Computer Programming for Everybody

This is the text of a revised funding proposal that we sent to DARPA in August 1999. In March, we heard that at least an earlier version of the proposal was accepted by DARPA; the work has begun late 1999 and will hopefully last two years, although we've only received funding for the first year (through October 2000). We're keeping our fingers crossed for the rest.

Please look at the Computer Programming for Everybody (CP4E) home page and the EDU-SIG home page (Python in Education Special Interest Group). This is where the current project status is described and/or will be discussed, and where you'll find pointers to more resources.

> **Note:**I have made one change to the text of the proposal: At the request of some supporters of other languages, I've withdrawn a language comparison chart that contained highly personal and sometimes unfounded opinions of other languages. The table was being used out of context in a way that some found objectionable. (Not all of the table is disputed, but it seems wiser not to engage in direct language comparisons without a lot more documentation.)

I've also removed some administrative details from the text and made some minor changes to accommodate HTML. I apologize for the writing style, which is at times more representative of funding proposals than most of the prose I write. I would like to thank Jeremy Hylton, Barry Warsaw, Al Vezza, Bob Kahn, Randy Pausch and David Beazley for contributions and suggestions that made this a successful proposal.

> --Guido van Rossum

Computer Programming for Everybody (Revised Proposal)

A Scouting Expedition for the Programmers of Tomorrow

Corporation for National Research Initiatives

July 1999

CNRI Proposal # 90120-1a

PI: Guido van Rossum

Point of Contact: Guido
van Rossum
Corporation for National
Research Initiatives
1895 Preston White
Drive, Suite 100
Reston, VA 20191-5434
Tel: (703) 620-8990
Fax: (703) 620-0913
Email:
guido@cnri.reston.va.us

Innovative Claims

In the seventies, Xerox PARC asked: "Can we have a computer on every desk?" We now know this is possible, but those computers haven't necessarily empowered their users. Today's computers are often inflexible: the average computer user can typically only change a limited set of options configurable via a "wizard" (a lofty word for a canned dialog), and is dependent on expert programmers for everything else.

We ask a follow-up question: "What will happen if users can program their own computer?" We're looking forward to a future where every computer user will be able to "open the hood" of their computer and make improvements to the applications inside. We believe that this will eventually change the nature of software and software development tools fundamentally.

We compare mass ability to read and write software with mass literacy, and predict equally pervasive changes to society. Hardware is now sufficiently fast and cheap to make mass computer education possible: the next big change will happen when most computer users have the knowledge and power to create and modify software.

The open source movement claims that peer review of software by thousands can greatly improve the quality of software. The success of Linux shows the value of this claim. We believe that the next step, having millions (or billions) of programmers, will cause a change of a different quality--the abundant availability of personalized software.

The tools needed for this new way to look at programming will be different from the tools currently available to professional programmers. We intend to greatly improve both the training material and the development tools available. For example, non-professional programmers should not have to fear that a small mistake might destroy their work or render their computer unusable. They also need better tools to help them understand the structure of a program, whether explicit or implied in the source code.

Our plan has three components:

- Develop a new computing curriculum suitable for high school and college students.
- Create better, easier to use tools for program development and analysis.
- Build a user community around all of the above, encouraging feedback and self-help.

These components come together in the scientific exploration of the *role of programming* in next generation computing environments.

We intend to start with Python, a language designed for rapid development. We believe that Python makes a great first language to learn: Unlike languages designed specifically for beginners, Python is also the choice of many programming professionals. It has an active, growing user community which has already expressed much interest in this proposal, and we expect that this will be a fertile first deployment ground for the teaching materials and tools we propose to create. During the course of the research we will evaluate Python and propose improvements or alternatives.

Rationale

CNRI proposes to undertake a research effort called **Computer Programming for Everybody** (CP4E). This effort intends to improve the state of the art of computer use, not by introducing new hardware, nor even (primarily) through new software, but simply by *empowering all users* to be computer programmers.

Recent developments in computer and communication hardware have given many people access to powerful computers, in the form of desktops, laptops, and embedded systems. It is time to give these users more control over their computers through education and supporting software. If users have a general understanding of computers at the level of software design and implementation, this will cause a massive surge in productivity and creativity, with a far-ranging impact that can barely be anticipated or imagined.

On a shorter term, the quantity and quality of available computer software will improve drastically, as the imagination and labor of millions is applied to the problem. Inventive users will be able to improve the software that supports them in their tasks, and share their improvements with their colleagues or--via the Internet--with others far away who are faced with the same tasks and problems. The ability to modify or customize software is important in crisis situations, when experts cannot be appealed to for help. It is also important for day-to-day activities: The number of unfilled programming jobs is currently estimated by some at 200,000 to 400,000.

Research Goals

The two major research goals are the development of a prototype of a new *programming curriculum* and matching prototype software comprising a highly user-friendly *programming environment.* We envision that the typical target audience will consist of high school and (non-CS major) undergraduate college students, although younger students and adults will also be considered. Course and software will normally be used together, so they should be tightly tuned to each other; each will also be usable on its own.

We will also explore the role of programming in the future. We are rapidly entering an age where information appliances, wearable computers, and deeply networked, embedded CPUs in everyday objects offer users control over their physical and information environments. End-user programmability will be the key to unlocking the potential of these technologies. (This is a common theme of DARPA's Information Technology Expeditions, cf. [Dertouzos].)

The research effort will not be done in isolation. We will engage academic research groups as well as several leading high schools. We will also build a larger community by making our course materials and software freely available on the Internet.

We plan to start by basing both components on Python, a popular free interpreted object-oriented language [Python] [Lutz] [Watters]. Originally developed at CWI in Amsterdam, Python is currently being developed and maintained by CNRI. Python is extremely suitable for teaching purposes, without being a "toy" language: it is very popular with computer professionals as a rapid application development language. Python combines elements from several major programming paradigms (procedural, functional and object-oriented) with an elegant syntax that is easy on the eyes and easy to learn and use. While we believe that Python is a good starting point, undoubtedly we will learn that improvements are possible. As part of our research, we will evaluate the effectiveness of Python for education and use by beginners, and design improvements or alternatives.

We expect that the fruits of our research will be demonstrable within two years and can impact society as a whole within a decade. This is the point when children who learn computer programming using our course in high school will start joining the work force (and the military). We expect the new mobile and embedded computing and communication technologies mentioned above to reach maturity around the same time. Thus, our timelines are nicely matched.

Motivation

In the dark ages, only those with power or great wealth (and selected experts) possessed reading and writing skills or the ability to acquire them. It can be argued that literacy of the general population (while still not 100%), together with the invention of printing technology, has been one of the most emancipatory forces of modern history.

We have only recently entered the information age, and it is expected that computer and communication technology will soon replace printing as the dominant form of information distribution technology. About half of all US households already own at least one personal computer, and this number is still growing.

However, while many people nowadays use a computer, few of them are computer programmers. Non-programmers aren't really "empowered" in how they can use their computer: they are confined to using applications in ways that "programmers" have determined for them. One doesn't need to be a visionary to see the limitations here.

An even more radical change is the introduction of computing and communications embedded in home and office systems. The number of devices that will contain programmable elements is expected to grow dramatically in the coming years. We must learn how to expose this programmability to users in a meaningful way and to make it easy for non-programmers to control and program these devices.

