Egyptian Informatics Journal 24 (2023) 191–203

Contents lists available at ScienceDirect

Egyptian Informatics Journal

journal homepage: www.sciencedirect.com

Synthesis of nested loop exercises for practice in introductory programming

Chinedu Wilfred Okonkwo, Abejide Ade-Ibijola [1]

Research Group on Data, Artificial Intelligence, and Innovations for Digital Transformation, Johannesburg Business School, University of Johannesburg, South Africa

ARTICLE INFO

Article history:
Received 13 May 2022
Revised 27 February 2023
Accepted 1 March 2023
Available online 10 March 2023

Keyword:
Novice programmer
Nested loop exercises
Practice problems
Introductory programming
Context-free grammars
Procedural content generation

ABSTRACT

Novice programmers struggle to comprehend specific programming constructs such as arrays, recursion, and loops. One way to address this challenge is to provide practice problems for students in these topics that are considered difficult to comprehend — such as *nested loops*. It is proven that practice aids program comprehension, hence, since it is time consuming to manually create many practice problems; synthesising these problems is an *Expert Artificial Intelligence Task* that is worth investigating. In this paper we present the synthesis of nested loop exercises in Python for practice using a context-free grammar. We defined the grammar rules for modeling program templates. Algorithms were designed and implemented for the generation of structurally different nested loop exercises based on the defined production rules of the context-free grammar. Each program generated consists of two or more looping statements, with a nesting, and the context-free grammar rules also allow for the spawning of infinitely many sub-loops for every loop. We checked for the correctness of the synthesised nested loop programs with an experimental procedure of iteratively supplying these programs to the Python interpreter. The first 120,000 iterations returned no syntactic bugs, hence, showing that these programs do compile and have outputs. Furthermore, an evaluation of the programs was carried out in a survey with 210 participants (novice and expert programmers). The results of the evaluation show that over 82% of the participants agreed that the synthesised exercises were correctly generated, solvable and can be used as practice exercises for novice programmers in introductory programming modules. 120,000 iterations of synthesised loop programs can be found at: `https://tinyurl.com/nestedloops2021`.

1. Introduction

Learning how to program is often a challenge for novice programmers [1]. Many students have problems in designing and writing clear programs [2], and they do not like programming subjects because they find them hard to learn [3]. Students in Computer Science and other related disciplines are required to be able to write codes correctly and fluently, as programming skills are becoming a significant competence in many professions [4]. Along with the emerging age of technology, there is need for programming abilities [5]. We have reasons to believe that the implementation of technological tools in learning increases students' comprehension of programming principles and can reduce difficulties in coding.

Various approaches have been proposed to help novice programmers learn how to program, including *pedagogy models* and *software tools* [6–8]. Some of the common pedagogy used in teaching programming include: Comprehension-first model - built on causal inference by displaying, describing and analyzing each program path [9], Design thinking - the process of generation of ideas for problem solving [10], Productive failure approach - allowing the students to solve complex problems on their own first before giving them the instructions or solution on how to solve them [11], Problem-first approach - teaching students how to solve problems first before teaching them how to write program [12], and Spreadsheet approach - teaching programming through the computational thinking embedded in spreadsheet (such as Excel) applications [13].

[1] Abejide Ade-Ibijola, Professor of Artificial Intelligence and Applications at Johannesburg Business School. Email: abejide@jbs.ac.za

Peer review under responsibility of Faculty of Computers and Information, Cairo University.

Production and hosting by Elsevier

https://doi.org/10.1016/j.eij.2023.03.001

Several software interventions have been designed to assist novice programmers, such as: intelligent tutors and Chatbot systems [6,14], syntactic generation of programs, automatic program narration [1], serious games concept, and program visualization [15]. These pedagogy models, tools, application programs and several other efforts have advanced educational setting [16] including this field of novice program comprehension.

While these models and tools are useful for teaching and learning programming, there is continuous need to create more tools for supporting novices, targeting specific programming constructs. Studies have shown that practice improves programming ability [17]. We also know that constructs such as nested loops are often difficult to comprehend for novices [3,18]; hence, the problem addressed in this paper is the synthesis of nested loop exercises. The synthesised exercises can be manually solved with pen and paper as a practical support for novice programmers. To do this, we have designed a context-free grammar (CFG) for the formalisation of specific syntactic patterns for nested loop exercises. We have also designed algorithms that makes use of the CFG rules in rendering valid nested loop exercises. These rules and the resulting exercises are targeted for Python programming language, as this is one of the most used languages in introductory programming around the world today [19].

Fig. 1 describes the process of synthesising nested loop exercises. The process begins by a user specifying the type of nested loop and the number of iterations they want, then CFG rules are used to generate these programs in python. The resulting programs are displayed as exercises to the user. The contributions of this paper are itemised as follows. We have:

1. designed a CFG for the syntactic generation of practice nested loop exercises in Python,
2. implemented the rules of the CFG in (1) above, and produced many iterations of valid nested loop exercises in Python,
3. application of this tool will help the student to understand the construct and flow of nested loop programs, and
4. the results of an evaluation of the synthesised programs to show that they are correct and can support novice program comprehension.

The remainder of this paper is organised as follows. Section 2 presents background information and Section 3 discusses related Works. Section 4 presents grammar design for the synthesis of nested loops and Section 5 explains the algorithms for loop synthesis. Section 6 describes the implementation and results, and Section 7 presents evaluation. Section 8 presents the conclusion and future work.

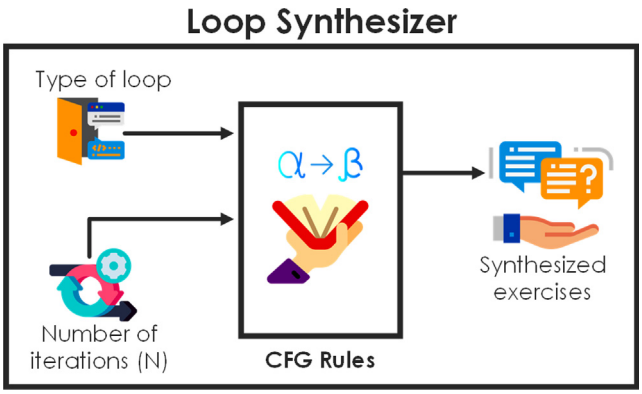

Fig. 1. Synthesis of loop exercise programs using the CFG rules.

2. Background

In this section, we discuss the difficulties faced by novices in learning how to write computer programs. We starting with general difficulties to specific ones around the comprehension of nested loop structures. We further discussed the importance of loops and highlighted the motivation behind this work.

2.1. Learning Programming and Difficulties

2.1.1. Learning Programming and its Importance

Learning programming includes a number of activities, such as learning the language syntax, program design techniques (i.e. paradigms), and program logic (i.e. semantics). Bittencourt et al. [20] stated that learning computer programming has been a longstanding subject of research and various methods have been designed to ease the novice programmers' initial learning process. Programming is not as intuitive as human spoken language – it requires a certain way of thinking and perception, as well as defined principles and techniques for novice programmers to learn [17]. Traditionally, programming teaching methods focused on syntax and semantics rather than problem-solving techniques [21], but novice program comprehension requires a balance attention on programming knowledge and strategies of solving problems [22]. For many years, scholars have been looking for a common teaching process to implement in order to improve novices' comprehension of programming concepts but there is still no method that is general accepted [23]. Understanding how to write basic computer programs is essential for novice programmers, as it helps them to know how to logically and creatively solve problems [17]. Programming is one of the key skills required in this era of the Fourth Industrial Revolution (4IR) to facilitate the new innovation trends [24]. Nonetheless, for most novice programmers, the abstract form of programming courses still remains a challenge [21].

