Python and Roles of Variables in Introductory Programming: Experiences from Three Educational Institutions

Uolevi Nikula
Lappeenranta University of Technology, Finland

Uolevi.Nikula@lut.fi

Jorma Sajaniemi and Matti Tedre,
University of Joensuu, Finland

Jorma.Sajaniemi@joensuu.fi
Matti.Tedre@joensuu.fi

Stuart Wray
Royal School of Signals, Blandford, England

swray@bournemouth.ac.uk

Executive Summary

Students often find that learning to program is hard. Introductory programming courses have high drop-out rates and students do not learn to program well. This paper presents experiences from three educational institutions where introductory programming courses were improved by adopting Python as the first programming language and roles of variables as an aid in understanding program behavior. As a result of these changes, students were able to write working code from the very beginning, they found programming easy and interesting, they passed exams with good grades, and drop-out rates were low. Students became interested in programming and some of them even changed their personal study plan to include more of programming and computer science.

The high abstraction level and complexity of the concepts to be learned in current elementary programming courses is a serious impediment to students. Python is a simple but powerful programming language that supports both procedural and object-oriented programming. It makes short programs straightforward to write while at the same time facilitating rapid development of powerful applications. Python has been found to make programming more fun and thus attract students. Some of the common arguments against Python include concerns about using an interpreted language, the different syntax from C++ and Java, the use of whitespace to represent code blocks, and the lack of static type checking. However, these concerns have not caused any significant problems in practice, though it may take some effort to adapt to the syntax of other languages.

Roles of variables are stereotyped patterns of usage of a variable. For example, in the role of a *stepper*, a variable is assigned a succession of values that is predictable and usually known in advance as soon as the succession starts. Roles provide students with programming knowledge in a compact form, which they can then apply in authoring

Editor: Janice Whatley

and understanding programs independently of the programming language used. In a classroom experiment, explicit teaching of roles has been found to result in better programming skills.

This paper also discusses the applicability of the roles to Python and notes some changes in the interpretation of individual roles required by some peculiarities of Python. In general, roles apply to Python well. However, the lack of constants in Python calls for a new role, and the use of lists in for-loops suggests a new interpretation for the stepper role. Finally, the difference between the two data structure roles, organizer and container, is smaller in Python than in other languages making their separation needless.

Keywords: computer science education, programming, CS1, roles of variables, Python.

Introduction

Undergraduate programming courses often have high drop-out rates and students have poorly developed programming skills. There has been a continual debate within programming education circles as to how to best solve these problems. The high abstraction level and complexity of the concepts to be learned is a serious impediment to students (for example, see (Rich, Perry, & Guzdial, 2004; Robins, Rountree, & Rountree, 2003) for reviews). However, even basic programming concepts such as variables and iteration are hard for many students to grasp. These problems had been encountered in the three university-level courses presented in this paper. The three courses vary in their contents, duration, and audience, but the changes made in the courses share two common issues: adoption of Python as the first programming language and the use of the "roles of variables" concept as an aid in understanding program behavior.

The high drop-out rates (25-50 %, (Herrmann et al., 2003; Nagappan et al., 2003; Rich et al., 2004)) and decreasing interest in computer science (Radenski, 2006) have recently been tackled in many universities by using Python as the first language. For example, in Georgia Institute of Technology a CS1 course was designed to attract the interests of women (Guzdial, 2003; Rich et al., 2004), and Chapman University attacked the perception that computer science is a dry and technically difficult discipline (Radenski, 2006) - and in both of the cases the Python language formed a central part of the new course implementation. There is also anecdotal evidence that Python makes programming more fun and thus attracts students (Reges, 2006). In any case, Python's simple syntax makes writing programs much easier than writing comparable programs in Java or C.

Roles of variables (Sajaniemi, 2002) provide students with programming knowledge in a compact form, which they can then apply in authoring and understanding programs independently of the programming language used. In a classroom experiment, explicit teaching of roles has been found to result in better programming skills (Byckling & Sajaniemi, 2006) and in better mental models of programs (Sajaniemi & Kuittinen, 2005). It has also been confirmed that roles of variables do correspond to classifications naturally used by programming experts (Sajaniemi & Navarro Prieto, 2005), i.e., roles belong to experts' tacit knowledge.

The rest of this paper is organized as follows. We first discuss Python as a first programming language and explain the *role* concept. Then we present the three introductory programming courses and summarize experiences obtained. Finally we discuss the applicability of the roles to Python and list changes and interpretations of individual roles as required by some Python peculiarities.

Python and Roles of Variables

Python as a First Programming Language

Donald E. Knuth (2005) looked at programming languages over the past four decades, and concluded that every decade seems to have a favorite programming language. Although Knuth made no predictions about the favorite language of the current decade, the Python language is a strong candidate. It is clear that no new programming language will be generally accepted before it has been proven technically capable in many applications. In a comparison of seven programming languages (Prechelt, 2000) some key issues of those languages were studied, including the program length, programming effort, runtime efficiency, memory consumption, and reliability. In this comparison Python proved to be a technically capable solution when compared to C, C++, Java, Perl, Rexx, and Tcl. For example, the programs written in Python were among the shortest in length, the variation in their length was among the smallest, and the programming time was also amongst the shortest.