In this "expedition into the future," we want to explore the notion that virtually everybody can obtain *some* level of computer programming skills in school, just as they can learn how to read and write.

There are many challenges for programming languages and environments to be used by a mass audience. If everybody is a programmer, poor programmers will surely abound. Coping with this situation adequately requires a rethinking of the fundamental properties of programming languages and development tools. Yet, we believe that there should be no clear-cut distinction between tools used by professionals and tools used for education--just as professional writers use the same language and alphabet as their readers!

Given the ever more pervasive use of computers and software in every aspect of society, we expect the need for programming skills will only increase. While most quality software will be produced by professionals, there will be a need for more programming and customizability by end users.

Examples of this drive for flexibility can be seen in both present-day computing and its likely future:

- Increasingly powerful applications for desktop and laptop computers use scripting and macro facilities.
- Growth of the Internet has led directly to greater need for programmability to create active and interactive Web content.
- End-user information appliances and networks of CPUs embedded in everyday objects--both will demand user control and personalization.
- Mobile and intelligent software agents will be commonplace and require customization by

users.

Our Vision

In the future, we envision that computer programming will be taught in elementary school, just like reading, writing and arithmetic. We really mean computer *programming*--not just computer use (which is already being taught). The Logo project [Papert][Logo], for example, has shown that young children can benefit from a computing education. Of course, most children won't grow up to be skilled application developers, just as most people don't become professional authors--but reading and writing skills are useful for everyone, and so (in our vision) will be general programming skills. For the time being, we set our goals a bit less ambitious. We focus on teaching programming to high school and (non-CS major) college undergraduates. If we are successful here, we expect that the lower grades will soon follow, within their limitations.

In addition to the goal of teaching how computers work, a course in computer programming will return to the curriculum an emphasis on logical thought which was once the main benefit of teaching geometry.

Two general computing trends of particular interest are the move towards information appliances and the growth of embedded CPUs in everyday machines and appliances--whether in the military or the civilian sector. The decreasing size of computing and the increasing reach of networking, particularly wireless networking, make it possible to share information between devices and to interact with them. The ability to program will greatly improve users' ability to control these devices. Imagine that users could make their own changes to the software embedded in, say, their GPS receiver or handheld organizer, rather than (or in addition to) downloading upgrades from a vendor or buying "canned" add-on applications from third parties. This would greatly empower people to improve their life by programming their personal tools to do exactly what they need them to do.

If we are successful, non-experts will be able use their computers and other intelligent devices much more effectively, reducing their level of frustration and increasing their productivity and work satisfaction. (New leisure possibilities will undoubtedly ensue as well!) Computer users will be able to solve their own computer problems more often, reducing the need for technical support.

Even if most users do not program regularly, a familiarity with programming and the structure of software will make them more effective users of computers. For example, when something goes wrong, they will be able to make a better mental model of the likely failure, which will allow them to fix or work around the problem. They will also be able to assess better when they can make the changes themselves and when they will need the services of an expert. They will be more able to converse with experts, since they will now share more of a common language. An analogy is obtaining basic literacy in automotive maintenance: you know enough to check your oil and add a few quarts if necessary, but you also know that you shouldn't try to change your own brakes. When the mechanic says "your rotors are warped and you need new pads," you understand what he is talking about.

If this effort is successful, there could be many millions, eventually billions of computer programmers, at various levels of proficiency. The effects this will have on the state of the art of software development is hard to imagine. The nature of software will change to accommodate the needs of these programmers, allowing customization through source code modifications--and personalizations will be plentiful.

The effort could also have a major impact on getting women and minorities into computer programming--currently, these groups are vastly underrepresented.

The recently popular open source movement [OpenSource] is promising to improve the quality of key software packages through the peer review of thousands, as well as the ability for programmers to "scratch their own itch" (i.e., tweak the software in a minor way that only one individual cares about). We expect that moving from thousands to millions or billions of programmers will further change the nature of the software development process. Personal programming will become more important (and feasible) at this scale, while mass peer review will become relatively less important, due to diminished returns (the logistics of integrating bug fixes from thousands of sources is already a formidable task). But most current software, open source or otherwise, is too complex to allow anyone to do personal customization without first investing a serious amount of effort and time into understanding the software they're using. We are interested in changes to the whole software development process that will fix this as well--in particular, development tools.

In addition, by enabling the programmability of applications by anybody, we will leverage economies of scale without sacrificing the desire of users for highly personalized software. Applications can be mass-produced, without forcing everyone to fit the same mold in their use of the software (or into just those eddies of customizability planned by the developers). Users will want to personalize their systems for a number of reasons; these include becoming more productive, solving a problem peculiar to their needs, or just expressing their creativity and setting themselves apart from their peers. They will be able to achieve this if they have the basic programming literacy we envision.

Challenges

Some broad questions help frame our specific research goals, such as: Will the programming language taught in schools resemble the programming languages we know today? Will it even be called a programming language? How will we teach it? Will there be only one language? What other tools are essential to the teaching and use of this language? Is it even possible to have a language and tools that are both good for teaching and useful for experts?

Just as interesting are questions like these: How and for what purposes will people use their programming skills? How will a near-universal ability to read and write computer programs change the structure and utility of computer software? (This is especially interesting in combination with future versions of the Internet, which promise high-speed ubiquitous access to computing and storage elements.) Will people be motivated to actually program their systems once they have the confidence that they can? Will they even be interested in the first place?

A clear concern is the expectation that, if most people are programmers, many of them will most likely be *poor* programmers. People who can't write understandable sentences in their native tongue or balance their checkbook are unlikely to write well-structured computer programs! Our intent, however, is to make programming accessible, if not easy, for everyone. Some users will employ or contract a third party programming and customization service. This is much like a homeowner contracting out for a remodeling job.

We therefore need to investigate ways to improve the quality of the interaction between the programmer and the system, to help even poor programmers get the most out of their computers. For example, you might want to write a program to customize your PDA or toaster, but you might be discouraged if a small mistake could wipe out your address book or set your house on fire. Safeguards against disasters are needed, as well as ways of backing out of unwanted changes to a system as a whole. ("Undo", while very powerful, usually only applies to one file at a time. Backing out of unwanted global system changes typically requires a reboot or even painful data restoration from back-up media.)

Another concern regards configuration management. Without superior configuration management, businesses are going to find themselves either unable to correct problems, or held hostage by programmers who have modified the operating system or applications in a manner that precludes either upgrading or making other changes. In general, all locally made changes to large software systems are currently in danger of being incompatible with future upgrades of the primary product. Even locally produced software may be rendered unusable when the primary developer leaves, due to a number of reasons including lack of testing or documentation.

Apart from the fear that something might go wrong, another concern for beginning programmers who are interested in customizing their computer is the daunting task of trying to understand a large piece of existing software. We need to look into user-friendly tools for program analysis; more about this later. Another intellectual challenge is visualization of (application-generated) data in ways that help novices. Spreadsheets are of great value here, but not all data fits the matrix form.

Scripting languages are growing in popularity among professional programmers [Ousterhout], but questions remain about performance, software reuse, and integration with components written in other languages. We can address these challenges by enhancing the facilities of JPython [Hugunin1], a Python dialect seamlessly integrated with Java, and SWIG, an interface generator that creates interfaces between scripting languages and systems languages like C or C++.

