at their institution and concur, stating that despite positive feedback from both students and instructors, they saw no measurable benefit in terms of a reduction in student submissions to programming problems [13].

To resolve these contradictory findings, Becker et al. [4] ran a controlled experiment where students identified and resolved syntax errors in code. They found that the enhanced errors did help, with students able to fix more errors with the enhanced support, but they also found that the enhanced errors did not reduce the number of non-compiling submissions or increase overall student performance. This suggests that compiler error messages can be improved but that more comprehensive supports are required.

A number of other efforts have utilized static analyses to identify and explain common errors, often with positive results. Flowers et al. [8] developed Gauntlet, a pre-compiler used to assist novice programmers by preprocessing Java code and providing more understandable explanations for common errors. From the use of Gauntlet for 18 months in a core CS1 course, it was determined that Gauntlet reduced the frustration caused during the debugging process, improved the quality if students' work, and lessened the amount of assistance sought after by students.

Truong et al. [19] created a static analysis tool to be used in aiding the students in writing better quality Java programs. The tool performed two analyses that examine the quality of the submitted code and how close the structure of the student's code is to a sample solution. At the time of the writing of their paper, the tool had been used in tutorial exercises with positive feedback overall.

Hristova et al. [9] created a static analysis tool, Expresso, intended to be used to assist novice programmers in identifying errors in their code in Java. The tool was created after research was performed on the most common Java errors as well as determining that the functionality of other existing tools did not coincide with the researchers' needs. Twenty of these errors were deemed essential and Espresso provides enhanced messages for these errors as well as suggested solutions. The authors of Espresso believe it will be useful for introductory Java classes, although at the time of the writing of the paper it had not yet been tested in a classroom environment.

Ayewah et al. [2] created FindBugs, an industrial strength tool for finding bugs in Java codebases. While not specifically built for novice programmers, it has been successfully applied to student code and found to be effective at finding such errors [17].

Schorsch [15] notes that even when there are no syntax errors in a program, logic errors can still result in an unexpected output. The Code Analyzer for Pascal (CAP) is a static analysis tool that was created to provide feedback on not only syntax errors, but logic and stylistic errors as well. This tool was used in a CS1 course of 520 students and a qualitative analysis of students' opinions were positive overall and instructors teaching the course all noticed in an increased quality of student programs. Delev and Gjorgjevikj [5] report their findings on common errors of C source code written by novice programmers detected by two different static analysis tools. Clang Static Analyzer and CppCheck were both run 13,960 source code files and the desired data was parsed from their reports. The results of the reports indicate a large presence of stylistic errors as well as errors that novices tend to not notice such as "uninitialized variable". The authors also note some of the students had more difficulty locating errors than fixing them, indicating the localization of errors through static analysis can be advantageous to the learning environment.

Most recently, Edwards et al. [7] collected errors from over 500,000 student program submissions using the Java static analysis tools Checkstyle and PMD. The authors highlighted some of the most common non-syntax related errors occurring in Java submissions and discussed how the frequency of errors detected by static analysis relate to factors such as experience with programming and time spent working on a problem. The authors also identify a number of problems with using static analysis for student programming problems, one of the largest of which is that students can perceive the errors detected by static analysis as purely cosmetic and begin to ignore them.

Overall, these efforts suggest that compiler error messages are difficult to use and that static analyses can provide additional information that students can use to more effectively debug programs. However, the messages presented must be carefully curated so that students value them, rather than dismissing them as addressing style. Furthermore, while a number of tools have been developed for a range of languages, relatively little work (excepting [7, 8] has been performed that demonstrates that these tools are effective within a classroom setting.

3 BACKGROUND

The two main tools described in this paper are PCRS [20] and PyTA [10]. PCRS is an open-source application for bundling interactive programming exercises with video-based instruction and has been presented previously. PyTA, the focus of this report, is a publicly available Python module[2] which uses existing static

[2]See https://github.com/pyta-uoft/pyta.