As an interpreted language, the memory consumption and running times of Python are of concern, but in the Prechelt (2000) study Python was about average in those aspects. Although the results of the study are not conclusive due to small sample sizes (between 4 and 24 programs in each language), there was no clear reliability difference between different languages, programs in Python were generally developed in half of the time it took to write the programs in traditional languages (C, C++, or Java), and the programs were also only half the size. The technical capability of Python is also supported by the observation that many serious software developing companies are using Python, e.g. Google, Nokia, New York Stock Exchange, NASA, Honeywell, and Philips (Python Software Foundation, 2006).

Some of the common arguments against Python (Donaldson, 2003) include concerns about using an interpreted language, the different syntax from C++ and Java, the use of whitespace – spaces and tabs – to represent code blocks, and the lack of strict type checking. The first two objections are mostly name-calling, since in the light of the language comparison (Prechelt, 2000) they have no technical consequences.

When compared to C there are some small but important differences in the syntax, but in many cases these are designed to reduce errors. For example, Python does not accept the statement "if (a = 0)..." but requires use of comparison operator in this expression: "if (a == 0)..." which is most often what the programmer actually intended to do. Use of whitespace to represent code blocks seems awkward at first, but if code blocks are indented properly in C, the outcome is actually the same. The consistent use of indentation has been shown to improve comprehension (Pane & Myers, 1996). In C it is easy to have indentation that is inconsistent with the block structure, which is a fertile source of confusion. In Python this is not possible.

The lack of static type checking is the subject of larger debate as, for example, explicit declarations have been argued against as being redundant information (Pane & Myers, 1996). Static type checking is also claimed (Donaldson, 2003) to make some techniques difficult to use, and require much more up-front design from the programmer. The static type checking does of course allow the compiler to catch type errors early in the programming process. Considering the usability issues of novice programming systems Pane & Myers (1996) suggests though, that Python has addressed many such issues, for example the conciseness mentioned above, general minimalist design, and support for debugging. Thus, overall there is a fair amount of evidence to support a decision to start using Python as the first programming language.

Introduction to Roles of Variables

Programmers spend a large part of their lives reading code. According to one estimate (Lefkowitz, 2005) more than half of software development time is taken up reading code. Programs must be written to be understood by other people, not just to be executed by computers. This is clearly a key skill possessed by the expert programmer, a skill that students must learn. As noted by Abelson, Sussman, and Sussman (1996):

> "First, we want to establish the idea that a computer language is not just a way of getting a computer to perform operations but rather that it is a novel formal medium for expressing ideas about methodology. Thus, programs must be written for people to read, and only incidentally for machines to execute."

Introductory programming courses where students are merely taught to mimic the mechanical operation of the machine often leave students confused. Clearly, expert programmers do not see programs as meaningless strings of symbols. They put a great deal of effort into making their programs meaningful. If students are to write programs like experts, with structure and meaning, they must learn to read programs like experts, to see the structure and meaning that have been written into those programs. One way in which students have been successfully helped to read more of the meaning in programs is the "roles of variables" concept.

Sajaniemi (2002) introduced the concept of "roles of variables" as a development of Ehrlich & Soloway's (1984) notion of "variable plans". Based initially on analysis of the patterns of variable usage in a number of novice programming textbooks, this classification system can be used for both teaching programming and analyzing large scale programs. A role is a stereotyped pattern of usage of a variable. For example, in the role of a *stepper*, a variable is assigned a succession of values that is predictable and usually known in advance as soon as the succession starts. The role concept concerns only the succession of values which a variable holds, their lifetimes and how subsequent values relate to earlier values. It is important to understand that roles are not related to programming language types of variables, or to the meaning of variables in a particular program. It is only the pattern of successive values which is described by the role of a variable.

As an example, consider the following program:

```
count = 0
while count <= 0:
    count = input("Enter count: ")

for n in range(count):
    print "Two times ", n, " is ", 2*n
```

This program uses two loops. In the first loop, the user is asked to enter a number, and repeatedly asked again until their input is valid (greater than zero). In this loop we would describe the role of the variable `count` as being a *most-recent-holder*, because the sequence of values that variable `count` can assume is unpredictable.

In the second loop, the value of `count` is used to select a range of integers over which the variable `n` will iterate. For example, if `count` was 5, then `n` will take on the values 0, 1, 2, 3, 4 in sequence. This sequence is predictable from the start of the for-loop, and cannot be influenced by any computations in the body of the for-loop, so we describe the role of the variable `n` as a *stepper*.

Table 1 gives a brief description of the different roles.

