A Grammar and Parameterization-Based Generator for Python Programming Exercises

Philipp Peess[1], Annabell Brocker[2], Rene Roepke[2] and Ulrik Schroeder[2]

Abstract: As the importance of programming education grows, the demand for a sufficient number of practical exercises in courses also increases. To accommodate this need without significantly increasing the instructors' workload, a programming exercise generator capable of generating exercises for independent practice is considered. This research mainly focuses on determining suitable generation methods and creating a modular and extensible generator structure. The current generator implementation uses parameterization and a grammar-based generation approach in order to provide generated exercises directly to students in their programming environment. Furthermore, the generator can act as a foundation for further research and be extended with additional generation methods, creating the possibility of exploring artificial intelligence for the generation of programming exercises.

Keywords: Automatic Generation; Programming Exercises; Python; JupyterLab

1 Introduction

An important aspect of programming education is the provision of practical exercises, which can improve both the students' theoretical and practical knowledge [BE15], tailored to varying skill levels, possibly through personalization [Of17]. Programming courses often require a substantial number of practice tasks, but manual creation demands considerable time and expertise and may not meet individual learning needs, such as the need for numerous tasks related to a specific programming concept. The work aims to design and develop `pygenaix`, an automatic generator for programming exercises. Importantly, students must be able to self-evaluate their solutions to these exercises to enable independent practice.

For the design and development of the system, an introductory programming course at a German university, in which Python is introduced to a wide variety of non-CS students, was choses as a reference. In this paper, the following two research questions are addressed as part of the design and development of the exercise generator: **(RQ1)** “Which task generation methods are suitable for exercises used in an introductory programming course?” and **(RQ2)** “How can the generator implementation be extensible and easy to use in the course's learning environment?”. For future developments and further research, the resulting source code of the exercise generator was made openly available.[3]

[1] RWTH Aachen University, Templergraben 55, 52062 Aachen, Germany, philipp.peess@rwth-aachen.de
[2] RWTH Aachen University, Learning Technologies Research Group, Ahornstr. 55, 52074 Aachen, Germany, {a.brocker, roepke, schroeder}@cs.rwth-aachen.de, https://orcid.org/{0009-0007-6708-0892, 0000-0003-0250-8521, 0000-0002-5178-8497}
[3] pygenaix, `https://doi.org/10.17605/OSF.IO/TKHR3`, last accessed 15.09.2023

doi:10.18420/abp2023-6

2 Related Work and Approaches to Automatic Generation

When reviewing related work on the automatic generation of programming exercises, it was found that so far only little research was conducted in this field, and among the existing works, a range of different generation methods is addressed. As such, [ROS10] employed *parameterization* by developing a generator which replaces placeholders in both the task description and an example solution to create new tasks with small but intentional variations. In other works, *models* were transformed to programming exercises, ranging from mathematical formulas to decision trees and complex graph structures (e. g. [So21; TS18]). Similarly, [Ad19] used a *context-free grammar (CFG)* to generate a large number of example programs by defining a grammar covering the core structure of select Python programming concepts. Importantly, this grammar does not yet allow for the generation of a task description, but was nonetheless capable of generating suitable code fragments focused on specific topics. In [Sa22], an *artificial intelligence (AI)*, *OpenAI Codex*[4], was primed using example exercises consisting of a task description, a solution, tests and keywords, which describe the scenario and used programming concepts. When also providing new keywords, a new task could be generated. Lastly, [WM15] and [Ha18] used *fault injection* to modify a sample solution to a task description by introducing errors and then providing this faulty solution to students, which would have to find all errors. Consequently, this method is not suited for generating free programming exercises for the considered course.

Besides these works, the use of content generation methods in other domains was investigated, e. g. for the generation of block-based programming exercises [Ba17] and quizzes [Ku20] in an educational context, or the generation of unit tests [Se19] as a more general application domain. As these generation approaches usually only focus on one specific aspect, they are unsuitable for the generation of whole programming exercises.

Among the presented methods found in related work, parameterization-, model-, grammar- and AI-based approaches were the most promising candidates for further exploration. To determine the suitability of the generation methods in the context of the considered course, a requirement analysis was conducted, focusing on the needs of tutors (i. e. instructors) and students. As such, interviews with tutors were conducted to collect insights into the current task creation process as well as their perception of students' knowledge levels and common problems when dealing with programming exercises. Additionally, the course materials and exercises were analysed to identify suitable task types for automatic generation.

