Why Complicate Things?
Introducing Programming in High School Using Python

Linda Grandell, Mia Peltomäki, Ralph-Johan Back and Tapio Salakoski

Turku Centre for Computer Science,
Lemminkäisenkatu 14 A, FIN-20520 Turku, Finland
Email: linda.grandell@abo.fi, mia.peltomaki@utu.fi, backrj@abo.fi, tapio.salakoski@utu.fi

Abstract

Deciding what to teach novices about programming and what programming language to use is a common topic for debate. Should an industry relevant programming language be taught, or should a language designed for teaching novices be used? Typically, these questions are raised at university level, but in this paper we address them from a high school perspective.

We present a case study with a twofold goal: (1) examining how programming can be introduced at high school level, and (2) evaluating how suitable the programming language Python is to support both teachers and learners in this process. During the school year 2004/2005, an introductory programming course was given to four student groups in two different high schools. The students enjoyed programming and learnt to think in terms of re-use and interfaces. In addition, we found that many features of Python facilitated both teaching and learning (for instance, a simple and flexible syntax, immediate feedback, easy-to-use modules and strict requirements on proper indentation).

Our findings support results from previous studies in that students have difficulties in dealing with abstract concepts - even though the syntax for implementing these is simple. In addition, compared to university students, high school students are young and have necessarily not yet developed the writing skills required for producing proper documentation. The course was designed to be well suited for high school students in general, but still all participants were boys. Since high schools should provide all-round learning to all students, we, as do all computer science teachers, face the challenge of making programming more appealing to girls.

1 Introduction

Computer Science in High Schools

Subjects that have previously been considered university level topics are increasingly introduced at lower levels of education; computer science (CS) is an example of such a topic (Ben-Ari 2004a). The aim of elementary and secondary education is to provide students with knowledge needed in every-day life, and considering the increasing role of computer technology and applications in today's society, CS can be seen as an essential part of this all-round learning. Consequently, efforts have been made in order to introduce CS at high school (HS)[1] level: for instance, in the USA (Merritt et al. 1993) and in Israel (Gal-Ezer et al. 1995), CS curricula for HSs were developed in the 1990s, and according to the information network on education in Europe, *Eurydice*,[2] most European countries specified CS and programming as part of HS education in 2004. As CS education has become more common at HS level, the interest in studying this topic has increased; for example in 2004, an entire issue of the journal *Computer Science Education*[3] was devoted to this topic.

Difficulties in Learning Programming

Although there seems to be a general agreement that CS should be taught at HS level, there are many differing ideas about what exactly should be taught. The main point of disagreement is usually whether programming should be included in CS curricula at HS level or not. Many studies have pointed out difficulties experienced by university students when learning programming. Spohrer and Soloway (1986) showed that a few bug types constitute the majority of the mistakes made by novice programmers: construct-based problems, which make it difficult to "learn the correct semantics of language constructs", and plan composition problems, which make it difficult to "put plans together correctly". Recent studies support these findings (for example (McCracken et al. 2001, Ben-Ari 1998)). Moreover, the results from a multi-national study conducted in 2004 (Lister et al. 2004) showed that students lack the skills needed to trace the execution of short pieces of code after taken their first course on programming. Other reported sources of difficulty or confusion have been related to, for instance, parameter passing (Fleury 1991), analogies (Halasz & Moran 1982) and mathematical notation (for example (Haberman & Kolikant 2001, Bayman & Mayer 1983)). For a further review of this topic see, for example, Robins et al. (2003).

Programming Languages in Education

The difficulties mentioned above seem to be related to the programming activity - not the programming language. Nevertheless, deciding what, and how, to teach novice programmers is not the only topic for debate. Equally common a topic is which programming language to teach. According to Palumbo (1990), the learning results in programming classes depend on the

Copyright ©2006, Australian Computer Society, Inc. This paper appeared at Eighth Australasian Computing Education Conference (ACE2006), Hobart, Tasmania, Australia, January 2006. Conferences in Research and Practice in Information Technology, Vol. 52. Denise Tolhurst and Samuel Mann, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

[1] In the Finnish educational system, high schools are referred to as upper secondary schools, providing education to students aged 16-19. The principal objective of these schools is to offer general education preparing the students for the matriculation examination, which is a pre-requisite for enrolling for university studies. http://www.edu.fi/english/page.asp?path=500,4699,4840,4845

[2] http://www.eurydice.org/Documents/KDICT/en/FrameSet.htm

[3] http://www.tandf.co.uk/journals/titles/08993408.asp

time devoted to actual programming and language access. In order to maximize the time spent on programming, one should avoid having to "waste" time on discussing irrelevant language constructs and syntax errors.

According to Milbrandt (1993), a programming language to be used in education should be easy to learn, structured in design, universal in use and powerful in computing capability. The language should also have a simple syntax, provide easy I/O handling as well as output formatting, use meaningful keywords, give immediate feedback, etc.

Pascal and Logo were both designed with education in mind, and many studies have indicated the suitability of these in education (for example (Schollmeyer 1996, Shaffer 1986)). Unfortunately, both languages suffer from drawbacks that have led to decreasing significance over the years. For instance, Logo is seen as a language for children, and Pascal has not followed the same pace of development as more recent languages (deRaadt et al. 2002).

Today, C, Java and C++ top the list of the most popular languages based on the availability of engineers, courses and third party vendors around the world.[4] Researchers have found that Java, Visual Basic, C++ and C are the languages most widely taught (or planned to be taught) at universities (deRaadt et al. 2002, Stephenson & West 1998). These results are further confirmed when looking at web pages of CS departments at universities worldwide. Despite their popularity, there has been much debate about the suitability of these languages in education, especially when introducing programming to novices (for example (Mody 1991, Hadjerrouit 1998, Biddle & Tempero 1998, Clark et al. 1998, Close et al. 2000)). The languages are, for example, considered too verbose, enforcing notational overhead that has little to do with learning to think algorithmically and to write structured programs. All the time spent on sorting out these issues is time away from actual programming.

Given the difficulties related to learning programming at university level, the skepticism against introducing the topic at HS level is understandable; after all, if something is considered problematic by university students, it is reasonable to assume that it is equally troublesome for younger students - especially when indications of the difficulties can be found also in studies conducted among HS students (for example (Aiken 1972, Ginat 2003, Haataja et al. 2001))). Most of the problems that have been discussed in the literature have, however, been based on the traditional instructional languages mentioned above; those who take a critical stand against programming being taught at lower levels might be doing so based on their experiences in teaching these languages. In that case, the criticism can be seen as aimed towards the *languages* and not at *programming* per se.