Table 1: Eleven most common roles of variables

Role	Informal description
Fixed-value	A variable which is assigned a value only once and the value of which does not change after that. (Sometimes known as a "single assignment" variable.)
Stepper	A variable stepping through a systematic, predictable succession of values.
Most-recent-holder	A variable holding the latest value encountered in going through a succession of unpredictable values, or simply the latest value obtained as input.
Most-wanted-holder	A variable holding the best or otherwise most appropriate value encountered so far. (For example the biggest value so far.)
Gatherer	A variable accumulating the effect of a series of individual values, for example a running-total. (Often known as an "accumulator".)
Follower	A variable that always takes its current value from the previous value of some other variable.
One-way-flag	A Boolean variable that is initialised to one value and may be changed to the other value, but never reverts to its initial value once it has been changed.
Temporary	A variable holding some value for a very short time only. (For example while values are rearranged between other variables.)
Organizer	A data structure storing elements so that they can be rearranged.
Container	A data structure storing elements that can be added and removed.
Walker	A variable used to traverse a data structure, for example a pointer running down a linked list.

These roles do not cover every possible pattern of variable use, but they cover the vast majority. It has been observed that with these roles it is possible to cover 99% of variable usage in novice-level programs (Sajaniemi, Ben-Ari, Byckling, Gerdt, & Kulikova, 2006), and that the roles *fixed-value*, *stepper* and *most-recent-holder* cover 70% of variable usage.

In addition to procedural programming, roles have also been applied to object-oriented and functional programming (Sajaniemi et al., 2006). In the object-oriented paradigm, roles are used for attributes of objects as well as for variables; objects which represent one conceptual entity (for example Java Strings) are considered unitary values rather than *containers*. In the functional programming paradigm, roles apply to the recursive behavior of parameters and return values. Roles can also be used when designing programs. For example, UML class and object diagrams are easier to understand when attributes are annotated with their roles. For a more comprehensive treatment, see the Roles of Variables home page (Sajaniemi, 2006).

Previous Experiences with Roles

Previous work (Byckling & Sajaniemi, 2006; Sajaniemi & Kuittinen, 2005) reported the results of classroom experiments comparing a conventional Pascal course with two variants: one using roles in lectures, the other with additional role-based program animation in lab exercises. The 91 students on this course were divided into three groups, referred to as "traditional", "roles" and "animation":

- The **traditional** group received normal lectures and Turbo Pascal debugger animation in exercises. This group formed a control group for the other interventions.
- The **roles** group received lectures using roles systematically plus Turbo Pascal debugger animation in exercises. Many students in this group correctly used role names in answers to examination questions, even though the questions did not mention roles. However, they performed only slightly better than students in the traditional group at problem solving. Their grasp of roles appeared to be too abstract.
- The **animation** group attended the same lectures as the roles group but their exercises used a role-based animation program, PlanAni. This program continuously displayed graphical reminders of the roles of variables and used pop-up reminders of key events such as variable creation or loop entry. Students in this group were observed to explicitly use their knowledge of roles when designing programs in pairs and were much more successful at building correct programs than either of the other groups. This group also used role names spontaneously in examination answers.

In both cases where roles were introduced, the students used their new vocabulary to understand the programs they read. Roles gave students a conceptual framework with which they could understand variable usage in a similar way to more experienced programmers. In addition, the animation group's superior performance in program construction indicates that their understanding of the roles concepts was deeper than the group which only learned about roles in lectures. This conclusion is also supported by the fact that the animation group tended to discuss program structures in a more expert way when they were programming in pairs.

The Three Courses

Course at Joensuu

Python was used this year for the first time as an introductory programming language in the "Introduction to Programming" course at the University of Joensuu, Finland. This course is offered to all students at the university taking computer science as their major or minor subject. Thus, there is a wide variety of participants ranging from science students to arts students. The course consists of 24 hours of lectures, 12 hours of lab sessions, and home assignments. There is no textbook but the slides of lectures are available for students on the web and quite a few students used a free, publicly available Python guide (Kasurinen, 2006) as a reference. Students use the PlanAni role-animation program to visualize the execution of four programs during the lab sessions.

The main idea is to make sure that everybody learns the basic concepts of programming and that everybody enjoys learning them. As a consequence, the lectures proceed slowly, complicated issues are avoided, and all example programs do some meaningful tasks. Technical details and programs demonstrating specific features of the programming language only are avoided. The course covers imperative programming but no abstraction mechanisms, i.e., functions and classes are not included.

Special care is taken to make sure that students have fun in lab sessions and home exercises. For example, syntax-related questions contain not only syntactically correct and incorrect items but

also brain-twisting items, e.g., a question might ask whether "`hundred = 20`" is a correct Python statement. There are also tasks that ask students to write programs that contain some given code fragment such as:

```
while year > 1958 and year < 1978:
```

with no other restrictions on the solution. Students find such tasks inspiring as they can use their creativity and compare their own solutions to ideas presented by others.

At the end of the course, students made a self-assessment on how well they had learned Python constructs and programming-related tasks in general. A four-point scale was used: not at all, badly, well enough, excellently. In almost all questions more than 85% of the students reported at least well-enough learning. Only debugging (82%), roles (71%), and for-loops (68%) got a lower score. At the end of the course, 98% of students passed the final exam with an average grade 3.6 where 1 is the lowest grade and 5 highest.

