PROMOTING COMPUTER LITERACY THROUGH PROGRAMMING PYTHON

by

John Alexander Miller

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Education)
in The University of Michigan

2004

The Joys of the Craft

Why is programming fun? What delights may its practitioner expect as his reward?

First is the sheer joy of making things. As the child delights in his mud pie, so the adult enjoys building things, especially things of his own design. I think this delight must be an image of God's delight in making things, a delight shown in the distinctness and newness of each leaf and each snowflake.

Second is the pleasure of making things that are useful to other people. Deep within, we want others to use our work and to find it helpful. In this respect the programming system is not essentially different from the child's first clay pencil holder “for Daddy's office.”

Third is the fascination of fashioning complex puzzle-like objects of interlocking moving parts and watching them work in subtle cycles, playing out the consequences of principles built in from the beginning. The programmed computer has all the fascination of the pinball machine or the jukebox mechanism, carried to the ultimate.

Fourth is the joy of always learning, which springs from the nonrepeating nature of the task. In one way or another the problem is ever new, and its solver learns something: sometimes practical, sometimes theoretical, and sometimes both.

Finally, there is the delight of working in such a tractable medium. The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds his castles in the air, from air, creating by exertion of the imagination. Few media of creation are so flexible, so easy to polish and rework, so readily capable of realizing grand conceptual structures. …

Yet the program construct, unlike the poet's words, is real in the sense that it moves and works, producing visible outputs separate from the construct itself. It prints results, draws pictures, produces sounds, moves arms. The magic of myth and legend has come true in our time. One types the correct incantation on a keyboard, and a display screen comes to life, showing things that never were nor could be.

Programming then is fun because it gratifies creative longings built deep within us and delights sensibilities we have in common with all men. (p. 7)

Frederic Brooks, Jr.
The Mythical Man-Month, 1975

DEDICATION

Dedicated to my wife, Jane, and our children, Jessa and Jason.

ACKNOWLEDGMENTS

There were many people who encouraged me while I worked on this dissertation. First, I'd like to thank my parents, Mervil and Barbara Miller, who provided a loving, supportive and stable family environment conducive to intellectual pursuits. "A good beginning is half the battle."

I'd also like to thank Charles and Genevieve Peterson, who extended an essential educational career opportunity to me, and continue to bless me with caring support to my family as it grows. I would also like to thank the numerous students I had in my ESL classroom: their kindness, generosity and respect revealed to me the true joy of teaching.

Next, I'd like to thank my advisor, Frederick Goodman, for his practical wisdom and guidance. With our every conversation, I come away enriched and refreshed from the depths of his memory and experiences. He is an educator's educator, and I am honored to have had the opportunity to work so closely with him. I would also like to thank the other committee members, Carl Berger, Jay Lemke, and John Swales, for their suggestions and recommendations as the dissertation evolved.

I would like to thank my colleagues at the Interactive Communications and Simulations group: Jeff Stanzler, Jeff Kupperman, Gary Weisserman, Roger Espinosa, Denise Conanan, Douglass Scott, Edgar Taylor, and Michael Fahy. Spending the early, heady days of the World Wide Web together engendered many stimulating and enjoyable conversations.

I would also like to thank Gil Oswald for the time he took reviewing a draft of this manuscript, and for his gracious and thoughtful suggestions and feedback.

Finally, I would like to thank my wife, Jane, who persevered knowing that I could indeed bring this dissertation to completion.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF APPENDICES

CHAPTER 1

COMPUTER LITERACY

True, a chimpanzee could not begin to design a car. But, come to think of it, neither could I. Nor could you or any other person working in intellectual isolation—without the help of books, conversations, directions, documents, explanations, and traditions—design a car. Or even a bicycle. Or a pair of shoes. Or a mousetrap. Apes work in intellectual isolation because they lack language. We have language, and therefore our creations and inventions and technologies become collective efforts and cultural products. With your brain alone, with my brain alone (minus language and a language-based tradition), we would consider ourselves very lucky indeed to think of cracking nuts between a stone hammer and a stone anvil. Our greatest human creation is not the tool but the word, not the technology that we so treasure and depend on but the language that has allowed us to talk about it. Language, not technology, is the most compelling artifact of the human intellect.