Firstly, parameterization provides a comparably low-complexity solution for generating programming tasks. It can easily be implemented on existing tasks by introducing placeholders and sets of values, thus creating task templates. Additionally, the process of replacing placeholder values is commonly used, even in other generation methods, and should therefore be implemented in any case as a possible post-processing step for the outputs of other generation methods. However, it also has major disadvantages, as the variability of the generatable tasks is strongly restricted by the fact that only specific sections of the input,

[4] OpenAI Codex, https://openai.com/blog/openai-codex, last accessed 30.05.2023.

i. e. the placeholders, can be changed. As such, another generation method offering more variability should be considered.

The remaining approaches offer a higher degree of variability, but also a high level of complexity. For model-based generation approaches, this may be controllable based on the specific model that is chosen for the generation process. More flexibility can be achieved using a grammar-based approach, as a grammar could also be used to generate different kinds of models. Additionally, a study showed that 93.1 % of students strongly agreed that the programs generated using a grammar-based approach can help them in practice and improve programming skills [Ad19]. Consequently, a grammar-based generation method was chosen as the second generation method the programming exercise generator offers.

AI-based generation methods were also considered, but due to the problems that occurred in the study presented in [Sa22], the reliability of the approach was questioned. In [Sa22], a sample set of 240 exercises generated by an AI was analysed and it was found that an example solution was missing in about 15 % of the cases and tests were missing in almost 30 % of the cases. Additionally, the sample solution passed the tests in less than a third of the cases it was generated with a solution, indicating that one of the two was faulty. Furthermore, it should be noted that AIs are generally black boxes, which makes it impossible to predict the next output for any given input. This makes it difficult to use AI as a direct source for programming exercises and makes an intermediate (human) control instance almost mandatory, which would likely introduce additional manual work. Also, when using a third-party solution, the training data is mostly unknown, making it hard to predict what the AI is capable of generating and may subsequently require manual rework to verify the usefulness of the task. Similar findings have been shown in another study where a purely AI-based approach with a large language model and prompts was chosen [SMB23]. While training an AI for programming exercise generation would generally be possible, it would be a complex problem and require sufficient training data. As such, we decided against implementing an AI-based generation approach in the initial version of the generator. However, with the recent advancements in large language models in AI that occurred during the research process of this work, this should be reevaluated. Additional research in this area should be conducted to investigate whether new AI-based tools are more capable of generating suitable and correct programming exercises.

3 Generator Design and Structure

The generator primarily focuses on creating exercises for novice programmers, emphasizing algorithm-oriented input-output tasks rather than complex software architecture challenges. Lecturers should be able to control the task topics themselves by specifying templates, so that, for example, tasks only focus on the topics of variable declaration and initialisation. A central goal in the design and implementation of a generator for programming exercises was to establish an extensible and modular software structure. Consequently, the generator was implemented in a plugin-based architecture, visualised in Figure 1.

As a first step, all external data was separated from the implementation of the generator itself. This includes the definition of the generation capabilities, such as data for replacing placeholders and the grammar definitions, as well as task blueprints, which act as templates for the generation process. These blueprints define the generation steps and additional metadata, such as topics a task may cover or potential filtering criteria like the task's difficulty. The resulting separation of the generator implementation and external data enables adding new content to the generator without modifying its implementation directly.

The generator itself is split into the main generator and multiple task generators. The main generator acts solely as a management component focused on loading and providing external data as well as controlling the generation process. When tasked with the generation of a new exercise, the main generator selects a blueprint and delegates the generation to the respective task generators. Each task generator implements a generation method, currently for both a parameterization- and a grammar-based generation approach. During the generation, multiple task generators can be used in sequence by using the output of one task generator as input for another task generator. This allows for the generation of more complex tasks while reducing the functionality each task generator has to provide, facilitating the introduction of new generation methods.