Why Teach a "General" Programming Language?

It is well understood that there is something of a dichotomy between "general" programming languages on the one hand and "domain-specific" languages on the other. For this discussion, we use the term "general" in a broad and loose sense, to include functional programming languages and possibly even logic programming languages, to the extent to which they are usable as a general programming tool. Turing-completeness is the key concept here.

The domain-specific category then contains everything else, from command line argument syntax to email headers and HTML. The distinguishing factor here is the presence of a relatively narrow application domain. In this category we also place things like Microsoft's "wizards" (really just sequences of predefined dialogs connected by simple flow charts) and the controls and dials on microwave ovens or nuclear reactors.

A typical property of domain-specific languages is that they provide excellent control in the application domain for which they were intended, and (almost) no freedom in unanticipated areas. For example, HTML has no inherent ability for conditional inclusion of text, or for variable expansion. (The fact that such features have been added many times as incompatible extensions merely proves this point.)

General languages, on the other hand, usually aren't as good in any particular domain. For example, it is much harder to write a program in a general language to format a paragraph of text than it is in HTML. However, general languages make up for this through their Turing-completeness, which makes it possible to solve *any* problem that might come up (assuming availability of sufficient resources). General languages are therefore ideal when used in *combination* with domain-specific languages.

For example, if cell phones were programmable, one would still use the regular domain-specific interface (the keypad) to dial a specific number, since that's the most convenient way to access that specific functionality. However, without programmability, there is no way to make it try several different numbers for a particular friend until one is answered, unless the cell phone vendor anticipated this particular feature.

Why Start with Python?

We propose to start by making it possible to teach programming in Python, an existing scripting language, and to focus on creating a new development environment and teaching materials for it. We have anecdotal evidence that Python is a good language to teach as a first programming language. Our effort will focus on creating tools and educational materials for this purpose and on fostering a community around those materials. This will allow us to study in what ways Python is a good (or bad) language for teaching, and instigate directions for future development.

Why start with an existing language? Our experience indicates that the design and implementation of a new language takes years--and that this work must be (nearly) completed before a user-friendly development environment and teaching materials can be created. So we jump-start our project by using an existing language. Depending on user feedback, we may make changes to Python or design a new language altogether during the project.

We already have some evidence of where changes might be necessary. Prof. Randy Pausch at Carnegie Mellon University (see below) has conducted some usability studies of Python within their limited problem domain. Their users seemed most confused by the case sensitivity of Python's variable names and by the truncation of integer division. More extensive and generalized studies will serve to drive specific changes to Python, or indicate the need for a newly designed language.

Python is a good language for teaching absolute beginners. It derives many of its critical features from ABC, a language that was designed specifically for teaching programming to non-experts [ABC] [Geurts]. The Python community has seen many reports from individuals who taught their children programming using Python. The consensus from these reports is that the language itself is well suited for this purpose--unlike, for example, C++, Java, Perl, Tcl, or Visual Basic, which are too cluttered with idiosyncrasies.

Table 1 on the next page is a (highly subjective) chart comparing a few relevant aspects of Python to some other languages. From this table (and our experience), we conclude that Python is a good *first choice* for teaching which also serves well as a language for more serious application development. Unlike other languages proposed for teaching to novices (e.g. Logo, LogoMation, even Python's ancestor ABC), Python isn't *just* a teaching language. It is suitable for developing large real applications, as shown by projects here at CNRI [Knowbots] [Mailman] as well as elsewhere. For example, Industrial Light and Magic has converted its entire tool base to Python and considers this an advantage over the competition.

Moreover, Python is extensible by modules written in other languages (e.g. C, C++, or Java), to mediate access to advanced functionality that is not easily accessible from Python directly (for example, high-speed 3-D computer graphics packages). While we don't expect students to write these extension modules, the *use* of such modules makes it possible to spruce up their learning experience greatly. This extensibility gives teachers an opportunity to tailor lessons to the interests of their students by providing them with guarded access to other software packages.

The fact that Python can be used to develop large applications plays into a different aspect of our vision, namely the development of open source application software that can be tailored by users who are not expert programmers, but have learned some programming skills. Although this is not the focus of our effort here, we hope that we will see at least some initiatives towards this goal, and we will encourage companies and organizations wishing to take steps in this direction. We expect that the existence of JPython will be an important enabling factor here.

Python's programming environment and documentation are less than ideal for teaching to novices. In particular, the existing program development tools and tutorials for Python (there are several of each) all assume that the user is a dyed-in-the-wool developer, who knows a suite of external tools to edit, run and debug programs, and who already knows one or more other programming languages and their development environments. This currently stands in the way of more widespread experimentation with Python as a first programming language.

Table 1. Language comparison chart

(withdrawn)

Approach

We will create a next-generation programming environment and teaching materials that empower ordinary users to write simple programs and to understand the structure and organization of larger programs. We will also explore how widespread programming literacy will affect the production and use of software in a ubiquitous computing environment.

Our work is organized into three distinct areas:

- a new computing curriculum suitable for high school and college students.
- better, easier to use tools for program development and analysis.
- a user community formed around the above, encouraging feedback and self-help.

As explained before, we will initially use the Python programming language. This will get the world ready for the next step. The "next generation computing environment" may not use Python, but Python and CP4E are useful experimental steps in the right direction.

As soon as initial versions of the newly developed course and tools are released to the community, the feedback channels will be opened. The initial feedback will mostly go into improvements of the environment and teaching materials. This is where the community building begins.

Curriculum Development

A key goal of the CP4E effort is the development of a curriculum for teaching programming literacy to a range of students, from non-computer science major undergraduates, down to secondary school and eventually lower- and middle-school students. The approach for each of these grade levels may differ so as to better relate to and reach students as they mature, but CP4E strives to provide a unified approach which will grow as the student grows, presenting richer and more in-depth topic material along the way. The initial effort will focus mainly on high school and undergraduate students.

CNRI will develop the basic materials for the curriculum, including software that will be used in the teaching environments, and tutorial and introductory material which may serve as the basis for textbooks on programming. CNRI will be working closely with educators experienced in producing textbooks and other teaching materials in order to best tailor these tools for the intended age groups.

Our goal is to take the software environments and tools used by more experienced programmers, and produce versions of these that will be useful in teaching programming skills. We are inspired by the existing Python interactive interpreter and by IDLE (a graphical development environment for Python), both of which can be used either as productivity tools for professional programmers, or as teaching aides when used in conjunction with tutorial material. Our new tools will provide useful functionality for novice and experienced programmer alike.

New computing course

CNRI proposes to work with the University of Chicago to develop a new course in computer science, using Python as the programming language for all levels of programming instruction. Python is a particularly appropriate language for this purpose because it is easy to learn, read, and use, yet powerful enough to illustrate essential aspects of programming languages and software engineering. Thus even young students could be taught the basics of programming using Python, but they would not be limited in their application domain as they would be with Logo. The use of Python would allow each student to explore and progress at their own pace. Especially exciting is the fact that gifted students would have a powerful programming language and environment already at their fingertips, should they become motivated to learn more, or at a faster pace.