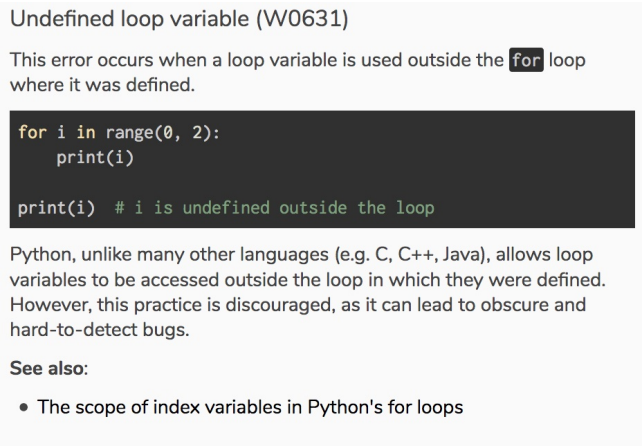

Figure 1: Example error message provided by PyTA

code analysis, provided by pylint [16], to help students find and fix common coding errors in introductory Python courses.

PyTA takes the form of a pylint wrapper [16] with numerous custom checkers and was developed with novice programmers in mind. It provides a number of custom analyses targeting common introductory errors, and it provides custom messages for warnings and errors that help identify possible causes for the message. Figure 1 provides an example of the information provided to help students diagnose errors. As shown in the figure, each message provides both a text description as well as an example and then information about why the error or warning is problematic.

To demonstrate how PyTA can be utilized by a variety of tools, we integrated it into PCRS so that students completing Python coding problems online would receive additional feedback. Integration into the system required about 20 hours of work by an undergraduate student unfamiliar with both projects.

When a student submitted a solution to a coding problem, they were presented with three pieces of feedback. The first was a message explaining if their submission passed or failed overall, along with the performance (pass or fail) of their submission on individual test cases. Second, they were provided with the errors, if any, emitted by the standard Python interpreter. Finally, for evaluation purposes, students were provided with a drop down box that, when clicked, provided the results of PyTA on their submission. The dropdown box allowed us to determine when students at least viewed the extra PyTA information.

3.1 Evaluation

We evaluated PyTA in a CS1 course taught at a large, research-intensive North American university. Over 11 weeks, students taking the course were provided with 40 Python coding exercises hosted on PCRS. The coding exercises consisted of a section of starter code for a function or class along with a docstring explaining what the final functionality of the function or class should be. The exercises were short with solutions averaging 6.30 lines, and students were given unlimited attempts to complete each one. These exercises accounted for 10% percent of a student's final grade.

Students were also provided with an introduction to PyTA in the form of additional exercises posted in PCRS. These exercises consist of a function or class with complete, but slightly erroneous, code. Students were expected to submit the code once, to see the PyTA output, and then to update the code to remove the PyTA error. The errors covered in these questions consisted of the most common errors that occurred in the previous offering of the course.

For each submission, a unique but not identifying student ID, the problem ID, the submitted code, the standard Python errors caused by the code, a time stamp of the submission, and the submission result (pass or fail) was recorded. Student interaction with the PyTA output was also stored: the output of PyTA and the number of times the PyTA dropdown box was clicked. Overall, 114865 submissions were collected from 750 students.

Control data was obtained from student submissions in a previous year. The content of the course was similar over both years, was taught by the same head instructor, and has comparable tutorial and PCRS exercises. However, PyTA output was not available for these students, so the control dataset includes PCRS submissions of coding exercises that were completed with no PyTA intervention. 106467 submissions were collected from 504 students.

3.2 Methods

To evaluate the impact of PyTA support, we take a quantitative approach. We count the number of submissions where a student responded to the intervention by clicking on the PyTA dropdown box to get additional information. We also investigate measures indicative of a student's performance on a programming exercise such as the number of submissions and the number of occurrences of particular errors. We compare these measures across the intervention and control groups.

Before analyzing the data, steps were taken to clean it. Since students are given unlimited attempts, they occasionally submitted an empty submission to examine the test cases or error messages as a form of hint. Similarly, students would occasionally submit identical code multiple times in a row, perhaps in frustration or in response to network latency. This inflates the number of errors and does not indicate anything of the students' capabilities, so all empty or repeated submissions were removed.