Should HS Students Learn Programming?

Prior programming experience has been shown to have a positive impact on university CS studies (Hagan & Markham 2000). However, programming skills are not only important to students pursuing careers as computer scientists. Researchers (for example (Mayer 1986, Treese 2003, Soloway 1993)) have studied the impact of programming on the development of thinking skills, and although the results vary, indications of the connection between programming and problem solving, algorithmic thinking and logical reasoning have been found. Students who will not become programmers will still benefit from these meta skills learnt in the process. Knowing how to program also implies understanding how software and computers work, which may reduce computer anxiety and frustration. Moreover, having some knowledge of programming makes it easier to communicate with CS professionals. In addition, we feel that the aforementioned difficulties in learning programming make it even more motivated to start teaching programming at HS level; doing so, we can deal with some of the problematic issues related to learning to program at an early stage and thus improve the prior knowledge and skills of future university students. In our opinion, programming should therefore be included in CS curricula at HS level. Even though some students would not take the courses, they should be given the chance to do so.

In this paper, we present an approach to introducing programming and algorithmic thinking in HS using the programming language *Python*. Our aim is twofold: (1) examining how programming can be introduced at HS level, and (2) evaluating how Python is suited for supporting this teaching process. We begin by motivating our research, particularly by reviewing previous work relevant for our two goals. Thereafter we present the course, its goals, material and syllabus. Starting in fall 2004, the course has been given to four groups of HS students, and we present and discuss the results from these courses. We briefly discuss our own experiences before concluding the paper with some final remarks and suggestions for future work.

2 Motivation and Related Literature

Although Pascal and Logo have lost much of their significance, the same things that one earlier hoped to achieve using these are still desirable in programming education. In many respects, the aims of Python can be compared to those of Pascal and Logo, with the difference of Python being a popular language.[5]

Python is a high-level scripting language, originally designed by Guido van Rossum to facilitate learning; van Rossum has even suggested that everybody could master programming using Python (van Rossum 1999). The language has many of the features characteristic for a language suitable for teaching programming, for instance the following ones:

Small and clean syntax Compared to languages such as Java or C++, Python has a more intuitive syntax (code listing 1).

Dynamic typing Python is dynamically typed, which further reduces the notation.

Expressive semantics Python's basic types are powerful: for example, lists can be introduced at the same time as other built-in types.

Immediate feedback The interpreter enables fast and interactive demonstration of programming concepts, and gives immediate and understandable feedback on potential errors.

Enforced structural design Python enforces an indented and structured way of writing programs, and the code resembles pseudo code.

Relevant open-source software Python is free and widely used.[6] It comes with a text editor (*IDLE*), and a large amount of tutorials, books, course material, exercises, assignments and extensive documentation is available on the web.

[4] http://www.tiobe.com/tpci.htm

[5] http://www.tiobe.com/tpci.htm

[6] http://www.pythonology.org/succes

Code listing 1 *Comparison of a simple output program in Python and Java.*

```
Python        Java
print "Hi!"   class Hi {
                public static void main (String args[]) {
                  System.out.println("Hi!");
                }
              }

              % javac Hi.java
              % java Hi
```

For a more detailed review of Python's features see the official website[7] or the articles cited below. As any language, Python naturally also has features that might be drawbacks:

Performance That Python is a scripting language makes it time saving to write and execute short programs, whereas large programs might suffer from a loss in performance. This should not be relevant in introductory programming courses, in which the programs written are relatively small.

No information hiding In Python, attributes are visible by default, thus making discussion on information hiding and encapsulation more difficult. This is not an issue while not dealing with object-oriented programming.

Dynamic typing Dynamic typing was mentioned as an advantage, but it might also turn out to be a drawback. In imperative programming, the possibility of assigning different types to the same variable can be seen as a weakness that might make programs more prone to errors.

These potential drawbacks must naturally be pointed out to the students. Although Python might be a good starting language, our goal was to introduce programming and not Python per se; it was therefore important to make students aware of at least the biggest differences so that they would know what to expect/not to expect from other languages they might use.

Many indications of Python's suitability as the first programming language can be found on the web.[8] During recent years, Python has also become increasingly popular in HS settings; for instance, the *Python Bibliotheca*,[9] a site acting as both a repository for learning materials and a virtual meeting place for teachers, includes a list of HSs in the USA that use Python.

Although seemingly on the rise in education, studies on teaching introductory programming using Python are still quite few. When adding the HS perspective to this research, the number of conducted studies decreases further. Some research papers on the use of Python in education (at both university and HS level) can be found (for example (Elkner 2001, Elkner 2002, Ceder & Yergler 2003, Oldham 2005, Stajano 2000, Agarwal & Agarwal 2005, Shannon 2003)), but these have mainly discussed experiences and provided suggestions, without presenting any results. Guzdial (2003) has developed a new type of introductory programming course in which a Java related version of Python (Jython) is used. Nevertheless, although research on Python or HS programming is by no means new, the combination of these seems to be a field in which much still remains to be explored. The aim of this paper is to begin this exploration.

3 The Introductory Programming Course

Since there is no compulsory programming curriculum for HSs in Finland,[10] we began by defining course goals according to what we feel HS students having taken their first course on programming should be able to do: (1) understand and use basic programming concepts and data structures, (2) develop specifications of problems to be solved, (3) write and test programs, and (4) communicate their work to others by thorough documentation.

Syllabus

According to the Finnish educational system, HS courses should include at least 28 hours of instruction. The course covered the basics of imperative programming: variables, types, boolean logic, I/O, subroutines, flow control, exception handling, modules and documentation. In addition, we began with a rather extensive introduction to algorithms, flowcharts and pseudo code, before moving on to actual programming. This theoretical introduction was considered essential, particularly for students who had no prior experience in CS-related topics. Only after understanding the concept of algorithms can one go on to translating these into programs. We also covered lists and dictionaries (a collection type similar to hashmaps in Java); topics that generally, using other languages, are not included in introductory programming courses.

Teaching Approach

Active learning[11] builds on *constructivist* principles, according to which students become active participants in their own learning process (Ben-Ari 1998). Instead of viewing learning as passive transmission of information from the teacher to the students, constructivists see learning as an active, reflective process in which the students themselves construct the knowledge by building further upon their prior knowledge.