Of the six roles that were introduced during the course, students reported *fixed-value*, *most-recent-holder* and *temporary* to be easy whereas *stepper*, *gatherer*, and *most-wanted-holder* were perceived as being harder. The hardness reported by the students might to be related to how long they had been using each role: the ones described as easy were introduced during the first three weeks of the course, the others in the last three weeks.

Teachers of the course found Python an excellent selection as a first programming language: Python makes it possible to immediately start with interesting, fully-fledged programs having meaningful functionality; students get experiences of discovery and success even in the first lab session; the simplicity of syntax enables students to write entire programs from scratch; basic programming constructs can be covered with a minimum of concepts, notations, and syntactic details. In contrast to Java, which was used in previous years, students now spontaneously tried out their own ideas and wrote their own programs, elaborating their knowledge and improving their programming skills. Finally, students that had attended the previous course reported that Python was clearly better than Java as a first language. The only negative feedback from the teachers concerned the use of the `range()` construct in for-loops. This was considered to be hard to understand and unnecessarily different from other programming languages.

Roles were introduced gradually during the course and the lecturer designed example programs based on the emergence of roles. He reported that this clarified his own thoughts and felt that this clarity was also transferred to students, too. The idea of asking the question "what does this variable do in this program" with an answer like "holds the smallest value so far" made the presentation of example programs easy and coherent for the lecturer.

Finally, the lecturer reported that the heavy use of roles in lectures paid off. The distinction between roles and variable types would not have been clear to students without continual reinforcement in lectures. The small number of roles with very clear meanings was also considered to be a key strength. The lecturer considered the central use of roles to be "a brilliant viewpoint to programming education".

Course at Lappeenranta

The introductory course on programming, Fundamentals of Programming, is delivered in two variants at Lappeenranta University of Technology (LUT), Finland. Course A is targeted at students who will need to be able to program later in their studies (computer science and electrical engineering majors). Course B is targeted at business and other students who need a basic understanding of programming. The courses have common lectures, the main difference being the practical work that the students must complete: course A has a more demanding project and assignments. The courses comprise 28 hours of lectures, plus weekly programming assignments and

quizzes. This arrangement with two courses has been used since 2001, and until fall 2005 the courses used C as their programming language.

The introductory programming courses at LUT have been problematic for some years: only 54% of the 186 students registered for the courses in fall 2006 were taking it for the first time and 21% of the students had registered for the courses previously at least twice. To overcome this problem, a major revision was undertaken in summer 2006 including a move from C to Python, inclusion of the roles of variables concept, integration of the weekly assignments and quizzes with the course project, development of a Finnish language programming guide for the students (Kasurinen, 2006), and aligning the course contents with ACM/IEEE Computing Curricula 2001 (Joint Task Force for Computing Curricula, 2001). Since many students were clearly not grasping the fundamentals, the new course focuses on understanding variables, input and output, iterations, files, and string handling. Abstraction is introduced through use of basic functions, classes, and lists.

The key results of the course implementations both in 2005 and 2005 are presented in Table 2. Since the courses tend to have students who register for the course but do not do any of the assignments, Table 2 uses the number of students who completed the first assignment as the reference point for calculations. *Dropout* refers to students who did not complete all compulsory assignments of the course while *exam failure* refers to students who completed them all but failed the exam. Table 2 shows a clear reduction in the exam and overall failure rates in both the courses. We also asked the students to complete questionnaires both half-way through the course and at the end. This feedback from the students showed that from a list of sixteen different programming topics - including variables and their types and roles, file handling, loops, etc. - the three easiest things were thought to be input, output, and calculations. Using a four-point scale (not at all, badly, well enough, excellently) in both the questionnaires with both the courses at least 94% of the students reported at least well-enough learning of the listed topics. These results are very encouraging, since in the previous courses using C many students found input and output very challenging.

Table 2: Key statistics of the course implementations at Lappeenranta in 2005 and 2006

	Course A				Course B			
	2005		2006		2005		2006	
First assignment done	138		129		60		57	
All assignments done	69	50 %	85	66 %	51	85 %	35	61 %
Exam & course completed	57	41 %	83	64 %	28	47 %	32	56 %
Dropouts before exam	69	50 %	44	34 %	9	15 %	22	39 %
Exam failures	12	17 %	2	2 %	23	45 %	3	9 %
Overall failures	81	59 %	46	36 %	32	53 %	25	44 %

The last question of both the questionnaires was reserved for qualitative feedback. This question was answered by about half of the students completing the course, and about half of the replies included positive feedback on the new course implementation. Students commenting on the language change from C to Python reported that in general the change was good and Python was found to be easier, clearer, and a more interesting language than C. The mid-term feedback included more comments on the Python-C –language topic, but final comments also supported the position that Python is a better fit for an initial course on programming than C. Only one student noted that he would have preferred a more general language, C++ or Java, as the programming language for the course. By and large the students did not consider the new course itself to be easy. Rather, the course was reported to require much more work than other courses and the programming assignments were found harder than previously. However, the course was considered

to have changed significantly in a favorable direction, and some senior students even considered the course to be the best course at LUT.

