BACHELOR'S THESIS, FINAL REPORT

PYTHON AS A PROGRAMMING LANGUAGE FOR THE INTRODUCTORY PROGRAMMING COURSES

Jussi Kasurinen Tite5

Student number 0237500

2006

ABSTRACT

Lappeenranta University of Technology
Department of Information Technology

Jussi Pekka Kasurinen

Python as a programming language for the introductory programming courses

Thesis for the Degree of Bachelor of Science in Technology

2006

39 pages, 8 tables, 4 figures, no appendices

Examiner: D.Sc. Uolevi Nikula

Keywords: computer science, programming, Python, introductory level, CS1, CS0

This thesis concentrates on the major issue on the field of basic computer science education, programming. Thesis takes a look on the introductory level education, and tries to seek out common problems amongst the courses. We also search for some ways to overcome these problems by using suitable tools to ease learning process, and search for technological accessories we could use to make things easier to comprehend.

This thesis starts from the hypothesis that the Python as a first programming language could possibly give us some edge over the traditional teaching languages such as C, C++ or Java. We will compare Python to these languages, and study what sort of benefits we would gain when we teach Python to the first year students, who generally have just a minimal if any experience on the programming. This thesis also studies what should be included to the course structures, and what in general should be taught on the first programming course, with some insight on the technical approach to the world of programming.

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto
Tietotekniikan osasto

Jussi Pekka Kasurinen

Python hyödyntäminen ohjelmoinnin perusopetuskielenä

Kandidaatintyö

2006

39 sivua, 8 taulukkoa, 4 kaaviota, ei liiteittä.

Tarkastaja: TkT Uolevi Nikula

Hakusanat: ohjelmointi, perusopetus, Python, ohjelmoinnin perusteet
Keywords: computer science, programming, Python, introductory level, CS1, CS0

Tämä kandidaatintyö tutkii tietotekniikan perusopetuksessa keskeisen aiheen, ohjelmoinnin, alkeisopetusta ja siihen liittyviä ongelmia. Työssä perehdytään ohjelmoinnin perusopetusmenetelmiin ja opetuksen lähestymistapoihin, sekä ratkaisuihin, joilla opetusta voidaan tehostaa. Näitä ratkaisuja työssä ovat mm. ohjelmointikielen valinta, käytettävän kehitysympäristön löytäminen sekä kurssia tukevien opetusapuvälineiden etsiminen. Lisäksi kurssin läpivientiin liittyvien toimintojen, kuten harjoitusten ja mahdollisten viikkotehtävien valinta kuuluu osaksi tätä työtä.

Työ itsessään lähestyy aihetta tutkimalla Pythonin soveltuvuutta ohjelmoinnin alkeisopetukseen mm. vertailemalla sitä muihin olemassa oleviin yleisiin opetuskieliin, kuten C, C++ tai Java. Se tarkastelee kielen hyviä ja huonoja puolia, sekä tutkii, voidaanko Pythonia hyödyntää luontevasti pääasiallisena opetuskielenä. Lisäksi työ perehtyy siihen, mitä kaikkea kurssilla tulisi opettaa, sekä siihen, kuinka kurssin läpivienti olisi tehokkainta toteuttaa ja minkälaiset tekniset puitteet kurssin toteuttamista varten olisi järkevää valita.

TABLE OF CONTENTS

ABSTRACT 1
TIIVISTELMÄ 2
TABLE OF CONTENTS 1
1 INTRODUCTION 1
2 BACKGROUND OF THE THESIS 4
2.1 Discussion concerning programming languages 4
2.2 Terminology 7
2.2.1 Integrated development environment 7
2.2.2 Virtual learning environment 7
2.2.3 Submission system 8
2.2.4 Tutoring system 8
2.3 Situation at fall 2005 at LUT 9
2.4 Summary and preliminary plans 13
3 THE NEW FRAMEWORK 14
3.1 Analysis of the collected data 14
3.2 Literature survey 15
3.3 The course design 16
3.4 Selection of the programming language 17
3.4 Realization of the course 25
3.5 Analysis of the course created 29
4 CONCLUSIONS 30
5 REFERENCES 32

1 INTRODUCTION

All over the world universities and colleges have faced the same problem: non-majors on the computer science classes complain about the increasing difficulties regarding the basic programming courses. And for that matter, those who are computer science majors are not doing that well either compared to the other introductory courses on the other majors such as the economics or the industrial engineering. Of all the students who study programming, generally half of them cannot seem to pass even the basic courses, and even when they do, they usually get quite appalling grades from them.[6] This unfortunate situation is truly a world wide phenomenon, and can also be observed at the Lappeenranta University of Technology (LUT).

Some universities, e.g. Louisiana State University, have created a first introductory course for the computer sciences called “Computer Sciences 0 (CS0)” to tackle this problem [1,7]. These pre-introductory courses tend to use flowcharts and simplified UML-models to introduce concept of algorithms and programs to the students, but even then the students have problems when they try to learn to actually program by themselves. This leads us to think that maybe the problem is not so much the concept, but the actual programming itself [1,7]. It seems that the learning should be reinforced with some sort of suitable tools that make the first steps of programming easier to comprehend [1]. Of course most of the actual programming tools are expensive and usually good integrated development environments (IDEs) are overly complex and have a tendency to be expert-oriented. In some cases, even the language’s own internal syntax is unnecessarily complicated for a novice [1].

As a result, some entry-level IDEs have been developed, but those environments usually aren’t extensive enough to be feasible on the long run [18]. They also

have the same constraints as all IDEs working on the compiler basis: students cannot see what their program is going to do until they get the code compiled. So is it sensible to stay with the compiler-driven languages, which requires user to be able to cope with structural elements even before he or she even gets the first 'Hello World' application to work?

In this situation help comes from the selection of the teaching appliances. First of all, teachers should be able to provide students with the best available tools to make sure that studying is not unnecessarily harder than it should be. Also, from the economical point of view, studying tools should be as cheap as reasonably possible, so that they would be affordable by even the smallest of the educational institutions and the students themselves. But at the same time these tools should have high enough quality and functionality to be actually sensible to use.