—*Dale Peterson,* Eating Apes

1.0 Introduction

Computers permeate our lives at work and at home, satisfying professional and recreational goals, enabling a kind of *cybernetic*[1] activity that one could only imagine a few decades ago. There is a rich variety to these activities including: systems modeling, graphic designing, multimedia content creation and organizing, word processing, financial accounting and transacting, database processing of myriad types of information, scientific hypothesis testing, not to mention near ubiquitous e-mail corresponding, web

[1] "goal-oriented feedback mechanisms with learning" (Pickering, 1995, p. 31)

surfing and instant messaging activities that keep us in touch with each other. For many, if not most, there seems to be a natural affinity with these machines which gratify a cognitive desire to store, retrieve, and manipulate information that is important to each of us. However, it is important to realize that the cybernetic activities described above are mediated through computer applications that have been *written*. This particular form of writing is called computer programming and constitutes the domain of this dissertation.

In order to communicate with a computer a programmer must use a language. Although there are many computer languages from which to choose, there are few designed with the beginning programmer in mind. One language so designed is Python, and it currently enjoys a burgeoning popularity among computer programmers. Since computer programming must be learned, and since Python was designed to be easy to learn, this dissertation explores *what considerations are most important in teaching Python as a first programming language* in a secondary school setting. In order to address this issue, a large corpus consisting of over three years worth of messages posted to a public newsgroup was analyzed using innovative techniques to reduce the data and isolate the relevant portions. This dissertation, then, is also a study of *what techniques are efficacious in reducing large, textual datasets* and their applicability to other researchers engaged with similarly large, unstructured textual data.

In order to address the first question, it will be worthwhile framing it in a larger context of literacy, that is, what considerations are most important in teaching people to become increasingly literate in the age of the computer? We shall see that understanding literacy as a way of knowing enables us to understand computer literacy as an alternative and supplementary way of knowing; and furthermore, that computer programming is an essential component of computer literacy, analogous to the way writing is essential to traditional literacy.

This first chapter is a comparison of traditional print literacy with the much newer computer literacy. We will see how computer programming is related to computer literacy, and how computer literacy is more related to the rhetorical function of using computers to express oneself than to knowing how computers themselves work. We then look at how print and computer literacies enable learning and why computer programming is an essential component of that learning process.

The second chapter discusses what skills and processes are involved in writing a computer program, focuses on some connectionist considerations surrounding the teaching of programming, takes a look at the Python programming language itself, and describes the genesis of the data source.

The third chapter explicates the methods and procedures used to analyze the data, distinguishes what was considered relevant from irrelevant in the data, and describes the eight categories that emerged and served as a framework for the analysis.

The fourth chapter provides the results of the analysis. Going category by category, the concerns of the posters that addressed this dissertation's topic are detailed and summarized. Also, the results of using the methods and procedures delineated in the third chapter are discussed.

The fifth and final chapter presents the conclusions drawn from the results and considers the larger role of programming as a literate practice.

1.1 Literacies

The term 'literacy' is somewhat charged with contested meanings. A sense of this can be gleaned from the Usage Note for the word 'literate' in the American Heritage Dictionary (1992):

> For most of its long history in English, literate has meant only "familiar with literature," or more generally, "well-educated, learned"; it is only during the last hundred years that it has also come to refer to the basic ability to read and write. … More recently, the meanings of the words literacy and illiteracy have been extended from their original connection with reading and literature to any body of knowledge. For example, "geographic illiterates" cannot identify the countries on a map, and "computer illiterates" are unable to use a word-processing system.