During the previous years many students wanted to change from the harder course A to course B. However, this time the course generated movement in the opposite direction. One student wanted to change from course B to course A since programming had started to interest him to such an extent that he wanted to change his minor to Computer Science - which requires participating in course A. Another student reported that she still needed one more course in her Master's Degree, and since this course had proven so interesting, she wanted to take another programming course that would be delivered along the same lines as this one. Combining these experiences with the comments in the feedback that programming can be fun, it seems that the course is changing to a favorable direction.

In Lappeenranta roles were introduced in lectures, followed by an example animation using PlanAni. Each role was also explained separately as it came up in the example programs. After that students were advised to study the roles on their own and ask advice, for example in the weekly exercises. Two specific encounters with roles can be noticed here: first, in the beginning of the course one assistant of the course raised serious doubts about the usefulness of the concept. However, when the course moved to the second half, this person commented spontaneously that the role concept appears justified and may well help in learning programming. The second observation comes from the final course feedback, where two students raised concerns about the usefulness of the concept. However, both of these students had been programming with the C language already before the course – just like the course assistant. Overall the feedback from the use of the roles concept was positive, but since the whole course has just gone through a major revision, it is too early to draw more formal conclusions.

Course at Blandford

The introductory programming course at the Royal School of Signals (RSS) at Blandford, England forms about half of a software engineering unit within two degree courses. One of the degree courses is a BSc (Hons) in Telecommunications Systems Engineering. This is an intensive first degree course compressed into 21 months. It is essentially an electrical engineering degree, with an emphasis on radio and telecommunications. The other course is an MSc in Communications and Information Systems Management. It is a one year postgraduate conversion course following a first degree in a subject unrelated to electrical engineering. Both degrees are accredited and awarded by Bournemouth University but taught by faculty at RSS.

The Python language is taught as 36 hours of lectures and lab sessions, with further work out of class. This introduces both procedural and object-oriented programming. After lectures on other aspects of software engineering, this is followed by a further 18 hours of lab sessions in which the students are helped to perform one iteration of a software development process (requirements, design, implementation and test), working in teams of 5 or 6 to build a simple distributed system for playing a board-game called “robots”.

Lectures are used to introduce Python language constructs which the students then explore in lab sessions, working through a sequence of “quiz” handouts in which they need to puzzle out what various fragments of Python code will do when executed. Some earlier exercises also require students to construct fragments of code themselves; in later exercises they modify programs and finally in the “robots” lab session, they write whole programs themselves. Students are encouraged to work in pairs, discussing the material with each other. Some of the handouts are “walk-throughs”, acting as a written reminder of how concepts introduced in lectures can be used to write programs in practice. All of the quizzes and exercises have accompanying “hints and explanations” handouts which are distributed to the students separately. These explanations make

heavy use of the roles concepts introduced in lectures. The textbook by Downey, Elkner, and Meyers, (2002) had been used in an earlier delivery of this course as the main text. This book was distributed to students for this course as a supplementary text, and some said that they found it helpful to have an alternative source of explanations.

Experience with Python at RSS has been very encouraging: compared to previous courses using Delphi, students seem to find the material more straightforward and are able to achieve more. The order in which topics were introduced for procedural programming was slightly unconventional (list-comprehensions, for-loops, while-loops then functions) but feedback from students was entirely positive on this approach.

Experience at RSS with roles in teaching Python has been more limited, only having been used on one course so far. Students used the role names spontaneously when talking about programs, and clearly were being helped to find structure in programs by doing so. However, the animation-based presentation of roles was not used, and it seems as a consequence that the students' ability to use roles successfully in program construction was rather limited. Some form of animation will be used in future courses.

Summary of the Courses and Findings

Experiences with Python have been very similar at all three educational institutions. Students are able to write working code from the very beginning of the course, they find programming easy and interesting, they pass exams with good grades, and drop-out rates are small. In all institutions Python has been found to be a better choice than the language used previously: Java at Joensuu, C at Lappeenranta, and Delphi at Blandford. This opinion is shared by both teachers and students. These findings are in line with other experiences reported in the literature (as noted earlier in the section "Python as a First Programming Language").

Teachers have, however, voiced some doubts regarding transfer to the next language in the curriculum. At the beginning of their studies, students' programming knowledge is fragile (Perkins & Martin, 1986) and changes in syntax may interfere with knowledge of semantics. A shift to some other language may therefore cause a temporary degeneration in students' ability to author programs. This question certainly needs further research but while waiting for that to be done, it is safer not to transfer to a second language during the first year.

Experiences with roles vary among the institutions. In Joensuu, where roles were used extensively during the course, the teachers found roles to help both themselves and students. The lecturer reported that roles were a brilliant viewpoint to programming education. At Blandford, the use of roles was more limited and role-based program animation was not used. Even though students were able to use role names in talking about programs, roles were not seen as a tool in program design. In Lappeenranta, the roles were not yet used extensively, but both the lecturer and teaching assistants found the roles useful in clarifying the use of variables. Thus, in the future the use of roles of variables will be increased in order to further improve the course.