Pedagogic studies have shown that many underlying issues can contribute to the personal learning process. Even if we don't think of it as a contributing factor, introducing too many subjects too soon, like object-orientation or dynamic memory handling, can be frightening at first to those who do not understand them. These studies clearly indicate that different approaches and different teaching methods contribute greatly to the learning factor of the courses.[6,17] These teaching methods include things like selection of the teaching appliances, such as programming language or the learning environment. If students can stay on the environment they are accustomed to, they will be focusing on the actual programming itself. If they have to use some unfamiliar programming environment, the learning of the environment will divert their attention, and all that is away from the programming.[7] Another important issue is also the feeling of getting something to work. At first, people should be able to overcome easy problems so that they can overcome difficult ones in the future.[7] This, of course, means that we should try to avoid unnecessary challenges during the early learning process. And by doing that, we may be able to improve the programming

capabilities of the vast majority of the students who participate on the course and pass it, as well as get some improvements on the grades.

Structure of the thesis is as follows: first, we are going to discuss about the selection criteria for the programming language and then take a look at the terminology that is used in this thesis. After that we take a look at the current situation at the Lappeenranta University of Technology and analyze some statistical data from the previous year.

On the third chapter we create a course framework based on the collected data and the additional literature survey on the proposed solutions for the course structure. On to the next section, where we take a look at the different programming languages, we conduct a theoretical comparison between some of the most prominent programming languages, including C, C++, Java and Python. After that, we introduce our proposal for the introductory level programming course and describe some of the major changes compared to the earlier designs. In the last chapter we also say a few words concerning the thesis itself and give some preliminary results from the actual course. Also a few words about the future development needs and concerns are considered.

2 BACKGROUND OF THE THESIS

Theoretical approach to this thesis is tied with the facts about teaching programming to the first year students, which of many are non-majors on the computer sciences. For example, Guzdial [7] and Agarwal [1] discuss this kind of situation, and both agree that traditionally teaching programming to the novices is a hard task to accomplish, not just because it is just a difficult concept to grasp, but also because most of the time we do not apply right tools for the job. Also the theorems concerning programming vary from several differently prioritized bottom-up-methodologies to the completely opposing top-down-systems. Even the Joint Task Force on Computing Curricula describes six different approaches, that all have some strong points and drawbacks. [3]

2.1 Discussion concerning programming languages

First part of the thesis revolves around the pros and cons of the programming languages we might want to use to teach programming. Of course, we can argue over the best programming language as long as we like because that is to some extend a matter of a personal preference. But if we take a look at the target group – the novice programmers – we can try to find a language that gives best support to the majority of them and fits their needs.

So how should we select the teaching language for the introductory programming course? The most common answer would probably be to select the most common and modern language available, but in the case of novice programmers, there are some other concerns. First and foremost should be simplicity. This would mean that the programming language has to be simple enough so that one could learn the basics of the language, such as the variables, the control flow methods or I/O-operations without much of an overhead that comes with the syntax and notation.

And on the other hand, simplicity should not be a constraint when we advance to the more complicated aspects.

Another important factor should be availability. We might have some good and easy development environments available at the university, but how about those people who want to -or has to- study at home? It is important to be able to give some common, publicly available tools that are practical, and in best case affordable or even free of charge. And if we consider this from the viewpoint of the educational institutions, it probably would not hurt to have tools that do not come with some sort of licensing fee. If the tools are available for several operating systems, the platform would not cause problems, which would also be a positive feature.

Last factor that contributes to the equation is the credibility. There are some development environments that use customized programming languages to teach the basics to the students, but these environments quickly become obsolete when people advance to the more complex aspects. Credibility factor also reflects to the smaller programming languages, which may have potential for being complete languages, but unfortunately lack the support from the software industry.

So in general the test group of programming languages for this thesis is selected so that they represent the most probable candidates for the introductory-level programming languages. More precisely, the preliminary criteria for inclusion was that the language had to fulfill the above requirements and should be either a popular and commonly used programming language such as Java or well established, matured language that has previously been proposed as a first programming language in the several earlier studies like Python.

So what programming languages did we select besides Python? The most common group of comparison, C, C++ and Java is a generally good group of

samples because each of them has some unique reasons to begin with. For example, C is the traditional choice, with C++ for those who wish to start with object-orientation. The Java is also selected because it's modern and flexible language, but also because it is usually the language people see as a new standard. These three were also important because they were our most probable non-Python choices. C was used earlier in our introductory courses when C++ was also usual and well-established language within our courses. Java was already in use on some advanced programming courses and therefore also potential choice.

On the other hand, Python is an interpreted language, whereas the before mentioned languages are compiler-based, so direct comparison between them was unfair. For this reason Perl, Rexx and Tcl [15] were recognized in the list to provide additional reference points in the case of interpreted languages against compiler-based languages in general. In the literature survey we even discovered that Matlab was represented on some of the earlier comparisons [5] even though it is not really a pure programming language in traditional sense.

2.2 Terminology

The course also applied a learning environment that provided exercises for the students. Therefore we should introduce some terminology on programming environments and teaching systems before advancing to the actual thesis. There are several different kinds of software-based teaching systems that are designed to either ease the programming process itself or offer some sort of aid to the struggling programmer. In this thesis we will mainly follow the categorization used in Rongas *et al.* [18] publication "Classification of tools for use in introductory programming courses". Their publication gave following description for categories of teaching appliances.

