pycheckmate – Addressing Challenges in Automatic Code Evaluation and Feedback Generation for Python Novices

Annabell Brocker[1] and Ulrik Schroeder[1]

Abstract: In academic settings, code assessment differs from traditional software testing by encompassing not only functional correctness but also critical structural aspects like naming conventions and programming concepts. Conventional static analysis tools like Pylint and Flake8, along with input-output unit tests, are deemed inadequate for introductory Python courses. To address this gap, this paper introduces `pycheckmate`, a library, tailored for automatic testing and targeted feedback in introductory Python programming courses.

Keywords: E-Assessment; Programming; Static Analysis; Automated Grading; CS1

1 Introduction

During the assessment process of programming code, automated testing is commonly employed to evaluate the code's static and/or dynamic properties. Statistical analyses are utilized to examine the properties without directly executing the code. Conversely, dynamic analyses involve the execution of the code to assess its functionality, often by comparing input and output [IL20]. The application of automatic assessment tests in introductory courses presents notable challenges, particularly when verifying fundamental aspects such as accurate variable and class naming, as well as the proper use of specific programming concepts [SG14]. Furthermore, code from novices often contains syntax errors, which complicate automated testing. Consequently, lecturers repeatedly create the same checks to initially evaluate the program code accordingly. In the Python programming language, code is dynamically interpreted, so that variables and other instances are not known to the compiler until runtime, whereas in other programming languages, such as Java, they are already known at compile time. Due to these traits, the analysis of specific programming concepts within Python can only be performed if the code has no syntax errors.

This paper therefore presents `pycheckmate`, a library that on the one hand helps lecturers to create automatic tests for introductory Python programming courses and on the other hand allows learners to get more specific feedback on their code (assessment).

2 Related Work

Automatic Assessments Tools The use of automatic assessment of programming tasks is becoming increasingly necessary, as manual corrections are very time-consuming and can

[1] RWTH Aachen University, Learning Technologies Research Group, Ahornstr. 55, 52074 Aachen, Germany
{a.brocker,schroeder}@cs.rwth-aachen.de, https://orcid.org/{0009-0007-6708-0892, 0000-0002-5178-8497}

doi:10.18420/abp2023-2

not be ensured with increasing numbers of novice programmers [Da18]. As a consequence, a wide spectrum of tools has been developed and is being used [SS22]. Reducing it to the programming language Python, still numerous tools are available for the purposes of static and/or dynamic testing, grading, and generation of feedback. An exemplar of an e-assessment system is *JACK*, which not only provides an integrated interface but also enables automated analysis of programming tasks, including Python and Java [Lo15; SG13]. It supports both static and dynamic tests, allowing learners to receive automatically generated feedback upon submission. JACK incorporates dedicated feedback mechanisms, such as variable allocation tables, specifically designed for programming tasks. The university of Turku designed its own web based learning environment called *ViLLE*, which includes task types for programming exercises that can be automatically assessed [LKR18]. Similar to JACK, ViLLE provides learners with immediate feedback upon submission. Additionally, learners have the capability to collaborate on multiple tasks and receive automatically shared feedback on their submissions. *Praktomat*[2], a programming course manager, provides support for the automatic assessment of programming tasks [BHS17]. Lecturers have the ability to define required, optional, and secret checks for submissions. Learners have the flexibility to submit their work at any time, with the required and optional checks being performed initially. If any of the required checks fail, the submission is not accepted. The results of the optional tests are immediately provided to the learner, while the results of the secret tests are revealed after the final accepted submission. The application of the required checks ensures that basic functionalities or requirements within the code are fulfilled, enabling subsequent tests to generate meaningful feedback. Unfortunately, this system lacks support for the assessment of Python programming tasks. Another autograding tool for Python assessments is *nbgrader*[3], which allows instructors to incorporate both public and hidden tests for static and dynamic analyses. It supports various packages, provided they are pre-installed. Learners receive a feedback file containing a comprehensive list of compiler messages, but this format may not be ideal for programming novices.

