Teaching Introductory Programming: A Quantitative Evaluation of Different Approaches

THEODORA KOULOURI, STANISLAO LAURIA, and ROBERT D. MACREDIE,
Brunel University

Teaching programming to beginners is a complex task. In this article, the effects of three factors—choice of programming language, problem-solving training, and the use of formative assessment—on learning to program were investigated. The study adopted an iterative methodological approach carried out across 4 consecutive years. To evaluate the effects of each factor (implemented as a single change in each iteration) on students' learning performance, the study used quantitative, objective metrics. The findings revealed that using a syntactically simple language (Python) instead of a more complex one (Java) facilitated students' learning of programming concepts. Moreover, teaching problem solving before programming yielded significant improvements in student performance. These two factors were found to have variable effects on the acquisition of basic programming concepts. Finally, it was observed that effective formative feedback in the context of introductory programming depends on multiple parameters. The article discusses the implications of these findings, identifies avenues for further research, and argues for the importance of studies in computer science education anchored on sound research methodologies to produce generalizable results.

Categories and Subject Descriptors: K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer science education; curriculum*

General Terms: Experimentation, Human Factors, Measurement

Additional Key Words and Phrases: Empirical studies, teaching strategies, novice programmers, learning programming, CS1, problem solving, programming languages, formative feedback

ACM Reference Format:
Theodora Koulouri, Stanislao Lauria, and Robert D. Macredie. 2014. Teaching introductory programming: A quantitative evaluation of different approaches. ACM Trans. Comput. Educ. 14, 4, Article 26 (December 2014), 28 pages.
DOI: http://dx.doi.org/10.1145/2662412

1. INTRODUCTION

The past decade has witnessed a boom in industry demand for computing expertise, driven by the dramatic expansion of computing use, the growing influence of computing on the economy, and the constant influx of new technologies into everyday life [CC2005 2005]. Yet, while the demand for computing graduates is growing, university Computer Science departments in the UK [National Audit Office 2007] and USA [DARPA-RA 2010] report declining enrollment and high attrition rates on their degree programs. Dropout and failure rates are exacerbated during and between the first and second years of these programs, reaching as high as 30%–40% [Beaubouef and Mason 2005] and meaning that even where students begin studying for Computer Science degrees, a high proportion do not go on to graduate.

Authors' address: T. Koulouri, S. Lauria, and R. D. Macredie, Department of Computer Science, Brunel University, Uxbridge, UB8 3PH, UK; emails: {theodora.koulouri, stasha.lauria, robert.macredie}@brunel.ac.uk.

ACM 1946-6226/2014/12-ART26 $15.00
DOI: http://dx.doi.org/10.1145/2662412

These high attrition rates are associated with considerable failure rates in introductory programming courses and/or disenchantment with programming [McGettrick et al. 2004; Nikula et al. 2011]. This disenchantment persists with graduates "expressing a dislike of programming and reluctance to undertake it" [McGettrick et al. 2004, p. 12; see also Ma et al. 2007]. This is also reflected in the outcomes achieved by students, with large, multinational studies reporting that even at the conclusion of their introductory programming courses, a large number of students show substandard performance in elementary programming tasks [McCracken et al. 2001; Lister et al. 2004]

The tension between contemporary society's need for a supply of skilled computer scientists who can program and the manifest issues in programming achievement demonstrated by many students in degree programs presents higher education institutions with opportunities and challenges in developing skilled computer scientists. Core to any computer science education is the nurturing of the ability to develop well-designed software, and all of the attendant conceptual and practical skills that underpin this activity. Programming is the practical mechanism through which these skills and abilities are realized. It forms an essential part of any Computer Science (CS) degree and, as already noted, its teaching typically begins in the first year of a degree program, with an introductory programming course (often referred to as CS1). The approach to beginning programming at this early stage in CS degrees is supported by students, CS and non-CS faculty, and near-time employers [CC2001 2001].

The changes to the computing industry associated with, *inter alia*, new technologies, and the problems associated with learning to program, have led to higher education institutions facing the pressing need to rethink their CS curricula, with special attention given to redesigning the CS1 course, which then has wider effects. Developing or revising an introductory programming course has a key role in, and impact on, the wider curriculum. An introductory programming course prepares students for, and underpins, subsequent courses, which are typically also revised in response to changes to CS1; and the overall degree prepares students for their future careers.

The complexity of teaching introductory programming is widely acknowledged among educators [Robins et al. 2003], being listed as one of the seven challenges in Computer Science Education (CSE) [McGettrick et al. 2004]. It is also understood that learning programming is a complex and multifaceted process. Novice programmers show a fragile ability to take a problem description, decompose it into subtasks, and implement them [McCracken et al. 2001]. They also have difficulty in tracing, reading, and understanding pieces of code and fail to grasp basic programming principles and routines [Lister et al. 2004]. The overhead of learning the syntax and semantics of a language at the same time, and difficulties in combining new and previous knowledge and developing their general problem-solving skills, all add to the complexity of learning how to program [Linn and Dabley 1989].

Taken together, these issues highlight the importance of identifying both the appropriate content and pedagogy for CS1 in order to prepare students for the rest of the degree programs and provide the foundation for effective computing careers. Different approaches to teaching programming which reflect the mix of pedagogy and content issues have been set out, notably in the IEEE and ACM Joint Task Force on Computer Curricula [CC2001 2001].

Debates around the effectiveness of these approaches to teaching introductory programming are well established, of course. Pears et al. [2007], for example, performed a large literature review on the teaching of introductory programming which led to the identification of four major categories relating to course development: curriculum, pedagogy, language choice, and tools for supporting learning. They reviewed studies in each of these four areas and discussed several unresolved issues before making recommendations in relation to them. First, they noted that there was a conspicuous lack

of an accepted framework/methodology to guide the processes of planning, designing, developing, revising, and implementing CS1 courses. Second, they pointed out that empirical data from small-scale studies are abundant in the literature, but that they rarely inform these processes. Pears et al. [2007] concluded that larger-scale systematic studies that provide significant new empirical results are needed, and that such research has the potential to provide valid and applicable recommendations for educators regarding what and how to teach novice programmers. Taking this argument as our starting point, the aim of this article is to identify an effective approach to teaching CS1 through applying a valid methodological framework. The remainder of the article is organized as follows. Section 2 reviews existing approaches to the design of CS1 courses to confirm the lack of systematic empirical studies in this area, providing the motivation for the study reported later in this article. The analysis reveals that the majority of studies are small scale, nonlongitudinal, or nonquantitative (i.e., they are either based on questionnaires or provide analysis that is nonstatistical). The section argues that relatively few studies that compare teaching approaches and, especially, approaches that employ simple and complex teaching languages, have reliably evaluated the outcomes of those approaches. As such, open questions remain with regards to what should be included in the CS1 curriculum and how it should be taught. The knowledge gaps identified in Section 2 give rise to the article's central research hypothesis, which is presented in Section 3. The article focuses on questions targeting three specific aspects of teaching introductory programming and develops relevant interventions. By reframing empirical problems as research problems, Section 4 formulates hypotheses that target three teaching interventions and seeks to provide a foundation from which to address teaching- and curriculum-related questions by developing an iterative methodology/framework for the design implementation and evaluation of a CS1 course. This is reported through a longitudinal (4-year), large-scale ($>$750 participants) study. To measure the outcome of each annual iteration—the extent of student learning—the approach uses quantitative/objective measures, in particular the frequency of (i) programming constructs, typically considered as fundamental concepts—that is, loops—conditionals, and libraries; and (ii) bugs, identified as semantic or syntactic errors in the software that students produce. Through the statistical analysis presented in Section 5, the study reveals the set of parameters that were found to improve student learning in the CS1 course. The article concludes by evaluating the effectiveness of two popular programming languages (Java and Python) for CS1 teaching, contributing to the language choice debate, and the potential benefit of formative feedback and of exposure to problem solving before exposure to programming. Finally, the article discusses how these parameters affected different elements and aspects of student learning. Taken together, the study aims to go beyond being an experience report of teaching a language or a comparison of two languages. Rather, it attempts to provide a model on which two languages can be compared in this context and to present a framework to guide the redesign of the CS1 curriculum.

2. LITERATURE REVIEW

One of the major issues influencing the design or restructuring of a CS1 course is the choice of programming language, covered in the IEEE and ACM Joint Task Force on Computer Curricula [CC2001 2001] within the context of "implementation strategies." CC2001 [2001, p. 24] describes and discusses the three different, typical approaches for introductory programming courses as follows: "'programming-first' implementations are an imperative-first approach that uses the traditional imperative paradigm; an 'objects-first' approach emphasizes early use of objects and object-oriented design; and a 'functional-first' approach introduces algorithmic concepts in a language with a simple functional syntax."

Arguments have been made for the value of each approach. For example, the advantage of the objects-first approach has been suggested to relate to the importance of object-oriented programming in industry, with early exposure to object-oriented principles and concepts considered an advantage for students. However, it has been recognized that curriculum implementations that employ an objects-first approach may be ill fated, because the languages typically used for delivery (such as C++ and Java) are much more complex in terms of syntax and semantics than the classical languages. Unless special care is taken, this complexity may prove overwhelming for novice programmers [CC2001 2001, p. 30]. This has led many educators and researchers to advocate that a necessary criterion for selecting a CS1 language is simple syntax and structure [Mannila and deRaadt 2006]. Within the movement toward less complex teaching languages, Python has gained much support [Zelle 1999].

Numerous studies have contributed to the debate on which paradigm (e.g., "objects-first" or "imperative-first") and on which language is most appropriate for teaching introductory programming, yet the results remain inconclusive. These two aspects—of paradigm and language—are, of course, not completely independent, meaning that it is not always possible to discuss them in isolation. The study reported in this article does not address the paradigm debate per se, but will focus on the choice of programming language, discussing evidence that supports the suitability of Python as a teaching language.

This is an important issue as there is also a lack of consensus with regard to which is the most appropriate programming language for first-year computing students. While Java remains a popular choice because of its association with commercial use, the difficulty in learning programming is aggravated by its previously noted complex syntax and semantics. As such, Java's notational overhead has been criticized as making it unsuitable for novice programmers. Python, on the other hand, has a simple and clean syntax and structure and other characteristics that make it appealing for teachers and learners, such as dynamic typing, powerful built-in functions and structures, and a simple development environment.