Certainly, the use of the term 'literacy' in this work is governed more by the later, more modern sense indicated above. In other words, I take computer literacy to be more inclusive than just 'familiarity with computers'; I explicitly include a 'reading and writing' component. In the context of computer usage, 'reading' roughly corresponds to the ability to use a computer's operating system and applications productively (which may include the reading and writing skills of traditional print literacy), while 'writing' roughly corresponds to the ability to program a computer.

The opposite of literacy is illiteracy (keeping in mind that these are measured in degrees rather than existing as categorical states), and it is often claimed that print illiteracy rates are rising in this country. Mihai Nadin (1997) expresses this complaint in *The civilization of illiteracy*:

> We notice that literate language use does not work as we assume or were told it should, and wonder what can be done to make things fit our expectations. Parents hope that better schools with better teachers will remedy the situation. Teachers expect more from the family and suggest that society should invest more in order to maintain literacy skills. Professors groan under the prospect of ill-prepared students entering college. Publishers redefine their strategies as new forms of expression and communication vie for public attention and dollars. Lawyers, journalists, the military, and politicians worry about the role and functions of language in society. … The major accomplishment of analyzing illiteracy so far has been *the listing of symptoms:* the decrease in functional literacy; a general degradation of writing skills and reading comprehension; an alarming increase of *packaged* language (clichés used in speeches, *canned* messages); and a general tendency to substitute visual media (especially television and video) for written language. (pp. 3-4)

However, instead of bemoaning the decline of literacy, Nadin embraces the notion that its decline is inevitable, and seeks to describe human life and interactions that emerge in an 'illiterate' world:

> The decline of literacy is an encompassing phenomenon impossible to reduce to the state of education, to a nation's economic rank, to the status of social, ethnic, religious, or racial groups, to a political system, or to cultural history. There was life before literacy, and there will be life after it. … My position in the discussion is one of questioning historic continuity as a premise for literacy. If we can understand what the end of literacy as we know it means in practical terms, we will avoid further lamentation and initiate a course of action from which all can benefit. Moreover, if we can get an idea of what to expect beyond the safe haven now fading on the horizon, then we will be able to come up

> with improved, more effective models of education. … This leads me to state from the outset—almost as self-encouragement—that literacy, whose end I discuss, will not disappear. … For the majority, it will continue in *literacies* that facilitate the use and integration of new media and new forms of communication and interpretation. The utopian in me says that we will find ways to *reinvent* literacy, if not save it. … We give life to images, sounds, textures, to multimedia and virtual reality involving ourselves in new interactions. Transcending boundaries of literacy in practical experiences for which literacy is no longer appropriate means, ultimately, to grow into a new civilization. (pp. 5-7)

It is within this sense of 'transcending boundaries of literacy' that I frame my argument for computer programming. I do not argue that the rise of illiteracy must be quelled, rather, I believe that computer literacy will arise as one of the new literacies Nadin speaks of, and that active promotion of computer programming will lead to computer *fluency,* which, due to its reliance on specific written syntaxes, will also reinvigorate traditional print literacy skills.

This reinvigoration is implied by Knuth (1992) and his thoughts on Literate Programming:

> Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.
>
> The practitioner of literate programming can be regarded as an essayist, whose main concern is with exposition and excellence of style. Such an author, with thesaurus in hand, chooses the names of variables carefully and explains what each variable means. He or she strives for a program that is comprehensible because its concepts have been introduced in an order that is best for human understanding, using a mixture of formal and informal methods that reinforce each other. (p.99)

A similar vision of intertwining literacies is offered by Olson (1985):

> Computers simply raise, by an order of magnitude, the requirement of making meanings explicit—a requirement that was begun by giving communication systems a semantics and by then making that semantics more explicit and elaborate through literacy. Far from being obsolete, literate competencies are basic to computational ones. To be intelligent in the society of computer users is to be skilled in making one's meanings explicit. In the 20th century, the world of computing, we are succeeding in doing what the 16th century, the world of literacy, aspired to do—make meanings clear and explicit. (p.7)

Knuth challenges programmers to write programs that are 'explicit' in *two* ways, explicit to the computer, in the sense Olson describes, but also explicit to other *human* readers of software programs. In a sense, I'm saying we can have our cake and eat it too: Computer literacy is emerging as an alternative literacy in the way Nadin describes, but as such, computer literacy (especially programming) is so strongly dependent upon traditional print literacy skills that promoting computer programming also strengthens those traditional skills.