We also removed all submissions to optional programming exercises, since students choosing to complete these problems may not be indicative of the general course population. Finally, the problems intended to be an introduction to PyTA were removed, as the code provided in these problems was intentionally erroneous and since these problems were not provided to the control group.

4 DATA

In this section, we discuss the the PyTA usage habits of the students in the intervention group and how the performance of these students differs from that of the control group. We also perform inter-group comparisons of the intervention group based on their usage of PyTA. In the comparisons, we focus on five exemplar programming exercises, numbered in order of increasing difficulty as determined by the average number of submissions required to successfully solve the problem.

Table 1: Number of PyTA Clicks in a Sequence of Submissions

Number of Clicks	Count of Sequences	% Sequences
0	21172	67.68
1	5476	17.51
2	2032	6.50
3	1031	3.30
4	537	1.72
5+	1032	3.30

4.1 Do students use PyTA?

In order to determine if students were making use of PyTA, an event was recorded whenever a student clicked on the drop down box that contained the PyTA output. Of the submissions recorded for the required problems 28.6% were followed by a click on the PyTA dropdown box. Usage per sequence of submissions is similar to usage per submission. A sequence of submissions is defined as the collection of submissions made by a student for a particular problem. There were 31280 sequences of submissions with 32.3% of these viewing the PyTA output at some point in the sequence.

We used usage statistics to identify three categories of students from the intervention group. The categories consist of students who use PyTA frequently, infrequently, and an average amount as determined by mean absolute deviation. The usage of PyTA was measured by the average number of submissions that occur between students examining the results of the static analysis. We classified frequent users as those that used PyTA more than once every 2.04 submissions, infrequent users as those that used PyTA less than once every 4.54 submissions, and average users as those who used PyTA between the amounts of the frequent and infrequent groups. There are 133 frequent users, 138 infrequent users, and 478 average users.

Table 2 shows the average grades that each of the groups received for their assignments and exams, displayed in the order in which these assessments were due or taken. We see that that frequent user groups consistently perform approximately 3-5% better than the average usage group, and the average usage group consistently performs approximately 8-10% better than the infrequent users group. For this table, we used two sample t-tests to compare the marks. The difference in mark is always significant when comparing the frequent and average users group to the infrequent users group, but is only significant for the midterm when comparing frequent and average users. In all tables, superscript X denotes a p-value of <0.05 when compared with the group in the same row in column X, and superscript X^* denotes a p-value of <0.01.

4.2 Do students repeat errors less often?

We define a repeated error as an error of a specific type (as identified by PyTA) that occurs on consecutive submissions. We do not consider whether the error appears in a different location in the code; if the same error is reported in two consecutive submissions, it is a repeated error.

A Wilcoxon two sided rank sum test was performed on the control and intervention groups for each of the problems; a rank sum was used since the distributions of submissions typically have a long tail (are not normal). Table 3 shows the average number of repeated errors that occur in a submission for the 3 categories of the intervention group as well as the control group for each problem. In the first four problems examined, the control group maintains the highest percentage of repeated errors, seeing increases of more than 9% compared to the overall intervention group. All such differences were found to be significant. The frequent users group mostly sees the fewest percentage of repeated errors (except problem 4), but differences between the frequent and average groups are only significant for problem 2. The infrequent initially repeat errors less than the average user. In the most difficult two problems, the difference is significant.

Table 2: Average grade received on assignments and examinations

Category	*Frequent*	*Average*	*Infrequent*
Assignment 1	83.09^{3^*}	80.34^{3^*}	$70.36^{1^*,2^*}$
Assignment 2	72.02^{3^*}	68.45^{3^*}	$60.63^{1^*,2^*}$
Assignment 3	44.99^{3^*}	40.38^{3^*}	$31.93^{1^*,2^*}$
Midterm	$58.55^{2,3^*}$	$53.37^{1,3^*}$	$43.94^{1^*,2^*}$
Term Test	70.52^{3^*}	67.68^{3^*}	$58.44^{1^*,2^*}$
Exam	59.20^{3^*}	54.80^{3^*}	$43.49^{1^*,2^*}$
Course	66.01^{3^*}	62.61^{3^*}	$52.45^{1^*,2^*}$

4.3 Do students require fewer submissions to correct errors?

We define the submissions required to fix an error as the number of submissions after an occurrence of an error before that error is completely resolved. Completely resolved means that no later submission contains an occurrence of that error. We focus on only the top ten most common errors.