Taken together with the results described in the section "Previous Experiences with Roles", these findings seem to suggest that roles are of most help to students when they are continuously referenced in teaching, when they are used in program design during lectures, and when role knowledge is elaborated with role-based animation.

A summary of the courses and their key characteristics is presented in Table 3.

Table 3. Summary of the three course implementations

	Joensuu	Lappeenranta	Blandford
Length in weeks	6	14	15
Number of lecture and laboratory hours	24/12	28/28	18/36
Number of students	58	129/57 (A/B courses)	19/5 (BSc/MSc students)
Team size for course work	Individual assignments	Individual assignments	Pairs and groups of 5-6 students
Target groups	Computer science major or minor students with no previous programming courses	All freshmen needing knowledge of programming	BSc (Hons) in Telecom Systems Engineering & MSc in Communications and IS Management
Course contents	Basic programming concepts; variables and their roles; no abstraction mechanisms	Basic programming concepts; variables and their roles; introduction to abstraction mechanisms	Basic programming concepts; variables and their roles; introduction to abstraction mechanisms including objects & inheritance
Programming paradigms	Imperative programming	Procedural programming	Procedural and object oriented programming
Previously used language	Java	C	Delphi
Other central ideas	Enjoy learning programming	Programming process; quality assurance	Programming process; quality assurance
Roles of Variables	Roles used as a starting point for program design. Lectures designed based on the emergence of roles	Introduced as an aid in understanding the different uses of variables in programming	Introduced as an aid in understanding the different uses of variables in programming
Animation	PlanAni demo & individual study during lab sessions	Quick PlanAni demo & individual study	Not used
Observed changes /improvements with Python	Students experienced the joy of discovery and success	Positive and excited feedback about programming with Python	Material more straightforward, students learn more
Observed problems with Python	range()-construct complicated and unique to Python	File handling using objects & methods	File handling using objects & methods initially confusing
Dropout rate	16%	34%/39% (A/B courses)	Nil
Exam failure rate	2%	2%/9% (A/B courses)	Nil

Roles and Python

Roles had previously been used to teach introductory Pascal and C programming. The authors expected that roles would be useful in teaching introductory Python classes, an expectation which has proven entirely correct. The authors also expected that the roles classification would carry over straightforwardly into Python. This expectation has proven largely correct, but the existing classification needed a few small changes to make it a more natural fit for Python. The roles *most-recent-holder*, *most-wanted-holder*, *gatherer*, *follower*, *one-way-flag*, *temporary* and *walker* are exactly the same. The roles *fixed-value* and *stepper* were slightly modified, and it is not clear

whether there is enough difference between *containers* and *organisers* in Python to make a useful distinction.

Taking *fixed-value* first, consider the following program fragment:

```
pii = 3.14
r = input("Enter the radius of a circle: ")
print "The area of the circle is", pii*r*r
```

By definition both variables `pii` and `r` are *fixed-values* even though `pii` is assigned with a literal value and `r` is obtained as input. Python does not allow us to declare that a variable is a constant which raises a temptation to reserve the role *fixed-value* for that purpose in Python. This raises a problem because in other languages *fixed-value* refers to variables that are assigned to only once, and now in Python a new role name would be required. A possible solution is to add a new role name, *constant*, and use it in Python to denote *fixed-value* variables that are assigned with a literal value and use the role name *fixed-value* for variables whose value depends on input but does not change after initialization.

In Python it is very easy to write programs using lists of data:

```
animals = ["aardvark", "albatross", "bee",
           "antelope", "zebra", "wombat",
           "lion", "giraffe", "elephant",
           "hippo", "rhino", "frog"]
```

We say that the variable `animals` is also *fixed-value*, provided it is not changed by any subsequent code in the program, even though `animals` is a list, and hence is mutable. It is not regarded as a *container* or *organiser*, unless elements are actually added, removed, or rearranged. In other languages it may be that the only way to dynamically create a list or array is to make an empty *container* and then add elements to it. However, it would be foolish to insist that Python lists always be regarded as *containers* merely because of the foibles of other languages.

The most significant change in classification when moving to Python was between what was called a *stepper* and what a *most-recent-holder*. In prior work, most of the variables considered to be *steppers* were integer variables, incremented explicitly each time around a loop. However, restricting *steppers* to only such variables would misclassify most loop variables in Python programs. Consider, for example, the following Python code which is a literal translation of an earlier Pascal program illustrating a *stepper*:

```
multiplier = 1
while multiplier <= 10 :
    print multiplier, "* 3 =", multiplier*3
    multiplier += 1
```

Although the variable `multiplier` certainly is a *stepper* in this example, the above code fragment is not idiomatic Python. A Python programmer would write that piece of program more like this:

```
for multiplier in range(1, 11):
    print multiplier, "* 3 =", multiplier*3
```

In this example we want to call `multiplier` a *stepper* too, because it is basically the same code, just better written. But this raises a problem, because `range(1, 11)` is actually a function call which returns the list [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. The variable `multiplier` simply refers to each element of the list in turn. This is a common pattern in Python.