Static Code Analysis Tools Apart from the automatic assessments tools, Python also has static analysis tools like *Pylint*[4], *Pyflakes*[5] or *flake8*[6]. A comparison conducted by [GP19] evaluates these and other static analysis tools for Python, considering diverse requirements, and provides recommendations accordingly. However, none of these tools are specifically designed for assessing tasks of novice programmers. This limitation arises due to the unique considerations required when examining code from novices, including proper variable, function, and class naming, as well as the correct usage of specific programming constructs [SG14]. *PyTA*[7], for example, is a static code analysis tools built on top of Pylint, but presents

[2] Praktomat, `https://github.com/KITPraktomatTeam/Praktomat/`, Last accessed: 14.06.2023
[3] nbgrader, `https://github.com/jupyter/nbgrader/`
[4] Pylint User Manuel, `https://docs.pylint.org/`, Last accessed: 02.06.2023
[5] Pyflakes GitHub, `https://github.com/PyCQA/pyflakes`, Last accessed: 02.06.2023
[6] Flake8: Your Tool For Style Guide Enforcement, `https://flake8.pycqa.org/en/latest/`, Last accessed: 02.06.2023
[7] PyTA, `https://github.com/pyta-uoft/pyta`, Last accessed: 28.08.2023

error and feedback messages in a manner that is more accessible to novice programmers compared to Pylint itself [LP19]. Currently, PyTA offers more than 100 distinct error and feedback messages, although this count may vary as development continues. However, PyTA is not suitable for assessment purposes due to challenges in configuration for lecturers and the absence of important checks required for automatic assessment, such as verifying correct parameter naming. Instead, PyTA focuses on providing valuable feedback to novice programmers within the context of conventional development processes. A further analysis tool for Python is *Scalpel*[8], which includes several features such as the construction of a control-flow graph and the representation of static single assignment [LWQ22]. This library proves to be particularly valuable for visualizing step-by-step code execution for novice programmers. Consequently, Scalpel not primarily intended for automatic evaluation, but rather serves as a means to automatically generate more detailed and comprehensive feedback. *ASPA*[9] is primarily dedicated to analyzing programming code generated by novice programmers in order to provide feedback [Lu22]. It offers a graphical user interface that facilitates the insertion of code files for examination. A study was conducted to evaluate the usefulness of ASPA, among other aspects. The findings revealed that ASPA significantly assisted the majority of novice programmers with their weekly exercises. However, the investigations conducted with ASPA were solely general in nature and did not include contextual investigations. Regrettably, the documentation available for this tool is insufficient, preventing a definitive conclusion on its effectiveness from being drawn at this stage.

Abstract Syntax Tree An Abstract Sytnax Tree (AST) serves as a hierarchical representation of program source code in the form of a tree structure. The Python programming language provides a built-in module called *ast*[10], which allows the conversion of Python code into a valid AST. By examining the transformed code within the AST, specific properties can be verified, such as the presence of particular names or the usage of specific libraries. For instance, Fig. 1 illustrates the AST derived from the one-liner `a = max (1, 2, 3)`. To determine whether the learner has implemented a variable named `a`, one can traverse the individual nodes of the tree until the corresponding nodes are found. In this case, it would be via `source_tree.body[0].targets.id` if the code has been previously parsed and stored in the `source_tree` variable. Unfortunately, only Python code devoid of SyntaxErrors can be successfully converted into a corresponding AST using the built-in ast module. Numerous existing tools, including Pylint, Pyflakes, and ASPA, rely on the conversion of code into an AST for their functionality.

Feedback In the domain of learning and teaching, feedback is an indispensable element for enhancing effectiveness [HT07]. This also applies in the area of automated assessment,

[8] Scalpel: The Python Static Analysis Framework, `https://github.com/SMAT-Lab/Scalpel`, Last accessed: 02.06.2023

[9] ASPA - Abstrakti SyntaksiPuu Analysaattori, `https://github.com/RoopeLuukkainen/ASPA`, Last accessed: 02.06.2023

[10] ast — Abstract Syntax Trees, `https://docs.python.org/3/library/ast.html`, Last accessed: 12.05.2023.