Python has recently tended to be the language of choice, typically within the "imperative-first" approach, perhaps because it started as an "academic" language, which, nevertheless, evolved to be "commercial" [Zelle 1999]. Further, Python was designed to have simple syntax and semantics leading to the elimination of the vast majority of errors commonly made by novice programmers, such as missing semicolons, bracketing problems, and variable type declaration errors [Jadud 2005]. Python is also increasingly being used in real-world applications (for instance, in high-profile organizations like Google and Nokia (see Miller and Ranum [2006]). Finally, since Python also supports the object-oriented paradigm, it can be used as a transition language for second-term or second-year courses that are based on C++ and Java [Goldwasser and Letscher 2008].

There are a number of empirical studies that document the process and outcomes of revising introductory programming courses, with an emphasis on switching to Python as a teaching language. In particular, Grandell et al. [2006] found that Python facilitated teaching and learning and increased student satisfaction. These conclusions were based on analysis of grade distributions, self-reports, identifying the use of constructs in students' code, and surveys of student attitudes toward programming. However, despite the multiple measures used, the authors did not provide evidence of statistical or grounded analysis, which may limit the strength of the conclusions that they draw from the study. Similarly, Kasurinen and Nikula [2007] reported that changes to the course and moving from C to Python led to higher grades, improved student satisfaction, and a decline in dropout and failure rates. A discussion and evaluation of the "Python first" approach by Radenski [2006] led to the development of a full online study pack

implementing this approach. The results of a survey administered to students who made use of the study pack revealed positive perceptions of Python as a programming language. Along the same lines, empirical studies by Patterson-McNeill [2006], Stamey and Sheel [2010], Miller and Ranum [2005], Oldham [2005], Goldwasser and Letscher [2008], and Shannon [2003] detail the choices involved in redesigning the curriculum and the shift from one language to another in their colleges and argue for dramatic improvements with the use of Python, but no explicit evaluation results are reported in any of these studies.

From the wider perspective of computing education research, Pears and Malmi [2009] argued that a large part of the literature to date has lacked methodological rigor. In particular, an extensive methodological review of 352 papers concluded that the majority of published findings cannot be considered reliable and, as a result, many educators and researchers are forced to "reinvent the wheel" (as also suggested by Almstrum et al. [2005]). The review found that one-third of studies did not use human participants or any type of analysis, but presented descriptions of interventions and anecdotal evidence, while the rest of the studies employed mostly questionnaires. Random sampling and control groups were rarely used, and even when quantitative approaches were adopted, only one third of these studies performed robust statistical analysis. Finally, more than half of the studies did not adequately describe their methods and procedures, while around 25% of them did not state research questions and/or review existing literature [Randolph et al. 2008]. Nevertheless, there has been a recent paradigm shift toward more systematic and focused investigations [Pears and Malmi 2009]. Notable examples include a study by Nikula et al. [2011]; building on a clear methodological framework, the Theory of Constraints, a 5-year longitudinal study was undertaken in which problems were diagnosed, and suitable interventions (mainly targeting course and student motivational problems) were designed, implemented, and evaluated through pass rates. Remarkable methodological rigor can also be found in the paper by Stefik and Siebert [2013] that reported four large-scale empirical studies involving randomized controlled trials. Relevant to the research focus of the present study, their analysis indicated that novice programmers perform better with Python-style syntax compared to C-style syntax and identified syntactic elements that are most problematic for learners. Beck and Chizhik [2013] also employed a robust methodology in order to compare traditional teaching methods and cooperative learning techniques using control groups and final grades.

Taken together, the review of the relevant literature clearly demonstrates that there are teaching approaches and programming languages that offer opportunities for better supporting teachers and learners. However, the evaluation results associated with the approaches implemented in these studies are, variously, missing, partial, based only on qualitative data, or have not been validated through statistical analysis. Moreover, most of these studies present the results of designing and implementing the new approach during one academic year in comparison with the results of the previous year's approach. Therefore, it is argued that while such studies may provide invaluable insights into good practices and innovative teaching techniques, if they do not select and apply a suitable research framework, they remain experience accounts, bound to a specific context, and may not be useful for offering reliable guidelines applicable to other higher education situations. Finally, the lack of conclusive findings in the field may be attributed, as underlined in Ehlert and Schulte [2009], to the lack of standard measures based on which the "old" and "new" teaching approaches can be fairly compared. Following this line of argument, we believe that appropriate key indicators to assess learning outcomes need to be identified.

The relation between education research, practice, and policy has become prominent, such that evidence-based education reform has been a key element in government

policies, which require schools to justify their choices and practices using findings from rigorous, experimental research [U.S. Department of Education 2001]. Thus, Slavin [2003] maintains that the validity of educational research should follow the criteria of other scientific fields, such as medical research, which include a control group, low variance among groups, large sample size, and statistically significant results. At the same time, it is recommended that applying research methodologies from other disciplines should be performed with caution, given the fact that education research aims to address problems in real-world, dynamic situations, and not in highly controlled settings, like laboratories [Collins et al. 2004; Reeves 2011].

Following these recommendations, the study reported in the remainder of this article aims to address the previously mentioned shortcomings through a number of means: first, by developing a suitable methodology, which includes multiple development iterations over more than 1 academic year; second, by defining objective parameters to measure the outcomes of the curriculum revisions; and third, by evaluating the data associated with the proposed measures through statistical analysis. In this way, the influence of the curriculum revisions can be assessed by impartially measuring and analyzing the variance of key indicators that are associated with student performance. To create a quasi-controlled experimental situation, each iteration in this study corresponds to an academic year and involves a single change in the module structure. As such, variance in the key indicators may be argued to be associated with a particular change.

The research adopts an experimental paradigm, which also takes elements from prominent iterative methodologies in education research, namely, action research [Stenhouse 1975] and design-based research [Brown 1992]. Like many experimental methodologies, once a problem is identified, action research and design-based research evolve through multiple cycles of analysis, design, and evaluation. In particular, action research and design-based research involve a spiral of self-reflective cycles of planning a change, acting, observing the consequences of the change, reflecting on consequences, replanning, acting, observing, reflecting, and so on. However, being mostly qualitative, these approaches have been criticized as being too context bound, only aiming to improve a situation but without producing generalizable findings [Koshy 2009]. Yet, these methodologies embrace the complexities, dynamics, and limitations of authentic real-world educational settings, which are not captured by laboratory-based experimental paradigms [Collins 1992; Collins et al. 2004]. Indeed, it is argued that insights and tools generated and evaluated within a normal setting are more likely to be useful and relevant [Reeves 2011].

As mentioned previously, the use of specific objective measurements as part of the observation process is an important aspect introduced in this article. This, in turn, should enable the potential bias issue (a common criticism of action research and design-based research paradigms) to be addressed during the evaluation and comparison stages. The aim of these objective comparisons introduced in the article is to quantitatively measure a student's ability to master basic programming concepts, and to achieve this, key indicators from student assessments are used in the evaluation. In effect, this study attempts to address the methodological gap identified by Pears et al. [2007] and Ehlert and Schulte [2009] and proposes a framework to guide course design and revision based on an established research paradigm and objective metrics.

Similar work has already been presented in Mannila et al. [2006] who used both a quantitative and qualitative approach in their investigation. The study involved the analysis of 30 programs written in Java and 30 programs written in Python produced by high school students in 2 different academic years. The results revealed that the Python programs contained fewer logic and syntax errors and more frequently fulfilled the required functionality. Interviews with eight of the students who had learned Python after Java revealed positive perceptions of the language.

In the same vein, the study reported in the remainder of this article uses quantitative key indicators as a measure of student programming proficiency; in particular, in addition to the frequency of syntax errors (bugs), the study also employs other quantitative indicators such as the presence of loop, conditional, and library constructs, which constitute key programming concepts. Another point of departure of the study is the sample makeup and size: The investigation involved more than 750 students from an undergraduate first-year degree level. Finally, in the study reported in this article, several cycles were introduced in the iterative process as discussed in this section. In the next section, details of the iterations are presented and discussed.

3. CENTRAL RESEARCH HYPOTHESIS

The discussion so far has stressed the greater-than-ever importance of programming skills, which increases the urgency for CSE to identify the fittest teaching approaches, especially pertaining to the CS1 module. However, there appears to be a limited knowledge base of tested teaching approaches on which practitioners at different educational institutions can draw. As such, the primary research question of the study was to determine whether a student's ability to learn how to program is dependent on the teaching approach of the CS1 module; this was the basis of the iterative paradigm employed in the investigation discussed in this article. The central research hypothesis was formulated as follows:

Ha: Programming proficiency of novice learners is dependent on the teaching approach of a CS1 module.

In order to address the central research hypothesis and produce teaching recommendations, the study involved three cycles in which in-class interventions were developed, implemented, and evaluated based on objective, quantifiable criteria.

A detailed description of the research methods, interventions, set of assumptions, and evaluation metrics is provided in the following sections.

4. RESEARCH METHODOLOGY

Central to this work was the selection and application of a clear research methodology. The nature of this research and the issues it aims to address were best served by a controlled experimental design. The appeal of such approach lies in its focus on hypothesis testing, statistical analysis of quantitative data, and fixed variables and procedures that can be replicated. Indeed, it is argued that educational studies that are not heuristic/exploratory, but have some very specific research questions or hypotheses ("is [teaching approach A] better than [teaching approach B] for novice programmers?") could benefit from using the type of controlled trials traditionally used in biomedical sciences [Stefik and Siebert 2013]. At the same time, the work also drew on studies within the tradition of action research and design-based research, which embrace the complexity of real-world learning environments that resist full experimental control. These approaches were developed to guide formative research, to test and refine educational designs and interventions based on theoretical findings from previous research. Their aim is not only to refine educational practice, but also to refine theory and to contribute to the field [Collins et al. 2004]. The research philosophy underpinning our efforts has been one of "progressive refinement," as also described and applied in action research and design-based research studies. This approach is cyclic and involves the formulation of a hypothesis, the design of an intervention, experimentation in the classroom, the analysis of the collected of data, the formulation of a new hypothesis, the design of a new intervention, and so on [Molina et al. 2007]. Within this paradigm, researchers and/or practitioners initiate a project because they have diagnosed a

problematic situation, such as an ill-suited curriculum. The diagnosis is often triggered by an external event, such as abnormally poor grades or feedback provided by students. Indeed, the efforts described in this article were triggered by the high failure rate of a student cohort (which was above 36%, as shown in Table IX). Subsequently, the formulation of the hypothesis to be tested is based on experience, previous research, and theoretical principles, as analyzed by the team of practitioners and researchers involved.