The University of Chicago would develop a series of courses that introduce programming and computer concepts to non-computer science students at the undergraduate and high-school level. Currently, courses at this level tend to emphasize either programming languages (with a strong mathematical flavor) or web-programming topics such as HTML and JavaScript. Unfortunately, both of these approaches have serious limitations. If a course is excessively formal and mathematical, it may only appeal to computer science majors and students of a technical mindset. On the other hand, Web-programming courses, while capitalizing greatly on the popularity of the Internet, tend to narrowly focus on specific technologies such as HTML, Perl, or JavaScript. As a result students learn little about computing within a greater context or gain the problem solving skills needed to solve the computational problems of the future.

The course to be developed at Chicago will address the aspects of computing that we feel everyone must know in order to be a knowledgeable computer user:

- Basic computer organization. Students will learn the basics of how computers work and how they are put together. Topics would include Boolean algebra, logic, and simple computer architecture (e.g. CPU, memory, I/O). Simply stated, this is the stuff going on "under the hood"--stated in easily understandable terms. Ultimately, we would hope to demystify computers as much as possible.
- An introduction to programming. This will introduce different ways people have programmed computers. Students will learn about procedural, functional, and object oriented programming, but in an informal manner. Rather than trying to turn students into professional programmers, the goal will be to introduce students to some of the ways people have tried to simplify the use of computers.
- Software architecture. In the future, it is increasingly likely that computers will be programmed largely by assembling existing software components and writing a small amount of glue code. In order to make this possible, an understanding of how software is put together will be essential. Student will learn about software design and software organization (just what in the heck are all those DLL's anyway?).
- Debugging and problem solving. How to survive when all else fails--without having to call customer support.

At all levels of education, it is vitally important that usability studies for both the software tools and the textbook material are conducted and evaluated. This is the only way that such materials and tools will be improved and tailored for the specific age and experience groups. Usability studies of the type conducted by Prof. Pausch of CMU will be developed and conducted at all three levels of teaching. Of course, we will also develop traditional tests to allow teachers to measure individual students' performance.

3D Game Playing

We intend to engage in small-scale teaching efforts ourselves, e.g. at the local high schools listed in the collaboration section, but we don't expect that we will be doing much teaching. If our experience with Python's popularity is any indication, we won't have to: others are eager to participate in this experiment.

The courses will use the new development environment described in the next section. As an incentive to make programming more "fun", we intend to connect the development environment to an existing programmable 3-D game-playing engine as used in popular computer games. Several such engines are or will likely become available for use with Python; we will select one and create an interface library for it suitable to our audience.

Why use a 3-D game-playing engine? The experiences with Logo show that graphics are a good way to catch a younger audience's attention, but its 2-D graphics look somewhat boring compared to the video games teenagers are familiar with these days. Alice is a good example of an engaging 3-D graphics environment.

Knowbot Programs

In addition to using a 3-D game as a testbed, we may use CNRI's Knowbot technology [Knowbots] as a motivating application for novice programmers. Knowbot programs are independent mobile programs capable of migrating between Knowbot "service stations" (specially equipped hosts) throughout the Internet. Service stations provide services to the Knowbot programs such as search services, digital object repositories, auction services, etc.

Another way of looking at a Knowbot program is as a small component working within the larger framework of the service station. A Knowbot program is a modular, independent program that can be easily written to move around the Internet, but which has powerful functionality due to its integration into the framework, and its use of the environments it encounters.

We will be exploring several ideas on how to use Knowbot programs in the teaching curriculum. We imagine cooperative gaming scenarios, where students can create Knowbot programs that exhibit certain behaviors and must work together to solve a common problem. This would be a great way to motivate students from all over the Internet to collaborate.

We might envision treasure hunts, where students have to apply the programming skills they just learned in order to discover and migrate to a service station, solve a puzzle at the site, and receive the treasure. We might design distributed virtual simulations, similar to MIT's Virtual Fishtank [Fishtank], where students can create their own discrete elements of a complex system (e.g. implement a virtual fish in a Knowbot program) and watch how their own elements interact with others. Because the Knowbot technology allows for highly distributed, very complex interactions across the entire Internet, it gives us a unique platform for experimenting with rich cooperative learning opportunities.

Programming Tools

We will design and build a programming environment specifically intended to support the teaching of programming to users with no previous programming experience. Our aim is provide tools to support users when they are learning programming and when they are employing those skills in their homes and offices.

We believe that most ordinary users will employ their programming skills to customized and extend their computing environment. Rather than writing new programs from scratch, most people will add new code to existing programs. There are three significant challenges that must be addressed by programming tools aimed at this audience.

First, the environment must significantly ease the burden of writing, installing, and debugging new programs. The current generation of development tools can be cumbersome for expert users, let alone novices. We must focus careful design and usability studies on the development of new programming environments.

The second challenge is to provide for the continual evolution and modification of software artifacts by consumers as well as producers. We will develop tools to help users understand the structure of large programs so that they can identify where to make changes and what impact those changes will have. Our tools will also help users manage and configure software, so that individual components can be replaced or upgraded over time. These tools will help users share new and modified programs by automatically tracking versions and dependencies.

The final challenge is to build tools that will be useful in a ubiquitous computing environment. The desktop computing environment will be rapidly overtaken by networks of computer-controlled devices and physical systems. This new environment exacerbates problems of installing, debugging, and managing software. It also poses new challenges for system designers to build software that allows end-user customization.

This section is organized around these three challenges. The first section discusses the proposed programming environment. The second section discusses program analysis and configuration management tools. The third section discusses application frameworks to support end-user programmability of ubiquitous computing environments.

Our approach to this problem will be to look at how traditional programming tools, such as editors, debuggers, and class browsers, can be augmented and enhanced by more advanced ways of analyzing, inspecting, and understanding programs.

Programming Environments

The most basic activities of programmers are editing source code, running the program to test it, and debugging the program. (Python doesn't have a separate compilation phase.) A programming environment must of course support these activities. We have been developing a portable programming environment for Python named IDLE, which allows the user to execute individual statements interactively. It is mostly targeted at experienced programmers, but will server as a starting point for an environment for absolute beginners.

IDLE only scratches the surface of the kind of programming environment needed to help novice programmers. For example, its source code colorization and indentation features could be replaced by a much more powerful program checker which would point out all syntax errors, undefined identifiers, type mismatches, and so on, while the user is typing (like a spelling checker). The debugger could support retracing execution steps, editing the source code of the running program, etc. The program editor could support a flexible form of template-based editing (the Alice group has very good experiences with this in their limited domain). The undo feature, which currently allows undoing source code changes only, could be extended to undo changes to the run-time state of the program or even side-effects to the environment (within reason--we can't expect to undo printing or the sending of an email message). As an example, the Alice software provides full undo for all actions involving changes to the 3D world it manages.

Two specific areas of work are undo and an extended type checker.

"Undo" is an extremely important tool for beginners because it is the programmer's first line of defense. Along with version control, auto-save, and other features, the ability to rollback an unlimited number of near-term changes means that the programmer has more leeway to experiment and learn. However, most undo implementations are quite limited in their scope. Our approach will investigate such concepts as selective and global undo. In a traditional undo system, the editor simply keeps a linked list of changes to a file, and those changes can be unapplied or reapplied by moving through this list (there are variations on this theme, including undo rings, and undo/redo). One of the problems with traditional undo is where some undesirable changes overlap with some desirable changes; the programmer often has to lose the desirable ones to eliminate the undesirable ones.