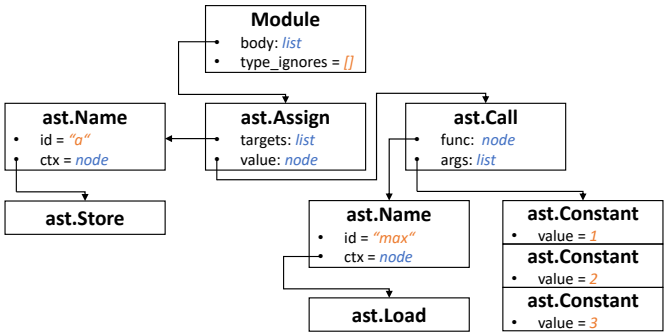

Fig. 1: Abstract Syntax Tree for Python code: `a = max (1, 2, 3)`

where feedback is generated automatically. Especially for programming novices, the display of an extensive callstack of compiler messages often proves to be unhelpful, as it can lead to cognitive overload [WLG12]. Additionally, novice programmers benefit from feedback that recognizes even the smallest accomplishments, providing positive reinforcement alongside identification of errors [MAG18]. Within programming courses, feedback can be classified into various types [KJH18; Na08]. Multiple learning environments, like previous presented tool JACK, have been investigated in terms of their feedback they provide and, based on this, developed guidelines on when and how to intervene [Je22]. Extensive research has demonstrated that elaborative feedback, such as compiler messages or their customization specifically for novice learners, greatly contributes to the learning process [Ha21].

3 pycheckmate

`pycheckmate` is a library designed to analyze Python code for various properties, focusing primarily on the evaluation of code authored by programming novices. A first prototype was developed as part of a bachelor thesis [St23]. Unlike other statistical analysis tools, `pycheckmate` does not primarily address code styling; instead, it enables the examination of SyntaxErrors, the usage of programming concepts, and the correct designation of variables, functions, classes, and other entities. All analysis are based on the AST of a Python program code. Through interviews with tutors who supervised programming lectures for Python programming novices, the most relevant aspects to investigate were identified and summarized in Tab. 1. To utilize the library, the code to be examined must be transformed once into an AST by initialising a `pycheckmate` object. This approach optimizes execution time and memory usage. Subsequently, each test can be invoked on the object instance, returning a dictionary indicating the success (`bool`) of the test and a corresponding feedback message (`str`). Intentionally, no exceptions are thrown to prevent the termination of subsequent tests, unless significant configuration is required. Within the scope of `pycheckmate`, the examination of code is limited to its current state. For instance, the analysis of indirect recursion within a function is contingent upon the presence of the

relevant functions within the code itself. `pycheckmate` is an actively evolving tool that is openly accessible to users[11].

Requirement	Explanation
Code Compilation	Check if the code has SyntaxErrors or not.
Variable Name	Check if a specific variable is defined (inside a function).
Function Name	Check if a specific function or multiple function are defined.
Class Name	Check if a specific class is defined.
Function / Class Parameters	Check if a function / class has specific parameters including correct naming, correct type as well as default value.
Module Import	Check if a specific module or a specific function from a module has been imported.
Module Usage	Check if specific function from a module has been used.
Function Usage	Check if specific function has been used.
Programming Concept Usage	Check if the the following programming concepts have been used: for including iteration range, while, lambda, comprehension, try-except.
Recursive Function	Check if a function is recursive (direct and indirect).
File Handling	Check if a file was opened, closed or both via with.

Tab. 1: Results from requirements evaluation.

Feedback Generation To cater to the needs of novice programmers, in line with the recommendations of [MAG18], the library provides static checks for even the most basic and successful programming activities. These checks include feedback messages specifically tailored for novice learners. For example, the function $function\_has\_parameters$, which configuration can be found in List. 1, informs novices which necessary named parameters are still missing or, if the parameters are available, whether the type and the default value are also correct:

Parameter 'param1' of function 'test_function' has wrong default.
Expected default: '0', got '1'.