Parameterization: Fundamentally, the parameterization-based task generator replaces placeholder values with randomly selected entries from predefined data sets. These placeholders contain identifiers, making it possible to link related placeholders. Additionally, the data set entries are not single words or values, but instead take on the form of dictionaries to provide data as key-value pairs. This way, different grammatical forms and metadata providing further information on the entry, like associated topics, can be grouped. Conditions may then be used to select only specific entries matching certain criteria, allowing the creation of consistent and complex tasks. To facilitate the process of defining templates, additional predefined commands were added to the generator, supporting operations for grammatical adjustments, e. g. choosing *a* and *an*, or common operations like randomly generating

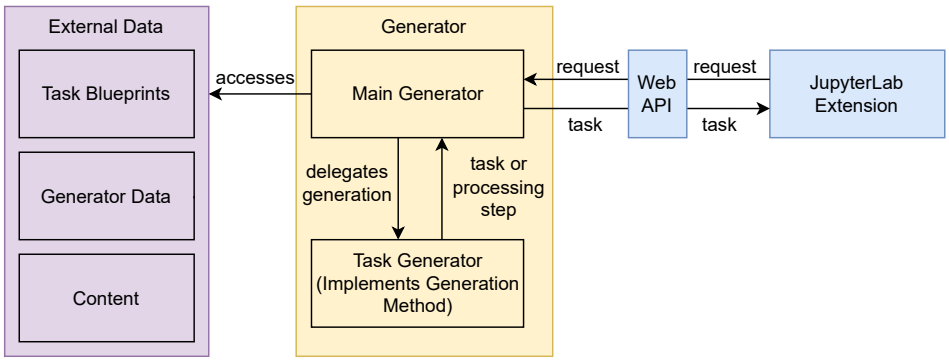

Fig. 1: System architecture visualising the different components and their interactions.

numbers. Furthermore, the parameterization-based generator supports the definition of any verification method, allowing for the automatic generation of tests and example solutions the students can then use to verify their solution.

Grammar-Based Generation: The foundation of the grammar-based generator is a modified version of a CFG (similar to [Si12]), which provides additional symbol types: *Meta symbols* may be defined to provide information for the post-processing steps and are not translated to an output value when evaluating the resulting symbols after the grammar evaluation; by using *direct symbols*, it is possible to define new content directly in the textual input instead of creating new grammar symbols, allowing for the creation of content for single tasks without bloating the grammar. During processing, both symbol types behave the same way a terminal symbol would and after resolving all non-terminal symbols, only terminals, meta symbols and direct symbols remain. Next, post-processing steps introduce variables and ensure newly generated functions and classes can be instantiated and called. This is done by analysing the current symbol sequence and searching for specific patterns that indicate the definition of functions and classes and determining their parameters and attributes. Any references to newly defined functions, e. g. function calls or class instantiations, may then be replaced with one of the generated elements. The processed grammatical representation of a Python program is then directly translated to source code. Importantly, the grammar also defines text values from which a task description can be derived, ensuring that the example solution matches the task description. Contrary to parameterization, the grammar-based approach allows only for the generation of an example solution and a matching task description. As such, an automatic verification is currently not possible. However, since practice tasks generally only encompass short tasks and code fragments, tools for generating unit tests from code, e. g. Pynguin[5], could potentially be used to generate tests automatically.

4 Available Interfaces for Stakeholders

To make the generator available for requests (e. g. from a learning environment), a web API was implemented using a *Flask*[6] server. It provides routes for all central functionality of the generator. Importantly, this includes a route for requesting new tasks from the generator which allows for configurations regarding the topic, the output format, requesting a download of the file as well as filtering criteria for the blueprint selection process. Additionally, it is possible to request information regarding the existing blueprints.

To simplify the process of generating exercises for students, a graphical user interface (GUI) in the form of a JupyterLab extension[7] was implemented. It groups different blueprints under topics and groups of topics under overarching categories corresponding to the course structure to guide students through choosing adequate tasks for their current knowledge

[5] Pynguin, https://pynguin.readthedocs.io/en/latest/index.html, last accessed 14.06.2023.
[6] Flask, https://flask.palletsprojects.com/en/2.3.x/, last accessed 31.05.2023.
[7] JupyterLab Extensions, https://jupyterlab.readthedocs.io/en/stable/extension/extension_dev.html, last accessed 05.06.2023.

level. For more control, attributes like the preferred difficulty of the task can be specified. Upon submission, a request is sent to the web API, which returns an ad-hoc-generated task matching the specification and opens it as a Jupyter Notebook in the lab environment.