For example, consider the following code:

```
animals = ["aardvark", "albatross", "bee",
           "antelope", "zebra", "wombat",
           "lion", "giraffe", "elephant",
           "hippo", "rhino", "frog"]

for animal in animals:
    print animal, "has", len(animal), "characters"
```

We want to call `animal` a *stepper* in this for-loop as well. Because a for-loop can iterate easily through lists containing any type of element, this is how loop variables in Python for-loops usually are used. Unlike Pascal or C, it is rare to have a for-loop that iterates through the subscripts to a list and then uses those subscripts to access the elements. It is far easier to iterate through the elements directly.

The logical conclusion of this argument is a revised classification of *steppers* in Python:

- a variable is always a *stepper* if it is the loop variable of a for-loop (or list comprehension), and
- sometimes *steppers* appear in particular circumstances involving while-loops. (But those other circumstances are rarely seen in practice.)

This means that in many loops, variables which would be classified as *most-recent-holders* in other languages will be *steppers* in Python. This reclassification is actually a realistic reflection of the nature of for-loops in Python: it is natural to separate out in one place the code which generates a sequence of items (e.g., `animals = […]`); then to use a for-loop in another place to actually process those items (`for animal in animals: …`). This separation of concerns does lead to simpler, more understandable code in both places and it seems right that the roles classification should be tailored to reflect this Python practice.

The authors found that the roles *organiser* and *container* were less obviously different in Python than in other languages; students also found the distinction somewhat subtle. The main form of "organisation" that you want to do to an *organiser* is to either sort the elements into order, or to reverse them. Both of these operations are built-in methods of Python lists, and hence trivially easy to achieve. Each data structure requires a construction phase which is typically implemented by a loop. Thus the construction part (where the data structure is *container*) is non-trivial whereas the re-organisation part (where the data structure is *organiser*) is trivial to implement even though more important for the functionality of the program. Thus, it is not that there is no difference in usage between an *organiser* and a *container*, it is just that the difference is not clear enough to justify two separate roles. Thus we suggest that only *container* is used with Python.

Conclusion

We have described how introductory programming education has been improved in three university-level courses. This has resulted in increased student satisfaction, lower drop-out rates, and better programming skills. The three courses are very different in their length, covered topics, and background of audience. The common elements in the changes were the adoption of Python as the first programming language and the introduction of roles of variables as an aid in program creation and comprehension.

Experiences with Python have been very similar at all three educational institutions. Students are able to write working code from the very beginning, and they find programming easy and interesting. In all three institutions Python has been found to be a better choice than the language used previously: Java at Joensuu, C at Lappeenranta, and Delphi at Blandford. This opinion is shared by both teachers and students.

Experiences with roles vary among the institutions. When roles were used extensively during the course, the teachers found roles helpful to both themselves and students. With more limited use of roles, students were able to use role names to talk about programs and to clarify the use of variables. However, students were unable to use roles as a tool in program design. These findings suggest that roles are of most help to students if they are uniformly employed in all aspects of teaching: during lectures, when discussing program design, and especially when elaborated with role-based animation.

The progression to a second programming language is an important open question. In the first year, students' programming knowledge is fragile and introduction of a new language may interfere with programming skills. Further work is required to establish how best to build on the material from the first course when introducing other languages, for example Java.

References

Abelson, H., Sussman, G., & Sussman, J. (Eds.). (1996). *Structure and interpretation of computer programs* (2nd ed.). Cambridge, MA: MIT Press.

Byckling, P., & Sajaniemi, J. (2006). Roles of variables and programming skills improvement, *37th SIGCSE technical symposium on computer science education (SIGCSE 2006)* (pp. 413-417). Texas, Houston, USA: Association for Computing Machinery.

Donaldson, T. (2003). *Python as a first programming language for everyone.* Paper presented at the Western Canadian Conference on Computing Education, , 1-2 May 2003, Courtenay, BA, Canada.

Downey, A., Elkner, J., & Meyers, C. (2002). *How to think like a computer scientist: Learning with Python*. Wellesley, MA: Green Tea Press.

Ehrlich, K., & Soloway, E. (1984). An empirical investigation of the tacit plan knowledge in programming. In J. C. Thomas & M. L. Schneider (Eds.), *Human factors in computer systems*. Norwood, NJ: Ablex Publishing Co.

Guzdial, M. (2003). A media computation course for non-majors. In D. Finkel (Ed.), *8th Annual conference on innovation and technology in computer science education* (pp. 104-108). Thessaloniki, Greece: ACM Press.

Herrmann, N., Popyack, J. L., Char, B., Zoski, P., Cera, C. D., Lass, R. N., et al. (2003). Redesigning introductory computer programming using multi-level online modules for a mixed audience, *34th SIGCSE technical symposium on computer science education* (pp. 196-200). Reno, Nevada, USA: ACM Press.

Joint Task Force for Computing Curricula. (2001, 15 December). *Computing curricula 2001 computer science*. Retrieved 29 November 2006, from http://www.acm.org/education/curricula.html

Kasurinen, J. (2006). *Python programming guide version 1 (In Finnish)* (Manual No. 7). Lappeenranta, Finland: Lappeenranta University of Technology.