2.2.1 Integrated development environment

Integrated development environment (IDE), are usually a complete set of software tools that form a full interconnected system that allows seamless software development process. These environments commonly come with at least editor, compiler or interpreter, debugger, and a set of pre-created libraries for easier accessibility to application protocol interfaces. These programs are not designed to be used as a teaching appliance, but they usually offer some additional support and automation to the programmer. However, these environments are usually flooded with expert-oriented tools and concepts, with somewhat little concern or guidance to the novice programmers. For example IDLE (for Python) and Eclipse (for Java) are both considered to be IDE:s.[18]

2.2.2 Virtual learning environment

Virtual learning environments (VLE) can be considered somewhat teaching-oriented systems, which hold all the tools teacher and students may possibly need in their courses. They usually have direct teaching-oriented functionalities, such

as submitting assignments, automated feedback processes, online editors and repositories for course-related data. Usually they also provide some sort of communication mechanisms between students and teachers, which is usually realized via a messenger service or forum. For example, Viope is a textbook example of such a system. [18]

2.2.3 Submission system

Submission systems can be considered somewhat limited versions of the VLE:s, because they usually just consist of submission and evaluation systems. In practice this means that submission systems only collect deliverable inputs of the students, evaluate their answers and mark some sort of scores to those who have given acceptable answer. Normally these kinds of systems are created to replace email- or paper-submissions from courses. They normally also exist as a part of the VLE:s, and carry little to no extra value to the learning factor. [18]

2.2.4 Tutoring system

Tutoring systems have two distinct variations: static and dynamic systems. Common to both of these types is that they aid students not by giving them exact information on what to do, but merely hinting and aiding students towards the actual solution just like tutoring people would do. Difference is, that static systems, often called *Constrain based models*, usually have large amounts of pre-recorded libraries of scenarios, from which they select the most appropriate one to pass on to the student, while dynamic systems, also called *model tracers*, analyze given answers and generate tips from their knowledge databases. Tutoring systems usually feature some sort of user interface to give inputs and the actual tutoring system, that is concentrated to some sort of server application. [10] However it should be noted that some systems also work on the basis that they give students opportunity to choose category they wish to get hint from, and by that way force students to actually consider what they have done wrong. [8]

2.3 Situation at fall 2005 at LUT

At the starting point for the thesis work, Lappeenranta University of Technology offered four programming courses that were meant for most of the students who are required to take programming classes. These courses were:

- Fundamentals of Programming A and B; both used C as a programming language. A was meant for the students that were supposed to continue programming courses or wanted to have a deeper insight on programming, such as Computer Science or Electrical Engineering majors. B was for the people who have only one mandatory programming course. They shared same lectures and were run simultaneously at first fall semester.
- Data structures and C, which was C-based advanced course on programming. Usually enrolled students were computer science -majors or -minors or read some similar major like digital techniques. Course was meant to be taken at first year’s spring.
- Fundamentals of Object-Oriented Programming, a C++ course, and Design of Algorithms, a C course, which both were second year courses designed to be taken only after first two. They both were compulsory to the majority of information technology students and also have some non-majors from time to time.

Basically this thesis focuses solely on the first course, Fundamentals of Programming. To actually realize a problem, we should take a look at the course participant statistics, and to the final grades given at the last year course that were collected from last year’s course.

Action Completed	A	B	Totals
Registration	170	79	249
Initial Questionnaire	138	60	198
Exercise assignments	114	51	165
Project assignment	69	-	69
Grade given	60	29	89

Table 1: Course completion statistics at the course Basics of Programming, Fall 2005 at LUT

As we can see from table 1, of the 249 students who started the course at September, only 89 were able to pass the course. And if we add to these statistics the grade information from table 2, we can make some interesting observations.

- Withdrawal or failure -group (WF-group) over all students were 64,3%
- WFD (withdrawal, failure or D) – rate 68,3% (D corresponds to grade 1)
- Grades at course A generally were above the average; mean value being 3,4, whereas at course B mean value was only 2,3.
- Of those who completed the course, 38% got a less-than-mediocre grade of 1 or 2 (D or C- correspondently).
- Of those who were able to pass the exercises at VLE, only 60,5% passed the project.

Given grade	Course A	Course B
1	5	5
2	9	15
3	19	5
4	9	4
5	18	0
Total	**60**	**29**

Table 2: Grades given at the course Basics of Programming, Fall 2005 at LUT

We also did a survey on the base level of the programming skills to those who participated on the course. This survey was sent to every student that had enrolled to the course, but was conducted as a voluntary part that gave one extra point to the respondents. Nearly four fifths of the course participants answered the questionnaire. Following data was collected:

Earlier participation on a programming course:	**Course A**		**Course B**	
	Count	Percentage	Count	Percentage
Never	38	25,7 %	14	25,0 %
Not previously at LUT	55	37,2 %	15	26,8 %
previously at LUT	55	37,2 %	27	48,2 %
Total	148	100 %	56	100 %

Table 3: Initial questionnaire result from the course Fundamentals of Programming, LUT 2005

As table 3 shows, the vast majority of students - 74,4 % at course A and 75 % at course B - has had at least some sort of previous background in programming. Of course, most of the people who chose "previously at LUT" come from previous years WF-groups. Another thing we asked on the survey was that in which year they participated first time to this course or its predecessors. This gives us some sort of an idea on how long it takes from an average student to pass the course successfully. And even if the student doesn't try the course every time he had a chance to do so, the survey gave us an approximate on how many times he has had a change to try it:

First year of participation			
Course A		**Course B**	
earlier than 2000	7	earlier than 2000	6
2000	1	2000	7
2001	11	2001	6
2002	11	2002	6
2003	31	2003	15
2004	32	2004	14
2005	76	2005	23
Total	169	Total	77

Table 4: First year of participation to the course Fundamentals of Programming, LUT 2005

Noticeable is the fact that 24,6 % of the course B's participant were at least fourth year students, earliest entry being at 1992.The same number at the course A was only 11,2 %, but still far over the reasonable level. Also the fact that on both courses the newcomers were a minority was somewhat concerning.

So why are people dropping these courses? We conducted separate survey on the year 2005's WF-group to find out main reasons why they thought that the course failed. This survey received total of 53 responses, citing that the most important reasons to drop out were overall difficulty, inability to find time to do all of the required tasks, schedule conflicts with other courses, own laziness and too few study credits to justify the workload.

2.4 Summary and preliminary plans

In this case we found, that there are many reasons to the generally weak results. We also concluded that there are no one or two main reasons that could be fixed, but the situation is more likely a result from many small problems that led to the current situation.

Our problem analysis led us to separate three stages for the improvement project: First, we would do a literature study to see what sort of research has been done on the similar problems and try to pinpoint reasons why programming course fails. Secondly we would try to find a reasonable solution that covers most of these problems and see if the change of programming language or environment would be reasonable. Finally, on the third stage we would build a design for completely new course structure that implements the theories and previously acquired data from literature and see if it can address the identified deficiencies and issues.