Lastly, to facilitate the development and the addition of new blueprints, a testing interface for instructors (particularly tutors) was needed. In this context, a command line interface was implemented, simplifying access to the generator functionality with additional options.

This way, generation can be done in bulk and can be restricted to specific blueprints. As such, the interface facilitates testing new blueprints locally instead of having to move them to the server first. It also supports the management of additional blueprints for exam task generation which are usually not shared with students for preparation.

5 Evaluation and Future Work

To determine the generator's suitability, a preliminary user evaluation was conducted with tutors that already participated in the initial interviews. In a second round of interviews, the tutors provided feedback regarding both the generatable content as well as the process of defining new task types as blueprints. Regarding the former, the evaluation focused on five criteria: (1) the possibilities for personalization, (2) the support of storytelling, (3) the variability of the generated content and their fit regarding (4) foundational topics (e. g., loops, recursion) and (5) advanced problems (e. g., sorting algorithms). The tasks were generally received favourably regarding these criteria, but in some cases, especially (4), the tutors had vastly different opinions, which underlines the need for a comprehensive evaluation with students and particularly novice programmers.

The evaluation also uncovered some problems of the implementation that still have to be addressed. On the one hand, task descriptions should be improved in two central ways: Firstly, the storytelling provided in the tasks was problematic, as the generated code did not necessarily make sense in the described scenario and therefore could be perceived as confusing or misleading by the students. Secondly, task descriptions were, at least in the case of the grammar-based generation method, largely very direct descriptions of the solution. However, with a growing knowledge level as well as task difficulty, more abstraction of task descriptions regarding the sample solutions would be favourable. An additional limitation of the grammar-based approach is the imperative nature of the task descriptions derived directly from the imperative code, which does not match natural language well in many cases. On the other hand, the generation can currently lead to unfavourable results, e. g. unnecessary conditions like the comparison of a variable with itself. To counteract this, static code analysis should be introduced to automatically detect and prevent these cases.

Furthermore, the interviewed tutors generally agreed that the task definition process is understandable, but noted the need for documentation to simplify the blueprint creation process. Especially the grammar-based approach requires sufficient knowledge of the existing

symbols in order to provide valuable blueprints. Consequently, the improved documentation now contains an extensive overview of all allowed symbols. However, there could be more assistance for defining tasks, e. g. in the form of a GUI.

Finally, the generator provides a foundation for follow-up research and the introduction of further generation methods due to its modular and plugin-based architecture. The capabilities of different generation methods should be explored further, especially with regard to the potential benefits of AI. Another benefit of AI would be for post-processing generated tasks and task descriptions, e. g. to improve current problems with storytelling. Further, it would be interesting to investigate the benefits of AI for the generation of blueprints to facilitate the process of defining new task types. Also, AI could be used to generate test cases for generated tasks. Lastly, another application of AI would be the translation of generated task descriptions into different languages, as the generator currently only supports English tasks.

Regarding the general capabilities of the generator, the number of generated instances depends on the definition of the task blueprints, the parameterization data and the grammar definition. The number tasks directly correlates to the number of key-value pairs available and the number of values per key. The grammar may contain rules with infinite recursions, although this was limited to prevent overly long tasks. Additionally, the generator already provides a sample solution to the task, allowing learners to independently compare their solution with the sample solution. Automated feedback could be included by extending the static test or unit test interface. Existing packages, such as Pynguin[5], pycheckmate[8] or PyTA[9], could be used to generate static as well as unit tests.

6 Conclusion

This work presents the design and implementation of `pygenaix`, a programming exercise generator supporting parameterization and a grammar-based approach, and provides answers to the formulated research questions. To determine suitable generation methods **(RQ1)**, related works were analysed and evaluated, leading to the selection of parameterization and a grammar-based generation approach. With regards to an extensible and easy to use implementation **(RQ2)**, a structure was developed which allows for the addition of new generation methods as well as new generatable content. Furthermore, a web-based API for requests was developed, which is used to connect the generator to a JupyterLab extension providing a GUI for the generator, enabling the students to access the generator easily. Future work entails improvements to the implemented generation methods before moving on to exploring further methods, like AI for refining storytelling, generating programming exercises or deriving programming exercise blueprints from available exercises.