2.1.2. General Difficulties

Main challenging aspects of programming include:

1. basic concepts (such as constructs [25], data structures [26], etc.),
2. program design styles and/or paradigms [26],
3. language syntax [3]
4. debugging, tracing [27] — *the so called "fight with the compiler"* [28], and
5. problem solving skills or algorithms [26].

2.1.3. Specific Difficulties with Loops

Students often struggle with the practical understanding of loops in introductory programming modules [3,18]. They find it difficult to understand the process of program iteration [26]. Some of the difficulties with loops faced by novice programmers include: misconception of loop construct and loop execution, and difficulties in solving problems with the use of a loop constructs [3]. When loops are nested, it becomes more difficult for novice programmers to understand [26].

2.2. Nested Loops

A loop is an iteration statement that allows a code block to be repeated a number of times [29]. Sometimes, you have to put one loop inside another, that is, you have to nest the loops. Nesting a loop simply means having a loop (outer loop) that has another loop(s) (inner loop(s)) inside its commands [30]. In this paper, six structurally different types of nested loops are covered. Fig. 2 shows the structure of these types of loops. In Fig. 2 (a), each arc

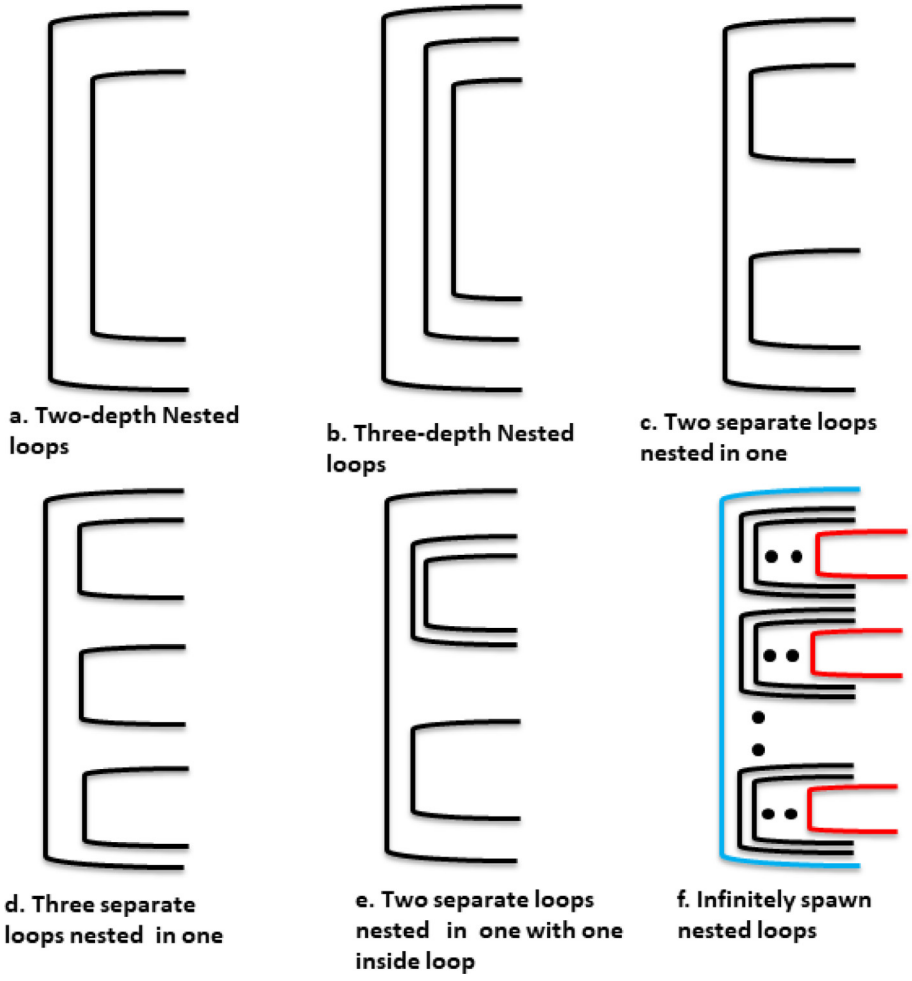

Fig. 2. Graphical representation of the types of nested loop programs/exercises synthesised in this work.

represents a separate loop, hence this is a nested loop depicted with two arcs, the smaller arc enclosed in the bigger one. This is the same for Sub Figs. 2 (b), (c), (d), and (e).

In each of these sub figures, arcs are placed in different structural designs to depict what loops are embedded in another. Sub Fig. 2 (f) describes an infinitely spawned structure of nested loops, where the loops are spawned in two dimensions; i.e. separate sub-loops (as shown in (c) and (d)) and continuous nesting (as shown in (b)). This is achieved by a recursive grammar rule design that is discussed later in Section 4.

2.3. The Importance of Loops

Loops are integral parts of many computer programs; i.e. most programs are likely to contain one or more loops [25]. A program is not generally limited to a linear series of instructions, as it can repeat code during its execution. Becoming a programmer requires that you understand the loop concepts in programming, the use of loops in a program, and the execution processes of loop statements. Having adequate knowledge of loop statements and their applications in the program will therefore help novice programmers to advance their programming skills. Hence, loop comprehension challenges amongst novice programmers is one of the motivations of this work. The next section discusses the motivation of this work in details.

2.4. Motivation

The following are the challenges that motivated this study.

Programming can be Challenging
Novice programmers find it difficult to understand Computer programming syntax, having to learn various sets of skills simultaneously [1,2]. One major challenge in studying programming is the need to concurrently acquire a number of different sets of skills [31].
Loop Comprehension Problems
Novice programmers find it difficult to understand the loop construct, the loop execution, and the use of loops in the program [3,25].
High Failures Rates
Studies have shown that the introductory programming courses have a high failure and drop-out rates [21] due to lack of adequate programming learning aids that can support the students [6].
Need for Practice
Hook and Eckerdal [32] concluded that, contrary to those who spent more time reading textbooks and attending lectures, students who spent more time doing problem exercises scored higher marks in the programming course examination. Practic-

ing tools aid programming [6,33], and continuous practice of program exercises aids program comprehension [34].

Teaching Method

The teaching style of programming courses affect students' comprehension ability, and in most cases, much attention is given to syntax and semantics whereas problem-solving strategies received less attention [31].

This work is motivated by these challenges and needs in learning programming. We have devised a formal technique (based on a CFG) for synthesising exercises of nested loop problems that novice programmers can use as practical problems. In the next section, we present research works that are related to this study.

2.4.1. Why Use Context-free Grammars?

Context-free grammars (or CFGs) are a class of formal languages that can be used to, formally, specify Context-free Languages (CFLs). The formal definition of CFGs is presented in Section 4.1. We have used CFGs as a medium of expressing the rules for formulating new problems in nested loops, for the following reasons:

Recurring Nature of Sub-loops

As shown in Fig. 2, loops can have an infinite number of sub-loops, eventhough we have considered going to the maximum depth of three sub-loops — i.e. we have not allowed nested loops to be generated to the depth that is $\geqslant 4$. CFGs are perfect for formalising these rules because of their power to express recurring patterns. For example, a simple rule to represent infinite number of sub-loops for the syntactic structure in Fig. 2 (a) and (b) is:

$$\langle loop\rangle \longrightarrow \langle start\_loop\rangle\langle loop\rangle\langle end\_loop\rangle| \quad (1)$$

$$\longrightarrow \langle loop\rangle \quad (2)$$

Where the variables $\langle start\_loop\rangle$ and $\langle end\_loop\rangle$ are symbols denoting the nesting of these loops. Rule 1 allows for an infinite replacement of the loop variable, that can be terminated with Rule 2 in the re-writing process of the derivation. Here, the depth of the loop can be generalised as a variable that specifies the number of re-writings in the derivation.