3 THE NEW FRAMEWORK

3.1 Analysis of the collected data

At the first stage we started the project with analyzing the data we already had collected and saw what we could make of it. At first we concluded the following hypothesis:

- Course project assignment was too hard to complete with available skills. Although it did only consist parts that were discussed at the lectures or practiced in exercises, the 39,5 % failure rate at this point pointed out that some changes had to be done.
- Lecture participation percentages were too low. The lecturer estimated that the initial group of over 200 participants decreased rapidly to the core group of less than fifty listeners. Also the exercises were suffering from the same kind of problem.
- Workload itself was in fact in scale when compared to the Computing Curricula 2001 templates or to the local study credit calculations.
- Students seemed to be somewhat divided to two main categories: those who actually understood the programming environment and basic aspects - and passed the course with a good grade - and those who had problems and ended up to the WFD-group. This phenomenon was clearly visible at course A; mean grade of 3,4 is extremely unlikely to be a result of a natural statistical distribution.

With these observations we decided to do a literature survey that would help us find some relevant information on similar situations. On early stages we also decided to actually focus on the two main possibilities: keeping the C as a

teaching language or adopting Python if enough substantial evidence were found to justify the adaptation work.

3.2 Literature survey

Literature survey was conducted over the period of three weeks and gave us a insight to the possible solutions. First, as mentioned in the introduction, some universities and colleges have introduced a entry-level pre-programming course called "Computer Science 0" [1,20,23]. For example Louisiana State University [1] and State University of West Georgia [21] had just applied Python to their basic programming courses with little to no changes over the traditional computing courses. On the other hand places like Centre College at Kentucky [20] and Georgia Institute of Technology (GIT) [6,7] had completely redesigned their programming courses to promote popular aspects of the programming, such as web-surfing, image manipulation or sound manipulation. Although at the GIT the course was designed for the non-majors. Both approaches gave promising result, with WFD-rates decreasing and overall results shown on table 5 showing that generally students were achieving better results.

Course	WFD-rate	Don't like it /Nothing	Enjoy content	Content is useful
Trad. CS1	43.0 % (27.8 %)*	18.2 %	12.1 %	0.0 %
Engineering CS1	19.0 %	12.9 %	16.1 %	25.8 %
Media Comp.	11.5 %	0.0 %	21.3 %	12.4 %

Table 5: Course comparison between three different CS1 courses at GIT (* long time mean value) [16]

What was interesting at the GIT's research was that their initial situation was similar to ours. Their traditional programming course was performing poorly with a WFD-rate of 27,8 % [17]. Also the general atmosphere around the course was

negative with non-major students questioning the reason for their compulsory programming course. Their solution was realized with a course "Introduction to Media Computation" that focused heavily to the media and tool manipulation, as opposed to the traditional software development focus. [17,23]

We can also make some interesting observations from table 5: First of all, Media Computation had lowest WFD-rate and it generated 0 "Don't like it"-opinions. Course itself had a two thirds female attendance and total of 120 students, of which only 3 students withdrew. Traditional CS1 had almost half of the class in WFD-category and even the Engineering-oriented course that was especially offered to the CS-majors had worse ratings that the Media Computation.

3.3 The course design

The literature survey gave us facts concerning programming courses that lead to the following corner stones concerning the design process:

- Object-first approach is too complicated to be feasible at the introductory courses. [19]
- Learning of the programming skills can be enhanced with teaching accessories. [18]
- Usually these accessories have to be novice-oriented, as opposed to the usual tools that are used at programming. In many cases these tools are clearly meant to be used by an expert and come with little to no help to the novice users.[4,18]
- Basic problem solving using programming language should be a major objective that comes even before knowledge of data structures and software development. [4,13,18]
- Feedback or some sort of guidance is important to the students. Dynamic feedback is also hindered by the fact that students have to create complete programs before they can get any information from the compiler or the

system. This also causes unnecessary overhead to the early learning experience. [13,18]

- Computing Curricula 2001 for Computer Science supports the idea of introducing programming language as early as possible as a means to solve problems in one of its frameworks. This framework is called imperative-first, and it is a sort of bottom-up-method for teaching programming. [3]

3.4 Selection of the programming language

So we decided to take a closer look at the currently available programming languages. Literature survey surfaced Prechelt's comparisons [15] that had previously given interesting results and at the same time it gave us some good insight to begin with. Overall, the tests were made with the selection of some traditional languages, such as C and Java, plus some others, for example Python and Perl. The initial participants and detailed list of languages can be found in Table 6.

language	programs	unusable	total
C	8	3	5
C++	14	3	11
Java	26	2	24
Python	13	0	13
Rexx	5	1	4
Tcl	11	1	10
Perl	14	1	13
Total	91	11	80

Table 6: Entries in Pretchel's comparisons [15]

The bar *programs* in Table 6 describes how many participants actually returned their solution to the given problem. *Unusable* on the other hand shows how many solutions were rejected either because they did not work at all or gave wrong answers. These added up leads to the bar *total*, which represent how many programs actually participated in the latter tests.

The project they were doing was a simple phone number-to-alphabet-conversion, with some linguistic rules and special cases. It was chosen so that the programmer had to spend some time designing his approach and implementation, but was still suitable and rational to be done with any of the selected languages. As Table 7 shows, these tests were done a few years ago, but still can be considered somewhat valid.

Version numbers of the used software
C GNU gcc 2.7.2
C++ GNU g++ 2.7.2
Java Sun JDK 1.2.1/1.2.2
Perl perl 5.005_02
Python python 1.5.2
Rexx Regina 0.08g
Tcl tcl 8.2.2

Table 7: Versions of compilers and interpreters used in Prechelt's test [15]

Following figures describe what results the tests gave. On the graphics, black dot describes a median value, circles individual results from test subjects, gray bar the middle half (25%-75%) of the results and black line the bounds of results minus top- and bottom ten percent. Checkered line with 'M' in it represents the arithmetic mean and plus/minus one error. The z1000 mentioned in some figures was a standardized data set that was given to all the participating programs to solve.