With selective undo, changes can be localized to a finer granularity. For example, suppose a programmer made three changes to function A at the top of the file intermingled with four changes to function Z at the bottom of the file. Now the programmer discovers that function A should never have changed; selective undo allows changes to function A to be rolled back without affecting the changes to function Z.

Global undo is similar to what version control provides, where system level changes can be tagged and rolled back when they adversely affect the system. Where global undo differs however is that no a-priori decision has to be made about tagging.

We also plan to enhance the development environment with type-checking tools that help programmers find mistakes in their code and improve the performance of compiled code. Python is a dynamically typed language, like Smalltalk or Scheme, that relies on extensive runtime checking to ensure the correct use of built-in operations. Soft typing [Cartwright] is a mechanism to statically check programs in dynamically typed languages to detect errors and eliminate unnecessary runtime checks; the analysis is integrated with the programming environment rather than with the language runtime. This mechanism has been applied to Scheme [Wright][Flanagan]. We will develop a similar type checking mechanism for Python. The key challenges for developing a soft type system for Python are extending the analysis to objects and modules and accommodating Python's extremely dynamic execution environment, which allows modification of classes and instances at runtime.

Preliminary work at CNRI demonstrated the value of type analysis for improving the performance of JPython programs. Hugunin [Hugunin2] demonstrated performance improvements of up to three orders of magnitude for JPython.

Program Analysis and Configuration Management

We will augment the basic programming environment with a collection of tools that aid users in understanding large programs, so that they can customize and modify them, and in managing the installation and configuration of software, so that they can upgrade the software without destroying their modifications. We will also develop and extend tools that build interfaces between low-level software components and scripting languages, which will enable greater user control over the low-level components.

It is important that these program analysis tools, aimed at beginners, are simple to use, and that they allow grasping the results of the analysis *without* requiring detailed understanding of how the analysis is done. At the same time, we strive for our tools to be powerful enough for experts too--they should accommodate increasing expert levels as a user becomes more familiar with the tool, and they should be able to handle large programs. An analysis tool that can effectively be applied to, say, the source code of the Netscape browser, is much more useful than one that only works well for small example programs.

Our focus on relatively inexperienced programmers requires that our tools contain excellent visualization modules, which can present the discovered design to the user without causing information overload. For example, current visualization tools often lack "common sense," and will mindlessly produce large tree or graph diagrams spanning many pages that consist of endlessly repeated similar substructures; this effect causes the user to lose sight of the forest through the trees.

We plan to focus initially on tools that work with Python programs. However, most of the techniques for program analysis that we expect to develop are essentially independent of the language being used. We will also investigate the use of tools that cross language boundaries, so that users can consider the effects of changes at the scripting level on low-level components written in C or Java.

The program analysis tools will help users understand the gross structure of programs by identifying the relationships between static program components. One example of this kind of analysis tool is Womble, which extracts object models from Java bytecodes [Jackson]. Womble extracts object model directly from source (or object) code, rather than from a formal specification. Because this method does not rely on the existence of a formal specification (such as a UML model), we believe it is more accessible to ordinary users. A similar approach can be used for analyzing Python programs, although Python's dynamic typing and "first class" functions and classes pose significant challenges.

We will also investigate program slicing [Tip] and program paths [Ball] as techniques to help users understand where to make changes and what impact those changes will have. Slicing is a well-known technique for identifying subsets of programs that affect a particular variable. Analysis via program paths shows the various possible execution paths through a body of code. Each technique has value for testing and debugging programs. Two challenges are to apply these techniques across language boundaries and to identify abstractions boundaries that are implicit in the code but expressed inexactly by the type system. Womble, for example, recognizes that Java container classes are not an interesting part of an application's object model; it merely represents a relationship between the object that uses the container and the contained objects. Similarly, a program slice could be performed with respect to certain abstraction boundaries and aspects of the code. A program slice that presents the functional aspect of a program without including concurrency-specific code may be useful for understanding the program structure. (Of course, the concurrency-specific code is important to understand, but may be a separate concern.)

A third area of work is one automatic generation of scripting language interfaces to low-level code such as C, C++, or Java. The SWIG tool [SWIG], developed by our collaborator David Beazley (see Collaboration Plans section below), helps users generate scripting language bindings from C and C++. We will work to improve and extend SWIG to improve the amount of automatic processing that can be done and to allow greater customization of bindings by users. We will also enhance SWIG to produce more natural language bindings; for example, when strings are passed as function parameters in C programs they are typically represented by two variables, a pointer and a length.

The SWIG bindings should automatically convert between those two C variables and a single Python string argument.

If users are empowered to modify and customize code, they will be challenged to maintain those changes when the underlying software is upgraded or when system components are replaced. Version control is already a vexing problem for software developers, who must ensure that their products are compatible with many operating systems and shared libraries. Users will also want to share their customizations with others - co-workers or friends and relatives. We will provide tools to help users maintain and share programs and modifications to programs.

One of the most important issues that our programming tools research will address is the burden that user customizability will place on system configuration management. When a user makes a change to an application, what assurances are there that a future update to the application by the vendor will be compatible with these changes? How does the user himself keep track of just what changes they've made to an application? What happens when a future version of the product adds a feature, previously missing, that the user has added (in a different form)?

Our tools will help users keep track of changes they've made, through successive revisions, and help users merge their changes back when the primary application itself has been modified or updated by the vendor. A key to this approach is identifying each version of software and a simple language for describing its properties and dependencies. For example, we intend to improve version control systems so that they track changes at different abstraction levels and granularities than current systems, e.g. labeling changes based on the features they implement instead of the source files (or parts of files) they modify. The tool will automatically identify dependencies on other libraries and component. We will investigate ways to integrate testing frameworks into the configuration management systems so that when the primary application is upgraded, each feature change a user has installed, will be merged and tested.

We will also create community-enhancing tools that make it easier to share, integrate, and use peer-developed software. Issues to be addressed include dependency management between packages, ease of installation (and de-installation!), coping with changes to both the operating system, applications, and libraries, version control, and package maintenance. Software must be described and classified, and forums for discussion, feedback, bug reports and patches (both to the maintainer and from the maintainer) must be established. Of greatest importance is that these community-building tools must be primarily managed by the community itself; they must be highly distributed, replicated, and secure. For example, users could share information about compatibility problems between various libraries; when coupled with automatic distribution tools, this information could prevent software upgrades that prevent other applications from working.

There are several existing tools that attempt to address some or all of these issues. The Comprehensive Perl Archive Network attempts to contain all the Perl material a Perl programmer would ever need [CPAN]. It is distributed and replicated, and it makes the job of installing Perl modules easier (though not always easy enough). It is not, however, very adaptable to other types of archival material. Within the Python community, the *distribution-utilities* special interest group is attempting to make the distribution and installation of third party software easier, but it does not address the wider issues involved, and again, it is narrowly focused on Python software [Distutils]. The most ambitious related work we are familiar with is the self-updating software [Liskov] project.