Furthermore, we can understand this strengthening, this interdependence of literacies in terms of semiotics as illustrated by Barthes and Lavers (1993) in *Mythologies* where they define myth as a second-order semiotic system. Using their system, I showed how computer literacy and print literacy are each second-order semiotic systems where mastery of the first-order semiotic system is a prerequisite for mastery of the second-order one (Miller, 2001). In this case, computer programming strongly depends on a specific written syntax, so at least minimal competence in print literacy skills is necessary for the budding programmer.

Besides extending print literacy skills, computer literacy can also extend scientific literacy skills. In *A New Kind of Science* (Wolfram, 2002) we are offered a vision of ways computer literate scientists can conduct experiments and perform research using computers. Wolfram suggests that much research in the future may be done this way:

> For with calculus there was finally real success in taking abstract rules created by human thought and using them to reproduce all sorts of phenomena in the natural world.
>
> But the particular rules that were found to work were fairly sophisticated ones based on particular kinds of mathematical equations. And from seeing the sophistication of these rules there began to develop an implicit belief that in almost no important cases would simpler rules be useful in reproducing the behavior of natural systems.
>
> During the 1700s and 1800s there was ever-increasing success in using rules based on mathematical equations to analyze physical phenomena. And after the spectacular results achieved in physics in the early 1900s with mathematical equations there emerged an almost universal belief that absolutely every aspect of the natural world would in the end be explained by using such equations.
>
> Needless to say, there were many phenomena that did not readily yield to this approach, but it was generally assumed that if only the necessary calculations could be done, then an explanation in terms of mathematical equations would eventually be found.
>
> Beginning in the 1940s, the development of electronic computers greatly broadened the range of calculations that could be done. But disappointingly enough, most of the actual calculations that were tried yielded no fundamentally new insights. And as a result many people came to believe—and in some cases still believe today—that computers could never make a real contribution to issues of basic science.
>
> But the crucial point that was missed is that computers are not just limited to working out consequences of mathematical equations. And indeed, what we have seen in this chapter is that there are fundamental discoveries that can be made if one just studies directly the behavior of even some of the very simplest computer programs. (pp. 44-45)

Thus, scientific computer simulations are *not* just the working out of mathematical equations, but rather, the execution of *algorithms* that may model physical phenomena that cannot be represented by equations. And it is the conceptualization and embodiment in code of those algorithms that will more and more constitute the life of the mind of future scientists.

Another way scientific literacy intersects with computer literacy is the growing field of genetic programming. In “Evolving Inventions” (Koza, Keane et al. 2003) write:

> The first practical commercial area for genetic programming will probably be design. In essence, design is what engineers do eight hours a day and is what evolution does. Design is especially well suited to genetic programming because it presents tough problems for which people seek solutions that are very good but not mathematically perfect. Generally there are complex trade-offs between competing considerations, and the best balance among the various factors is difficult to foresee. Finally, design usually involves discovering topological arrangements of things (as opposed to merely optimizing a set of numbers), a task that genetic programming is very good at. (p. 54)

Furthermore, as such computer literate scientists perform algorithm-based, simulation-based and genetic programming-based experiments with their computers (rather than, or alongside, mathematical equation-based ones) they can help educate a new generation of student/scientists who are ‘looking over their shoulders’ via internet-based legitimate peripheral participation (Lave and Wegner, 1991). The rise of computer-mediated communication systems enables a kind of collaboration-at-a-distance not only between scientists who are peers, but also between master scientists and student apprentices.