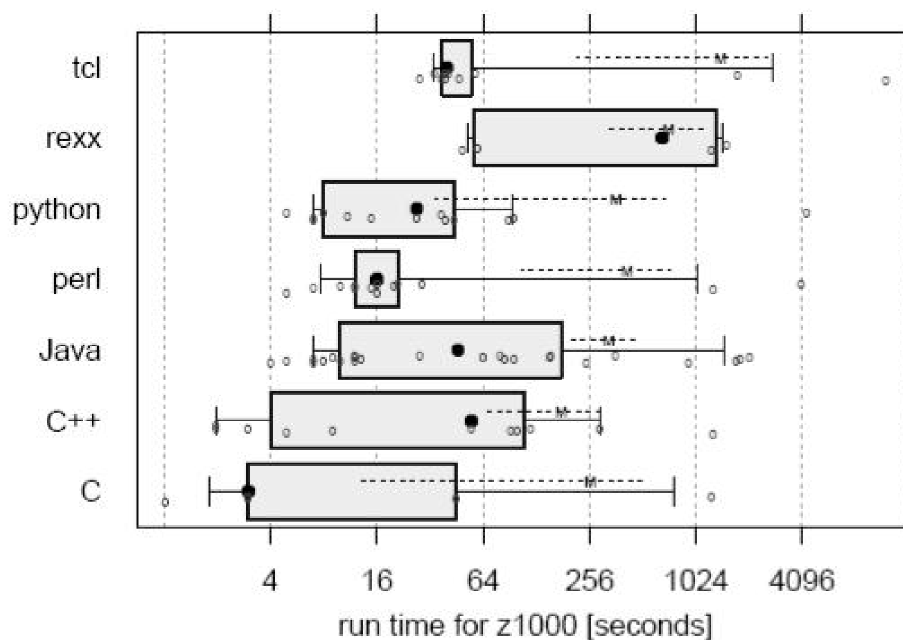

Figure 1: Run time for the programs with standardized z1000-set. [15]

The average program run time on the z1000 data set was roughly between thirty and sixty seconds. Three programs were timed out with no output after about 21 minutes. One Tcl program took 202 minutes. On this test we can see that although C was dominant in the runtime test, the Python actually was quite competitive against Java and C++. And if we concentrate on the middle half (the gray bar), we also see that Python was in fact in the same scale as C.

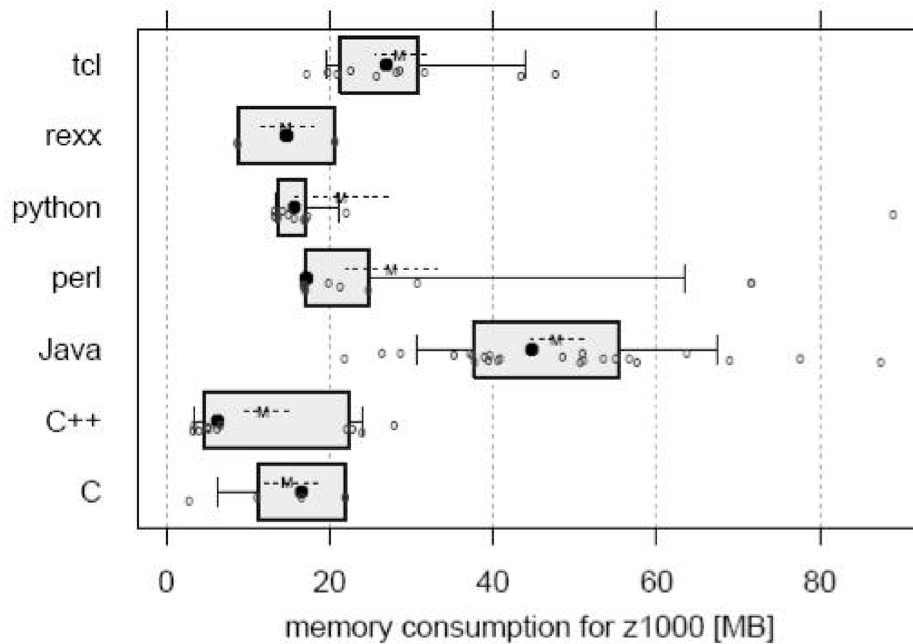

Figure 2: Memory consumption during the z1000 test. [15]

Figure 2 describes the amount of memory required by the program, including the interpreter or run time system, the program itself, and all static and dynamic data structures. Also in this case, Python was competitive with other main competitors. Actually the results were quite good; roughly the same with the C, but only with smaller variance. In this test Java had a big disadvantage because of its runtime environment.

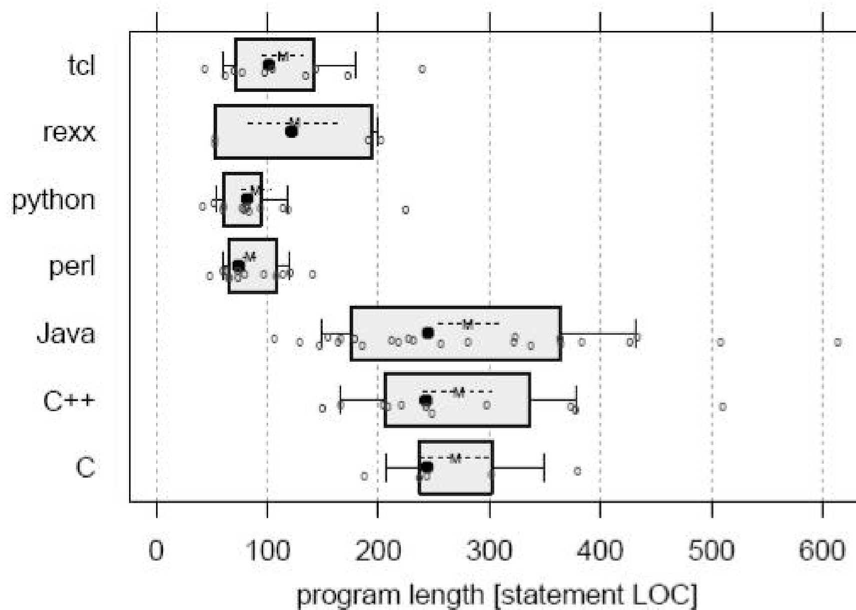

Figure 3: Program length measured in actual lines of functional code. [15]

Figure 3 represents the program length, measured in number of non-comment source lines of code. Once again, Python performed well against the other languages. In this particular case the Python's legacy as a scripting language was the reason why it outperformed Java, C and C++ with such a good margin. As can be seen from the Figure 3, when comparing to the other languages, even the longest script-based solutions are typically shorter than the shortest codes.