For this research, the body of findings discussed in Section 2, suggesting that the complexity of Java may be overwhelming for learners and favoring Python as a suitable teaching language, led to the development of the first hypothesis (presented in Section 4.2). The second hypothesis and related intervention were also the products of careful review of literature indicating the pedagogical benefit of formative feedback. This iteration cycle is discussed in Section 4.3. Finally, as described in Section 4.4, the intervention of the third iteration targeted the problem-solving skills of learners, exploring the hypothesis of a correlation between problem-solving and programming, informed by analysis of existing literature.

As detailed in Section 4.5, the study involved four experimental groups of CS1 students, which corresponded to four full student cohorts in four consecutive years: a control group that was taught using Java, a group that was taught using Python, a group that received formative feedback, and a group that received initial problem-solving training.

4.1. The Iterative Process

The methodological approach consisted of three iterations, in which a single parameter that could potentially improve programming proficiency was identified, introduced, and evaluated. Sections 4.2 through 4.4 provide detailed descriptions of each iteration and, in particular, set out the theoretical and research findings that motivated the intervention and led to the formulation of the associated research hypothesis, and the design and implementation of the intervention. Students' programming ability was measured through their use of specific key programming constructs and the presence of bugs in the implementation of a required computer program. The computer program was part of a final, formal assessment (as explained in Section 4.7). The key concepts included basic programming constructs such as conditionals and loops, and are discussed in Section 4.8.

4.2. First Iteration: Python is Introduced as the Introductory Programming Language Replacing Java

4.2.1. Motivation for the Intervention and Research Hypothesis. Java was used as the introductory programming language with the previous cohort of students (the "control group" for the study). However, as indicated by the findings considered in Section 2, the use of Java to introduce students to the basic aspects of programming may be problematic, not least because Java is heavily coupled with object-oriented concepts of which this may interfere with the basic aim of an objects-later strategy. Java "forces" some of the more advanced concepts into the foreground—concepts which teachers do not typically want to introduce at an early stage [Kölling 1999]. As a consequence, a student's focus may switch from learning the basic programming concepts to learning the language's syntax. Drawing on the evidence that Java may not be well suited for education, especially when introducing programming to novices [Siegfried et al. 2008], the use of an alternative programming language with a lower syntactic burden was considered. As discussed in the previous sections of the article, several educators have argued that scripting languages could offer a more effective alternative. Python is one example, offering a simple and expressive language with support for procedural

Table I. Comparison of Programs in Java and Python

Concepts Taught	Examples of Implementations of the Concepts	
	Java	Python

Variables, primitive data types, and data structures (arrays and lists)	`public class ExampleClass`
	`{`
	`  public static void main(String[]args)`
	`  {`
	`  int x = 42;`
	`  System.out.println(x);`
	`  }`
	`}`
		`x = 42`
		`print x`

Control structures: selection	`int x = 42;`
	`if (x%2==0)`
	`{`
	`  System.out.println(x + " is even");`
	`}`
	`else`
	`{`
	`  System.out.println(x + " is odd");`
	`}`
		`x = 42`
		`if x%2==0:`
		`  print x, "is even"`
		`else:`
		`  print x, "is odd"`

Control structures: iteration	`for (int i = 0; i <42; i++)`
	`{`
	`  System.out.println(i);`
	`}`
		`for i in range (0,42):`
		`  print i`

Sub-programs: procedures/ methods	`public static boolean evenOdd(int n)`
	`{`
	`  if (n%2==0)`
	`  {`
	`  return true;`
	`  }`
	`  else`
	`  {`
	`  return false;`
	`  }`
	`}`
		`def evenOdd(n):`
		`  if n%2==0:`
		`  return True`
		`  else:`
		`  return False`

Importing libraries	`import java.util.Random;`
	`Random randy = new Random();`
	`int r= randy.nextInt(10);`
		`import random`
		`r= random.randrange(10)`

programming. It can be argued that the lower overhead associated with Python should provide a gentler introduction to the basic concepts of programming. By using Python, a greater emphasis on core principles was expected with less of an unwanted focus on syntax. As already noted, Python is also widely used in industry and is therefore considered to be attractive to students. In Table I, a set of programs are written in Java and Python to exemplify the syntactic and semantic differences between the two languages. These example implementations correspond to the concepts taught in the course (please refer to Table IV and Section 4.6). The juxtaposition of these programs suggests that Python has a simple and intuitive syntax. On the other hand, a basic program in Java may expose students to notation and concepts that cannot be understood until well into their study. As such, Python is expected to provide a speedier, less overwhelming, and possibly more effective startup platform for novice learners.

For all of these reasons, Python was selected as the programming language to replace Java in the first iteration where, in effect, we observed the impact of the complexity of programming languages. The associated research hypothesis is as follows:

Ha0_1: Programming proficiency increases for novice learners using Python compared to Java as the introductory programming language in a CS1 module.

4.2.2. Implementation of the Intervention. The experimental design required a "common basis of comparison." This meant that a large number of learning and teaching parameters needed to be equal or equivalent between cycles. In particular, in all iterations, the same programming concepts were taught in the same order (see Section 4.6). This was only possible for the first 10 weeks of instruction. As such, this period formed the basis on which the four cohorts would be compared. Similarly, the assessment, which involved a 1-hour laboratory test in week 11, marking procedures, and module organization were kept constant (see Section 4.7).

4.3. Second Iteration: Formative Feedback is Introduced

4.3.1. Motivation for the Intervention and Research Hypothesis. In educational settings, feedback is crucial to improving knowledge and skill acquisition as well as motivating learning [Shute 2008]. Formative feedback is the information provided to students about their strengths and weaknesses in order to improve their learning and performance, and does not usually contribute toward the final grade of the student [Lilley and Barker 2007]; this is in contrast to summative feedback which, as part of a formal assessment such as a final exam or coursework, contributes to the final grade. There appears to be a general consensus in educational research and policy that formative feedback is paramount for student learning [Black and William 1998; Quality Assurance Agency for Higher Education 2003], and the benefits of formative feedback have been found to be particularly pronounced for novice students [Moreno 2004; Shute 2008]. Students also appear to value and expect feedback [Weaver 2006; Higgins et al. 2002]. Within the context of teaching programming to novices, Corbett and Anderson [2001] showed that for formative feedback to be most effective for programming students, it has to be immediate and point to individual steps and features in the learner's work.

Based on this literature, for the second iteration, it was anticipated that introducing formative feedback would be beneficial for the students. This leads to the next research hypothesis:

Ha1_2: Programming proficiency increases for novice learners receiving regular formative feedback in a CS1 module.

4.3.2. Implementation of the Intervention. Students were encouraged to submit the programs that they produced as solutions to weekly laboratory exercises (see Module Organization in Section 4.6). These students received formative feedback on their submitted programs within 7 days. A two-stage, semiautomatic process was used to generate the feedback. First, an ad-hoc application ran every program and compared its output with the expected output. If this output was different from that expected, or if the application located a bug (runtime and compile time—syntax—error), it would produce feedback and would flag the program for manual inspection. Second, the human inspector assessed the code and provided detailed comments, which were added to the automatically generated feedback. The formative feedback involved identifying both errors and successful aspects of a student's submitted program and, where appropriate, referred to the lecture material and useful sources and provided hints toward the completion of the task, but did not give the solution. As such, students could attempt the tasks and receive feedback any number of times. Table II contains an example of a syntax error (missing quotation marks around a string value), and the feedback and information that would be automatically and manually generated for this error.

It should be noted that the same tasks were given to students in the previous iteration. To minimize the likelihood of this activity being perceived as a form of monitoring students' outcomes, no grading was associated with it. However, a consequence of the

Table II. Example of Student Submission and Formative Feedback Provided

Task	Student Submission	Automated Feedback	Manual Feedback
Write a program that prints the string Hello World	`s = Hello World` `print s`	global name 'Hello World' is not defined	Strings should be enclosed in quotation marks for the data to be recognized as a string and not a variable name. Please refer to week 2 lecture notes (slides 5–10).

noncompulsory element of the activity was that the submission rate was not as high as desired; only 50% of the students engaged at least once in the process.

4.4. Third Iteration: Problem-Solving Training Is Introduced

4.4.1. Motivation for the Intervention and Research Hypothesis. The purpose of an introductory programming course is to develop students' problem-solving skills and introduce them to primary concepts of design and programming. Yet, this aim is not always clear, and a common misconception of students is that the course is a "Java class" or a "C++ class" [Zelle 1999]; a tendency sometimes attributed to the complexity of these languages [Zelle 1999]. According to the CC2001 [2001], problem solving is the key skill in the discipline, but is also where the greatest weakness of novice students lies. It is, again, recognized that language syntax can detract from developing problem-solving concepts [CC2001 2001, p. 23]. As such, it is recommended that all introductory courses (independent of the underlying—"objects-first" or "objects-later"—paradigm or choice of language) include sessions in which problem solving and problem-solving processes are covered. Thus, at this stage in the iterative process, it was noted that a lack of problem-solving skills could be one of the underlying reasons for students having difficulty in utilizing key concepts, such as loops and conditionals. Therefore, in the last cycle of the iterative process the focus was on problem-solving training. This resulted in the final research hypothesis:

Ha2_3: Programming proficiency increases for learners receiving problem-solving training before the beginning of a CS1 module.

4.4.2. Implementation of the Intervention. During the final cycle, a "crash course" was introduced during induction week (i.e., just before the formal start of the module at the beginning of the academic year), and lasted three hours. The aim of this preparatory session was to reinforce the idea of the course as training in problem solving and algorithmic thinking as opposed to simply learning a programming language.

As such, the design of the session followed the four principles (phases) of problem solving described in Polya [1973]. In particular, the session referred to and used the four phases to guide each stage of the activity. First, the students were presented with the "problem." They were asked to control a robot through a basic set of natural language instructions to complete a given task within a virtual environment. Question-answering prompts were used by the staff to encourage discussion with and among the students in order to develop their understanding of the problem. The problem involved specific constraints; that is, the natural language processing abilities of the virtual robot were restrained such that the robot could only turn left and move one step at a time. Second, the students were asked to devise a plan by considering different strategies (drawing a picture, working backwards, route decomposition, etc.). Third, the students implemented their plan with the virtual robot. Finally, the students were asked to reflect on, and evaluate, each step of their plan—the solution to the "original problem." In this last part of the session, the students reviewed the steps again and attempted to

Table III. Research Hypotheses
The overarching hypotheses of the study and those formulated for each iteration of the study.