As seen in Table 4 the control group requires more submissions to resolve these errors in every case when compared to the overall intervention group. In the majority of these cases the differences are significant. In addition, the frequent users group solves errors in the fewest number of submissions as compared to the average and infrequent users. The differences in submissions required to solve any error between these groups never surpasses 1, although the largest differences are seen when comparing the frequent and infrequent groups for errors E9996 (always returning in a loop) and W0311 (bad indentation).

4.4 Do students pass exercises in fewer submissions?

Table 5 shows that the overall intervention group always requires less submissions than the control group, and in particular requires 2.54 fewer submissions for problem 2. Only the first three problems see significant differences between the two groups. Comparing the students in the intervention group: the frequent user group consistently requires the fewest submissions to solve each of the problems when comparing to the average and infrequent users groups, with the average users often requiring at least 25% more submissions than the frequent users and the infrequent users requiring more

Table 3: Average percent of errors repeated per submission

Problem	*Frequent*	*Average*	*Infrequent*	*Overall*	*Control*
1	11.11	18.00	11.43	15.86^{5*}	27.22^{4*}
2	17.13^{2}	22.85^{1}	19.47	21.34^{5*}	43.00^{4*}
3	12.95	14.924	15.44	14.82^{5*}	26.55^{4*}
4	16.37^{3}	12.28^{3*}	$23.11^{1,2*}$	15.77^{5*}	24.66^{4*}
5	18.35^{3*}	18.96^{3*}	$29.45^{1*,2*}$	21.28^{5}	23.96^{4}

Table 4: Average number of submissions to correct an error

Error Code	Description	*Frequent*	*Average*	*Infrequent*	*Overall*	*Control*
E0001-1	Invalid syntax	1.64	1.77	1.79	1.76	1.78
E0001-2	Unexpected indent	1.49	1.56	1.77	1.61^{5*}	1.73^{4*}
E0602	Undefined variable	$1.35^{2*,3*}$	$1.48^{1*,3}$	$1.58^{1*,2}$	1.49	1.54
E1101	No member	1.44^{3}	1.79	1.87^{1}	1.71	1.82
E9996	Always returning in a loop	$1.58^{2*,3*}$	2.06^{1*}	2.22^{1*}	2.06^{5*}	2.46^{4*}
W0104	Pointless statement	1.49	1.71	1.89	1.73^{5*}	1.97^{4*}
W0311	Bad indentation	1.72	1.90	2.23	1.98^{5}	2.50^{4}
W0612	Unused variable	$1.45^{2*,3*}$	1.78^{1*}	1.89^{1*}	1.78^{5*}	2.17^{4*}
W0613	Unused argument	1.53^{3*}	1.61^{3*}	$1.83^{1*,2*}$	1.65^{5*}	1.86^{4*}
W0621	Redefined outer name	2.20	2.27	2.26	2.26^{5*}	2.83^{4*}

than double the amount of submissions for problems 4 and 5. Nearly all of such differences are found to be significant, with the exception of the differences seen in the first problem. The average users group also see significantly fewer required submissions when compared to the infrequent users.

5 DISCUSSION

In each point of comparison, the intervention group performs better than the control group, and the users who frequently use PyTA perform better than the average and infrequent users. While this is positive and aligns with our hopes that providing additional, improved feedback from PyTA would lead to improved performance, the evidence only describes correlations; we cannot, with our study design, claim a causal relationship. It is, indeed, likely that frequent users would have higher course performance simply because they are likely to be students who use any and all resources available to master the course material.