Feedback messages are also generated for passed tests, and not only for failed tests to honour successful actions. When certain checks rely on the presence of others, for instance, a specific function's existence, such interdependencies are communicated to the learner as well. The messages are currently provided in English, but should also support multilingualism in the future. If the pre-configured feedback messages are not sufficient, they can be overwritten or extended by the lecturers. The timing of feedback message delivery to learners is presently determined by the lecturer due to the underlying architecture.

Performance Measures `pycheckmate`'s performance was rigorously assessed through extensive testing, measuring execution time and memory usage. Execution time was recorded as the duration between the start and end of each check, and memory consumption was

[11] pycheckmate, `https://doi.org/10.17605/OSF.IO/BR68W`, last accessed 28.08.2023

analyzed using the `tracemalloc` module. We conducted 1000 iterations on various devices, including a Windows 10 Pro laptop (i7-10510U, 16 GB RAM) and a Raspberry Pi 4 (4 GB RAM) running Raspberry Pi OS. Results showed execution times were device-dependent, as detailed in Table 2. Nevertheless, `pycheckmate` performed well on typical devices encountered in practice.

	Laptop	Raspberry Pi 4
Execution Time Average	~2.5 ms	~11.2 ms
Memory Usage Average	~0.15 MB	~0.05 MB

Tab. 2: Performance measurements of `pycheckmate`.

Use Cases The presented library exhibits versatile applicability in various scenarios. Firstly, it can be traditionally employed for evaluating programming tasks, enabling lecturers to seamlessly integrate the library into their existing frameworks and leverage the pre-defined tests. This integration offers lecturers enhanced accessibility and efficiency in automating the assessment of Python code. Additionally, the library finds utility in both summative and formative e-examinations, where automatic evaluation mechanisms are employed for subsequent analysis. One notable challenge in utilizing automatic evaluation mechanisms in e-examinations, where learners are restricted to using basic text editors without execution capabilities, lies in the inability to automatically evaluate the code itself due to syntax errors or incorrect variable and class naming. To address this, the library can be employed to verify specific code specifications, allowing students to receive feedback on those aspects, similar to the required checks implemented in the Praktomat programming course manager. However, incorporating student executability would entail certain trade-offs, such as rendering certain tasks, like assessing code reading comprehension, unfeasible within the examination context. For e-examination systems conducted through web applications, the library can be integrated using Pyodide[12], provided that appropriate interfaces are established beforehand within the e-examination system. An example to integrate `pycheckmate` can be found in List. 1.

```
from pycheckmate import PyCheckMate

#store code as str in variable source_code
with open("testing_file.py") as file:
    source_code = file.read()

reqs_args = {
    'param1': { 'default': 0 },
    'param2': { 'type': int }
}

pcm = PyCheckMate(source_code)
```

[12] Pyodide, `https://pyodide.org/en/stable/`, Last accessed: 26.05.2023

```
check_func_name = pcm.has_function("test_function")
check_func_params = pcm.function_has_parameters("test_function",
    required_args=reqs_args, required_vararg=False, required_kwarg="
    kwarg")
```

List. 1: Example configuration to integrate `pycheckmate` into an automatic assessment tool.

4 Conclusion and Future Work

In this paper we discussed problems of automatic assessment for introductory programming courses and presented the library `pycheckmate` which is designed to analyze Python code accordingly. The library not only checks for the existence of basic programming concepts, but also generates feedback suited to novices, which can, however, be adapted by the lecturer according to the context. Up to now, the feedback messages have not yet been evaluated with the actual user group (programming novices). This must be done retrospectively in order to also prove the positive effect of the messages adapted to the novices. As part of a requirements analysis, fundamental features were identified and implemented. Nevertheless, the library is in continuous development, so that further features will be added in the future. Furthermore, it might be possible to integrate code styling checks based on existing static analysis tools, bundling the most important code styling checks for programming novices in one library.

Bibliography

[BHS17] Breitner, J.; Hecker, M.; Snelting, G.: Der Grader Praktomat. In: Automatisierte Bewertung in der Programmierausbildung. Digitale Medien in der Hochschullehre 6, Waxmann Verlag GmbH, pp. 159–172, 2017, URL: https://www.waxmann.com/automatisiertebewertung/.