Moreover, both constructivism and active learning are related to the cone of experience[12] developed by Dale in the 1940s. This model suggests that learners retain more information by what they "do" (90 %) compared to what they "read" (10 %), "hear" (20 %) or "observe" (30 %). Consistently with this model, recent literature on active learning (for example (Ramsden 1992)) suggests that most students cannot learn unless they are actively involved.

As a result of the issues discussed above, we took a "learning by doing"-approach,[13] and did not give traditional lectures, but instead introduced so called *lab-lessons* in which theory and practice were interweaved during class. The lab-lessons were interactive, hands-on sessions, which took place in a computer lab. The exercises and other activities were *problem-oriented*, engaging the students in both problem-solving and analysis, in addition to writing the actual

[7] http://www.python.org

[8] http://www.cs.ubc.ca/wccce/Program03/papers/Toby.html, http://zen.sandiego.edu:8080/luby/papers/python.pdf, http://www.python.org/sigs/edu-sig/miller-dissertation.pdf, http://mcsp.wartburg.edu/zelle/python/python-first.html

[9] http://www.ibiblio.org/obp/pyBiblio/schools.php

[10] CS is not included as an individual subject in the curriculum, but is instead integrated in other subjects. Schools also have the opportunity to arrange optional courses in CS and related topics.

[11] http://www.ericdigests.org/1992-4/active.htm.

[12] http://www.mc.uky.edu/pharmacy/edinnovation/pdf/Step Dales Cone.pdf

[13] The concept of "learning by doing" was introduced by John Dewey in the early 1900s (Dewey 1910).

code. Each lab-lesson started with the teacher formulating a problem, covering topics likely to be relevant and motivating to the students, including simple games and web programming. A solution was developed collaboratively by gradually refining the code. The classroom thus made up a community, in which authentic problems were solved in a "real" context. This can be seen as an implementation of *situated learning*, a theory introduced by Lave and Wegner in the early 1990s, and discussed by Ben-Ari (2004b).

We chose the lab-lesson approach in order to address issues that have emerged from literature: Winslow (1996) states that "[g]ood pedagogy requires the instructor to keep initial facts, models and rules simple, and only expand and refine them as the student gains experience". According to Spohrer and Soloway (1986), "students are not given sufficient instruction in how to 'put the pieces together."' Lahtinen et al. (2005) have stated that "[t]he more practical and concrete the learning situations and materials are, the more learning takes place". The lab-lessons were designed to make it easier for the students to see the reason for introducing new concepts: why new constructs are needed, and when, and how, these are to be used. Programming concepts and structures were covered in the order needed to be able to solve the problem; not just because they "had to" be introduced according to the order of the chapters in a book. In this manner, new features were learnt one at a time, at the same time as they were combined with previously learnt programming structures using known strategies. This approach of mixing theory and practice is not unique (Haberman & Kolikant 2001), but to our knowledge not common in Finnish HSs. The more traditional way of instruction is to separate the two parts by first covering theory and only then implementing the newly learnt ideas in practice as a separate part of tuition.

Material

A tutorial was written for each problem containing a problem description and theory about the concepts needed to solve the problem, also providing a strategy for solving the problem at hand. Although the course was held in class, all material was made available on a web page in the course framework *Moodle*.[14] The course page acted as a repository for course material, examples, assignments, announcements and links. The students were to document 1) what they had experienced as most difficult and 2) what they had learnt after each lab-lesson in their private online learning diary (called *progress report* since we wanted to use a more neutral term that would not have direct associations with "traditional" diaries). The purpose was to make it easier for us to see when students had problems; in our experience, HS students are seldom willing to express their uncertainties or admit their difficulties in front of their classmates.

Examination

The final grade was based on the level of participation in the lab-lessons, a final project (including documentation) and a two-part exam. The final project was a larger assignment that the students completed at the end of the course. Since we wanted to meet the individual needs and interests of the students, they were allowed to choose the topic for the project quite freely. The students were allowed to complete the project either individually or in pairs. In cases where the students collaborated, documentation had to be precise, clearly stating who had done what. The final exam consisted of one part to be completed using only pen and paper and another, in which the students were to write the solution programs on the computer.

[14] http://www.moodle.org

4 Study Methodology

Action Research

Our research is loosely built on the principles of *action research* (Clear 2004). In action research, practitioners in a field improve practice by doing or changing something and reflecting on the results. The improvement can be in three areas: "improving a practice; improving the understanding of a practice [...] and improving the situation in which the practice takes place" (p. 106). The main purpose is to collect data and experience that help in gaining a better understanding of the practice. Our study follows a practical action research model, in which "the researcher facilitates reflection by individual practitioners upon some aspect of their practice" (p. 109). In our case, the researchers and the practitioners were the same persons.

Settings

The introductory course presented in the previous section was given four times as an optional HS course during the school year 2004/2005 (four different student groups in two different schools). In total, 42 boys aged 16-19 have completed the course; unfortunately the course did not attract any female students. The students in our study constituted nearly 40% of all HS students taking a basic programming course in Turku (the center of the third largest metropolitan area in Finland) in 2004/2005.

When teaching programming, one is bound to be faced with a situation in which the students are on different levels; some students might have programmed a lot, whereas others might not have any programming background at all. There were also differences between the groups: all students in groups A-C had taken at least one basic CS course, our programming course was part of their curriculum, and they met regularly thrice a week during a period of $1\frac{1}{2}$ months. The students in group D had not had any previous CS courses, and due to school arrangements, they met only once a week for three hours in the late afternoon, making the course span a period of three months.

Data Collection

The learning results were mainly derived by analyzing the progress reports, the final projects and the exam. In order to gather additional information we conducted two surveys during each course. The aim of the pre-course survey was to collect background data of the students, such as information about age, programming background and mathematical skills. More general questions, such as concerning the benefits experienced from learning to program and the students' awareness of Python, were also included. The purpose of the post-course survey was to gather information about the students' attitudes towards the course, its difficulty level and Python as the language of instruction. Students who had some pre-course programming background were also asked to compare Python with the languages used earlier.