Sharing Grammar Rules

An important part of science communication is giving a "*shoulder for the community to stand on*". Using CFGs will allow other scientists that intend to synthesise other types of program fragments to re-use some sub-structures of the syntactic structures embedded in the CFG rules.

Applications of FLAT

Formal Languages and Automata Theory (FLAT) is a well known aspect of Theoretical Computer Science. This aspect has also been known to have many mathematical or computational theories with little or no real life applications. However, adopting an aspect of FLAT (CFGs) in this work shows that FLAT can find new applications in topics like procedural content generation, and computer science education.

CFGs and Programming Languages

The syntactic definitions of programming language recognisers and parsers are done with CFGs. This is preferred over Regular Languages (or RLs) because it handles more complex structures such as the language of balanced parenthesis. The use of CFGs in parsing programming languages is not new, and like any other formal language, they can be used for both recognition and generation tasks [35]. The application of CFGs for recognition can be found in compiler constructions, and its applications in generation can be found in Ade-Ibijola [1].

3. Related Works

Although there is no research on synthesis of nested loop practice programs, or exercises; some related works have been carried out using similar techniques in the following areas:

Artefacts — Gulwani [36] synthesised a range of objects with interface models and algorithmic methods for repetitive drawings and mathematical concepts (such as algebraic identities, bit vector algorithms, compass based geometry constructions, etc.). Ade-Ibijola [37] proposed an algorithm for the automatic generation of hypothetical social network graphs that were graphically rendered using the Microsoft Automatic Graph Layout API. This algorithm produced random hypothetical names as vertices, and random directed edges connections between them. Sunbeom and Hakjoo [38] developed a tool for the generation of different patterns of characters through synthesis of pattern program. The algorithm synthesised the desired program through the integration of enumerative search, constraint response, and program exploration. Another work by Ade-Ibijola [39] presented the synthesis of social media profiles using a probabilistic CFG (or PCFG) — using Facebook as a case study, the author used PCFG rules to formulate reasonable Facebook personas. A tool for the synthesis of advertisement on social media was also presented by Kabaso and Ade-Ibijola [40].

Exercises — In an attempt to synthesise programming exercises, Edwards *et al.* created a tool called `Phanon` which teaches programming basics through non-class online exercises [41]. A data-driven tutoring platform was introduced in the provision of personal assistance to novice programmers. This method used state abstraction and path building to automatically synthesise customised hints for students, even when the provided states have not previously occurred in the data [42]. Ade-Ibijola [1] generated infinitely many unique practice programs in Python using CFG rules to model the program templates. Gulwani *et al.* [43] proposed a light-weight system that generates feedback on performance issues in the introductory programming tasks. The method uses a dynamic analytic-based approach to check whether a student's program fits the requirements of the instructor. Kim *et al.* [44] developed a system that automatically explains mistakes in programming task called `Apex`. The system will automatically synthesise the reasons for programming assignment errors, including where the errors are and the causes of the errors.

3.1. Comparisons of Related Work

Table 1 compares some of previous research works that are related to our work. Although some of the previous works had some form of synthesis algorithms, and some with CFGs but none has attempted to address the loop synthesis problems, or to formalise this process.

3.2. The Gap

From the literature, no attempt has been made to formalise the process of synthesising nested loop problems using CFGs; nor has there been any efforts toward the generation of nested loops practice exercises in Python. This is the gap addressed in this work.

Table 1
Comparison of Related Work.

S/N	Title	Author(s)	Comparison
1	Synthesis from examples: Interaction models and Algorithms.	Sumit Gulwani (2012)	It uses algorithmic methodologies (Bitvector, Algebraic identities, etc.) to synthesised examples-based artefacts and does not use CFG.
2	Synthesis of hypothetical sociograms for social network analysis.	Abejide Ade-Ibijola (2018)	The tool uses an algorithm to generate hypothetical social network graphs. The study applied a synthesis technique but does not use formal grammar rules.
3	Synthesizing pattern programs from examples.	Sunbeom So and Hakjoo Oh (2018)	The tool uses domain-specific language (DSL) and algorithm to synthesised pattern programs. Synthesis technique was implemented but does not use grammar rules to design the problems.
4	Synthesis of social media profiles using a probabilistic context-free grammar.	Abejide Ade-Ibijola (2017)	The probabilistic/stochastic CFG was used in the formulation of the grammar rules for the automatic generation of social media profiles. The tool synthesised personal attributes and not nested loop instance programs.
5	Synthesis of loop-free programs.	Gulwani et al. (2011)	It uses a component-based approach to generate the synthesis constraint and then uses an algorithm to solve the constraint. The CFG was not used.
6	Data-driven hint generation in vast solution spaces: a self-improving python programming tutor.	Kelly Rivers and Kenneth R Koedinger (2017)	The tool uses state abstraction, state reification, and path building to automatically produce suggestions/hints for student programming problems. It does not use CFG.
7	Syntactic Generation of Practice Novice Programs in Python.	Abejide Ade-Ibijola (2018)	The tool uses a synthesis technique to generate many unique procedural programs using CFG. Though a similar approach was used but procedural programs were synthesised and not nested loops exercises.
8	Feedback generation for performance problems in introductory programming assignments.	Gulwani et al. (2014)	It uses an algorithmic strategy to automatically generate feedback on students' performance on a programming assignment. First, a matching specification was defined, and an algorithm is used to check student implementation to generate automatic feedback. It does not use CFG.
9	Apex: automatic programming assignment error explanation.	Kim et al. (2016)	An instance program matching technique was used to compare student assignment with the correct implementation of the instructor. The tool automatically generates errors with explanations if detected. It does not use CFG.
10	Sell-Bot: An Intelligent Tool for advertisement Synthesis on Social Media	Kabaso and Ade-Ibijola (2020)	It uses an algorithmic approach to generate natural or human languages based on CFG. It deals on human language generation and not instance program

4. Grammar Design for Nested Loop Synthesis

In this section, we present the design of a CFG for the synthesis of nested loop exercises in Python. The reason for using a CFG (a formalism for describing context-free languages) were stated in Section 2.4.1.

4.1. Definition of Terms

Some of the terms used in this section include: *symbol*, *alphabet*, *string*, and *context-free grammar* (or CFG). We define these terms as follows. A *symbol* is an item or single token; an *alphabet* is a finite set of symbols; *strings* consist of a concatenation of zero or more symbols (when it is zero, its called an empty string, represented as λ), and *grammar* ($\mathscr{G}$) is a formal way of representing rules describing the syntax of a language [45].

$\mathscr{G}$ is a four-tuple, given as $\mathscr{G}= (V, \Sigma, R, S)$. Here V is a finite set of nonterminal variables, Σ is a finite set of terminal symbols (disjoint from V, i.e. $\Sigma \cap V = \varnothing$), R is a set of production rules, where each production rule maps a nonterminal variable to a string, i.e. $R \in (V X \Sigma)$, and S is the start symbol ($S \in V$) — *the origin of all derivations.*

4.2. Building Blocks

We have adapted the production rules for simplistic symbols (such as letters, identifier names, arithmetic ($\langle Arth\_Expr \rangle$), and logical ($\langle Logical\_Expr \rangle$) expressions) from the work of Ade-Ibijola [1]. These symbols appear in the rules of the new CFG presented in subsequent sub-sections.

4.3. Rules for Nested Loop Synthesis

Here we present rules for the generation of nested loop exercises. These rules cover three types of loops in Python, namely: `For`, `While`, and `For-Each` loops.