Application Frameworks

We will examine the impact of future changes in computing and communications on the way users control computers, and the implications of such developments as near infinite bandwidth, greater accessibility to much more powerful computers, the ubiquity of computing resources, and a much greater level of internetworking, even at the micro-device level. These changes affect both what kinds of programs users will write and what kinds of computers those programs will run on.

Our approach in this area will be to work with experimental and prototype systems in order to understand how end user programmability should be exposed. As these technologies mature, we may incorporate our experiences into teaching materials.

Many non-programmers begin writing small programs that are used within specific domains and within the contexts of application frameworks. The spreadsheet is perhaps the best example of programming in a limited context. It provides a limited language targeted specifically at manipulating tables of numerical data. The application provides the coordination framework--managing the display, control flow, I/O, etc.--and lets the user concentrate on a specific problem. A macro facility, which is one part of a spreadsheet's programming suite, aids the use of many applications, allowing users to automate repetitive tasks. A second example can be found in MS Word macros, which are also a common form of user customization.

The Internet has exposed many non-professionals to programming, especially in domain-specific languages like HTML and active content languages like JavaScript, PHP and CGI. In these situations, too, non-programmers are typically creating small programs which fit within the larger context, for example, of a Web server which takes care of details such as managing the TCP/IP connections to the clients, setting up the environment in which the user programs run, handling error situations, and adhering to standard protocols.

In the future, non-programmers will be using a plethora of information appliances. One example area where programmability of these devices will significantly improve their usefulness is in the management of information flow--and perhaps more crucially, the *limiting* of information flow. A person may have messages arriving over many media--text, voice, video--and access them via many devices such as PDA, mobile phone, and computer screen. We expect seamless interoperability, so that for example a mobile phone can be used to follow up to an email with a voice response without having to look up the number. Small programs could be used to customize the interfaces on these devices and to filter and limit the flow of messages through them.

Some examples include:

- Trying several different phone numbers to reach a person, perhaps depending on time of day or day of week.
- Diverting incoming phone calls to voice mail or email when you do not want to be disturbed-but still letting important calls through.
- Lowering the volume on TV or radio when you answer the phone.
- Recording copies of certain phone calls, perhaps by invoking a speech-to-text converter.
- Limiting the amount of personal information transferred to stores or companies as part of an e-commerce transaction.

Each of these interface features might be accommodated by the designers of a particular device. However, feature-rich interfaces are harder to use, so designers may intentionally limit the interface. It is also likely that inventive users will always think of features the designer has overlooked. Allowing for end user programmability and customization lets the user adapt the device to her particular needs.

These examples indicate an important pattern in programming scope: users will be customizing applications and appliances in the context of the system's computational model and software framework. Thus our expedition will explore ways to modularize applications, and organize these modules so that users can add the small bits of functionality they want without having to concern themselves with the operation of the system as a whole.

One of the key goals of empowering non-programmers to modify and customize software in the context of application frameworks or embedded devices is to reduce the cognitive load required to understand how the modification fits into the larger program. This is true even for experienced programmers new to an application's code base. Applications must be modular and provide sufficient high level abstractions so that their constituent parts can be understood quickly and independently. This lets people concentrate primarily on the parts that need to be changed. In our approach we will explore several ideas intended to improve the modularity of software.

We intend to look at existing techniques, such as object-oriented programming and component composition as ways to organize the software. In theory it is much easier to swap out a black box component for some different functionality, as long as the interfaces and input/output semantics are maintained. In reality, it is currently very difficult to write classes and components that are independent of the rest of the system. We will also be exploring new concepts such as Aspect Oriented Programming [AOP], a new way of modularizing software based on cross-cutting concerns. Perhaps some combination of these, or new modularizing techniques will prove effective, for example, by organizing each function as an aspect of a component.

Community Building

Besides working with selected partners, we will seek the involvement of the community at large. We will do this by sharing prototypes of the developed courses and software through a website, and by soliciting feedback on those materials through a variety of channels such as newsgroups and mailing lists. CNRI has considerable experience with community involvement through the web and via other means. For example, the key focus for the Python community is the Python website [Python], which also hosts many Python-related mailing lists; an important focus for the digital library community is the D-Lib website [D-Lib]. Both sites are run by CNRI. The existing Python community is already showing great interest in the CP4E proposal, and we expect that this will be the perfect place to bootstrap community activities specific to the CP4E effort.

Such explicit facilities for community involvement will greatly benefit our research, and allow the community to reap the most benefit from our research. Benefits of early and large-scale community involvement for our research will include: volunteers who help "test-drive" our courses and software prototypes; new courses developed by community members aimed at specific target audiences or aimed at teaching specific skills or subjects; localized variants, translations etc. of existing courses; new or modified examples (you can never have enough examples, and examples that are tailored to specific audiences are more effective); new applications developed by and for students of our courses using our program development software; and so on. In the Python community, we receive many such contributions (including complete foreign-language translations of key documents) completely unsolicited!

Benefits for the community include early access to new courses and software, and help for teachers in teaching and in convincing their management of the importance of teaching computer programming to all students (as opposed to the advanced placement crowd). We also expect that the proposed community facilities will foster a large amount of self-sufficiency among community members. For example, in co-tutoring projects, students in need of tutoring help will find volunteer tutors on the net (often other students who are more advanced), and teachers will be able to exchange their experiences directly. This kind of tutoring activity already occurs in the Python community, and will help defray the load on our research team caused by repeated requests for help with simple questions.

In the spirit of "mass computer literacy" and the open source movement [OpenSource], our website will make the courses and software widely and freely available. In addition to the website, we will create and maintain one or more mailing lists with archives, and perhaps a "chat" service for users. We will actively participate in the mailing lists in order to foster a community, and also collect and analyze the feedback provided by the community to us through these (and other) channels.

It is clear that we consider community involvement essential for the success of this project. Therefore, we want to go beyond the typical website setup. We plan to create an automated archival site for teachers, students and programmers, which can be used to exchange course notes, examples, useful software, and so on. We also plan to develop prototype software to aid users in maintaining consistent collections of software packages, both locally developed and downloaded, on one or more machines. The expected frequent exchange of extensions, upgrades, patches, end user modifications, and so on, would cause a nightmare of version control problems using the existing practice. This is elaborated in the subsection "Sharing and Maintaining" above.

Collaboration Plans

We will build into the program a certain amount of planned cooperation with other academic and non-academic institutions in order to ensure the success of the research. In particular, we propose to subcontract some activities to groups at Carnegie Mellon University and the University of Chicago. The Alice group at CMU has successfully used Python as an end user programming language for their popular virtual reality software; the University of Chicago has considerable experience and interest in CS curriculum development. We also plan to work informally with other academic groups and to open up the participation to others as practicable.

Carnegie Mellon University

The Alice group [Alice] [Pausch], under leadership of Prof. Randy Pausch, develops affordable 3-D graphics and virtual reality software for Windows, originally at the University of Virginia and now at Carnegie Mellon University. They use Python both for end user programming and in the implementation of large parts of their system (almost everything except the rendering engine). They probably have the most extensive and best-documented case study of teaching Python programming (albeit in a limited domain) to users with no prior programming experience, and their enthusiasm for Python has been a great encouragement for us to consider using Python in a general programming curriculum. They are also one of the only groups to apply usability testing to programming languages and Application Programmer Interfaces.