5 Results and Discussion

In this section we present and discuss the learning results and the data extracted from the surveys. We also discuss our experiences from teaching Python

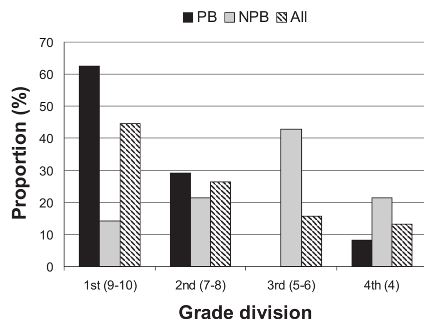

Figure 1: *The distribution (in percent) of students in divisions based on whether they had programmed before (PB) or not (NPB).*

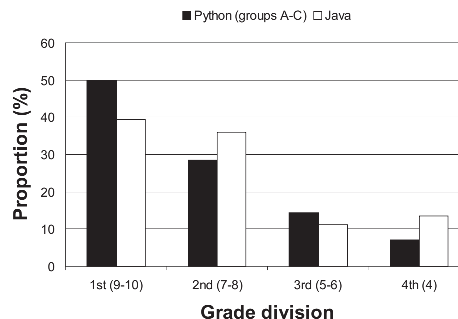

Figure 2: *Comparison of the grade distribution (in percent) from introductory programming courses using Python and Java.*

compared to prior experiences using other languages such as C, C++ and Java.

5.1 Learning Results

Course Grades

The course grade was given on the scale 4-10 (4 = failed), and was derived from the different parts of the examination mentioned in section 3. When calculated based on the data from all four student groups, the average course grade was 7.71. Over 85 % of the students passed the course. The grades were divided into four divisions (1st (9-10), 2nd (7-8), 3rd (5-6) and 4th (4)), which resulted in a distribution following a reversed Gaussian curve; this finding is well in line with our experiences from university level programming courses. Our data indicate that the results for the students who had some programming background were good: nearly 90 % achieved the grade 8 or higher. Although the diagram in figure 1 does not provide any comparable data, since this was the first time we gave these courses, it does show that nearly 80 % of students who had not programmed before passed the course. For students who achieved one of the lowest grades, the course exam was the weakest point of the examination parts. This calls for a discussion on how to prepare students for written exams: should tuition perhaps include elements of writing programs using only pen and paper? This would enforce the students to really think their programs through and trace the execution, test and debug the code - all by hand.

The number of students that failed the course was somewhat higher than for HS courses in general, but completely normal for programming courses, and therefore an expected result. The number of students in the first division was, however, larger than what can be seen as normal for any HS course (especially for programming courses): almost 45 % of all students passed with one of the two highest grades.

Since this was the first time a programming course was given at the school of group D, we did not have any data to compare the results of this group with. However, programming courses have been given at the school of the other groups (A-C), most recently using Java. For these courses, all other factors but the language have been the same as for groups A-C in our course (at least one background CS course, same teacher, same course requirements, scheduled and frequent meetings). When comparing the results, the differences were notable. As figure 2 shows, the proportion of students that failed decreased using Python, whereas the number of students achieving one of the two highest grades increased: half of the students in groups A-C achieved a grade in the first division. The reliability of this result might naturally also depend on other variables, such as motivation level, prior programming experience and mathematical skills, of the students. However, given that the prerequisites and the practical arrangements of the courses were identical, as was the teacher, the results should be comparable.

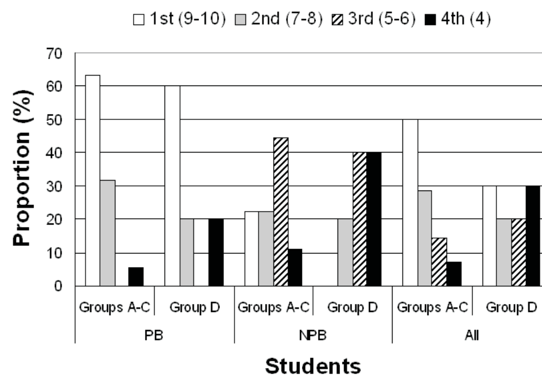

Figure 3: *The distribution (in percent) of grades achieved by students in different groups with regard to whether they had programmed before (PB) or not (NPB).*

As expected, there were differences between the grades achieved by students in groups A-C and those in group D. Figure 3 shows that the overall result for the students in groups A-C was better than for those in group D. The biggest difference was among students who had not programmed prior to the course; whereas only 11 % of the novices in groups A-C failed, 40 % of those in group D failed the course. On the other hand the percentage of students achieving a high grade is just as good for students with some programming background in both groups A-C and group D. Our findings thus suggest that the differences in the environment (for example meeting rarely late in the afternoon) and the lack of previous CS-courses, complicated the learning process among true novices. Moreover, not meeting more frequently resulted in the students finding it difficult to remember the syntax and programming in general. Considering, in addition, that over 75 % of all students stated that they programmed most in class, one can conclude that the students in group D were not exposed to programming as much as would have been needed. However, when starting on the final project, most students also worked at home. Since the students enjoyed working on the project, it could be beneficial to let the students work on smaller projects throughout the course to ensure that they program continuously.

Progress Reports

We were pleasantly surprised to see that most students took the progress report seriously. Naturally, there were also non-informative comments, but most students filled out the report regularly, stating what they had learnt and what they had experienced as most difficult. The analysis of the reports showed that students had in general been positive. Coding itself was frequently mentioned as the most enjoyable part of the lab-lessons. No difficulties were mentioned at the beginning of the course, whereas some students started to report difficulties when dealing with subroutines and exceptions.

An interesting observation was that the weaker students more often wrote comments such as "rather easy", "did not sound too difficult", "it will be OK when I practice", whereas the more advanced students were able to point out the difficulties more precisely if they had any. One explanation for this could be that weaker students do not dare admit their difficulties, but it might also be that they do not recognize which things are problematic. This could imply that teachers should try more actively to make students aware of their difficulties, for example by arranging more mid term quizzes and discussing the results - perhaps even in private.

Characteristics of Student Programs

We are currently in the process of analyzing the students' final projects, and we will later report on our findings from this analysis elsewhere. Our preliminary results indicate that the programs have been of at least the same standard as during similar HS-courses given using other languages. Most final projects include different control structures, self-made and built-in subroutines, modules as well as some documentation. Several projects can be ranked as particularly good considering both quality and complexity: programs that consist of several modules interacting at runtime, providing user guides and thorough documentation including testing and debugging reports.