4.3.1. Statements and Indentation

Here we present rules for the generation of varying types of statements and enforcing a structural indentation between these statements — *ensuring that loops 'know' which statements they hold.*

$$\langle NL \rangle \longrightarrow newline \tag{3}$$
$$\langle Var\_Init \rangle \longrightarrow GetSetOfInitialisedVariables() \tag{4}$$
$$\langle Stmt\_Block \rangle \longrightarrow \langle Var\_Init \rangle \langle Arth\_Expr \rangle | \langle Logical\_Expr \rangle \tag{5}$$
$$\langle Opt\_Stmt\_Block \rangle \longrightarrow \langle Stmt\_Block \rangle | \lambda \tag{6}$$
$$\langle Loop\_Type \rangle \longrightarrow For | For\_Each | While \tag{7}$$
$$\langle Loop\_Instance \rangle \longrightarrow GetLoopStatement(\langle Loop\_Type \rangle, \langle Var\_Init \rangle) \tag{8}$$
$$\langle Struc\_Str \rangle \longrightarrow \langle NL \rangle \langle Tab \rangle \tag{9}$$

Rule 3 to Rule 9 defines structures such as newline characters, variable initialisation (this is done with a function presented in Algorithm 1). Rules 5 and 6 describe statement blocks. In Rule 5, variables are initialised first, and these variables are used in arithmetic expressions or as an alternative, logical expressions are generated with Boolean variable initialised in Rule 4. Rule 6 allows for an optional statement block to be generated. This is later used in building nested loop programs that may or may not have statement blocks in specific parts of the source code. Rule 7 enumerates the types of loops covered, and this is passed to Rule 8 for the synthesis of an actual loop instance using the function `GetLoopStatement()`, defined in Algorithm 2. To maintain the structure of a nested loop, the newline and tab characters are combined in Rule 9 to manage indentation of code.

4.3.2. Two-depth Nested Loop

In this section, we leverage on the rules defined in Section 4.3.1 to define the structure of a simple nested loop with a depth of two — *see Fig. 2(a)* for the graphical description of this loop structure.

$$\begin{aligned} \langle 2\_depth \rangle \longrightarrow & \langle Loop\_Instance \rangle \\ & : \langle Stru\_Str \rangle (\langle Opt\_Stmt\_Block \rangle (\langle Loop\_Instance \rangle \\ & : \langle Struc\_Str \rangle \langle Stmt\_Block \rangle) \langle Opt\_Stmt\_Block \rangle) \end{aligned} \tag{10}$$

Rule 10 defines this type of nested loops by weaving a loop instance with an optional statement block, and allowing another loop instance to appear inside it, using parenthesis to manage the nesting.

4.3.3. Three-depth Nested Loop

To define a simple nested loop structure with a depth of three, we use the rules described in Section 4.3.1 — *see* Fig. 2(b) for the graphical description of this loop structure.

$$\begin{aligned}\langle 3\_depth\rangle \quad &\longrightarrow \langle Loop\_Instance\rangle : \langle Stru\_Str\rangle \\ &(\langle Opt\_Stmt\_Block\rangle \\ &\langle Loop\_Instance\rangle : \langle Struc\_Str\rangle \\ &(\langle Opt\_Stmt\_Block\rangle \\ &\langle Loop\_Instance\rangle : \langle Struc\_Str\rangle(\langle Stmt\_Block\rangle) \\ &\langle Opt\_Stmt\_Block\rangle) \\ &\langle Opt\_Stmt\_Block\rangle)\end{aligned} \tag{11}$$

The syntax of nested loops of the depth of 3 is described in Rule 11. Here, three loops are nested in a concentric formation. Each loop is defined with the ⟨*Loop_Instance*⟩ and the ⟨*Opt_Stmt_Block*⟩ allows for lines of code to appear between each loop when they start, and before the end of each loop. The innermost loop has a compulsory ⟨*Stmt_Block*⟩, as this cannot be empty — the innermost loop needs statements to execute.

4.3.4. Two Loops Nested in One

Here we describe the syntax of two loops nested in one loop in Rule 12. To separate the two sub-loops, we have also enclosed an optional statement loop between them, and at the start and end of both sub-loops.

$$\begin{aligned}\langle 2\_in\_one\rangle \quad &\longrightarrow \langle Loop\_Instance\rangle : \langle Struc\_Str\rangle \\ &(\langle Opt\_Stmt\_Block\rangle \\ &(\langle Loop\_Instance\rangle : \langle Struc\_Str\rangle \\ &\langle Stmt\_Block\rangle) \\ &\langle Opt\_Stmt\_Block\rangle \\ &(\langle Loop\_Instance\rangle : \langle Struc\_Str\rangle \\ &\langle Stmt\_Block\rangle) \\ &\langle Opt\_Stmt\_Block\rangle)\end{aligned} \tag{12}$$

4.3.5. Three Loops Nested in One

Rule 13 is similar to Rule 12, with the difference of now having three loops as opposed to two. Each of these three loops nested in one, have a compulsory statement loop in them for reasons discussed earlier. A graphical description of this syntactic definition is presented in Fig. 2(d).

$$\begin{aligned}\langle 3\_in\_one\rangle \quad &\longrightarrow \langle Loop\_Instance\rangle : \langle Struc\_Str\rangle \\ &(\langle Opt\_Stmt\_Block\rangle \\ &(\langle Loop\_Instance\rangle : \langle Struc\_Str\rangle \\ &\langle Stmt\_Block\rangle) \\ &\langle Opt\_Stmt\_Block\rangle \\ &(\langle Loop\_Instance\rangle : \langle Struc\_Str\rangle \\ &\langle Stmt\_Block\rangle) \\ &(\langle Loop\_Instance\rangle : \langle Struc\_Str\rangle \\ &\langle Stmt\_Block\rangle) \\ &\langle Opt\_Stmt\_Block\rangle)\end{aligned} \tag{13}$$

4.3.6. Two Loops Nested in One With One Inner Loop

In Rule 14, we have allowed one of two sub-loop structures (the first sub-loop) to have an inner loop. The graphical illustration of this is shown in Fig. 2(e). The innermost sub-loop gets a compulsory statement block. Other syntactic definitions for substructures are as described in the previous production rules.

$$\begin{aligned}\langle 2\_in\_one\_one\rangle \quad &\longrightarrow \langle Loop\_Instance\rangle : \langle Struc\_Str\rangle \\ &(\langle Opt\_Stmt\_Block\rangle \\ &((\langle Loop\_Instance\rangle : \langle Struc\_Str\rangle \\ &\langle Opt\_Stmt\_Block\rangle) \\ &(\langle Loop\_Instance\rangle : \langle Struc\_Str\rangle \\ &\langle Stmt\_Block\rangle) \\ &\langle Opt\_Stmt\_Block\rangle) \\ &\langle Opt\_Stmt\_Block\rangle \\ &(\langle Loop\_Instance\rangle : \langle Struc\_Str\rangle \\ &\langle Stmt\_Block\rangle) \\ &\langle Opt\_Stmt\_Block\rangle)\end{aligned} \tag{14}$$

At this stage, a question that comes to mind is: what if we want more than two sub-loops? Or, what if we want to define these rules such that it allows infinitely many sub-loops to be derivable from the start symbol? Hence, it becomes compelling to define the sub-loops recursively. Rules for this is presented in Section 4.4.