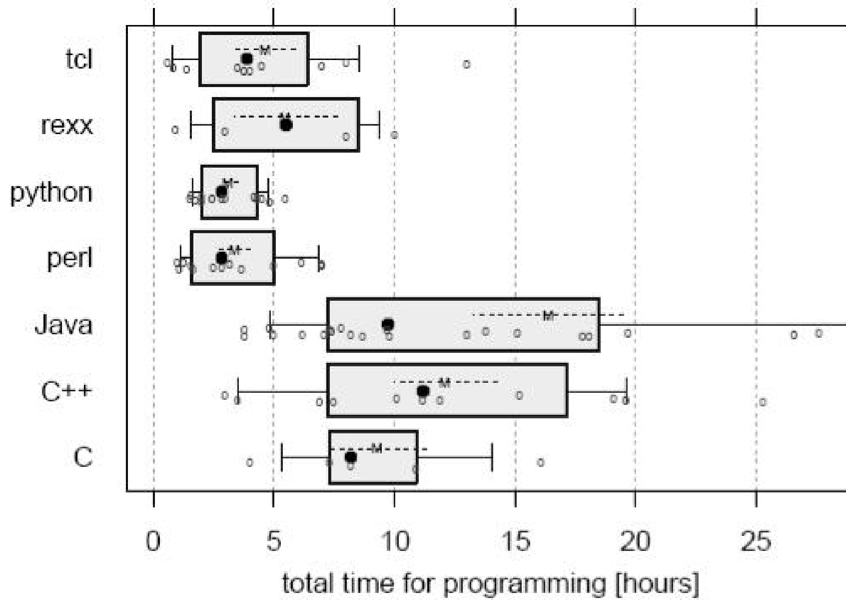

Figure 4: Total time in hours used in making the program [15]

Figure 4 shows total working time for realization of the program. Method for timing the used hours was based on the programmers' own calculations with few exceptions. In script-group the timing relied solely on programmers own approximations of used hours, whereas with non-scripts the experiment was performed in controlled manner. This means that the time used could be calculated from the experiment participation hours by approximating how much of the time the programmer was actually programming. This time again, Python was successful giving one of the best results in test. Of course we can see that the correlation between lines of code and used hours is somewhat strong, but noticeable is that some individual works in Java and C++ were in same time-scale as Python and the script-based languages. This phenomenon unfortunately was visible on the other end of the scale too; total times with Java at 40, 49, and 63 hours are not shown.

The programmers in this study were voluntary participants that were collected from various sources on the Internet. It is also plausible to believe that they all were fairly competent programmers, due to the recruiting processes used in the experiment [15]. It seems that in this scenario with Python, all results are within

reasonable timescale of 2-6 hours regardless of the individual who programmed the code, whereas with Java and C++, the personal abilities and experience contribute greatly to the time used.

What was also interesting was that Python generated no unusable answers. Although the amount of entries was relatively low, these results indicated that Python could compete with other languages. It also suggested that when compared to the more popular languages like C or Java, Python could run in the same amount of resources as efficiently as they, and it also had generally shorter source code implementation which was created in shorter amount of time. Also the processing time and memory usage was proven to be in the tolerable levels. Overall Python as a programming language was competitive in all tested categories against the common C, C++ and Java. [15]

Donaldson [4] had also come up with the similar results as Pretchel. He also suggests that the reason why Python is suitable for the introductory courses comes from its simplicity. Compared to Java, Python has much lighter structure, which can be demonstrated easily with following example:

To create a program that generates phrase “Hello world!” to the prompt, Java uses following source code:

```
class HelloWorld {
public static void main(String[] args)
{
System.out.println("Hello, world!");
}
}
```

Whereas Python comes up with following code:

```
print “Hello World!”
```

It is easy to see why some students feel that object-orientation is hard concept to grasp by novice programmers - the language-generated overhead is astonishing. For example, from above code you could see that the students have to have some sort of an idea concerning classes, static declarations, void functions and system hierarchy even before they can achieve something as easy as printing strings. This problem is also discussed in a paper by the Sajaniemi and Hu named "Teaching Programming: Going beyond "Objects First"" [19], which indicates that object-orientation is a problematic way to approach introductory courses. This lead us to the situation where the most likely feasible scenarios on developing the course were limited to (1) updating the traditional C-based course or (2) seeing if converting to Python would work. The Centre College's - which had similar situation to ours - experiences with Python [20] where quite promising, so we decided to see what possible positive outcomes Python would provide. Overall we came up with the following conclusions:

:

- Python has solid background as an entry level programming language and it can be used in introductory level programming courses. More importantly, there are several authors and studies that support this view. [1,4,13,20,23]
- Interpreter-based programming languages are usually more user friendly due to their possibility to dynamic feedback on partial program code than compiler-based counterparts. [4,11,17]
- Automated memory handling is an advance on the introductory level course. Also the fact that anything more complex involving data structures requires memory handling can be a problem. [11,19,21]

This led us to decide to convert our introductory level courses to Python.

3.4 Realization of the course

So we selected the option of converting our course to Python. This also meant that the topics the course covered where somewhat changed from previous years. Dynamic memory handling, linked lists, pointers and such were replaced with wider range of basic-level topics and some supporting ones such as computing time, debugging and general discussion over programming-related topics. Overall the course was somewhat simplified, mostly because the language-based requirements weren't any longer in such a dominant role.