Code listing 2 *for-loops in Python*

```
Python                Java
for element in set:   for (i = 0; i < set.length; i++) {
   print element            System.out.println(set[i]);
                      }
```

Some interesting findings have been made already at this point of our analysis. First, the "off-by-one bugs" (Spohrer & Soloway 1986) seem to be less frequent in students' programs using Python compared to our previous experiences in, for instance, Java. One evident explanation can be found in the differing format of the for-loop in Python. When writing for-loops that are truly "Pythonish", no indexes are needed (code listing 2). Whereas the for-loop in Java can be seen as quite cryptic, the Python for-loop might remain unclear due to the missing indexes: the syntax does not give an intuitive understanding for what the computer does and how many times it repeats it. The avoidance of indexes naturally cleans up and simplifies the code, but the Python version might even be too simplified. In this sense, the for-loop of Pascal can be seen as a good compromise: simplified, but still containing indexes, thus forcing the students to get used to "thinking" in terms of the computer and its memory. However, the lack of indexes can also be seen as an advantage: when teaching programming as an all-round learning skill in HSs (to students, of which by no means all will become programmers), the most important thing is for students to learn how to solve problems and translate solutions into program code. The question is whether indexes play an important role in this. If the answer is yes, the Python for-loop can indeed be regarded as too simple; otherwise, its simplicity is an additional feature due to which Python can be found suitable for novice programmers.

Another interesting issue concerns the use of I/O. When teaching HS programming using Java and C++, our students have frequently found user input difficult. Difficulties related to input statements have also been reported on by several researchers (for example (Bayman & Mayer 1983, Haberman & Kolikant 2001, Rosenberg & Kölling 1996)). In Python, no input or output streams have to be opened or closed. Getting input from the user is accomplished by using simple built-in functions (code listing 3), and the students learnt to use I/O effectively in an hour. The same comments as for the for-loop can be made concerning I/O as well; while Python's solution is compact and simple, it can be discussed whether it has been made even trivial.

Code listing 3 *I/O in Python*

```
# Prints out a question and reads user input as a string
name = raw_input("What is your name? ")

# Prints out a question and reads user input as a numeric
age = input("How old are you? ")
```

5.2 Survey Results

Difficulty Level

The data from the post-course survey showed that the course was not considered particularly difficult. The students graded the difficulty level of the course on a five-point scale (1 = very easy, 5 = very difficult), giving a mean of 2.58 (std dev = 1.03).

Students rated the difficulty level of each course topic on the same scale as the course. The data (table 1) showed that no topic had been regarded as particularly difficult, the averages lying between 1.33 and 3.11 on the scale 1-5 (1 = very easy, 5 = very difficult). Subroutines, exception handling and documentation were considered the most difficult topics. Open questions established this finding: whereas variables and control structures were considered rather simple, abstract topics such as algorithms, exception handling, documentation and subroutines were regarded as more difficult.

Topic	Mean	Std Dev
Algorithms	2.68	0.87
Variables	1.72	0.97
Lists	2.51	1.21
Boolean expressions	2.32	1.04
For-loops	2.32	0.93
While-loops	2.05	0.94
If-statements	1.33	0.74
Subroutines	2.76	1.08
Modules	2.44	1.02
Dictionaries	2.51	1.07
Files	2.69	1.15
Exception handling	2.89	1.09
Documentation	3.11	0.85

Table 1: *The perceived difficulty level of course topics (1 = very easy, 5 = very difficult)*

Difficulties in Abstract Topics

Subroutines are known to be difficult among novices, for example due to parameter passing (Fleury 1991). Another plausible explanation is that it might be difficult to understand why subroutines are needed when writing short programs. The same goes for exception handling: students might not grasp why such a thing is needed. Finding out what might go wrong in one's programs, and recognizing the parts of the code in which errors could occur, can be difficult. Exception handling has traditionally not been seen as one of the first things to cover in programming courses, resulting in it being introduced rather late in the schedule. This might make it even more difficult for the students to understand why it is suddenly needed; after all, their programs have "worked" before (provided that the user has given the correct input).

As for the difficulties experienced with documentation, researchers have suggested different reasons: students find writing comments and documentation useless for small programs, and is also considered to slow down the programming process (Deveaux et al. 1999). Moreover, Weinberg (1998) stated that "the good documenter has to be a good programmer" (p. 169). Clearly, not all students having taken their first programming course can be regarded as "good programmers" and explaining one's programs, ideas and logic in one's own words is completely different from writing code. In our case, the fact that the students were young might be an additional reason: students of this age are simply not used to writing thorough documentation, and have, most likely, not yet developed the necessary writing skills.

Many of our findings are supported by previous studies: in a study made by Haataja et al. (2001) among students in a web-based introductory programming course in Java, 60 % of the respondents stated that methods (in other words, subroutines in Java) were one of the most difficult topics. According to a study conducted by Lahtinen et al. (2005), students find error handling (exception handling) and libraries (modules) more difficult than, for example, selection and looping structures when learning to program in Java or C++.

Although many of our findings are supported by results from earlier studies, some differences can be found: compared to the results reported by Lahtinen et al. (2005), variables and selection structures were considered easier by our students (average difficulty of 1.72 and 1.33 vs. 2.10 and 1.98). Loop structures were considered to be of similar difficulty level in both studies. Moreover, our students found the if-statement easy, whereas 48% of the respondents to the study of Haataja et al. (2001) perceived if-statements as one of the most difficult topics. It thus seems that although some of the difficulties are the same across studies, there are differences as well, which, in addition, can be contradictory. These differences can naturally be due to various factors, the differing programming language being only one. More research comparing students' attitudes towards the different topics in introductory programming courses would be needed to make clearer the potential impact of the language.

Petre et al. (2003) have found indications of topics being introduced early in a course to be generally perceived as "easy" by students, whereas later topics are usually considered more difficult. They found two possible explanations for this: 1) students have more time to truly learn the early topics and 2) the early topics are emphasized more by the teacher. In this light, our findings and the results from previous afore-mentioned studies suggest that the "difficult" topics would benefit from being introduced earlier in the syllabus. For instance, exception handling could be introduced when discussing user input, to let the students get used to including it in their programs from the beginning. Later on in the course, we find it imperative that the students are made aware of the different types of exceptions that can occur and learn how to look for parts in the code that could be prone to errors. This could potentially result in students paying more attention to debugging and trying to write correct programs. Documentation could also be emphasized more in the very beginning of the course, in order for the students to get used to describing the goals and the process of their programs. This can be done for small programs, but takes away some time from actual programming. One must then consider the benefits of introducing these "difficult" topics earlier - is it worthwhile to rethink the way in which we teach programming if we can give students better opportunities to learn the difficult topics? We are tempted to say "yes", or as one student stated in the post-course survey: "Is it truly necessary to spend so much time on printing in the beginning of the course? The simple issues will still be practiced throughout the rest of the course[...]". It would be interesting to see if the results were to be different if the "simple" topics were covered quickly, and the ones that have commonly been viewed as "difficult" were given more time.