Hypotheses	
Ha	Programming proficiency of novice learners is dependent on the teaching approach of a CS1 module.
Ha0_1	Programming proficiency increases for novice learners using Python compared to Java as the introductory programming language in a CS1 module.
Ha1_2	Programming proficiency increases for novice learners receiving regular formative feedback in a CS1 module.
Ha2_3	Programming proficiency increases for novice learners receiving problem-solving training before the beginning of a CS1 module.

Table IV. Module Content and Teaching Sequence
The topics taught in each session.

Stage	Topics
T1	Overview
T2	Variables, primitive data types, and data structures (arrays and lists)
T3	Control structures: selection
T4	Control structures: iteration
T5	Subprograms: procedures/methods
T6	Importing libraries
T7	Syntax errors and exceptions

extend the solution for a real robot scenario (which represented the "future problem"). The set of instructions was fed to and translated by an application in order to be executed by the robot in real time. The session did not touch on programming concepts such as loops, conditionals, and arrays. For this last iteration, the impact of introducing these problem-solving tutorials was again assessed using the same approach as for the other cycles.

The research hypotheses tested in the study are consolidated in Table III.

4.5. Participants

4.5.1. The Groups. Four distinct groups of students took part in the different iterations of the study; the experience of each group is outlined next.

G0: For this group, during the 10 weeks before the test, teaching emphasized mastery of basic programming skills with Java as the implementation language. In particular, lecture and laboratory sessions and exercises focused on loops, conditionals, use of libraries and packages (e.g., to generate random numbers and for input and output), and so on (the complete list of topics can be found in Table IV). These students used BlueJ as the learning environment. This cohort served as the control group for the study. The size of this cohort was 157 students (with 19% being female).

G1: The G0 and G1 groups had essentially the same structure to their teaching/learning experience, with the programming language used being the only difference for the groups. Python, rather than Java, was used for the G1 cohort. These students used Python IDLE as the learning environment. This group consisted of 195 students (with 18% being female).

G2: The G1 and G2 groups had essentially the same structure to their teaching/learning experience. Throughout the term, the groups were given weekly, ungraded tasks (exercises to be implemented using Python), but students in the G2 group were encouraged to also submit their solutions. Formative feedback was provided for the submitted work. Task submission was not compulsory. The size of this group was 193 students (25% being female).

G3: The G3 group had essentially the same structure to their teaching/learning experience as the G2 group but had an additional 3-hour problem-solving "crash course" before the beginning of the module. The G3 group consisted of 216 students (with 22% being female).

4.5.2. Student Background. The four groups corresponded to four full cohorts of students from consecutive years, enrolled in the first year of the suite of computing degrees offered at a single university. The students comprising the groups had comparable backgrounds and levels of prior programming proficiency. Within a cohort, individual students may have had greater proficiency than others, but this was true in equal measure for each group. As such, the research assumed that the average levels of initial programming proficiency for each student cohort were similar. This assumption was supported by the fact that the admissions criteria applied for the particular programs of study were the same for the duration of the study and the admission qualifications profiles of the cohorts were very similar. No student was in more than one cohort.

4.6. Module Content and Organization

4.6.1. Organization. The organization of the module was one of the fixed parameters of the experimental design. The module consisted of weekly lectures (duration: 1 hour), laboratory sessions (duration: 2 hours). While the whole cohort attended the same lecture session, for the laboratory sessions the cohort was split into three smaller groups. Each week, the lecture preceded the laboratory session and introduced the topic/concept. During the laboratory session, students were asked to complete self-paced tutorials. These tutorials provided short explanations of the concepts and instructions on how to implement, run, and debug simple programs based on the concepts. Depending on the cohort, students used BlueJ or Python IDLE as the interactive development environment (see Sections 4.2–4.4). The tutorials also included exercises—simple and more challenging tasks—without solutions. Depending on the cohort, students had the opportunity to submit their solutions and receive feedback. Students also used the university e-learning environment, which provided access to resources such as lecture notes, the laboratory tutorials, discussion forums, and additional learning materials.

4.6.2. Content. Table IV shows the module content and teaching sequence followed. The topics, the sequence, and the depth of coverage of topics were kept constant across the groups. As can be seen from Table IV, seven topics were taught over a period of 10 weeks. Then, in week 11, students took a formal assessment (as explained in Section 4.7).

It should be noted that the implementation strategy followed in all groups corresponded to an "objects-later" approach; that is, fundamental programming constructs were taught without any explicit reference to object-oriented programming, or to language-specific elements (such as types and access modifiers). While object-oriented programming aspects may "creep in" in even simple programs in Java, students were not expected to understand or produce any such element.

In week 12, all groups were formally introduced to object-oriented principles, but their curriculum was no longer comparable given that the three "Python" groups (G1–G3) were being exposed to Java for the first time at this point.

4.6.3. The Teaching Staff. The teaching team consisted of a module leader, two supporting lecturers, and six graduate teaching assistants. The module leader delivered all lectures covering the seven fundamental topics during the first 10 weeks (with the supporting lecturers being involved later in the term). The module leader and

supporting lecturers ran and coordinated the laboratory sessions, and along with the graduate teaching assistants, they provided one-to-one help to students. The module leader and supporting lecturers were the same for every cohort, with G0 being the first cohort taught by the team. Not all of the graduate teaching assistants remained until the end of the 4-year study. Finally, only the module leader was involved in all of the marking/grading to ensure consistency.

4.7. Assessment and Marking

Students were assessed on their ability to implement programs. A laboratory test was chosen as the assessment method, as it has been found to be an accurate assessor of programming ability [Chamillard and Braun 2000; Califf and Goodwin 2002; Bennedsen and Carpensen 2007], and superior to written exams and assignments [Daly and Waldron 2004]. During the test, students had one hour to complete the set task and they could use the same environment as the one used during their laboratory tutorials (BlueJ or Python IDLE, depending on the group). As such, for the duration of the test the students were required to implement, run, and debug their programs. The students submitted their programs electronically. The students had access to both the standard Java/Python documentation and to module resources (lecture notes, laboratory examples, etc.). The tasks were kept as similar as possible in each cohort assessment.

The test covered the basic programming language aspects addressed during the previous 10 weeks (see Table IV). The aim of the assessment was to evaluate the ability of the students to translate the task given into a program that executed successfully. Moreover, students were tested on their ability to apply basic concepts such as loops, conditionals, etc. The test consisted of a set of subtasks, with some subtasks targeting a specific basic concept and others targeting the ability to combine some of these basic concepts. For example, a task such as "*print the string 'Hello World' 20 times*" would have been used to evaluate the students' understanding of applying a loop, while a task such as "*generate two random numbers Y and X and print the value of Y if* $Y < X$" would have been used to evaluate their ability to combine two basic concepts together (importing and using libraries, and understanding of conditionals).

Within the 60-minute time limit, possible submissions by the students for the first task ("*print the string 'Hello World' 20 times*") included the following: no code implemented for this task, the code included 20 print statements, a loop that repeated the print statement 20 times. For the second example ('*generate two random numbers Y and X and print the value of Y if* $Y < X$'), possible solutions included initiating Y and X with the student's own values instead of random values, generating the random values but printing all values of Y, instead of when $Y < X$, and so forth.

4.7.1. Grading. Each task in the laboratory test was simple enough to have a "perfect solution." As noted previously, this solution would require the use of one of the taught concepts or a combination of them. Full marks were awarded if a student submitted a program which reflected the perfect solution. Half marks were awarded if a student submitted a program which produced the correct output, but without using the anticipated concept. For example, if the task was "*print the string 'Hello World' 20 times*," a solution which included a loop that repeated the print statement 20 times would receive full marks, while a solution that contained 20 print statements would receive half marks. The grading process involved two stages. First, an autograder filtered out the programs that did not run (because of runtime or compile-time errors) and the programs that gave incorrect output (semantic errors). Then, a human marker (the module leader) examined the programs that ran and gave correct output and awarded marks based on the criteria set out earlier.

4.8. Evaluation Metrics/Learning Indicators

As explained at the start of Section 4, the research paradigm employed in this study involved the collection of "hard," objective data to evaluate a particular intervention and test the underlying hypothesis. While the literature review presented in Section 2 offered only a "bird's-eye" view of the field of computing education, it revealed that researchers have used a variety of metrics to evaluate the impact of their interventions on learning. Among the evaluation metrics employed in related studies are assessment results (of examinations and coursework assignments), final grades, and dropout rates. There are some limitations with these metrics that discouraged their use in this study. In particular, if viewed in isolation, results and grades appear too coarse and reveal little about qualitative elements of the programs produced by students, and which of the taught concepts were acquired and which were not. Second, as assessment content and methods differ between institutions, meaningful comparisons using test results are not possible. Third, dropout rates are influenced by many factors in addition to, or irrespective of, the CS1 course [Nikula et al. 2011]. Finally, data on final grades and dropout rates are collected after the completion of a CS1 course, which means that learning problems are diagnosed when it is already too late for a particular cohort. In effect, we argue that evaluation metrics for teaching interventions should be reframed to be "indicators of learning"; this could shift the focus from the teaching intervention—the process—to learning, which is, after all, the end point of interest. Taken together, the study sought to identify metrics that could indicate that a concept taught was also understood; correspond to concepts that are universally taught to students at this level and, as such, could be applied by different institutions to measure performance; could be derived using minimal resource requirements; and could be also used remedially.

4.8.1. Fundamental Programming Concepts: Looping, Conditionals, Libraries. These characteristics guided the decision to look at whether students were able to appropriately use the most important concepts taught in the course *up to that point*. These concepts were looping, conditionals, and library use. The additional advantage of selecting these concepts was that their building blocks are discrete keywords (looping: for, while, and do; conditionals: if, case, ?: operator; library use: import), which can be located and counted in a program without human intervention and effort. Therefore, the frequencies of appropriate use of these constructs in the assessment were used as indicators. Simply put, a task in the laboratory test was constructed such that its perfect solution would include the use of particular concept(s) taught (for instance, a loop and/or a conditional). If the student used the associated construct ("if," "for," "while," etc.) in his/her solution, this could be indicative of the student understanding the concept and when it should be used. This type of indicator was independent of syntactic elements and errors. It should be noted that the exam tasks were simple enough to have model answers, and if a model answer required one type of construct (e.g., one conditional), but the submitted program contained more than one, this would not increase the count.