4.4. Infinitely Spawned Nested Loops and Sub-Loops

Here, we define rules for spawning sub-loops infinitely (graphically illustrated in Fig. 2(f)). We begin by describing the innermost loop — as this is the only loop that needs a compulsory statement block — in Rule 15. We then present a recursive definition of sub-loops in Rule 16 where the symbol ⟨*Recursive_SubLps*⟩ has a reference to itself. Still in Rule 16, we make sure the rewriting of this nested loop derivation can be stopped with the option of generating the inner most loop, ⟨*Innermost_Lp*⟩.

$$\begin{aligned}\langle Innermost\_Lp\rangle \quad &\longrightarrow (\langle Loop\_Instance\rangle : \langle Struc\_Str\rangle \\ &\langle Stmt\_Block\rangle)\end{aligned} \tag{15}$$

$$\begin{aligned}\langle Recursive\_SubLps\rangle \quad &\longrightarrow \langle Loop\_Instance\rangle : \langle Struc\_Str\rangle \\ &(\\ &\langle Opt\_Stmt\_Block\rangle\langle Recursive\_SubLps\rangle \\ &\langle Opt\_Stmt\_Block\rangle \\ &)| \\ &(\langle Opt\_Stmt\_Block\rangle \\ &\langle Recursive\_SubLps\rangle \\ &\langle Opt\_Stmt\_Block\rangle) \\ &(\langle Recursive\_SubLps\rangle \\ &) \\ &|\langle Innermost\_Lp\rangle\end{aligned} \tag{16}$$

In conclusion, we present the formulation of six types of nested loops as discussed in Fig. 2. To complete the rules of $\mathscr{G}$, we define the start symbol ($S \in V$, where $S \equiv \langle Nested\_loop\rangle$) in the following productions.

$$\langle Nested\_loop\rangle \longrightarrow \langle 2\_depth\rangle| \tag{17}$$
$$\longrightarrow \langle 3\_depth\rangle| \tag{18}$$
$$\longrightarrow \langle 2\_in\_one\rangle| \tag{19}$$
$$\longrightarrow \langle 3\_in\_one\rangle| \tag{20}$$
$$\longrightarrow \langle 2\_in\_one\_one\rangle| \tag{21}$$
$$\longrightarrow \langle Recursive\_SubLps\rangle \tag{22}$$

Rules 17–20 allow for the derivation of all six classes of nested loops from the start symbol, ⟨*Nested_loop*⟩. Now, we proceed to an example of how a nested loop can be derived from ⟨*Nested_loop*⟩ in Example 1.

Example 1 (*Derivation of a Nested Loop*). The following is an example of how a nested loop can be derived from the start symbol, ⟨*Nested_loop*.

$$\begin{aligned}\langle Nested\_loop\rangle &\Rightarrow \langle Recursive\_SubLps\rangle[Rule22]\\ &\Rightarrow \langle Loop\_Instance\rangle : \langle Struc\_Str\rangle\\ &\quad \langle Opt\_Stmt\_Block\rangle\langle Recursive\_SubLps\rangle\\ &\quad \langle Opt\_Stmt\_Block\rangle\\ &\quad [Rule16] \qquad (23)\\ &\Rightarrow \texttt{foryinrange(3,50,12):}\\ &\quad \langle Opt\_Stmt\_Block\rangle\langle Recursive\_SubLps\rangle\\ &\quad \langle Opt\_Stmt\_Block\rangle \qquad (24)\\ &\quad [Rule8]\\ &\Rightarrow \texttt{foryinrange(3,50,12):}\\ &\quad \texttt{c = y * 2}\\ &\quad \langle Recursive\_SubLps\rangle \qquad (25)\\ &\quad \langle Opt\_Stmt\_Block\rangle\\ &\quad [Rule5\&6]\\ &\Rightarrow \texttt{foryinrange(3,50,12):}\\ &\quad \texttt{c = y * 2}\\ &\quad \langle Innermost\_Lp\rangle \qquad (26)\\ &\quad \langle Opt\_Stmt\_Block\rangle\\ &\quad [Rule15]\end{aligned}$$

$$\begin{aligned}&\Rightarrow \texttt{foryinrange(3,50,12):}\\ &\quad \texttt{c = y * 2}\\ &\quad \langle Loop\_Instance\rangle : \langle Struc\_Str\rangle\langle Stmt\_Block \qquad (27)\\ &\quad \langle Opt\_Stmt\_Block\rangle\\ &\quad [Rule15]\\ &\Rightarrow \texttt{foryinrange(3,50,12):}\\ &\quad \texttt{c = y * 2}\\ &\quad \texttt{while(c <= 100):}\\ &\quad \texttt{k = c + 10} \qquad (28)\\ &\quad \langle Opt\_Stmt\_Block\rangle\\ &\quad [Rule8]\\ &\Rightarrow \texttt{foryinrange(3,50,12):}\\ &\quad \texttt{c = y * 2}\\ &\quad \texttt{while(c <= 100):} \qquad (29)\\ &\quad \texttt{k = c + 10}\\ &\quad \texttt{print(k)}\end{aligned}$$

In this example, a nested loop python program was derived from the start symbol of the grammar, $\mathscr{G}$; i.e. $S \Rightarrow^{*} 29$. This derivation process was automated with the aid of the algorithms described in Section 5, and the implementation details are described in Section 6.

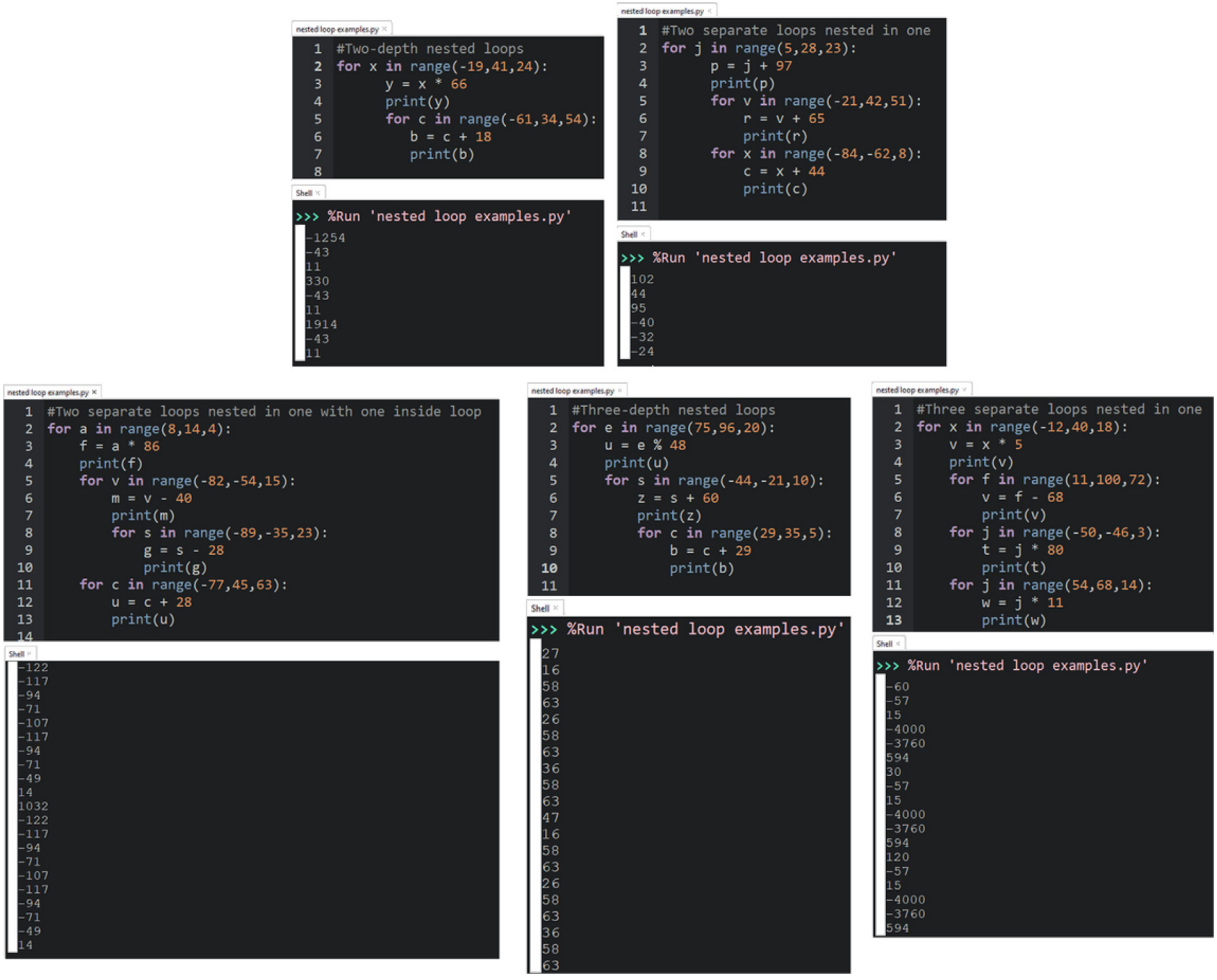

Fig. 3. Solutions to the examples of the synthesis programs (Listing 1–5).