Attitudes towards Python

According to the pre-course survey data, almost half of the students had never heard of Python before. About 65 % of the students mentioned a language they would prefer to use, the majority mentioning C++, C or Java.

The students who had prior experience in programming (60 %) were asked to compare the languages used previously to Python. We used a five-point scale (1 = totally disagree, 5 = totally agree) and the results were in Python's favor: compared to other languages, Python was perceived as easier to learn and also as somewhat more pleasant to use. The students strongly agreed with the statement "Python was easier to learn [than languages previously used]" (mean of 4.1) and they also agreed with the statement "Python was more fun to use" (mean of 3.7). Naturally, new languages might be considered "easier" when one has already learnt to program in another one. However, the fact that Python was regarded as at least somewhat more fun to use is a positive finding. According to the students, aspects such as simplicity, compactness (resulting in shorter code), clarity and a rich range of modules made Python better than other languages; that is, the kind of features that initially suggested that Python would be suited as a first language. The other languages were, on the other hand, perceived as more flexible and "hard-core", enabling programming on a lower level. However, the course being an introductory one, no "hard-core" details could (or even should) be covered.

Nearly 80 % of the students that filled out the post-course questionnaire planned to continue programming, and over 65 % of these stated that they would do so using Python. Other languages mentioned were C++, C, Java and PHP. Half of the students who had programmed prior to the course (in some other language than Python) reported that they would continue programming using Python after taking this course. This can be seen as an interesting result, which could imply that Python was considered better than the languages used previously; one could expect that one does not lightly change one's preferences concerning what language to use.

Student Satisfaction

The post-course survey showed that the course had fulfilled the expectations of most students (90 %). Over half of the students had learnt exactly as much as they had expected in the course, whereas 23 % felt they had learnt more and 20 % felt they had learnt less than expected. The students found writing code, working on the final project and "getting a program to work" most satisfying. The lack of graphical programming was the main reason why the expectations had not been fulfilled. Considering again the fact that the students were young, one can assume that some did not have any expectations at all, and some had expectations focusing on technical and "cool" details.

Collaborative or Single Programming

The results from the post-course survey did not indicate any preference to work alone or with a friend. In groups A-C, all students worked individually on the project, whereas most students in group D worked in pairs. One explanation for most students working alone could be the additional requirements on documentation, since, as stated earlier, the students found documentation difficult. We believe that collaborative programming could facilitate producing documentation: when working together with someone else, one is bound to express thoughts and ideas to the other(s), that is, exactly the same things that one does when documenting programs. In the future, we will therefore consider guiding the students into working in pairs or groups.

5.3 Our Experiences

Powerful Built-in Constructs

Compared to for instance teaching Java, it was possible to introduce the powerful list data type at an early stage, thus enabling writing more advanced programs. The students did not use the same variables for different data types, although this is allowed, and dynamic typing did, thus, not turn out to be a drawback as was presumed in section 3.

Ease of Debugging and Error Detection

We had expected that Python's easy and clear syntax would facilitate students' learning, but we had not considered the consequences for us as instructors. The indented structure of Python programs forced the students to write structured programs, since incorrect indentation results in syntax errors. The clear indentation facilitated debugging, error detection and made correcting students' programs easier than, for instance, in Java or other languages where program blocks are denoted by curly brackets. In such languages, the compiler does not put any requirements on the structure of the program; one can even write an entire program on one single line, provided that all semicolons and brackets are in the right places. Programs of this kind are nearly impossible to check. However, such horror code cannot be written in Python. We feel that writing structured programs, which are easy to check, follow and maintain is one of the main lessons in any programming course. Using Python, this lesson is taught automatically.

Interactive Mode

The interactive mode was useful when introducing new constructs and showing examples. Since each concept could be demonstrated in isolation, no other syntax "got in the way". In addition, the students frequently used the interactive mode for checking name spaces and reading documentation.

Ease of Re-Use

The core of a programming language should not be overly extensive (Weinberg 1998): all necessary constructs and concepts should be compact. To be truly useful, the language should, however, be extendable. Most of the languages today can be extended by individual programmers, who are also given the possibility to re-use code written by others or themselves. However, in Python the advantages of re-use are even more evident, since all programs automatically become modules that can be used in other programs. Our students were able to share their programs with others and re-use programs written earlier already during the first parts of the course. The students recognized the importance of also writing small and seemingly unusable programs, by seeing how these are used when building more complex software.

Our suspicions of the negative impact of the lack of information hiding proved to be without grounds. Even though the students found writing documentation difficult, they learnt to efficiently read and make use of formal documentation, and were able to use ready-made subroutines correctly without knowing what was going on "behind the scenes". This made it possible for us to discuss abstraction and information hiding, which suggests that the students managed to develop some level of abstract thinking skills in this short time - even though object orientation was not included in the syllabus.

Code listing 4 *Use of the webbrowser module*

```
import webbrowser

options = {"A" : "http://www.google.com",
           "B" : "http://www.yahoo.com",
           "C" : "http://www.altavista.com"
           }

for site in options:
    print site + " : " + options[site]

try:
    choice = raw_input("\nChoose a site: ")
    webbrowser.open(options[choice])

except:
    print "You did not pick a valid alternative."
```

In addition, using modules that appealed to the students made it possible to teach the basics of programming in a motivating way. Since the modules are easy to use, and do not enforce notational overhead, these did not take the students' attention away from the programming itself. In our experience, when trying to incorporate external packages or modules in other languages, the extra notation usually turns the students' focus to learning the specific package instead. In Python, the modules and the functionality they provide can really be used as what they should be, in other words, as tools, without becoming the focus of attention. One example is the *open*-function of the `webbrowser`-module, which can be used to open web pages using a browser. The sample program in code listing 4 illustrates a program that displays a menu of search engines of which the user chooses one by simply entering the corresponding menu letter, whereas the web page is opened using the default browser. As the code illustrates, this lets the students practice important concepts such as user input,

iteration, dictionaries and exception handling in an interesting way, without extra notation. In our opinion, modules truly supported teaching and we feel that this ease of making tuition interesting is a great advantage for a language used by novices.