4.8.2. Bugs. Bugs were the second type of indicator used in this study. Bugs refer to logic, compile-time, and runtime errors. However, given the level of the students, the content of the 10 weeks of instruction, and of the exam content, the bugs present in student submissions were largely limited to compile-time errors. So, in effect, this metric solely consisted of syntax errors. This resonates with previous research findings that indicate that the majority of student errors in CS1 are compile-time errors, and, in particular, syntax errors. The analysis by Jadud [2005] found that the three most common student errors are, in fact, rather "trivial": semicolons, typographic errors in variable names, and bracketing (accounting for 42% of errors found in students' work). The study also reported that these mistakes are made repeatedly.

Syntax errors are an important area of programming pedagogy research. Experienced programmers rarely make syntax errors, and when they do, they have clear strategies to correct them very quickly. However, syntax errors are significant for novice programmers [Sleeman et al. 1988; Kummerfeld and Kay 2003]; correcting them is a time-consuming process and often leads to random debugging behavior, also influenced by the fact that students do not understand compiler messages [Kummerfeld and Kay 2003]. It is recognized that logical and runtime errors are much deeper, more important, and difficult to correct. However, a novice programmer has to get past compiler errors first.

As suggested previously, failure to use the correct construct for a task—such as a loop to solve a loop problem—is indicative of inadequate knowledge/learning, but this is not the only type of error based on inadequate knowledge/learning. Kummerfeld and Kay [2003] indicated that the ability to efficiently fix syntax errors necessitates knowledge of the programming language in order to understand error messages. Taken together, it is argued that failure to correct simple syntax errors is also indicative of inadequate knowledge/learning, based on the basic assumption that a competent student should be able to locate and correct simple errors within a time limit.

While the number of syntax errors in a program is a valuable indicator, its use as a metric is problematic when comparing the effectiveness and suitability of teaching languages since some languages, like Java and C++, have more elaborate syntax. As such, it is argued that syntax errors as a metric of programming and debugging ability should not be used in isolation, but in conjunction with other metrics, like the ones proposed earlier in this section.

While arguing these four metrics are useful and appropriate for the study, the article is not suggesting that this is an exhaustive set with higher intrinsic value than any others. Rather, for the specific CS1 content of the 10 weeks, which was the basis of comparison of the 4 iterations/years, these metrics were considered adequately consistent, robust, and objective to assess learning and level of understanding of programming of CS1 students to that point in their degree programs. Assessing the ability of CS2 or CS1 term, two students would have demanded a different set of metrics; most probably, a larger number of, and more sophisticated and fine-grained metrics to correspond to students' more advanced knowledge.

The study by McCracken et al. [2001] served as an inspiration to this study not only because it demonstrated the extent of the problem, but also because it did so using a robust and clear methodological approach. The study aimed to evaluate the programming ability of CS2 or CS3 students in several universities in different countries (which obviously used different teaching techniques and curricula). In order to be able to meaningfully compare the outcomes of these cohorts, a framework of general evaluation criteria was developed.

While the adoption of existing validated and reliable frameworks should be preferred over "home-grown" ones, the use of the criteria of McCracken et al. was not possible. In the study presented in this article, there was relatively little variation within the compared groups (as described in Section 4.5). Moreover, having been created for CS2 and CS3 students, the dimensions of the metrics developed by McCracken et al. [2001] could not be readily applied to CS1 programmers. In addition, the study by McCracken et al. developed and applied two subjective evaluation measures (a rating scale for tutors and a question of "perceived difficulty" for students). However, as mentioned previously, a methodological decision was taken such that the scope and focus of the present study would be on "hard," objective data. As such, at this stage of the work, all subjective evaluation data were purposely excluded. At the same time, as argued in Section 7, there is immense value in mixed methods as their use can reinforce the strengths of quantitative and qualitative approaches while minimizing their respective weaknesses.

Table V. Observed Frequencies Summary
Columns show the frequencies of programs containing a key indicator. Rows show the measured values for each cohort. The last column indicates the total size of each cohort.

Group	Loop	Conditional	Import	Bugs	Total Cohort Size
G0	27	55	56	90	157
G1	106	77	100	64	195
G2	113	92	117	60	193
G3	148	157	146	36	216

Table VI. Group Comparisons
The groups compared for testing each hypothesis are indicated.

Hypotheses	Groups compared
Ha	G0 vs. G1, G2, G3
Ha0_1	G0 vs. G1
Ha1_2	G1 vs. G2
Ha2_3	G2 vs. G3

Table VII. χ-squared Analysis Summary
This table shows the p values obtained by the separate χ-squared analyses performed for each key indicator to test each hypothesis. The last column indicates the overall rejection decision for the hypothesis at a 0.05 significance level. A hypothesis is accepted if the majority of the values for a key indicator are below the significance level.

Hypothesis	Loop	Conditional	Import	Bugs	Result
Ha	<0.001	<0.001	<0.001	<0.001	Supported
Ha0_1	<0.001	0.478	0.005	<0.001	Supported
Ha1_2	0.431	0.117	0.071	0.82	Not supported
Ha2_3	0.046	<0.001	0.172	<0.001	Supported

5. RESULTS

Table V shows a summary of the measurements for each key indicator relative to each cohort. In particular, the cells in the table contain the number of programs that had one (or more) bug, loop, conditional, and import statements. The values in the last column indicate the sample size of each cohort.

To validate each hypothesis, the differences in the frequencies observed for different groups/iterations were calculated. Table VI outlines which groups in the iteration were compared in order to address each hypothesis.

To statistically validate our results, χ-squared tests were conducted at a 0.05 significance level. R [R Development Core Team 2010] was used to carry out these statistical tests. It must be noted that the simultaneous occurrence of different key indicators could be observed for every participant (e.g., when the program submitted contained a loop and a conditional). Therefore, key indicators are not mutually exclusive. A solution is to calculate the χ squared for each indicator separately [Agresti and Liu 1999]. A hypothesis is then seen as accepted if the majority of the p values for a key indicator are below the significance level. Table VII shows a summary of the results.

The results suggest that programming proficiency (as measured by the presence of bugs and use of key programming concepts such as loops, conditionals, and importing libraries) depends on the module structure (Ha supported). Moreover, both hypotheses Ha0_1 and Ha2_3 were confirmed, while the analysis failed to support Ha1_2. Therefore, the results suggest that the use of a simple programming language such as Python

Table VIII. Calculated Percentage Differences
Columns show the percentage difference for each key indicator using the observed frequency values from Table V. Data from Table V was normalized by turning each frequency value into a percentage of the total.

Groups	Loop (%)	Conditional (%)	Import (%)	Bugs (%)
G1-G0	37	4	15	−24
G3-G1	14	33	16	−16

Table IX. Grade Distribution of All Groups

		Grade				
		A	B	C	D	E/F
Group	*G0*	7 *4.5%*	15 *9.6%*	22 *14.0%*	56 *35.7%*	57 *36.3%*
	G1	5 *2.6%*	18 *9.3%*	29 *14.9%*	78 *40.2%*	64 *33.0%*
	G2	14 *7.1%*	23 *11.7%*	50 *25.5%*	78 *39.8%*	31 *15.8%*
	G3	25 *11.6%*	34 *15.7%*	66 *30.6%*	68 *31.5%*	23 *10.6%*

can have a significant impact on novices learning how to program. Also, while formative feedback did not provide an observable benefit, problem-solving training resulted in an improvement of programming ability.

However, the frequencies of each key indicator in Table V indicate a more intricate pattern of differences; it appears that the magnitude of the effect of each module change is different for each key indicator. These differences generate richer questions with regard to which module changes influence and bring a greater benefit to the use and understanding of a particular programming concept. Thus, further analysis was undertaken in order to tease apart the individual effects on the learning of programming concepts and to understand the underlying reasons behind these differences.

The analysis involved the calculation of the differences between the frequency values of each key indicator between two groups. This analysis aimed to elucidate the individual effects on each key concept of (i) programming language and (ii) problem solving. Because the module change implemented in G2 (formative feedback) was not found to produce a significant effect, only groups G0, G1, and G3 were considered in this analysis. The differences in the frequencies of each key indicator between these groups (G0 vs. G1 and G1 vs. G3) were calculated. Each observed frequency from Table V was normalized by turning the value into a percentage of the total. Table VIII shows the percentage difference for each key indicator using the observed frequency values from Table V.

Table VIII suggests that the impact of the programming language used is not the same for each key indicator. In particular, when Python was introduced, replacing Java as the teaching language, variations in the frequencies related to the use of loops were higher than the variations related to conditionals. At the same time, when problem-solving training was introduced, the difference in the use of conditionals was the most pronounced.

To provide a complete picture of the cohorts, the distribution of final grades achieved by the students in each group is presented in Table IX. χ-square analysis yielded a significant association ($\chi^2_{(12)} = 77.862$, $p < 0.001$), and the data appear to reiterate the pattern of results of the analysis presented previously. The final grades reveal a steady improvement in students' overall performance. As mentioned in Section 4, the research project was initiated because of high failure rates (36.6% for G0). With

this figure dropping to 10.6% (for G3) during the last iteration, it is assumed that the interventions produced a clear positive outcome.

Final grades have been traditionally used in CSE research to measure the success of an intervention. However, this study proposed and applied an alternative combination of metrics/learning indicators. The fact that the results of both analyses (of final grades and the identified set of metrics) coincide supports the validity of the proposed metrics for evaluating whether learning outcomes have been achieved, but it may also indicate that these metrics may hold predictive power for students' overall performance. In this light, the proposed metrics may also be used as monitoring mechanisms and to trigger intervention during the module.

6. DISCUSSION

There have been suggestions that higher education institutions appear to be unable to cope with increasing industry demands for graduates with programming skills. This has been attributed to students' dissatisfaction with, and failure rates in, programming modules, which are pronounced in, and after, their first year, and to a lack of adequate programming skills even after graduation. As such, universities are impelled to rethink their introductory programming curricula. In order to undertake such changes on the basis of a sound evidence base, the study reported in this article aimed to undertake rigorous research and provide robust evidence to underpin recommendations with regard to content and methods for an introductory programming (CS1) module. This section discusses the findings of the study in light of related literature and distills them into recommendations for the design of CS1 curricula.