We plan to provide the Alice group with limited funding to continue this part of their research in a mutually beneficial way. We expect to benefit from their experience in teaching Python to novice users, from the simple but effective programming tools they have built into the Alice system, and from their proficiency in user testing--very important for both the tutorial and the software we plan to develop. They will benefit from our tutorial (giving Alice users broader instruction in software development) as well as from our software (more powerful program construction tools).

University of Chicago

The University of Chicago can contribute to the CP4E research effort in two core areas: curriculum development and software development tools. We also plan to work with the University of Chicago Laboratory School, a private K-12 school operated by the University of Chicago. Our liaison at the University of Chicago is Prof. David Beazley, an experienced Python user. We plan to provide the University of Chicago with limited funding for curriculum development. This will be to our benefit because of their interest and experience in teaching; their benefit will be the use of a superior programming language and tools.

Local Schools

Finally, we plan to work directly with selected local schools to "test-drive" the developed materials. Working with local schools makes regular face-to-face meetings with both teachers and students possible; we consider this essential for the evaluation of our prototype course and software. The Yorktown High School in Arlington County, Virginia, has already shown interest in this proposal [Yorktown].

Another possible candidate might be the Thomas Jefferson High School for Science and Technology, a public magnet school in Fairfax County, Virginia, which already offers a computer science curriculum to advanced students [TJHSST]. We will be contacting other local schools in Maryland, Virginia, and the District of Columbia for collaboration purposes during the course of this project.

Other Research Groups

We also plan to work informally with academic and other research groups who are developing personal computing hardware and software for the future ("ubiquitous computing" projects); these projects all envision end user programmability of sorts. Some examples of such projects are Project Oxygen at MIT [Dertouzos], Portolano/Workscape at Xerox PARC and the University of Washington [Portolano], and Invisible Computing at CMU.

We expect that the main benefit for us of such cooperation will be early deployment of our technology in advanced systems, while their benefit will be improved end user programmability of the systems they are developing. Note that the timing is excellent here: widespread deployment of the personal, embedded systems as envisioned in e.g. Project Oxygen is expected around the same time that our curriculum and software could be in widespread use.

Comparison to Other Research

ABC. Python's predecessor, ABC, was designed in the early eighties as a teaching language. Its motto was "stamp out Basic"--acknowledging the main competition in languages for non-experts at the time. ABC's designers had a lot of experience teaching "classic" programming languages like ALGOL to novices. They found that their students were often so overwhelmed by the incidental details of using a computer language (such as running the compiler, dealing with different numeric formats, arcane I/O operations, and low-level memory management) that they never managed to concentrate on the essentials of good program and algorithm design.

To counteract this effect, ABC's designers went back to first principles. They set out to design a language and an environment for that language that would take care of all the incidentals, leaving the student more time to learn what's essential in programming independent of the programming language at hand, such as clear control flow and powerful data structures, and focusing on the elegant expression of programs. They proposed both a new language design and new terminology that deviated radically from what was (and still is) current among computer scientists and programmers. In fact, the single largest reason why ABC didn't make as much of an impact as expected is probably that they deviated too much from current practice. The people who had access to the hardware that was needed to run ABC (initially it only ran on Unix system, although it was later ported to the Mac and PC) were often experienced computer users who felt frustrated that ABC didn't "speak the same language" as the rest of their applications.

About a decade later, Python grew out of this frustration. It shares ABC's focus on elegance of expression, fundamentals of programming, and taking away incidentals, but adds object-orientation, extensibility, and a powerful library of modules that interface to other applications, via many different mechanisms: shared files, program embedding, RPC interfaces like CORBA or COM, and network protocols (supporting all the protocols typically used on the Internet).

Logo. Really a family of languages related to Lisp and mostly developed at MIT, Logo is of course the most well-known programming language for children. It has a rich tradition, strong roots in schools, and a number of commercial offerings. There is ongoing research being done by the Epistemology and Learning Group at the MIT Media Lab, e.g. the "programmable brick" (in cooperation with LEGO).

A key difference between Logo and our proposal lies in our vision that millions of (amateur) programmers will be developing open source software together--Logo appears content with teaching limited programming skills to younger children, for whom computer programming is mostly a way to train their mind in abstract thinking. Logo also has limited applicability in the real world of software development.

LogoMation. A company called Magic Square sells LogoMation, a language not unlike Logo, with a similar emphasis on turtle graphics. It comes with an excellent tutorial suitable for children from 8 up. LogoMation's syntax is similar to Python (more so than Logo's syntax); which suggests that we're on the right track with Python.

But like Logo, LogoMation is limited in the growth path it offers. It doesn't directly address the issue of "what next," expecting its users to move on to other programming languages for real work.

Alice. The testimonials on the Alice website clearly indicate that Alice is successful at teaching programming to children as well as to adults with no prior computer experience. It also indicates the importance of a "fun" environment (and Alice's 3-D graphics are more attractive than Logo's turtle graphics). Alice also gives us some hints on what aspects of Python could be improved: for example, their experiences suggest that Python's case-sensitivity is a problem.

However, the emphasis of the Alice project is on 3-D graphics--the Alice tutorial doesn't really teach much in the way of program or data structuring techniques. While we agree that 3-D graphics are a great way to create and keep a young audience captive, we are interested more in teaching programming in general, not just graphics. For this reason, the emphasis in our initial work will be on the development of a programming environment and tutorial where 3-D graphics is just one of the possible uses for a computer.

DrScheme. The TeachScheme! Project at Rice University [TeachScheme] aims to develop a new introductory computing curriculum based on the Scheme programming language. A central part of the Rice effort is the development of DrScheme [Findler], a programming environment targeted at beginning students. The focus of TeachScheme is on a relatively narrow audience--college students who have a solid grounding in high school algebra and an interest in studying computing and its application to scientific problems. We envision a much wider audience, where the assumptions about a strong math background and interest in scientific problems do not hold. We also expect that Scheme, a language that excels in exposing the fundamental building blocks of computation for pedagogical purposes, would be inappropriate for a mass audience.

It is interesting to note, however, that one of the key parts of the TeachScheme project is a development environment. While the audiences and approach are different, our project and TeachScheme share a sense that the development environment is a crucial component. There is a need for an interactive read-eval-print loop, a powerful debugger, and tools to understand how programs work.

Facilities

We will use CNRI's existing computing infrastructure for development and distribution of the proposed materials, augmented with desktop workstations and a web server purchased specifically for this project. We will use the Internet and the World-Wide Web for all distribution of materials.

List of Key Personnel

Guido van Rossum will lead this effort at CNRI. He is a group leader at CNRI, and is the creator of Python, for which he still serves as the key developer. He is also the lead designer of the Knowbot mobile agent system. In the past he has worked on ABC, a programming language developed for teaching purposes, and Amoeba, a well known distributed operating system developed in the 80s. He has a Masters' degree in mathematics and computer science from the University of Amsterdam.

Jeremy Hylton is a senior member of the CNRI technical staff. He is one of the designers of the Knowbot mobile agent system, and has designed and implemented several agent-based information management applications. He received a M.Eng. in electrical engineering and computer science and an S.B. in computer science and engineering from the Massachusetts Institute of Technology, both in 1996.