5. Algorithms for Loop Synthesis

In this section, we introduce the algorithms for the automatic generation of program variables and nested loop instances from the productions defined in Sections 4.3 and 4.4.

Algorithm 1: GetSetOfInitialisedVariables

```
1  Function GetSetOfInitialisedVariables():
       Data: N (number of variables)
       Data: lwrbnd (the least initial value)
       Data: uprbnd (the most initial value)
       Data: spread (the minimum distance between least and most initial values )
       Result: L (list of initialised variables)
2      L ← empty()
3      i ← 0
4      check ← lwrbnd + spread
5      while ((i <= N) & (check < uprbnd)) do
6          var ← α ∈[A-Za-z] ∋: (α ∉ [l, o] ∧ α ∉ L);
7          value ← rnd(lwrbnd, uprbnd)
8          stmt ← Concat(var, "=", value)
9          L.Add(stmt);
10         i++;
11     end
12     return L
```

5.1. Generation and Initialisation of Variables

This section describes an algorithm (Algorithm 1) for automatic generation and initialisation of program variables. The algorithm takes (as parameters): the number of variables that needs to be generated and initialised, two parameters that denote the least and the most initial values for each randomly generated variable, and a spread that ensures a minimum numeric distance between the least and the most initial values — this ensures that the range of selection of a random value is wide enough to have access to a variety of random values. The variable *L* on Line 3 is an empty list that stores the generated variables and their respective values. In Line 5, a loop starts the search for valid variables and values, with the conditions that the number of iterations is not yet reached, and the *lwrbd*, *uprbnd*, and *spread* all satisfy the conditions for spread. In Line 7, a random alphabet is selected with the exception of those that look like digits — i.e. *l*, and *o*. Line 8 finds a value for this variable, and Line 9 constructs the assignment statement, i.e. `variable = value`. This initialisation is added to the list *L*, and this process continues until we have *N* initialisation.

5.2. Generation of Loop Type

In this section, we present an algorithm (Algorithm 1) for the generation of loop types defined earlier in Rule 7. Algorithm 2 describes the `GetLoopStatement()` function used to perform the operation defined in Production 8. This algorithm takes five parameters described on Line 1 and returns a loop statement. One Line 2, a syntactic end of line character in python is initialised, and the loop string is set to null on Line 3. The switch statement that begins on Line 4 allows for a selection of loop type. Loops for_1,for_2,for_3 (defined on Lines 5, 7, and 10 respectively) allows for the generation of three variants of the for-loop — *using the* `BRAC` *function to introduce the needed brackets/parenthesis to the generated strings*. Another function in Algorithm 1 is `getValidLogicalStmt` which generates a valid logical statement. Line 15 and 17 allows two additional loop types, i.e. the foreach and the while loops. In all cases, loop strings are built such that they produce valid loops as per the earlier presented production rules of $\mathscr{G}$.

Algorithm 2: GetLoopsStatement

```
1  Function GetLoopsStatement():
       Data: Var_Init (List of 2-tuple, i.e. variable and value, variable initialisations)
       Data: LoopType ∈ {for_1, for_2, for_3, foreach, while}
       Data: Var_Collection (2-tuple list variables and corresponding list of items)
       Data: spread (the minimum distance between least start and end of loop counters)
       Data: mesh (the number of desired iterations per loop)
       Result: loop (a valid string of loop instance)
2      loop ⟵ null
3      switch LoopType do
4          case for_1 do
5              loop ⟵ concat_with_Infix_Spc("for", Var_Init[var], "in range",
                BRAC(Var_Init[value]))
6          case for_2 do
7              UpLmt ⟵ n ∋: UpLmt ∈ [Var_Init[value], Var_Init[value] + spread]
8              loop ⟵ concat_with_Infix_Spc("for", Var_Init[var], "in range",
                BRAC(Var_Init[value], UpLmt))
9          case for_3 do
10             UpLmt ⟵ n ∋: UpLmt ∈[Var_Init[value], Var_Init[value] + spread]
11             Step ⟵ Ceil(UpLmt - Var_Init[value]) / Mesh
12             loop ⟵ concat_with_Infix_Spc("for", Var_Init[var], "in range",
                BRAC(Var_Init[value], UpLmt, Step)
13         case foreach do
14             loop ⟵ concat_with_Infix_Spc("for", Var_Collection[var], "in",
                Var_Collection[List])
15         case while do
16             loop
                ⟵ concat_with_Infix_Spc("while",BRAC(getValidLogicalStmt(Var_Init))
17         end
18         return loop;
19
```

Algorithm 3: Synthesis of Nested Loop Exercises

```
1  Function DeriveLoopClass():
       Data: N_loop
       Result: List of loops of size N
2      Return_list ⟶ empty : i = 0;
3      while i <N do
4          Goto start symbol S ∈ G
5          for nonterminal N ∈ RHS(S) do
6              Nested_loop ⟶ replace N with RHS of production
7              if RHS(N) contains any nonterminal then
8                  replace nonterminal with production
9              end
10             Return_list.Add (Nested_loop)
11         end
12     end
13     return Return_list
```

Algorithm 3 traces the steps of derivation of the grammar, $\mathscr{G}$. This algorithm takes the number of loops to be derived as a parameter and returns a list of these loops. At each step, it attempts to replace all nonterminals in the grammar rules with terminal productions or symbols. Line 10 adds the final string derived from $\mathscr{G}$ to a list of nested loop strings. This list is returned on Line 13.

6. Implementation and Results

In Section 5 we introduced algorithms for the synthesis of nested loop instances, based on the rules presented in Section 4.

6.1. Implementation Details

Here we present the implementation details of these algorithms and showcase the results from these implementation. We implemented all the rules and algorithms with Python. This program was tested severally, and 120,000 iterations of valid nested loop programs that were synthesised from this program can be viewed at: https://tinyurl.com/nestedloops2021.

6.2. Comment on Uniqueness and Correctness

.

Uniqueness We demonstrated the generation of 120,000 iterations of different types of nested loop exercises consisting of six categories of nested loops including two-depth nested loops, three-depth nested loops, two separate loops nested in one, three separate loops nested in one, two separate loops nested in one with one inside loop, and infinitely spawned nested loops. All the instances of the generated programs were unique during execution, and there was no repeated program. According to Ade-Ibijola [1], one could provide a theoretical proof of uniqueness based on the branches of the parse tree of the rules, as well as infer uniqueness (or speculate) from the almost impossible chances that the same program can be generated in over a million iterations of program generation.

Correctness To validate the correctness of the generated programs, we tested these programs with the Python 3.9.0 interpreter, we checked for syntax errors in the exercises and it was found that the synthesised program instances were syntax error-free. Fig. 3 shows examples of synthesised programs that were tested using the Python interpreter.

Furthermore, this tool was practically evaluated to determine its usefulness and the obtained results were discussed in the next section.

7. Evaluation and Application of the loop Synthesiser

7.1. Evaluation

This section presents the results from a survey-based evaluation of the students' perception of the generated nested loop exercises and their possible usefulness. The survey was carried out at the University of Johannesburg South Africa, and the target audience includes students taking the introductory course in Python programming as well as those who have completed the course previously. This demographic was selected because they are programming at an early stage. The survey can be found in this link: `https://tinyurl.com/nested-loops-exercises`.