6.1. Research and Practice in CSE

The study reported in this article embraced principles of action research and design-based research methodologies, which aim to improve practice and to learn through action; planned change is implemented, monitored, and analyzed in cycles. These approaches recognize that teaching programming is a real-world situation that contains multiple dependent variables that coexist, although not all of them need to be investigated. The study reported in this article involved a process of progressive refinement undertaken in three iteration cycles; in each cycle a single change was identified, implemented, and the effects of the change were evaluated. Each proposed change was formulated as a research hypothesis, and its evaluation was performed based on statistical analysis of verifiably reliable data on measures associated with programming ability. Unlike previous small-scale studies, this research was driven by data obtained during 4 consecutive years and student cohorts (with 761 participants in total).

By adopting this approach, the study aimed to provide valid conclusions that may help CS1 curriculum developers in different educational settings. It also attempted to provide observations and guidelines that may be implemented as part of any teaching paradigm adopted in an institution (functional, objects-oriented, or imperative). Finally, even if the conclusions are not in line with an institution's strategies, this study serves to highlight the importance of approaching the issue of teaching introductory programming as one would any other research problem; that is, it should be built on careful consideration of published findings, and suitable, explicit and rigorous experimentation and evaluation.

Much of the controversy in the field of education research concerns the issue of generalizability. As argued previously, approaches based on quantitative and reliable measurement, large samples, and valid statistical analysis, like the one reported in this article, tend to be less susceptible to such criticism. Still, quantitative studies are associated with many limitations and misconceptions. In relation to educational research, it may be problematic that quantitative studies do not seek to control or

interpret all variables that operate in the real-world educational setting being analyzed, which may confound the success or contribution of a particular intervention [Denzin and Lincoln 1998]. Like qualitative approaches, the involvement of the experimenter in the practice may also introduce bias. As such, it is important that researchers do not overinterpret results derived from quantitative analysis, taking them at face value. Instead, such findings should be triangulated with existing knowledge and, where possible, complemented by qualitative methods. Finally, studies need to present clear and sufficient information regarding methods, outcomes, assumptions, and situational parameters, in order for educators to assess the applicability of the observations to their own circumstances and for future research to reliably review, compare, and replicate them.

6.2. The Benefit of a Syntactically Simple Language

The analysis presented in this article first confirmed the basic premise of this research: The choice of language has an impact on the development of programming skills of novices. Moreover, the analysis compared Java and Python, and revealed that Python facilitated students' learning of the fundamental programming concepts and structures.

In effect, this article makes the case for Python, a syntactically simple language, which, at the same time, is not a purely educational language, since it is increasingly being used in real-world applications. Python offers the possibility to write and run programs without the notational overhead imposed by Java, because of the straightforward syntax and development environment. The aim of an introductory programming course is not to teach a language *per se*, but to teach the basic concepts of programming, improve algorithmic thinking, and prepare students for the remainder of their studies. As such, by no means do we argue against the value and necessity of teaching Java, but we maintain that Python is more suited for an introductory programming module and can provide the necessary foundations for students from which they can move on to Java (or a similarly complex language) in the second term or year, being better equipped and confident.

Among the objective metrics used to assess learning were the frequencies of use of loop and conditional constructs. The analysis showed that the changes implemented in the iterations resulted in effects of different magnitude on the use of these constructs. In particular, when Python replaced Java as the language of instruction, the use of loop structures increased more dramatically than the use of conditional statements. One possible explanation for this phenomenon is that loops are problematic because novices often fail to understand that "behind the scenes" the loop control variable is being updated [Du Boulay 1986]. Therefore, a simpler syntax could have a greater impact because it makes loops easier to use, and the underlying concept easier to understand.

It is perhaps to be expected that a more syntactically complex language will result in novice programmers making more syntax errors. However, it is only recently that empirical studies have begun to investigate the extent of the impact. Denny et al. [2011] found that syntax presents a major barrier for novice learners; in a large-scale experiment, students with excellent final grades submitted noncompiling programs almost 50% of the time, even in simple exercises, while low performing students submitted noncompiling code 73% of the time. As the authors state, it appears that syntax is a greater challenge for all novice students than anticipated. The same study found that there is a negative correlation between syntax error frequency and perceptions and attitudes about programming, while suggesting that spending excessive amounts of time on correcting syntax errors hinders learners. Denny et al. [2012] also explored the idea that "not all syntax errors are equal." Indeed, while all students make the same

types of syntax error, top students can quickly fix missing semicolons (the most common syntax error), while less able students take twice as long to correct the problem. However, students of all abilities find it equally difficult to correct the second and third most common syntax errors. So, if, indeed, novice programmers largely make syntax errors which they then repeat or are unable to resolve within a specified time, it may be argued that a language which is more likely to lead to syntax errors may be unsuitable as a teaching language for novice programmers.

Taken together, the recommendation derived from these findings is that Python is an effective language for teaching introductory programming in the first term or year of a CS degree, since it enables solid acquisition of fundamental concepts and constructs and debugging skills, and, as such, it can be integrated within the paradigm of choice of an institution (e.g., functional, object-oriented, or imperative paradigms).

6.3. Formative Feedback for Improving Programming Skills

In the second iteration, students had the possibility to submit their weekly work and receive feedback on successful and erroneous aspects of their code, as well as pointers to help them address any errors. It was predicted that this group of students would outperform the previous cohort. However, the analysis failed to confirm the related hypothesis. Similar results were reported by a 3-year study exploring the effect of formative feedback using the exam scores of CS students, with the author concluding that there was little or no correlation between the provision of formative feedback and student achievement [Irons 2010].

There are two possible explanations for the absence of the effect of formative feedback in our study: the first stems from students and the broader context, and the second relates to the quality of feedback. First, the participation in the formative feedback process was lower than expected and this could explain the lack of significant improvements in student learning for the G2 cohort. This may be related to the fact that young students appear to be less keen to take advantage of opportunities unless there is a tangible benefit associated with the task. Indeed, several studies have found that most undergraduate CS students are "externally motivated," that is, they are driven to work in order to perform well in summative assessments that contribute to their final grades and degree outcome rather than from a "thirst to develop knowledge" [Carter and Boyle 2002, p. 1; Bostock 2004]. Additional reasons for the low engagement of students with the feedback process may include fear of participation, seeing feedback as a bad sign, and negative attitudes toward learning [Black 1999]. Research has also indicated that students may find feedback difficult to understand [Lillis and Turner 2001], or not even read it [Ecclestone 1998]. As such, it is argued that simply offering feedback has limited value; in order for students to make the most of it, a broader change in their motivations and perceptions should take place [Black 1999], while Rust [2002] also recommends that students should be required (not encouraged) to actively engage with it. In addition to providing feedback, then, structured opportunities should be provided to students to understand, reflect on, and discuss the feedback with tutors.

A second explanation may lie in the nature of the feedback provided in this study. Research by Corbett and Anderson [1990, 2001] shows that results are mixed with regards to whether introductory programming students benefit from feedback; for instance, parameters that determine whether feedback leads to improvement have to do with whether it is provided on demand, automatically, at the end of each line, or the program, consists of goal hints or explanations, and so forth. These parameters may also interact with the proficiency level and learning style of students. Indeed, such observations give rise to richer research questions, with regard to feedback parameters that can provide a true advantage to learners of programming.

In light of the findings of this study and those of previous literature, it is advised that for formative feedback to yield observable benefits to their performance, novice students of programming may need to be externally motivated and guided. Moreover, the amount, nature, and timing of feedback should be fine-tuned to the particular characteristics of the task and students in order to be effective and worthwhile.

6.4. The Benefit of Problem-Solving Training Before Programming

In the third iteration of this study, the week before the formal start of the module focused on problem-solving activities that introduced algorithmic concepts and constructs, independent of a particular programming language. The analysis compared the performance outcomes of the groups in the two relevant iterations (G3 and G2; with and without problem-solving training). The results confirmed that problem solving yields an actual, quantifiable benefit in the programming ability of students by the end of the 10 weeks. A possible explanation is that novice programmers normally tend to begin coding without considering and analyzing the problem. So, it may be the case that through even a brief exposure to problem-solving procedures before programming sessions began, students were able to understand that an exercise is a problem that can be solved if decomposed into steps that can be then translated into lines of code. In addition, having already gained some experience with algorithms and concepts relating to data and control may subsequently have facilitated the students' easier and more rapid progress through the remainder of the introductory programming course. This inference is also supported by the finding that the use of conditionals was mostly facilitated by problem-solving training. In particular, the analysis revealed that the benefit in the use of conditionals was more pronounced compared to the use of loops. Thus, it appears that variations in the use of conditionals may be less associated with programming language complexity and more associated with abstract thinking skills. This implies that beginners need to acquire logical and mathematical structures in order to be more proficient in the use of conditional statements, a point also noted by Rogalski and Samurçay [1990]. This seems to be achieved when developing the students' problem-solving skills before programming is addressed, suggesting that this ordering should be followed in the design of the module.

Preoccupation with syntactic detail and the executable performance requirements may detract from students thinking of the algorithm and, then, mentally following its execution. At the same time, most introductory programming courses, at least in the beginning, involve simple tasks with straightforward solutions, with no weight given to design and analysis, which gives students a false impression of the programming process [CC2001 2001]. The link between problem-solving and programming skills is well known [Tu and Johnson 1990]. The practice to cover problem-solving concepts at the beginning of an introductory programming course has been recommended by CC2001 [2001] and evidence of the approach's value has been demonstrated by Turner and Hill [2007], who adopted it and reported positive student perceptions.

Palumbo [1990] discussed the "reverse" causal relationship of programming and problem solving; that is, whether learning programming improves problem-solving skills. His conclusions, however, may be viewed from both sides. In particular, he suggested that the development of problem-solving skills could be facilitated if, after learning the syntax of the language, the student learns how to (i) break the program into subprograms, (ii) use iteration, and (iii) use conditions to solve the programming problem. From these points, it could be inferred that if, indeed, problem-solving training is linked to iteration and conditions, it is sensible to suggest that even limited duration training in problem solving can help students to grasp the particular fundamental concepts that students were taught in this course.

In conclusion, it is recommended that novice learners of programming undertake a brief problem-solving training course before the beginning of programming sessions. This approach helps students to develop a basic understanding of analysis and design, algorithmic, and problem-solving concepts without relating them to a particular executable language. Having a conceptual foundation facilitates and accelerates the transfer to an executable programming context within a functional, imperative, or object-oriented paradigm.