Barry Warsaw is a senior member of the CNRI technical staff. He has been a contributing designer to several CNRI projects including the Application Gateway System, the Knowbot Operating Environment, and JPython. He has contributed to development of the Python language and to the Grail Internet Browser. He received a B.S. in computer science from the University of Maryland in

1984. Previous to CNRI, he worked on robotic systems operator interfaces at the National Institute of Standards and Technology from 1980 through 1990, and on medical database information technology at the National Library of Medicine from 1990 through 1994.

Statement of Work

CNRI will perform the following work:

- Develop a prototype programming environment for Python, including program analysis and management tools suitable for use by novices.
- Develop a prototype tutorial to teach programming using Python to non-programmers, especially in high school or college, using the above programming environment.
- Develop example software aimed at the above audience; for example, a Python extension that allows the manipulation of a third-party 3-D game-playing environment.
- Set up and maintain a website and mailing lists promoting the above software and tutorial and soliciting feedback.
- Collaborate with selected high schools and universities.
- Engage in small-scale teaching and user testing efforts.
- Collect feedback regarding the above software and tutorial from users, students and teachers and use this to improve the software and tutorial. Also use it to improve the Python language itself or to propose a better language.
- Publish a final report documenting the research, the lessons learned, the results of the research, and the recommended follow-on research.

The Alice group at CMU will perform the following work:

- Integrate CNRI-developed tools into Alice.
- User testing of CNRI-developed tools and tutorial.

The University of Chicago will perform the following work:

- Work with CNRI to develop the CNRI-developed tutorial into a CS curriculum suitable for high school, and separately into a CS course suitable for non-CS undergraduates in college.
- Teach classes using the developed curriculum in order to provide feedback.

In order to maximize access to the materials produced, all software, educational materials, and reports produced for this project will be made freely available on the World-Wide Web as open source material.

Schedule

We envision the following schedule for the effort:

- Year 1. Initial research at CNRI: develop initial prototype programming environment; design program analysis and measurement tools; develop first version of tutorial; develop contacts with other researchers and with interested teachers.

- Year 2: At CNRI: initial implementation of program analysis and measurement tools; collect feedback from first software and tutorial experiences. At CMU: start working on integrating CNRI-developed tools into Alice and start user testing. At U of C: start working on curriculum development for high school and college students.

- Year 3: Continue above activities. In addition, at CNRI: integrate first program analysis and measurement tools into programming environment; small-scale roll-out of enhanced programming environment; develop initial example applications; define evaluation criteria for success. At CMU: user testing of program analysis tools. At U of C: integrate use of program analysis tools into curriculum.

- Year 4: Continue above activities, completing most of them. In addition, at CNRI: large-scale roll-out of enhanced programming environment; refine example applications; start large-scale collection of end user feedback; start working on Python language changes. At CMU: Completion of user testing, integration into Alice. At U of C: completion and roll-out of curriculum.

- Year 5: Project completion: report experiences, evaluate success, technology transfer to educational world and industry.

Optional Tasks

There are no optional tasks in the proposal.

References

[ABC]

http://www.cwi.nl/~steven/abc/

[Ball]

Thomas Ball and James R. Larus. Programs Follow Paths. Microsoft Research Technical Report MSR-TR-99-01, Jan. 1999.

[Alice]

http://www.alice.org/

[AOP]

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin. **Aspect-Oriented Programming.** In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP),* Finland. Springer-Verlag LNCS 1241. June 1997.

http://www.parc.xerox.com/spl/projects/aop/

[Blair]

http://www.mbhs.edu/

[Cartwright]

Robert Cartwright and Matthias Felleisen. Program verification through soft typing. ACM Computing Surveys, June 1996, 28(2): 349-351.

[CPAN]

http://www.cpan.org/

[Dertouzos]

Michael L. Dertouzos. The Future of Computing. *Scientific American*, Aug. 1999.

[Distutils]

http://www.python.org/sigs/distutils-sig/

[D-Lib]

http://www.dlib.org/

[Findler]

Robert Bruce Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. DrScheme: a pedagogic programming environment for Scheme. In *Proceedings of the 1997 Symposium on Programming Languages:*

Implementations, Logics, and Programs, Southampton, UK, Sept. 1997. (Lecture Notes in Computer Science, Vol. 1292.)

[Fishtank]

http://el.www.media.mit.edu/groups/el/projects/fishtank/

[Flanagan]

Cormac Flanagan and Matthias Felleisen. *Componential Set-Based Analysis. ACM Transactions of Programming Languages and Systems, to appear.*

[Geurts]

Leo Geurts, Lambert Meertens, Steven Pemberton. *The ABC Programmer's Handbook.* Prentice-Hall, 1990.

[Hugunin1]

Jim Hugunin. Python and Java - The best of both worlds. In *Proceedings of the 6th International Python Conference*, San Jose, Ca., Oct. 1997, pp. 11-20.

[Hugunin2]

Jim Hugunin. JPython Update. Invited talk, 7th International Python Conference. Houston, Tex., Nov. 1998. PowerPoint slides: http://www.jpython.org/jpython-talk-1.ppt

[Jackson]

Daniel Jackson and Allison Waingold. Lightweight Extraction of Object Models from Bytecode. In Proceedings of the International Conference on Software Engineering, Los Angeles, Ca., May 1999.

[JPython]

http://www.jpython.org/

[Knowbots]

http://www.cnri.reston.va.us/home/koe/

[Liskov]

http://sdg.lcs.mit.edu/~dnj/research/self-updating.html

[Logo]

http://el.www.media.mit.edu/groups/logo-foundation/

[LogoMation]

http://www.magicsquare.com/LM2/

[Lutz]

Mark Lutz. Programming Python. O'Reilly, 1996.

[Mozilla]

http://www.mozilla.org/

[Mailman]

http://www.list.org/

[Ousterhout]

John K. Ousterhout. Scripting: Higher Level Programming for the 21st Century. *IEEE Computer*, March 1998.

[Papert]

Mindstorms: Children, Computers, and Powerful Ideas. New York: Basic Books, 1980.

[Pausch]

Randy Pausch, Tommy Burnette, A.C. Capeheart, Matthew Conway, Dennis Cosgrove, Rob DeLine, Jim Durbin, Rich Gossweiler, Shuichi Koga, Jeff White. Alice: Rapid Prototyping System for Virtual Reality. ***IEEE Computer Graphics and Applications***, May 1995.

[Portolano]

http://www.cs.washington.edu/research/projects/portolano/

[Python]

http://www.python.org/

[SWIG]

http://www.swig.org/

[Tip]

Frank Tip. A survey of program slicing techniques. Journal of Programming Languages, 3(3):121-189, September 1995.

[TeachScheme]

http://www.cs.rice.edu/CS/PLT/Teaching/

[TJHSST]

http://www.tjhsst.edu/

[Yorktown]

http://yhspatriot.yorktown.arlington.k12.va.us/

[Watters]

Aaron Watters, Guido van Rossum, Jim Ahlstrom. *Internet Programming with Python.* MIS Press/Henry Holt, 1996.

[Wright]

Andrew K. Wright, Robert Cartwright. A Practical Soft Type System for Scheme. ACM Transactions of Programming Languages and Systems, Jan. 1997, 19(1): 87-152.