The aim of the assessment was to determine students':

1. perception of computer programming,
2. perception of generated nested loop exercises,
3. if the generated programs are correct and solvable, and
4. Whether the practice of the synthesised exercises helps them to improve their programming skills.

7.2. Participants' Profile

The survey included 210 university undergraduates, both first-year students (87.7%) and returning or old students (13.3%). The sample population consists of male and female students of Computer Science and other relevant courses who are taking the introductory programming course for the first time and those who have previously taken it. This audience is appropriate for the study since the goal of the research is to aid novices in overcoming challenges in program comprehension.

Students were asked about their perception of:

Computer programming
This tested the general perceptions of students on Computer programming courses to assess their feelings about learning the courses. 71% believed that the courses are difficult, 15.2% find the courses to be easy and 13.8% were indifferent. This suggests that the students considered that it was difficult to learn computer programming courses. See Fig. 4a.
Python programming
Here the students were asked to decide if they are doing the Python programming course at the moment or if they did it before. 86.7% agreed that they are currently doing the course or did the course before, while 13.3% did not do the Python programming courses before. This implies that a large number of students involved in the survey have knowledge of Python programming. See Fig. 4b.
Use of looping statements in programming
It tested student perceptions of the use of looping statements in programming. 76.2% believed that it is difficult to use looping statements in programming, 11.9% found it easy to use and 11.9% were indifferent. This suggests that the majority of novice programmers found it difficult to use loops in programming. See Fig. 4c.
Expertise in programming
The level of the students' knowledge of programming was tested. 70% are taking programming course for the first time, 14.3% did it 2–5 years ago but stopped coding, 11.9% did it 2–5 years ago but are still coding, and 3.8% considered themselves to be expert in programming. See Fig. 4d.
Practice aids learning
Here the students were asked to determine if practice aids the learning of programming. 82.4% agreed, 6.6% disagreed, and 11% were indifferent. This means that the practice of exercise helps in program comprehension. See Fig. 4e.
Correct synthesised exercises
This tested the generated exercises to see if they are correctly formulated and can be solved using pen and paper. 87.6% agreed that the generated exercises are correct, 3.8% assumed that the questions are incorrect, and 8.6% cannot say if the exercises are correct or not. This implies that the developed system is capable of generating correct nested loop exercises. See Fig. 4f.
Synthesised nested loop exercises
The effect of using the synthesised exercises for practice in Python programming was checked. 85.7% agreed that practicing nested loop exercises using these synthesised problems can help in comprehension of Loops, 4.8% disagreed, and 9.5% were indifferent. This suggests that the practice of nested loop exercise has a positive impact on improving students' learning skills in programming. See Fig.4g.

The findings show that a large proportion of the students involved in this survey are at the early stage of learning programming including those who are currently doing the course and those who did it before. Computer Programming courses have been shown to be difficult for the majority of students to learn, particularly the use of loops, suggesting that students need the support of learning aids. This result compare well with the reports of [3,26] which indicated that students struggle to understand nested loops at early stage of programming. The synthesised nested loop exercises were correctly formulated and are solvable with pen and

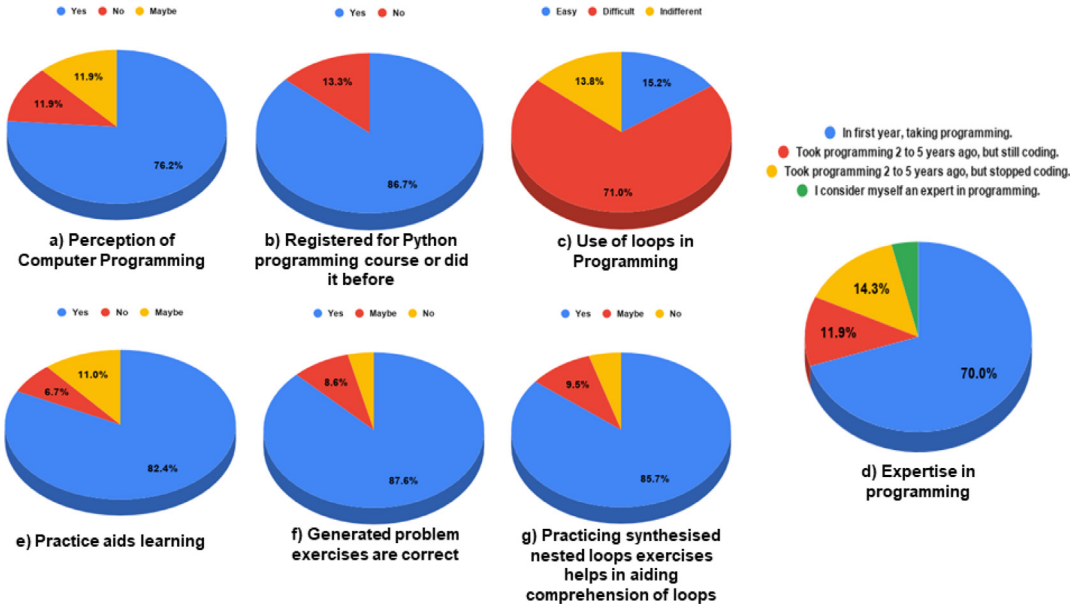

Fig. 4. Evaluation results.

paper, indicating that the tool was well-designed to achieve the purpose of the study. By practicing the generated exercises, novice programmers can learn more about the use of loop statements in programs, making it easier for them to improve their programming skills in Python. This finding agrees with the suggestion of [17] that practice improves programming ability.

7.3. Application

The synthesised nested loops exercises can be used as:

1. practice exercises for students,
2. test problems for students, and
3. examination problems in introductory Python programming course for undergraduate.

8. Conclusion and Future Work

8.1. Conclusion

This work presented the synthesis of nested loop exercises for practice in introductory programming for novice programmers. We defined the context-free grammar (CFG) rules for modeling program templates and applied these rules to synthesise different loop programs. New algorithms (based on the CFG rules) were developed and used to generate the exercises. These exercises can be administered to novice programmers to be solved with pen and paper. We demonstrated the functionality of the tool by synthesising 120,000 of unique nested loop problems. The tool was evaluated through a survey and from the students' perceptions, the generated exercises are correct and can help the students to improve their programming skills.

Future Work

In this work, we concentrated on loop statements in Python; future work will attempt to explore other loop statements in other programming languages and other programming constructs.

Funding

This work is supported by the National Research Foundation of South Africa (Grant Number: 119041).