Knuth, D. E. (2005). *Art of computer programming, Volume 1, Fascicle 1, The: MMIX -- A RISC computer for the new millennium.* Addison Wesley Professional.

Lefkowitz, R. (2005). The semasiology of open source (part 2). Talk at *O'Reilly open source convention held in Portland, Oregon, August 1-5, 2005.* Retrieved 15 November 2006, from http://www.itconversations.com/shows/detail662.html

Nagappan, N., Williams, L., Ferzli, M., Wiebe, E., Yang, K., Miller, C., et al. (2003). Improving the CS1 experience with pair programming, *34th SIGCSE technical symposium on computer science education* (pp. 359-362). Reno, Nevada, USA: ACM Press.

Pane, J. F., & Myers, B. A. (1996). *Usability issues in the design of novice programming systems* (No. CMU-CS-96-132). Pittsburgh, Pennsylvania: School of Computer Science, Carnegie Mellon University.

Perkins, D. N., & Martin, F. (1986). Fragile knowledge and neglected strategies in novice programmers. In E. Soloway & S. Iyengar (Eds.), *Empirical studies of programmers* (pp. 213-229): Ablex Publishing Co.

Prechelt, L. (2000). An empirical comparison of seven programming languages. *Computer, 33*(10), 23-29.

Python Software Foundation. (2006). *Python programming language*. Retrieved 29 November 2006, from www.python.org

Radenski, A. (2006). "Python first": A lab-based digital introduction to computer science, *11th annual SIGCSE conference on innovation and technology in computer science education* (pp. 197-201). Bologna, Italy: ACM Press.

Reges, S. (2006). Back to basics in CS1 and CS2, *37th SIGCSE Technical Symposium on Computer Science Education* (pp. 293-297). Houston, Texas, USA: ACM Press.

Rich, L., Perry, H., & Guzdial, M. (2004). A CS1 course designed to address interests of women, *35th SIGCSE Technical Symposium on Computer Science Education* (pp. 190-194). Norfolk, Virginia, USA: ACM Press.

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education, 13*(2), 137-172.

Sajaniemi, J. (2002). An empirical analysis of roles of variables in novice-level procedural programs, *IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC02)* (pp. 37-39). Arlington, Virginia, USA: IEEE Computer Society.

Sajaniemi, J. (2006). *Roles of variables homepage*. Retrieved 17 November 2006, from http://cs.joensuu.fi/~saja/var_roles/

Sajaniemi, J., Ben-Ari, M., Byckling, P., Gerdt, P., & Kulikova, Y. (2006). Roles of variables in three programming paradigms. *Computer Science Education, 16*(4), 261-279.

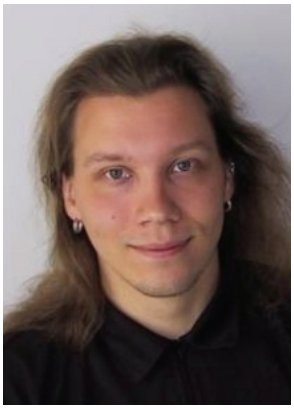
Sajaniemi, J., & Kuittinen, M. (2005). An experiment on using roles of variables in teaching introductory programming. *Computer Science Education, 15*(1), 59-82.

Sajaniemi, J., & Navarro Prieto, R. (2005). Roles of variables in experts' programming knowledge. In P. Romero, J. Good, S. Bryant & E. A. Chaparro (Eds.), *17th annual workshop of the psychology of programming interest group (PPIG 2005)* (pp. 145-159). University of Sussex, Brighton UK: University of Sussex, UK.

Biographies

Uolevi Nikula received the degree of Doctor of Science (Engineering) in 2004 from the Lappeenranta University of Technology, Finland. Since 1999 he has been working at the Lappeenranta University of Technology as a researcher and senior assistant in software engineering. His main research interests include software process improvement, requirements engineering, and diffusion of innovations both in industry and in education. Before returning to academia he was working as a programmer, software specialist, and project manager in industry over five years.

Jorma Sajaniemi received the Licentiate degree in computer science in 1975 from the University of Helsinki, Finland. Since 1979, he has been associated with the Department of Computer Science at the University of Joensuu, Finland, as an associate professor and a full professor. He has obtained industrial experience in Softplan Ltd and Karjalan Tietovalta Ltd. His main area of research is psychology of programming and he has focused on mental models and cognition-based tool support in programming.



Matti Tedre received a PhD degree in computer science in 2006 from the University of Joensuu, Finland. Since 2002 he has been working in the Department of Computer Science at the University of Joensuu as an assistant, researcher, and lecturer, and spent two years in South Korea visiting the universities of Yonsei and Ajou. Earlier he has worked as a programmer and as a software analyst. His research interests include social studies of computer science, the history of computer science, and the philosophy of computer science.

Stuart Wray received a BA in computer science in 1981 and a PhD in computer science in 1986, both from the University of Cambridge, U.K. Since then he has worked in research, at ORL and the University of Cambridge Computer Laboratory, and in product development, at Virata, Marconi and BAE Systems. He is currently a senior lecturer at the Royal School of Signals in Blandford.