In addition to limitations in the design of the evaluation of this tool, there are a number of other threats to validity that must be considered. The higher performance of the intervention group, overall, is more suggestive, but we still cannot guarantee that the difference is caused by PyTA. The instructors believed that students would perform better with the assistance of PyTA, so they did not feel that they could, ethically, restrict access to the tool. All students were allowed equal access. This made it necessary to perform a comparison with a previous year rather than separating students into those who receive assistance from PyTA and those who do not. The difference in populations is, by itself, a possible threat to validity. Performance did not improve gradually; the intervention groups solve problems faster and with less errors than the control group from the first week.

This effect could be caused by differences in instruction between the two years. While most of the material remained the same, one major difference between the two years was the inclusion of additional practice problems in the form of an introduction to PyTA. While the problems did not require students to write new code, they did require that students identify and fix errors, which may have resulted in the immediate improvement in performance in the problems students were required to submit.

A similar pattern holds within the three intervention groups themselves. The users who frequently use PyTA tend to outperform both the average and infrequent users. While we hope this is a result of their PyTA usage habits, their performance is higher than the other groups at the start. This suggests that the performance of students in these groups is a result of their attitudes or habits – not just PyTA usage. For example, the students who utilized PyTA the most probably also make maximal use of other resources (including the introduction to PyTA), leading them to perform better than students who do not, as a habit, use all of the information available.

Another possibility is that student performance may have increased as they received problems similar to those given in the previous year, so answers to these problems may have become more available. The online problems could be completed anywhere, and as a result, we cannot guarantee that students did not obtain assistance from outside sources. Since the solutions only involve a few lines of code, it is very difficult to tell if it is not the students' own work. Requiring that exercises be completed during tutorial sessions under the supervision of someone such as a teaching assistant may have helped prevent these cases. However, the purpose of tutorials is to proceed through more complicated exercises and review coding concepts, so this option was unavailable.

Table 5: Average number of submissions to pass an exercise

Problem	*Frequent*	*Average*	*Infrequent*	*Overall*	*Control*
1	1.31	1.50	1.55	1.47^{5^*}	1.68^{4^*}
2	$3.17^{2^*,3^*}$	$4.22^{1^*,3}$	$5.50^{1^*,2}$	4.24^{5^*}	6.78^{4^*}
3	$2.71^{2^*,3^*}$	$3.64^{1^*,3}$	$5.08^{1^*,2}$	3.70^{5^*}	5.04^{4^*}
4	$3.46^{2^*,3^*}$	5.23^{1^*}	8.07^{1^*}	5.35	5.91
5	$4.20^{2^*,3^*}$	$6.23^{1^*,3}$	$8.46^{1^*,2}$	6.19	6.91

Our final concern was our inability to determine if students were actually paying attention to the information provided to them in the PyTA dropdown box. Analyzing the click events can only tell us which students are accessing PyTA messages. It does not tell us if students are reading the messages or are just giving it a passing glance. In the future, eye tracking technology could be used to determine if the students are paying attention to the messages and whether they spend significant time perusing the messages.

6 CONCLUSION

The default Python syntax and runtime errors is a limited resource that is not easily accessible to novice programmers. Python makes use of a small number of error messages, making it difficult for students to use the messages to identify the source of the error. In addition, the errors do not detect many logical mistakes novices make. This insufficient feedback results in novices struggling with debugging and repeatedly making the same errors in logic, style, and syntax.

In this report, we present PyTA, a static analysis tool designed to check for common student errors and to present warnings and errors with effective, easy-to-understand messages. To evaluate the effectiveness of PyTA in a CS1 context, we integrated it into PCRS and used it to provide additional feedback when students completed online programming exercises. We compared student submissions in a year where PyTA was available to one where it was not and found that, overall, there is a significant reduction in the number of repeated errors per submission, submissions to pass a programming exercise, and submissions required to solve the most common errors. These effects were also seen when we compared students who used PyTA frequently to those who used it infrequently.

These results suggest that PyTA, and static analysis in general, may be an effective complement to language errors for students and may assist them with identifying and resolving common errors. We were able to integrate PyTA into our existing toolset with a minimal amount of effort and hope that other institutions will consider adopting it for their Python courses.