6 Summary

When choosing a programming language for educational settings, one has to consider many different factors: although learning aspects should be most important, institutions, pressured by for instance industry, usually feel a responsibility to select a language with market appeal. The competition is heavy, and individual institutions do not dare to make radical changes or try something new - even if they wanted to; the risk of losing students to a competitor is too big. Python is not one of the most widely used languages today, but statistics indicate that it is among the top ten languages.[15] But should this even be an issue at HS level? This is again a question of what we want our students to learn. If we focus on a certain language, we are most likely primarily teaching the language and its syntax, the core concepts of programming becoming a secondary objective. Should it not be the other way around? Should we not focus on teaching programming skills and leaving the language to be something of subordinate importance? The fact still remains - we need a language when teaching programming, and the challenge is to find the most suitable one. In this paper we have presented a case study of using Python to introduce programming in HSs, with emphasis on programming, not Python, and we will now summarize our main findings.

Introducing Programming in High School...

Our intention was to introduce programming as an all-round learning skill at HS level, and to some extent we succeeded. The students experienced feelings of success, learnt to write structured programs and think in terms of re-use and interfaces; a highly needed and appreciated skill in today's society.

Our findings indicate that students find it difficult to deal with abstract concepts, regardless of language. Naturally, the easier the syntax, the better, but in order to truly understand, one cannot focus on pure syntax or technical details; these can be looked up in a book and copied into one's programs. However, using abstract concepts properly and in the correct place of the code is nearly impossible without understanding the underlying principles of why these concepts are needed and in which situations. Abstract topics should thus be given more time in the syllabus, even early on in the course.

Another source of difficulty was related to the young age of the students: HS students have not necessarily yet developed good writing skills required, for example, for producing proper documentation. A final issue is that of attracting girls into CS: so far, all students have been boys, and considering the role of HSs as all-round learning institutions, this is alarming. We tried to build a programming course that would be suited for everybody, but were not able to attract any female student - even though both teachers were women. A future challenge is therefore, as is probably the case for all CS faculty, trying to find ways to make programming more appealing to girls.

...Using Python

Python's simple and flexible syntax significantly reduced notational overhead compared to other languages we have used in instruction, and thus left more time for actual coding. In addition, the interactive environment offers immediate feedback even with limited language experience, and lends itself to in-class coding, testing and demonstration. This interaction also supports active, hands-on learning, since the students can easily try out and analyze different solutions. We identify the advantages of using compiled system languages such as Java and C++ at higher levels of education and in industry, but we do believe that one should remember that there is quite a difference between teaching professionals or experienced programmers, and high school students, who are about to take on programming for the first time. The main goals of programming education at HS level are (1) to teach the basics of programming and algorithmic thinking as general all-round skills, and (2) to prepare the students for future studies.

Moreover, the ease of re-use has many benefits: the easy-to-use modules makes it possible for the students to practice basic concepts in a motivating manner, and the students learn how smaller programs are crucial when building larger systems. Although the final product might be very complex, the building process must be easy; using Python this is something that seems possible to achieve. Python is an open-source language, and this openness further increases its suitability for teaching purposes. Books, course readings, syllabi, exercises and other material can be downloaded from the Internet free of charge, and news groups and mailing lists provide forums for discussions and questions. The international community may also have welcome side effects: reading material in foreign languages and exchanging ideas with persons with the same interests in other countries develop language skills and promote understanding for other cultures. In some cases, Python's syntax can be seen as perhaps even too simplified, but as a whole, our initial experiences indicate that Python could be suitable as a language for novices.

7 Future Work

It is always difficult to judge the effects of curricular changes objectively, and the case study presented in this paper is too small for giving any significant statistical results; that was, however, not our original goal either. We set out to investigate programming at HS level and the suitability of Python in this process. Our experience has been positive, but to be able to make more conclusive statements, we are actively continuing our work in the field of using Python in HS programming courses.

Since the four courses presented in this paper, we have given a HS continuation course in programming covering object-oriented topics and graphics using Python. As we speak, we are also conducting a new study, similar to the one presented here. Some changes have been made, based on the results from this first study, for example by rethinking the syllabus in order to introduce abstract topics earlier. We will also continue the analysis of the student projects from the courses given so far. Finally, some of the students that have participated in our Python courses are now studying Java, and we are following up on their progress.

References

Agarwal, K. K. & Agarwal, A. (2005), 'Python for CS1 CS2 and Beyond', *J. Comput. Small Coll.* **20**(4), 262–270.

Aiken, R. M. (1972), Experiences and observations on teaching computer programming and simulation concepts to high school students, *in* 'SIGCSE '72: Proceedings of the 2nd SIGCSE technical symposium on CS Education', ACM Press, pp. 67–71.

[15] http://www.tiobe.com/tpci.htm

Anthony Robins, Janet Rountree, N. R. (2003), 'Learning and teaching programming: A review and discussion', *Computer Science Education* **13**(2), 137–172.

Bayman, P. & Mayer, R. E. (1983), 'A diagnosis of beginning programmers' misconceptions of basic programming statements', *Commun. ACM* **26**(9), 677–679.

Ben-Ari, M. (1998), Constructivism in Computer Science Education, *in* 'Proceedings of the 29th SIGCSE technical symposium on CS education', Atlanta, Georgia, United States, ACM Press, pp. 257–261.

Ben-Ari, M. (2004a), 'Computer Science Education in High School', *Computer Science Education* **14**(1), 1–2.

Ben-Ari, M. (2004b), 'Situated Learning in Computer Science Education', *Computer Science Education* **14**(2), 85–100.

Biddle, R. & Tempero, E. (1998), 'Java pitfalls for beginners', *SIGCSE Bull.* **30**(2), 48–52.

Ceder, V. & Yergler, N. (2003), Teaching Programming with Python and Pygame, *in* 'PyCon 2003'.

Clark, D., MacNish, C. & Royle, G. F. (1998), Java as a teaching language - opportunities, pitfalls and solutions, *in* 'ACSE '98: Proceedings of the 3rd Australasian conference on CS education', The University of Queensland, Australia, ACM Press, pp. 173–179.

Clear, T. (2004), *Computer Science Education Research*, Taylor and Francis Group, chapter 2, Critical Enquiry in Computer Science Education, pp. 101–125.

Close, R., Kopec, D. & Aman, J. (2000), Cs1: perspectives on programming languages and the breadth-first approach, *in* 'CCSC '00: Proceedings of the 5th annual CCSC northeastern conference on The journal of computing in small colleges', New Jersey, United States. Consortium for Computing Sciences in Colleges, pp. 228–234.

de Raadt, M., Watson, R. & Toleman, M. (2002), Language Trends in Introductory Programming Courses, *in* 'Informing Science and IT Education Conference', pp. 329–337.

Deveaux, D., Fleurquin, R. & Frison, P. (1999), Software engineering teaching: a "docware" approach, *in* 'ITiCSE '99: Proceedings of the 4th annual ITiCSE conference', Cracow, Poland, ACM Press, pp. 163–166.