[8] pycheckmate, `https://doi.org/10.17605/OSF.IO/BR68W`, last accessed 28.08.2023
[9] PyTA, `https://github.com/pyta-uoft/pyta`, Last accessed: 28.08.2023

Bibliography

[Ad19] Ade-Ibijola, A.: Syntactic Generation of Practice Novice Programs in Python. In (Kabanda, S.; Suleman, H.; Gruner, S., eds.): ICT Education. Springer, Cham, pp. 158–172, 2019, ISBN: 978-3-030-05813-5.

[Ba17] Bart, A. C. et al.: BlockPy: An Open Access Data-Science Environment for Introductory Programmers. Computer 50/5, pp. 18–26, 2017, ISSN: 1558-0814.

[BE15] Berglund, A.; Eckerdal, A.: Learning Practice and Theory in Programming Education: Students' Lived Experience. In: Int. Conf. on Learning and Teaching in Computing and Engineering. LaTiCE'15, IEEE, New York, pp. 180–186, 2015.

[Ha18] Habibi, B. et al.: Using Fault Injection for Programming Task Generation. In (Auer, M. E.; Guralnick, D.; Simonics, I., eds.): Teaching and Learning in a Digital World. ICL'17, Springer, Cham, pp. 559–566, 2018, ISBN: 978-3-319-73204-6.

[Ku20] Kurdi, G. et al.: A Systematic Review of Automatic Question Generation for Educational Purposes. en, Artificial Intelligence in Education 30/1, pp. 121–204, 2020, ISSN: 1560-4306, URL: https://doi.org/10.1007/s40593-019-00186-y, visited on: 04/05/2023.

[Of17] Offutt, J. et al.: A Novel Self-Paced Model for Teaching Programming. In: 4th ACM Conf. on Learning @ Scale. L@S '17, ACM, New York, pp. 177–180, 2017, ISBN: 978-1-4503-4450-0, URL: https://doi.org/10.1145/3051457.3053978, visited on: 10/03/2022.

[ROS10] Radošević, D.; Orehovački, T.; Stapić, Z.: Automatic On-Line Generation of Student's Exercises in Teaching Programming. In: Central European Conf. on Information and Intelligent Systems. CECIIS'10, Varaždin, 2010, URL: https://papers.ssrn.com/abstract=2505722, visited on: 09/27/2022.

[Sa22] Sarsa, S. et al.: Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models. In: 18th ACM Conf. on Int. Computing Education Research. ICER '22, ACM, New York, pp. 27–43, 2022, ISBN: 978-1-4503-9194-8, URL: https://doi.org/10.1145/3501385.3543957, visited on: 09/27/2022.

[Se19] Serra, D. et al.: On the Effectiveness of Manual and Automatic Unit Test Generation: Ten Years Later. In: IEEE/ACM 16th Int. Conf. on Mining Software Repositories. MSR'19, IEEE, New York, pp. 121–125, 2019.

[Si12] Sipser, M.: Introduction to the Theory of Computation. Cengage Learning, 2012, ISBN: 978-1-133-18779-0.

[SMB23] Speth, S.; Meißner, N.; Becker, S.: Investigating the Use of AI-Generated Exercises for Beginner and Intermediate Programming Courses: A ChatGPT Case Study. 2023 IEEE 35th International Conference on Software Engineering Education and Training (CSEE&T)/, pp. 142–146, 2023, URL: https://api.semanticscholar.org/CorpusID:261433946.

[So21] Sovietov, P.: Automatic Generation of Programming Exercises. In: 1st Int. Conf. on Technology Enhanced Learning in Higher Education. TELE'21, pp. 111–114, 2021.

[TS18] Tiam-Lee, T. J.; Sumi, K.: Procedural Generation of Programming Exercises with Guides Based on the Student's Emotion. In: IEEE Int. Conf. on Systems, Man, and Cybernetics. SMC'18, IEEE, New York, pp. 1465–1470, 2018.

[WM15] Wakatani, A.; Maeda, T.: Automatic generation of programming exercises for learning programming language. In: IEEE/ACIS 14th Int. Conf. on Computer and Information Science. ICIS'15, IEEE, New York, pp. 461–465, 2015.