The actual course design was done with some standards in mind. Most important of these standards were the Computing Curricula 2001 Computer Science Volume [3] that gave us some straightforward standards on what to include to the basic courses. It gave us base platform on the design and dictated that an introductory course should have the same basic topics that would be covered in the course.

Topic	Core Hours reserved
Fundamental programming constructs	10 hours
Algorithms and problem solving	3 hours
Fundamental data structures	2 hours
Machine level representation of data	1 hour
Assembly level machine organization	2 hours
Overview of operating systems	1 hour
Introduction to net-centring computing	1 hour
Introduction to language translation	1 hour
Declarations and types	3 hours
Abstraction mechanisms	3 hours
Foundations of human-computer interaction	1 hour
Fundamental techniques in graphics	1 hours
History of computing	1 hour
Social context of computing	1 hours
Professional and ethnical responsibilities	1 hour
Intellectual property	1 hour
Software design	3 hours
Software tools and environments	2 hours
Software process	1 hour
Total	**39 hours**

Table 8: Computing Curricula 2001's recommendations for topics [3]

Structure described in Table 8 was our basis on designing the course. Noticeable is the fact that this structure has 19 different categories timed for 39 hours of lecturing, whereas we only had 14 lectures - 28 hours- available. By combining related topics such as software design and software process and by narrowing non-crucial subjects such as assembly- and machine level- representation of data, we were able to create a structure that consisted all of the required topics. The final version of the structure for lecture topics and related CC2001 categories are listed on the Table 9.

Week	Lecture topic	Related CC2001 categories
1	Introduction to Python and software tools applied. General discussion on the programming as a profession.	History of computing, Introduction to the language translation. Software tools and environments
2	Basic idea of programming; good structure of the code, basic inputs and outputs, numeric variables.	Fundamental programming constructs, Declarations and types, Foundations of human-computer interaction
3	Formatted output; string variables, slicing and general discussion on the related topics.	Fundamental programming constructs, Declarations and types
4	Source code branching; logical-structures, logical operators and Boolean variables.	Fundamental programming constructs Algorithms and problem solving
5	More on control flow; while- and for-structures. Roles of variables.	Fundamental programming constructs Algorithms and problem solving
6	Functions; splitting the source to the logical parts. Code reusability.	Fundamental programming constructs
7	File operations; Basic I/O-operations with files. Overview on how files work.	Fundamental programming constructs, Fundamental data structures
-	Midterm exam	
8	Data structures; list, manipulating data within structure.	Fundamental data structures
9	Module libraries; importing functionality, command line parameters.	Introduction to net-centring computing, Fundamental programming constructs,
10	Error handling; catching errors, creating exception rules.	Fundamental programming constructs,
11	Bit-level operations; different data-structures, basics of the computer operations. Basics of the memory handling.	Overview of operating systems, Machine level representation of data, Assembly level machine organization
12	Testing; Debugging. Basics of complexity, computing times.	Software tools and environments, Abstraction mechanisms
13	Documentation; defining programs, designing testing sequences and applying requirements.	Intellectual property, Software process, Software design, Social context of computing, Professional and ethnical responsibilities, Abstraction mechanisms
14	Last lecture; what comes on the following courses, what is yet to learn, Introduction of Graphical User Interface.	Software process, Social context of computing

Table 9: Designed lecturing topics with corresponding CC2001-categories. [3]

Another important issue was the weekly exercises. These exercises were created to force the students to actually start working with the programming environments. These exercises were toned down to the level where every participant was expected to be able to complete them with none to little help from teaching staff. This meant that each week we would give out a quiz and three relatively easy programming assignments for students to complete during the

week so that they would actually have to look up the topics discussed on the lectures and at the same time do some programming. Also the exercises were divided to two different styles of exercise groups according to the initial survey done with students. Group 1 had Viope hinting service available at VLE whereas group 2 had assistant-led exercises where they could ask help from teaching staff. This also meant that group 1 was expected to do their exercises at home, when group 2 participated to the weekly non-compulsory exercises. Both groups were allowed to use the same course material and use helpdesk for assistance. This division was done to test difference between self-study and organized studies; results of these tests will be reported on later publications.

Programming exercises were also backed up by the programming manual [9] that was created to give the students some sort of Finnish tutorial on the Python programming which otherwise would not have been available. The manual was also divided to categories following the lecture structure, so that every week student had to learn the next chapter and understand the previous ones. Also the helpdesk-services and hint-based tutoring system that employed the Hume-categorized hints [8] were implemented to the Viope Virtual Learning Environment, [18] which provided us with the submission system for the exercises. If this tutoring system proves to be helpful, it could be integrated as a part of the course in the following years, giving the students additional source of support aside from the assistant-lead exercises.

3.5 Analysis of the course created

Literature survey indicates that the best way to give introductory programming course is somewhat debatable, but there are some frameworks that can be used as a backbone when creating a course. These frameworks propose different approaches, from which we chose the imperative-first-method as our basis. This selection was based on the results, which indicated that Python can be used with "hands-on"-approach to the programming. Also the literature supported this idea [1,4,7,12]. Other addition was the hinting system [8] that was implemented to the course.

Selection of the programming language is for many a matter of opinion, but still some languages can be easier to understand than others. This, in fact, is the basis of this thesis. Our own experiences, while getting into Python and the supporting research material gave us confidence to undergo the whole process of redesigning the course with Python.

This, of course meant also that the exercises were redesigned with learning abilities in mind. This also led to the creation of the beginner's programming manual, so that students would have some sort of reference material with only the relevant information so that abundance of information would not scare them at the first place. Also the weekly quizzes were issued to actually enforce the weekly learning process with compulsory assignments. These assignments were generally designed to be relatively easy, with a slowly increasing difficulty curve, so that they would have the feeling of actually getting something done. We also deliberately designed course so that the last completely new code-oriented topic would come in week 12, giving the students few weeks to finalize their project assignment, which was previously considered difficult.

4 CONCLUSIONS

Summarizing results on this case is difficult due to the fact that at this point we can only offer theoretical results. The original hypothesis that the Python can be useful as a programming language seems to be somewhat debatable subject. However, there are studies which provide convincing support for the benefits of Python. Also the test result from the similar scenarios from places such as Georgia Institute of Technology and Louisiana State University give good references.