CRediT authorship contribution statement

Chinedu Wilfred Okonkwo: Methodology, Data curation, Project administration, Writing - original draft. **Abejide Ade-Ibijola:** Conceptualization, Methodology, Software, Supervision, Formal analysis, Investigation, Project administration, Validation, Writing - original draft, Writing - review & editing.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] A. Ade-Ibijola, Syntactic generation of practice novice programs in python, Communications in Computer and Information Science (CCIS) (2018) 158–172.
[2] A. Altadmri, N.C. Brown, 37 million compilations: Investigating novice programming mistakes in large-scale student data, in: Proceedings of the 46th ACM Technical Symposium on Computer Science Education, ACM, 2015, pp. 522–527.
[3] Qian Y, Lehman J. Students' misconceptions and other difficulties in introductory programming: A literature review. ACM Transactions on Computing Education (TOCE) 2017;18:1–24.
[4] M. Rahmat, S. Shahrani, R. Latih, N.F.M. Yatim, N.F.A. Zainal, R. Ab Rahman, Major problems in basic programming that influence student performance, Procedia-Social and Behavioral Sciences 59 (2012) 287–296.
[5] T.-C. Huang, Y. Shu, S.-H. Chang, Y.-Z. Huang, S.-L. Lee, Y.-M. Huang, C.-H. Liu, Developing a self-regulated oriented online programming teaching and learning system, in: 2014 IEEE International Conference on Teaching, Assessment and Learning for Engineering (TALE), IEEE, 2014, pp. 115–120.
[6] F. Clarizia, F. Colace, M. Lombardi, F. Pascale, D. Santaniello, Chatbot: An education support system for student, in: International Symposium on Cyberspace Safety and Security, Springer, 2018, pp. 291–302.
[7] X. Chen, H. Xie, G.-J. Hwang, A multi-perspective study on artificial intelligence in education: Grants, conferences, journals, software tools, institutions, and researchers, Computers and Education: Artificial Intelligence (2020) 100005.
[8] Okonkwo CW, Ade-Ibijola A. Chatbots applications in education: A systematic review. Computers and Education: Artificial Intelligence 2021;2: 100033.
[9] G.L. Nelson, B. Xie, A.J. Ko, Comprehension first: evaluating a novel pedagogy and tutoring system for program tracing in CS1, in: Proceedings of the 2017 ACM Conference on International Computing Education Research, 2017, pp. 2–11.
[10] Wrigley C, Straker K. Design thinking pedagogy: The educational design ladder. Innovations in Education and Teaching International 2017;54:374–85.
[11] Sharples M, de Roock R, Ferguson R, Gaved M, Herodotou C, Koh E, Kukulska-Hulme A, Looi C-K, McAndrew P, Rienties B, et al. Innovating pedagogy 2016: Open University innovation report 5. Institute of Educational Technology: The Open University; 2016.
[12] G.J. Hill, Review of a problems-first approach to first year undergraduate programming, in: Software Engineering Education Going Agile, Springer, 2016, pp. 73–80.
[13] Tahy ZS. How to teach programming indirectly–using spreadsheet application. Acta Didactica Napocensia 2016;9:15–22.
[14] C.W. Okonkwo, A. Ade-Ibijola, Python-bot: A chatbot for teaching python programming., Engineering Letters 29 (2020).
[15] Sorva J, Karavirta V, Malmi L. A review of generic program visualization systems for introductory programming education. ACM Transactions on Computing Education (TOCE) 2013;13:1–64.
[16] G.-J. Hwang, H. Xie, B.W. Wah, D. Gašević, Vision, challenges, roles and research issues of artificial intelligence in education, 2020.
[17] Malik SI, Mathew R, Al-Nuaimi R, Al-Sideiri A, Coldwell-Neilson J. Learning problem solving skills: Comparison of e-learning and m-learning in an introductory programming course. Education and Information Technologies 2019;24:2779–96.
[18] Grover S, Jackiw N, Lundh P. Concepts before coding: non-programming interactives to advance learning of introductory programming concepts in middle school. Computer Science Education 2019;29:106–35.
[19] Guo P. Python is now the most popular introductory teaching language at top us universities (2014). Blogs: Communications in ACM; 2017.
[20] Bittencourt RA, dos Santos DMB, Rodrigues CA, Batista WP, Chalegre HS. Learning programming with peer support, games, challenges and scratch. In: 2015 IEEE Frontiers in Education Conference (FIE). IEEE; 2015. p. 1–9.
[21] S. Iqbal Malik, Role of ADRI model in teaching and assessing novice programmers, Technical Report, Deakin University, 2016.
[22] Malik SI, Coldwell-Neilson J. A model for teaching an introductory programming course using ADRI. Education and Information Technologies 2017;22:1089–120.
[23] Bogaerts SA. One step at a time: Parallelism in an introductory programming course. Journal of Parallel and Distributed Computing 2017;105:4–17.
[24] Komarova N, Zamkovoi A, Novikov S. The fourth industrial revolution and staff development strategy in manufacturing. Russian Engineering Research 2019;39:330–3.
[25] S. Grover, S. Basu, Measuring student learning in introductory block-based programming: Examining misconceptions of loops, variables, and boolean logic, in: Proceedings of the 2017 ACM SIGCSE technical symposium on computer science education, 2017, pp. 267–272.
[26] Medeiros RP, Ramalho GL, Falcão TP. A systematic literature review on teaching and learning introductory programming in higher education. IEEE Transactions on Education 2018;62:77–90.
[27] G. Salvaneschi, M. Mezini, Debugging for reactive programming, in: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), IEEE, 2016, pp. 796–807.
[28] Simon L, Chisnall D, Anderson R. What you get is what you c: Controlling side effects in mainstream c compilers. In: 2018 IEEE European Symposium on Security and Privacy (EuroS&P). IEEE; 2018. p. 1–15.
[29] J. Sundnes, Loops and lists, in: Introduction to Scientific Programming with Python, Springer, 2020, pp. 19–34.
[30] Poole M. Extending the design of a blocks-based python environment to support complex types. In: 2017 IEEE Blocks and Beyond Workshop (B&B). IEEE; 2017. p. 1–7.
[31] S. Iqbal Malik, J. Coldwell-Neilson, Impact of a new teaching and learning approach in an introductory programming course, Journal of Educational Computing Research 55 (2017) 789–819.
[32] L.J. Höök, A. Eckerdal, On the bimodality in an introductory programming course: An analysis of student performance factors, in: 2015 International

Conference on Learning and Teaching in Computing and Engineering, IEEE, 2015, pp. 79–86.
[33] Ade-Ibijola A. Synthesis of regular expression problems and solutions. International Journal of Computers and Applications 2018:748–64.
[34] Lucariello JM, Nastasi BK, Anderman EM, Dwyer C, Ormiston H, Skiba R. Science supports education: The behavioral research base for psychology's top 20 principles for enhancing teaching and learning. Mind, Brain, and Education 2016;10:55–67.
[35] K. Chowdhary, Natural language processing, in: Fundamentals of Artificial Intelligence, Springer, 2020, pp. 603–649.
[36] Gulwani S. Synthesis from examples: Interaction models and algorithms. In: 2012 2012 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing. IEEE; 2012. p. 8–14.
[37] A. Ade-Ibijola, Synthesis of hypothetical sociograms for social network analysis, in: 2018 5th International Conference on Soft Computing & Machine Intelligence (ISCMI), IEEE, 2018, pp. 79–83.
[38] S. So, H. Oh, Synthesizing pattern programs from examples., in: IJCAI, 2018, pp. 1618–1624.
[39] Ade-Ibijola A. Synthesis of social media profiles using a probabilistic context-free grammar. In: 2017 Pattern Recognition Association of South Africa and Robotics and Mechatronics (PRASA-RobMech). IEEE; 2017. p. 104–9.
[40] S. Kabaso, A. Ade-Ibijola, Sell-bot: An intelligent tool for advertisement synthesis on social media, in: The Disruptive Fourth Industrial Revolution, Springer, 2020, pp. 155–178.
[41] Edwards JM, Fulton EK, Holmes JD, Valentin JL, Beard DV, Parker KR. Separation of syntax and problem solving in introductory computer programming. In: 2018 IEEE Frontiers in Education Conference (FIE). IEEE; 2018. p. 1–5.
[42] Rivers K, Koedinger KR. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. International Journal of Artificial Intelligence in Education 2017;27:37–64.
[43] S. Gulwani, I. Radiček, F. Zuleger, Feedback generation for performance problems in introductory programming assignments, in: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014, pp. 41–51.
[44] Kim D, Kwon Y, Liu P, Kim IL, Perry DM, Zhang X, Rodriguez-Rivera G. Apex: automatic programming assignment error explanation. ACM SIGPLAN Notices 2016;51:311–27.
[45] L. Kallmeyer, Parsing beyond context-free grammars, Springer Science & Business Media, 2010.