ACKNOWLEDGMENTS

We gratefully acknowledge the contributions of the many undergraduates at our institution who have contributed to the development of PyTA and PCRS. Special thanks goes to Kaveh Parsaie for his work integrating PyTA into PCRS and for his efforts to collect and analyze data on its impact.

REFERENCES

[1] Marzieh Ahmadzadeh, Dave Elliman, and Colin Higgins. 2005. An analysis of patterns of debugging among novice computer science students. *Acm sigcse bulletin* 37, 3 (2005), 84–88.
[2] Nathaniel Ayewah, David Hovemeyer, J David Morgenthaler, John Penix, and William Pugh. 2008. Using static analysis to find bugs. *IEEE software* 25, 5 (2008).
[3] Brett A Becker. 2016. An effective approach to enhancing compiler error messages. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. 126–131.
[4] Brett A. Becker, Kyle Goslin, and Graham Glanville. 2018. The Effects of Enhanced Compiler Error Messages on a Syntax Error Debugging Test. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18)*. ACM, New York, NY, USA, 640–645. https://doi.org/10.1145/3159450.3159461
[5] Tomche Delev and Dejan Gjorgjevikj. 2017. Static analysis of source code written by novice programmers. In *Global Engineering Education Conference (EDUCON), 2017 IEEE*. 825–830.
[6] Paul Denny, Andrew Luxton-Reilly, and Dave Carpenter. 2014. Enhancing syntax error messages appears ineffectual. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*. 273–278.
[7] Stephen H. Edwards, Nischel Kandru, and Mukund B.M. Rajagopal. 2017. Investigating Static Analysis Errors in Student Java Programs. In *Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER '17)*. 65–73. https://doi.org/10.1145/3105726.3106182
[8] Thomas Flowers, Curtis A Carver, and James Jackson. 2004. Empowering students and building confidence in novice programmers through Gauntlet. In *Frontiers in Education, 2004. FIE 2004. 34th Annual*. T3H–10.
[9] Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. 2003. Identifying and correcting Java programming errors for introductory computer science students. In *ACM SIGCSE Bulletin*, Vol. 35. 153–156.
[10] David Liu. [n. d.]. PyTA. https://github.com/pyta-uoft/pyta. Accessed: 2018-08-31.
[11] Laurie Murphy, Gary Lewandowski, Renée McCauley, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: the good, the bad, and the quirky–a qualitative analysis of novices' strategies. In *ACM SIGCSE Bulletin*, Vol. 40. 163–167.
[12] Marie-Hélène Nienaltowski, Michela Pedroni, and Bertrand Meyer. 2008. Compiler error messages: What can help novices?. In *ACM SIGCSE Bulletin*, Vol. 40. 168–172.
[13] Raymond S. Pettit, John Homer, and Roger Gee. 2017. Do Enhanced Compiler Error Messages Help Students?: Results Inconclusive.. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17)*. ACM, New York, NY, USA, 465–470. https://doi.org/10.1145/3017680.3017768
[14] Peter C Rigby and Suzanne Thompson. 2005. Study of novice programmers using Eclipse and Gild. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*. 105–109.
[15] Tom Schorsch. 1995. CAP: an automated self-assessment tool to check Pascal programs for syntax, logic and style errors. In *ACM SIGCSE Bulletin*, Vol. 27. 168–172.
[16] Sylvain Thénault. [n. d.]. Pylint. https://www.pylint.org/. Accessed: 2018-08-31.
[17] Marco Torchiano, Maurizio Morisio, et al. 2010. Assessing the precision of findbugs by mining java projects developed at a university. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*. IEEE, 110–113.
[18] V Javier Traver. 2010. On compiler error messages: what they say and what they mean. *Advances in Human-Computer Interaction* 2010 (2010).
[19] Nghi Truong, Paul Roe, and Peter Bancroft. 2004. Static analysis of students' Java programs. In *Proceedings of the Sixth Australasian Conference on Computing Education-Volume 30*. 317–325.
[20] Daniel Zingaro, Yuliya Cherenkova, Olessia Karpova, and Andrew Petersen. 2013. Facilitating Code-writing in PI Classes. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)*. 585–590.