7. CONCLUSIONS AND FUTURE WORK

How to approach the teaching of introductory programming is a multidimensional problem and has given rise to enduring debates. The outcome of the investigation reported in this article has contributed to this body of literature through findings derived from a rigorous and relatively large scale, longitudinal research study which offers benefits over many of the existing studies in the field. First, the article has introduced the argument that some objective measurements should be considered to assess the effects of the strategies adopted. Second, the reported study integrated principles of experimental and applied research methodologies, arguing that to evaluate effectively the effects of any didactic change in the module structure, it is crucial to rigorously control the process in a naturalistic setting. This led to iteration through a cycle of a single change being implemented, then the effects being observed, and to the next change being defined based on these objective observations, before repeating the cycle. Third, quantitative evidence has been produced from these cycles of data collection and analyzed to frame recommendations for the design of CS1 curricula. The article further argues that, by approaching questions of what and how to teach in this context primarily as "research problems," the resulting findings of such research can be generalized and the recommendations used by educators across different settings.

From the iterative process adopted, the following outcomes emerged. First, the choice of programming language seems to affect student learning. This may be because some programming languages are too complex (in terms of syntax, semantics, etc.) for beginners and may distract them from learning basic programming concepts, which may, in turn, have a lasting impact on students' confidence in, and perception of, their programming abilities. Second, introducing problem-solving concepts before teaching more specific programming aspects has an impact on how students learn to program. This strategy could help students realize that it is essential to develop an ability to both break down complex problems into subtasks and produce the correct sequence of actions and accelerate consolidation of concepts, such as data and control structures, introduced later in the course. From this perspective, programming problems can then be solved as a systematic implementation of tackling one subtask at a time and backtracking afterwards. It could also be argued that the positive effect of early exposure to problem solving on programming ability arises independently of the language used in the programming course. However, an empirical investigation of possible interaction effects is worthwhile; that is, an interesting question is whether the benefit of problem solving is present, grows, or diminishes when different instruction languages and techniques are used later in the course. Third, formative feedback may not be necessarily as effective as expected unless students are ready to have a proactive role in seeking and responding to feedback. In the context of introductory programming, effective formative feedback should have particular characteristics in terms of timing and targeting specific features of the code. Identifying the parameters of effective feedback is an interesting future research challenge. The findings of this study have also identified other potential avenues for further research, outlined in the following.

The study does not argue that the proposed three interventions constitute the only possible set, or even a superior set compared to other candidate interventions. Research

in, and development of, a course is a long-term endeavour. Interventions should be continuously planned and tested until as many aspects of the problematic space as possible have been addressed and the course improved in a sustainable way. With the field of CSE maturing, educators are gaining access to a growing knowledge base of solid findings to guide them through making judicious decisions regarding teaching interventions. For example, there is robust evidence that pair programming and related cooperative learning instructional techniques lead to improvements in students' performance, self-efficacy, and perceptions [Braught et al. 2011; Beck and Chizhik 2013]. At the same time, well-documented previous efforts give educators the resources, as well as the confidence, to apply "nontraditional" pedagogical approaches. These include the media computation approach, in which students learn how to program through the manipulation of visual and audio media [Sloan and Troy 2008; Guzdial 2009]; and peer instruction, which relies on students working in small groups [Lee 2013; Simon et al. 2010; Porter et al. 2011]. Originating from non-CS classroom environments, these approaches have been found to increase achievement, attainment, and motivation in CS1 students.

The data in this study were obtained from a (relatively short) teaching cycle of 10 weeks. Therefore, how these findings affect the settling time of the learning curve is one aspect that merits further investigation. A next step in this area would be to explore the correlation between the choice of the programming language used in the CS1 course and the length of the transient time for the learning curve. Another interesting, related research area would be to investigate how using Python first has an impact on learning Java at a later stage (as also addressed in Mannila et al. [2006]). Finally, a follow-up study about students' performance in the second and third years of their studies could help provide an appreciation of the long-term effects of curriculum changes introduced in the CS1 course.

This article has argued for the importance of choices and practices informed by reliable analysis of experimental data. Richer insight and deeper understanding could, though, be gained if objective measurements were inspected in light of findings from analysis of qualitative data, such as video recordings, interviews, and verbal protocols [Renumol et al. 2010].

There are also areas of further study associated with extending the variables that are considered. For example, human factors in the student population, such as gender and ethnicity, were not considered in the reported study. However, gender and ethnic background are consistently found to correlate strongly with attrition rates of first-year students in computing, with female and minority-ethnic students being particularly vulnerable [Talton et al. 2006; Rosson et al. 2011]. Therefore, it is planned to further develop this study by analyzing the impact of gender and ethnic background on the strategies adopted to teach programming to beginners. Through this investigation, it may be possible to identify the teaching methods and curriculum changes that equally support and are suitable for all students, without adversely affecting a particular demographic group. For example, it has been consistently found that pair programming greatly benefits female programmers while not impairing the performance of male students [Berenson et al. 2004; McDowell et al. 2003].

REFERENCES

A. Agresti and I. M. Liu. 1999. Modeling a categorical variable allowing arbitrarily many category choices. *Biometrics*, 55, 3 (Sept. 1999), 936–943.

V. L. Almstrum, O. Hazzan, M. Guzdial, and M. Petre. 2005, Challenges to computer science education research. *ACM SIGCSE Bull.* 37, 1 (2005), 191–192.

T. Beaubouef and J. Mason. 2005. Why the high attrition rate for computer science students: Some thoughts and observations. *ACM SIGCSE Bull.* 37, 2 (2005), 103–106.

L. Beck and A. Chizhik. 2013. Cooperative learning instructional methods for CS1: Design, implementation, and evaluation. *ACM Trans. Comput. Educ. (TOCE)* 13, 3 (Aug. 2013), Article 10.

J. Bennedsen and M. E. Caspersen. 2007. Assessing process and product: A practical lab exam for an introductory programming course 1. *Inn. Teaching Learning Inf. Comput. Sci.* 6, 4 (Oct. 2007), 183–202.

S. B. Berenson, K. M. Slaten, L. Williams, and C. W. Ho. 2004. Voices of women in a software engineering course: Reflections on collaboration. *J. Educ. Res. Comput. (JERIC)*, 4, 1 (March 2004), Article 3.

P. Black. 1999. Assessment, learning theories and testing systems. In *Learners, Learning and Assessment*. Paul Chapman Publishing, London, 118–134.

P. Black and D. Wiliam. 1998. Assessment and classroom learning. *Assess. Educ.* 5, 1 (1998), 7–74.

S. J. Bostock. 2004. Motivation and electronic assessment. *Effective Learning and Teaching in Computing*. Routledge Falmer, London. 86–99.

G. Braught, T. Wahls, and L. M. Eby. 2011. The case for pair programming in the computer science classroom. *ACM Trans. Comput. Educ. (TOCE)*, 11, 1, Article 2 (Feb. 2011).

A. L. Brown. 1992. Design experiments: Theoretical and methodological challenges in creating complex interventions in classroom settings. *J. Learn. Sci.* 2, 2 (1992), 141–178.

M. E. Califf and M. Goodwin. 2002. Testing skills and knowledge: Introducing a laboratory exam in CS1. *ACM SIGCSE Bull.* 34, 1 (Feb. 2002), 217–221.

J. Carter and R. Boyle. 2002. Teaching delivery issues: Lessons from computer science. *J. Inf. Technol.* 1, 2 (2002), 65–90.

A. T. Chamillard and K. A. Braun. 2000. Evaluating programming ability in an introductory computer science course. *ACM SIGCSE Bull.* 32, 1 (March 2000), 212–216.

A. Collins. 1992. *Toward a design science of education*. Springer, Berlin, 15–22.

A. Collins, D. Joseph, and K. Bielaczyc. 2004. Design research: Theoretical and methodological issues. *J. Learn. Sci.* 13, 1 (2004), 15–42.

Computing Curricula. 2001. *IEEE CS, ACM Joint Task Force on Computing Curricula*. IEEE Computer Society Press and ACM Press. Retrieved from http://www.acm.org/education/curricula.html.

Computing Curricula: The overview report. 2005. *IEEE CS, ACM Joint Task Force on Computing Curricula*. IEEE Computer Society Press and ACM Press. Retrieved from http://www.acm.org/education/curricula.html.

A. T. Corbett and J. R. Anderson. 1990. The effect of feedback control on learning to program with the LISP tutor. In *Proceedings of the Twelfth Annual Conference of the Cognitive Science Society*. 796–803.

A. T. Corbett and J. R. Anderson. 2001. Locus of feedback control in computer-based tutoring: Impact on learning rate, achievement and attitudes. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, New York, 245–252.

C. Daly and J. Waldron. 2004. Assessing the assessment of programming ability. *ACM SIGCSE Bull.* 36, 1 (March 2004), 210–213.

DARPA. 2010. Computer Science—Science, Technology, Engineering, and Mathematics (CS-STEM) Education Research Announcement (RA). DARPA-RA-10-03.

P. Denny, A. Luxton-Reilly, and E. Tempero. 2012. All syntax errors are not equal. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'12).* 75–80.

P. Denny, A. Luxton-Reilly, E. Tempero, and J. Hendrickx. 2011. Understanding the syntax barrier for novices. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education (ITiCSE'11).* ACM, New York, 208–212.

N. K. Denzin and Y. S. Lincoln, (Eds.). 1998. *The Landscape of Qualitative Research: Theories and Issues*. Sage, Thousand Oaks, CA.

B. Du Boulay. 1986. Some Difficulties of Learning to Program. *J. Educ. Comput. Res.* 2, 1 (1986), 57–73.

K. Ecclestone. 1998. "Just tell me what to do" barriers to assessment-in-learning in higher education. In *Scottish Educational Research Association Annual Conference, University of Dundee*. 25–26.

A. Ehlert and C. Schulte. 2009. Empirical comparison of objects-first and objects-later. In *Proceedings of the 5th International Workshop on Computing Education Research Workshop*. ACM Press, New York, NY, 15–26.

M. H. Goldwasser and D. Letscher. 2008. Using Python to Teach Object-Oriented Programming in CS1. *Innov. Technol. Comput. Sci. Ed.* (June 2008), 30–32.

L. Grandell, M. Peltomäki, R. J. Back, and T. Salakoski. 2006. Why complicate things?: Introducing programming in high school using Python. In *Proceedings of the 8th Australasian Conference on Computing Education—Volume 52*. Australian Computer Society, Inc., 71–80.