1.2 Literacy Analogy

In order to make the connection between print literacy and computer literacy, I would like to elaborate on an analogy:

```
Reading : Writing :: Using a computer : Programming a computer
```

In order to precisely discriminate what is meant by each term in the analogy, it helps to define each of them as having two aspects: an external function and an internal function. Reading, for example, has the external, or mechanical, function of seeing text on a surface (or, more generally, perceiving signs in an environment) *and* the internal function of interpreting that text to assign meaning(s) to it (usually, but not always, with the aim of reproducing meaning(s) intended by the author in the reader's mind). Writing also has an external, or mechanical, function of manipulating a writing instrument (pen, typewriter, word processor) on a surface (paper, screen) to create text *and* an internal function of strategizing which words, phrases, ideas, and rhetorical devices to 'textualize' in order to create the written artifact.

Likewise, the other two, computer-based terms in the analogy have a dual aspect akin to reading and writing, so that we may say 'using a computer' has the external, mechanical function of seeing signs on a surface *and* the internal function of assigning meaning to those signs, while 'programming a computer' has the external function of manipulating a device (computer keyboard and mouse) *and* the internal function of devising and designing data structures and algorithms using a computer language's syntax and vocabulary to create a programming artifact.

Although both sets of literacy skills, reading and writing text, and using and programming computers, involve the encoding and decoding of signs (discussed shortly), the salient difference between them is the object of their attention. With written artifacts the object of attention is *what is said (written)*, whereas with programmatic artifacts, the object of attention is *what is done*. Suppose, for example, we are examining a PDF (Portable Document Format) document on a computer screen. To the extent that our attention is focused on the textual content of the document, we are engaged in a traditional print literacy skill: reading. However, to the extent that our attention is focused on jumping to page 25 of that document, or adding a bookmark to it, or adding a personal comment, or copying a passage, or zooming in for an enlarged view, or clicking on a hyperlink, or emailing the document to a colleague, or printing selected pages, or even using the Help menu to find out how to do any of these actions with the document, we are engaged in a computer literacy skill: using a computer.

1.3 Computer Literacy Conventions

In the past two decades, computers have become almost ubiquitous (in the US and other developed countries). Computers enhance and extend traditional print (and other kinds of) literacy skills by greatly augmenting one's ability to manipulate the symbols of literacy: alphabetic characters, words, paragraphs, documents, references, numerals, equations, charts, maps, images, graphics, sounds, video, etc. However, these machines also

Table 1 Typical literacy conventions

Alphabet	Footnotes
Spelling	Endnotes
Punctuation	Indexes
Grammar	Tables of Contents
Sentences	Parenthetical comments
Paragraphs	Authors
Rhetorical Devices	Textbooks
Chapters	Reference Works
Quoting	Periodicals
Stories	Letters
Poetry	Publishers
Novels	Anthologies

impose an additional cognitive load on the various literacy skills they enhance. For example, before computers, being literate consisted of learning a wide-ranging set of conventions applied to written text (see Table 1).

The pervasive expectation threading throughout all these literary conventions was that the symbols being manipulated were either immutably printed, or relatively fixed, on paper. This, in turn, imposed certain limitations on the degree of interactivity with those symbols by both writer and reader. With computers and word-processing applications and the Internet, these limitations have greatly diminished, and expectations of what the literacy skill set consists of have risen concomitantly. For instance, cutting and pasting text from one document to another, formatting a document for publication in a journal, finding and replacing all occurrences of the word 'skill' with the word 'ability' throughout a document, inserting a bar chart into a document, and posting a version of an article on a webpage are all examples of computer-assisted literacy skills that can reasonably be

Table 2 Typical computer literacy conventions

Files	Folders
Undo	Copy and Paste
Find and Replace	Multimedia Presentation
'Desktop'	'Clipboard'
Attachment	Spreadsheet

expected of a literate person these days. In other words, these procedures and their artifacts constitute additional conventions that could be added to a list like the one above (see Table 2).

This raises a question. Do these additional operating system and word-processing skills constitute an *extension* of what it means to be literate, or do they, along with other computer-enhanced symbol-manipulation skills such as using a spreadsheet and presentation software and digital photo editing, constitute a new *kind* of literacy, computer literacy?