The fact that we chose Python as the programming language caused some basic localization problems. After all, Python is quite a new language, so acquiring good course books and other reference material was difficult. In fact, we had to order most of our handbooks and reference material from abroad. In Finnish there was no material available, which led us to compile the programming guide. Although some other Finnish universities use Python as a part on some of their courses, for example University of Lapland and University of Jyväskylä, we still had to create our own guidebook because theirs were aimed at advanced courses that relied on the English material available on the Internet.

The actual results of the redesigning process will be available only after the first run with the Python-based Fundamentals of Programming course is completed. But based on the following observations it seems reasonable to expect some sort of improvements in general scores:

After the first seven weeks into the course it seems that things are going rather well when compared to the last year. Course has 186 active participants, which means that these persons have responded to the preliminary survey. First week the lectures had 117 participants (62,9% of the active group) and at the week eight,

there were still 81 participants (43,5%), which is a significant difference when compared to the last year's situation. Overall, the trend seemed to be somewhere in the magnitude of 60-70 participants per lecture.

At the exercises, 97 participants where assigned to the exercise groups based on their own wishes. Of those 97, at week seven 42 persons (43,3% of the exercise-participating group) visited exercises. Also from the exercise assignments themselves, only 26 people of 186 had either dropped the course (no returned assignments) or is in danger of not meeting the required minimum of 40% returned (so far less than 33% returned). Also in the midterm exams, 147 (79,0%) persons participated.

The future research will study the actual final results and surveys from the course and do a follow-up on how the course introduced in this thesis worked out and how it should be improved. If the framework proves to be functional, then it should be studied even further to see what was the actual cause of improvement. Also the outcome of two different exercise-groups and how their results differ will be addressed. It's also probable that the manual created for this course will be upgraded and developed further on.

5 REFERENCES

1. Agarwal, Krishna. Agarwal, Achla (2006) Simply Python for CS0. Journal of Computer Sciences in Colleges, Volume 21, Issue 4. Pages 162-170

2. Agarwal, Krishna. Agarwal, Achla (2005) Python for CS1, CS2 and beyond. Journal of Computing Sciences in Colleges, Volume 20, Issue 4, pages 262-270

3. Chang, Carl, Denning Peter *et al.* (2001) Computing Curricula 2001 Computer Science Volume. Journal on Educational Resources in Computing, volume 1, ISSN:1531-4278.

4. Donaldson, Toby (2003) Python as a first programming language for everyone [Presentation] Western Canadian on Computing Education 2003. Available at: http://www.cs.ubc.ca/wccce/Program03/papers/Toby.html [referenced 31.05.2006]

5. Fangohr, Hans (2004) A Comparison of C, MATLAB, and Python as Teaching Languages in Engineering. ICCS 2004, LMCS 3039, sivut 1210-1217

6. Guzdial, Mark (2003), Media computation course for non-majors. Education Proceedings of the 8th annual conference on Innovation and technology in computer science education, pages 104-108.

7. Guzdial, Mark (2005), Designing media computation course for non-majors. Proceedings of the 36th SIGCSE technical symposium on Computer science education, pages 361-365.

8. Hume, Gregory *et al.* (1996) Hinting as a Tactic for One-on-One Tutoring. The Journal of the Learning Sciences, volume 5, issue 1, pages 23-47

9. Kasurinen, Jussi (2006) Python –ohjelmointiopas, versio 1. Lappeenranta University of Technology, Department of Information Technology: Manuals 7. ISBN 952-214-286-7

10. Kodaganallur, Viswanathar, Weitz, Rob, Rosenthal, David (2006) Tools for building intelligent tutoring systems. Proceedings of the 39th Annual Hawaii International Conference on System Sciences

11. Lindstrom, Greg (2005), Programming with Python. IT Professional, Volume 7, Issue 5. Pages 10-16.

12. Maurer, W. Douglas (2002) Technical Symposium on Computer Science Education. Proceedings of the 33th SIGCSE technical symposium on Computer science education. Pages: 336 -340 ISBN:1-58113-473-8

13. Miller, Bradley N. Ranum, David L. (2005) Teaching an introductory Computer Science Sequence with Python Available at: http://www.cs.luther.edu/~bmiller/Papers/paper20.pdf [referenced 01.11.2006]

14. Oldham, Joseph D. (2005) What happens after Python in CS1. Journal of Computing Sciences in Colleges, Volume 20 , Issue 6 Pages: 7 – 13

15. Prechelt, Lutz (2000) Comparison between 7 programming languages. Computer, Volume 33, Issue 10. Pages 23-29. ISSN:0018-9162.

16. Raymond, Eric (2000) Why Python? Linux journal website, published 01.05.2000. Available at: http://www.linuxjournal.com/article/3882 [referenced 31.05.2005]

17. Rich, Lauren. Perry, Heather. Guzdial, Mark. (2004) A CS1 course designed to address interests of women. Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education, pages 190-194. ISBN:1-58113-798-2

18. Rongas, Timo. Kaarna, Arto. Kälviäinen, Heikki (2004) Classification of tools for use in introductory programming courses. Lappeenranta University of Technology, Reseach Report 91.

19. Sajaniemi, Jorma. Hu, Chenglie (2006) Teaching Programming: Going beyond “Objects First”. University of Joensuu Technical Report, Series A, Report A-2006-1.

20. Shannon, Christine (2003) Another Breadth-first approach to CS1 using Python. Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education, pages: 248 – 251. ISSN:0097-8418

21. Ventura, Phil. Ramamurthy, Bina (2004) Wanted: CS1 students. No experience required. Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education, pages 240-244. ISSN:0097-8418

22. Zelle, John M. (2004) Simple, not Simplistic; Squeezing the most from CS1 with Python. [Presentation] Available at: http://mcsp.wartburg.edu/zelle/python/ccsc-handout.pdf [referenced 01.11.2006]

23. Zelle, John M. (1999) Python as a first language. [Publication] Available at:http://mcsp.wartburg.edu/zelle/python/python-first.html [referenced 01.11.2006]