Dewey, J. (1910), 'How We Think'. Retrieved June 21, 2005 from http://spartan.ac.brocku.ca/~lward/Dewey/Documents.html.

Elkner, J. (2001), Using Python in a High School Computer Science Program, *in* '9th International Python Conference'.

Elkner, J. (2002), Using Python in a High School Computer Science Program - Year 2, *in* '10th International Python Conference'.

Fleury, A. E. (1991), Parameter passing: the rules the students construct, *in* 'SIGCSE '91: Proceedings of the 22nd SIGCSE technical symposium on CS education', San Antonio, Texas, United States, ACM Press, pp. 283–286.

Gal-Ezer, J., Beeri, C., Harel, D. & Yeduhai, A. (1995), 'A high-school program in computer science.', *Computer* **28**(10), 73–80.

Ginat, D. (2003), The novice programmers' syndrome of design-by-keyword, *in* 'ITiCSE '03: Proceedings of the 8th annual ITiCSE conference', Thessaloniki, Greece, ACM Press, pp. 154–157.

Guzdial, M. (2003), A media computation course for non-majors, *in* 'ITiCSE '03: Proceedings of the 8th annual ITiCSE conference', Thessaloniki, Greece, ACM Press, pp. 104–108.

Haataja, A., Suhonen, J., Sutinen, E. & Torvinen, S. (2001), 'High School Students Learning Computer Science over the Web', *Interactive Multimedia Electronic Journal of Computer-Enhanced Learning* **3**(2).

Haberman, B. & Kolikant, Y. B.-D. (2001), Activating "black boxes" instead of opening "zipper" - a method of teaching novices basic cs concepts, *in* 'ITiCSE '01: Proceedings of the 6th annual ITiCSE conference', Canterbury, United Kingdom, ACM Press, pp. 41–44.

Hadjerrouit, S. (1998), 'Java as first programming language: a critical evaluation', *SIGCSE Bull.* **30**(2), 43–47.

Hagan, D. & Markham, S. (2000), Does it help to have some programming experience before beginning a computing degree program?, *in* 'ITiCSE '00: Proceedings of the 5th annual ITiCSE conference', Helsinki, Finland, ACM Press, pp. 25–28.

Halasz, F. & Moran, T. P. (1982), Analogy considered harmful, *in* 'Proceedings of the 1982 conference on Human factors in computing systems', Gaithersburg, Maryland, United States, ACM Press, pp. 383–386.

Lahtinen, E., Ala-Mutka, K. & Järvinen, H.-M. (2005), A study of the difficulties of novice programmers, *in* 'ITiCSE '05: Proceedings of the 10th annual ITiCSE conference', Capacrica, Portugal, ACM Press, pp. 14–18.

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K., Seppälä, O., Simon, B. & Thomas, L. (2004), A Multi-National Study of Reading and Tracing Skills in Novice Programmers, *in* 'ITiCSE-WGR '04: Working group reports from the ITiCSE conference', Leeds, United Kingdom, ACM Press, pp. 119–150.

Mayer, R. E., Dyck, J. L. & Vilberg, W. (1986), 'Learning to program and learning to think: what's the connection?', *Commun. ACM* **29**(7), 605–610.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., Laxer, C., Thomas, L., Utting, I. & Wilusz, T. (2001), 'A Multi-National, Multi-Institutional Study of Assessment of Programming Skills of First-Year CS Students', *SIGCSE Bull.* **33**(4), 125–180.

Merritt, S. M., Bruen, C. J., East, J. P., Grantham, D., Rice, C., Proulx, V. K., Segal, G. & Wolf, C. E. (1993), 'Acm model high school computer science curriculum', *Commun. ACM* **36**(5), 87–90.

Milbrandt, G. (1993), 'Using problem solving to teach a programming language in computer studies', *Journal of Computer Science Education* **8**(2), 14–19.

Mody, R. P. (1991), 'C in education and software engineering', *SIGCSE Bull.* **23**(3), 45–56.

Oldham, J. D. (2005), 'What Happens after Python in CS1?', *J. Comput. Small Coll.* **20**(6), 7–13.

Palumbo, D. B. (1990), 'Programming Language/Problem-Solving Research: a Review of Relevant Issues', *Review of Educational Research* **60**(1), 65–89.

Petre, M., Fincher, S. & et al., J. T. (2003), My criterion is: Is it a boolean?: A card-sort elicitation of students' knowledge of programming constructs., Technical Report 6-03, Computing Laboratory, University of Kent, Canterbury, Kent, UK.

Ramsden, P. (1992), *Learning to Teach in Higher Education*, London: Routledge.

Rosenberg, J. & Kölling, M. (1996), I/o considered harmful (at least for the first few weeks), *in* 'ACSE '97: Proceedings of the 2nd Australasian conference on CS education', The Univ. of Melbourne, Australia, ACM Press, pp. 216–223.

Schollmeyer, M. (1996), Computer programming in high school vs. college, *in* 'SIGCSE '96: Proceedings of the 27th SIGCSE technical symposium on CS education', Philadelphia, Pennsylvania, United States, ACM Press, pp. 378–382.

Shaffer, D. (1986), 'The use of logo in an introductory computer science course', *SIGCSE Bull.* **18**(4), 28–31.

Shannon, C. (2003), Another breadth-first approach to cs i using python, *in* 'SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on CS education', Reno, Navada, USA, ACM Press, pp. 248–251.

Soloway, E. (1993), 'Should we teach students to program?', *Commun. ACM* **36**(10), 21–24.

Spohrer, J. C. & Soloway, E. (1986), 'Novice mistakes: are the folk wisdoms correct?', *Commun. ACM* **29**(7), 624–632.

Stajano, F. (2000), Python in Education: Raising a Generation of Native Speakers, *in* 'Proceedings of the 8th International Python Conference'.

Stephenson, C. & West, T. (1998), 'Language choice and key concepts in cs 1', *Journal of Research on Computing Education* **31**(1), 89–95.

Treese, W. (2003), 'Programming literacy: is it for everyone?', *netWorker* **7**(2), 15–17.

van Rossum, G. (1999), Computer Programming for Everybody. CNRI: Corporation for National Research Initiatives, 1999.

Weinberg, G. M. (1998), *The Psychology of Computer Programming*, Dorset House Publishing.

Winslow, L. E. (1996), 'Programming Pedagogy - a Psychological Overview', *SIGCSE Bull.* **28**(3), 17–22.