[Da18] Dawson, J. Q. et al.: Designing an Introductory Programming Course to Improve Non-Majors' Experiences. Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE)/, pp. 26–31, 2018.

[GP19] Gulabovska, H.; Porkoláb, Z.: Survey on Static Analysis Tools of Python Programs. In: Software Quality Analysis, Monitoring, Improvement, and Applications (SQAMIA). 2019.

[Ha21] Hao, Q. et al.: Towards understanding the effective design of automated formative feedback for programming assignments. Computer Science Education 32/, pp. 105–127, 2021.

[HT07] Hattie, J. A. C.; Timperley, H. S.: The Power of Feedback. Review of Educational Research 77/, pp. 112–81, 2007.

[IL20] Ismail, M. H.; Lakulu, M. M.: A Critical Review on Recent Proposed Automated Programming Assessment Tool. Psychology and Education/, pp. 1049–1060, 2020.

[Je22] Jeuring, J. et al.: Towards Giving Timely Formative Feedback and Hints to Novice Programmers. Proceedings of the 2022 Working Group Reports on Innovation and Technology in Computer Science Education/, 2022, URL: https://api.semanticscholar.org/CorpusID:255226835.

[KJH18] Keuning, H.; Jeuring, J.; Heeren, B.: A Systematic Literature Review of Automated Feedback Generation for Programming Exercises. ACM Transactions on Computing Education (TOCE) 19/, pp. 1–43, 2018.

[LKR18] Laakso, M.; Kaila, E.; Rajala, T.: ViLLE – collaborative education tool: Designing and utilizing an exercise-based learning environment. Education and Information Technologies 23/, pp. 1655–1676, 2018.

[Lo15] Lohmann, E.: Erweiterung eines E-Assessment-Systems um eine Prüfkomponente für die Programmiersprache Python. In: Proceedings of the Second Workshop "Automatische Bewertung von Programmieraufgaben". 2015.

[LP19] Liu, D.; Petersen, A.: Static Analyses in Python Programming Courses. In: Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE). Pp. 666–671, 2019.

[Lu22] Luukkainen, R. et al.: ASPA: A Static Analyser to Support Learning and Continuous Feedback on Programming Courses. An Empirical Validation. 2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)/, pp. 29–39, 2022.

[LWQ22] Li, L.; Wang, J.; Quan, H.: Scalpel: The Python Static Analysis Framework. ArXiv abs/2202.11840/, 2022.

[MAG18] Mattheiss, S. R.; Alexander, E. J.; Graves, W. W.: Elaborative feedback: Engaging reward and task-relevant brain regions promotes learning in pseudoword reading aloud. Cognitive, Affective, & Behavioral Neuroscience (CABN) 18/, pp. 68–87, 2018.

[Na08] Narciss, S.: Feedback Strategies for Interactive Learning Tasks. In: Handbook of Research on Educational Communications and Technology (3rd ed.) Pp. 125–144, 2008.

[SG13] Striewe, M.; Goedicke, M.: JACK Revisited: Scaling Up in Multiple Dimensions. In: European Conference on Technology Enhanced Learning (ECTEL). Pp. 635–636, 2013.

[SG14] Striewe, M.; Goedicke, M.: A Review of Static Analysis Approaches for Programming Exercises. In: Computer Assisted Assessment. Research into E-Assessment (CAA). Pp. 100–113, 2014.

[SS22] Strickroth, S.; Striewe, M.: Building a Corpus of Task-Based Grading and Feedback Systems for Learning and Teaching Programming. Int. J. Eng. Pedagog. 12/, pp. 26–41, 2022.

[St23] Stuermer, M.: Development of a library to analyze Python code used in ACE editor, Bachelor's Thesis, RWTH Aachen University, 2023, URL: https://doi.org/10.18154/RWTH-2023-03637.

[WLG12] Watson, C.; Li, F. W. B.; Godwin, J. L.: BlueFix: Using Crowd-Sourced Feedback to Support Programming Students in Error Diagnosis and Repair. In: International Conference on Advances in Web-Based Learning (ICWL). 2012.