M. Guzdial. 2009. Education: Teaching computing to everyone. *Commun. ACM*, 52, 5 (2009), 31–33.

R. Higgins, P. Hartley, and A. Skelton. 2002. The conscientious consumer: Reconsidering the role of assessment feedback in student learning. *Studies Higher Educ.* 27, 1 (2002), 53–64.

A. Irons. 2010. *An Investigation into the Impact of Formative Feedback on the Student Learning Experience* Doctoral dissertation. Durham University, Durham, UK.

M. C. Jadud. 2005. A first look at novice compilation behaviour using BlueJ. *Comput. Sci. Educ.* 15, 1 (2005), 25–40.

J. Kasurinen and U. Nikula. 2007. Lower dropout rates and better grades through revised course infrastructure. In *Proceedings of the 10th IASTED International Conference on Computers and Advanced Technology in Education*. ACTA Press, Calgary, Canada. 152–157.

M. Kölling. 1999. The problem of teaching object-oriented programming, Part 1: Languages. *J. Obj. Orient. Prog.* 11, 8, 8–15.

V. Koshy. 2009. *Action Research for Improving Educational Practice: A Step-by-Step Guide*. Sage, Thousand Oaks, CA.

S. K. Kummerfeld and J. Kay. 2003. The neglected battle fields of syntax errors. In *Proceedings of the Fifth Australasian Conference on Computing Education—Vol. 20*. Australian Computer Society, Inc., 105–111.

C. B. Lee. 2013. Experience report: CS1 in MATLAB for non-majors, with media computation and peer instruction. In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education*. ACM Press, New York, NY, 35–40.

M. Lilley and T. Barker. 2007. Students' perceived usefulness of formative feedback for a computer-adaptive test. *Electron. J. e-learning* 5 (2007), 31–38.

T. Lillis and J. Turner. 2001. Student writing in higher education: Contemporary confusion, traditional concerns. *Teaching Higher Educ.* 6, 1 (2001), 57–68.

M. C. Linn and J. Dalbey. 1989. Cognitive consequences of programming instruction. In *Studying the Novice Programmer*, E. Soloway and J. C. Spohrer (Eds.). Lawrence Erlbaum Associates, Hillsdale, NJ, 57–81.

R. Lister, E. S. Adams, S. Fitzgerald, W. Fone, J. Hamer, M. Lindholm, and L. Thomas. 2004. A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bull*. 36, 4 (2007), 119–150.

L. Ma, J. Ferguson, M. Roper, and M. Wood. 2007. Investigating the viability of mental models held by novice programmers. *ACM SIGCSE Bull.* 39, 1 (March 2007), 499–503.

L. Mannila and M. de Raadt. 2006. An objective comparison of languages for teaching introductory programming. In *Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling 2006*. ACM, New York, 32–37.

L. Mannila, M. Peltomki, and T. Salakoski. 2006. What about a simple language? Analyzing the difficulties in learning to program. *Comput. Sci. Educ.* 16, 3 (2006), 211–227.

M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz. 2001. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bull.* 33, 4 (Dec. 2001), 125–180.

C. McDowell, L. Werner, H. E. Bullock, and J. Fernald. 2003. The impact of pair programming on student performance, perception and persistence. In *Proceedings of the 25th International Conference on Software Engineering*. IEEE Computer Society, 602–607.

A. McGettrick, R. Boyle, R. Ibbett, J. Lloyd, G. Lovegrove, and K. Mander. 2004. Grand challenges in computing: Education. British Computer Society.

B. N. Miller and D. L. Ranum. 2005. Teaching an introductory computer science sequence with Python. In *Proceedings of the 38th Midwest Instructional and Computing Symposium*.

B. Miller and D. Ranum. 2006. Freedom to succeed: A three course introductory sequence using Python and Java. *J. Comput. Sci. Colleges* 22, 1 (2006), 106–116.

M. Molina, E. Castro, and E. Castro. 2007. Teaching experiments within design research. *Int. J. Interdisc. Social Soc. Sci.* 2, 4 (2007), 435–440.

R. Moreno. 2004. Decreasing cognitive load for novice students: Effects of explanatory versus corrective feedback in discovery-based multimedia. *Instruct. Sci.* 32, 1–2 (2004), 99–113.

National Audit Office. Staying the course: The retention of students in higher education. 2007. *Report by the National Audit Office*, 44. Retrieved from http://www.nao.org.uk/report/staying-the-course-the-retention-of-students-in-higher-education/.

U. Nikula, O. Gotel, and J. Kasurinen. 2011. A motivation guided holistic rehabilitation of the first programming course. *ACM Trans. Comput. Educ. (TOCE)* 11, 4 (2011), 24.

J. D. Oldham. 2005. What happens after Python in CS1? *J. Comput. Sci. Colleges* 20, 6 (2005), 7–13.

D. B. Palumbo. 1990. Programming language/problem-solving research: A review of relevant issues. *Rev. Educ. Res.* 60, 1 (1990), 65–89.

H. Patterson-McNeill. 2006. Experience: From C++ to Python in 3 easy steps. *J. Comput. Sci. Colleges* 22, 2 (2006), 92–96.

A. Pears and L. Malmi. 2009. Values and objectives in computing education research. *ACM Trans. Comput. Educ. (TOCE)*, 9, 3 (2009), 15.

A. Pears, S. Seidman, L. Malmi, L. Mannila, E. Adams, J. Bennedsen, M. Devlin, and J. Paterson. 2007. A survey of literature on the teaching of introductory programming. *ACM SIGCSE Bull.* 39, 4 (Dec. 2007), 204–223.

G. Polya. 1973. *How to Solve It: A New Aspect of Mathematical Method*. Princeton University Press.

L. Porter, C. Bailey Lee, B. Simon, Q. Cutts, and D. Zingaro. 2011. Experience report: A multi-classroom report on the value of peer instruction. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*. ACM, New York, 138–142.

Quality Assurance Agency for Higher Education. 2003. Learning from subject review. Retrieved from http://www.qaa.ac.uk/Publications/InformationAndGuidance/Documents/learningFromSubjectReview.pdf.

R Development Core Team. 2010. R: A language and environment for statistical computing. R Foundation Statistical Computing.

A. Radenski. 2006. Python First: A lab-based digital introduction to computer science. *ACM SIGCSE Bull*. 38, 3 (2006), 197–201.

J. Randolph, G. Julnes, E. Sutinen, and S. Lehman. 2008. A methodological review of computer science education research. *J. Inf. Tech. Educ.: Res.* 7, 1 (2008), 135–162.

T. Reeves. 2011. Can educational research be both rigorous and relevant. *Educ. Des.* 1, 4 (2011), 1–24.

V. G. Renumol, D. Janakiram, and S. Jayaprakash. 2010. Identification of cognitive processes of effective and ineffective students during computer programming. *ACM Trans. Comput. Educ. (TOCE)* 10. 3 (2010), 10.

A. Robins, J. Rountree, and N. Rountree. 2003. Learning and teaching programming: A review and discussion. *Comput. Sci. Educ.* 13, 2 (2003), 137–172.

J. Rogalski and R. Samurçay. 1990. Acquisition of programming knowledge and skills. *Psych. Prog.* 18 (1990), 157–174.

M. B. Rosson, J. M. Carroll, and H. Sinha. 2011. Orientation of undergraduates toward careers in the computer and information sciences: Gender, self-efficacy and social support. *ACM Trans. Comput. Educ. (TOCE)* 11, 3 (2011), 14.

C. Rust. 2002. The impact of assessment on student learning: How can the research literature practically help to inform the development of departmental assessment strategies and learner-centred assessment practices? *Active Learn. Higher Educ.* 3, 2 (July 2002), 145–158.

C. Shannon. 2003. Another breadth-first approach to CS I using python. *ACM SIGCSE Bull*. 35, 1 (Sept. 2003), 248–251.

V. J. Shute. 2008. Focus on formative feedback. *Rev. Educ. Res.* 78, 1 (2008), 153–189.

R. M. Siegfried, D. Chays, and K. G. Herbert. 2008. Will there ever be consensus on cs1. In *Proceedings of the 2008 International Conference on Frontiers in Education: Computer Science and Computer Engineering–FECS*. Vol. 8, 18–23.

B. Simon, M. Kohanfars, J. Lee, K. Tamayo, and Q. Cutts. 2010. Experience report: Peer instruction in introductory computing. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*. ACM, New York, 341–345.

R. E. Slavin. 2003. A reader's guide to scientifically based research. *Educ. Leadership* 60, 5 (2003), 12–16.

R. H. Sloan and P. Troy. 2008. CS 0.5: A better approach to introductory computer science for majors. *ACM SIGCSE Bull.* 40, 1 (2008), 271–275.

D. Sleeman, R. T. Putnam, J. Baxter, and L. Kuspa. 1988. An introductory Pascal class: A case study of students' errors. *Teaching and Learning Computer Programming: Multiple Research Perspectives.* RE Mayer (Ed.). Lawrence Erlbaum Associates, Hillsdale, NJ, 237–257.

J. Stamey and S. Sheel. 2010. A boot camp approach to learning programming in a CS0 course. *J. Comput. Sci. Colleges* 25, 5 (2010), 34–40.

A. Stefik and S. Siebert. 2013. An empirical investigation into programming language syntax. *ACM Trans. Comput. Educ. (TOCE)* 13, 4 (2013), 19.

L. Stenhouse. 1975. *An Introduction to Curriculum Research and Development*. Heinemann, London.

J. O. Talton, D. L. Peterson, S. Kamin, D. Israel, and J. Al-Muhtadi. 2006. Scavenger hunt: Computer science retention through orientation. *ACM SIGCSE Bull*. 38, 1 (2006), 443–447.

J. J. Tu and J. R. Johnson. 1990. Can computer programming improve problem-solving ability? *ACM SIGCSE Bulletin* 22, 2 (1990), 30–33.

S. Turner and G. Hill. 2007. Robots in problem-solving and programming. In *Proceedings of the 8th Annual Conference of the Subject Centre for Information and Computer Sciences*.

U.S. Department of Education. 2001. No Child Left Behind Act. Retrieved from http://www2.ed.gov/policy/elsec/leg/esea02/index.html.

M. R. Weaver. 2006. Do students value feedback? Student perceptions of tutors' written responses. *Assess. Eval. Higher Educ.* 31, 3 (2006), 379–394.

J. M. Zelle. 1999. Python as a first language. In *Proceedings of 13th Annual Midwest Computer Conference*. Vol. 2.

Received July 2013; revised July 2014; accepted August 2014