I argue that facility with computers is indeed a new kind of literacy. The earlier literacies (print literacy, numeric literacy, media literacy) represent ways of communicating with others where 'others' in those cases are usually assumed to be 'other humans' (with the rare exception being chimpanzees or dolphins). However, in this case, computer literacy represents a way of communicating with, not another human (directly), but with a machine (or perhaps more accurately, a machine system, if the computer is attached to a network)[2]. Even though it is probably most often the case that the reason the user is interacting with a machine is to create an artifact (or message) which will be communicated to another human, the concepts involved in accomplishing this task are different enough from those of traditional print literacy that it constitutes a qualitatively different experience which deserves a 'literacy' appellation. We expect computer users to design artifacts, retrieve information, configure operating systems, program and use applications, maintain hard drives, as well as have some sort of understanding of how their computers, networks, applications, file systems and digital objects work. That is, there now exists a rich set of abstract symbols constituting operations on 'computable resources' which manipulators ('writers' and 'readers') of those symbols need to learn in order for meaningful communication (between human and machine) to occur and which we can refer to as computer literacy.

For example, sending an e-mail attachment to a newsgroup constitutes a symbol manipulation that requires a certain amount of care and understanding that is different from what is involved with sending the same information through postal mail. There are a host of variables to consider when performing this seemingly simple act, and becoming computer literate involves anticipating and dealing with those variables. First there is the

[2] It could be argued that it is humans who created the computer and the communication protocols for interacting with it, so communication with a machine is indirectly communication with the person (or committee, or company) that created the machine and its protocols.

attachment's file format. Most any recipient will likely be able to open a simple text file. However other text documents created by proprietary word processors require the same or similar application on each recipient's machine (or at least some way of getting it opened remotely). There is also the issue of how the newsgroup server handles attachments: some let them through; many others discard them. How is the attachment to be compressed, if at all? What about language encoding issues? What about the perceived risk of viruses hidden in attachments? Is there another way to distribute the information?

Understanding these and other issues means understanding in some way how computers operate, and how our interactions with computers affect the way our messages and other artifacts are handled. Understanding many of these issues involves understanding *conventions* which, when consistently invoked by a critical mass of users, becomes part of the knowledge base for interacting with computers. These conventions, in turn, create expectations about the process of creating, sending and receiving messages and artifacts mediated through computers.

Three kinds of signs

Computer code consists of sequences of signs generated by humans in a computing environment. Charles Sanders Peirce postulated that there were three kinds of signs: icons, indexes, and symbols. These three kinds are distinguished by the relationship they hold with their 'dynamic object,' or that which the sign refers to (Boyarin, 1992, p. 115-116). He identified icons as signs that are determined by their dynamic object by virtue of their own internal nature and are characterized by qualities of feeling and unity. For example, icons easily identify many corporations, flags serve as iconic representations of countries, and many traffic *signs* are iconic rather than textual. In this context, we note that computer interactions utilizing a graphical user interface heavily rely on the use of icons to identify applications and document types (for example, .pdf, .doc, .qt, .psd and .zip files); eventually, these icons become conventional.

Peirce says that the second kind of sign, indexes, are determined by their dynamic object by virtue of being in a real relation to it and are characterized by the experience of effort and action. For example, proper names serve as indexes to the people they refer to, as do symptoms of diseases. In computer literacy, the naming of the documents we create is a primary example of indexical signs. Often, the extensions at the end of the filename are used to identify the kind of file it is, and these indexical extensions also become conventional (for example, .jpg, .pdf, .doc, .gif, .txt, etc.) A similar convention exists for the naming of network domains.

However, it is the third kind of sign we are most interested in, symbols, which are signs that are determined by their dynamic object only in the sense that they will be so interpreted. In other words, a symbol's meaning is wholly arbitrary, and as such, depends on convention, habit, or a natural disposition of the interpretant. This, of course, includes programming languages such as Python where the symbols used are wholly arbitrary (yet resonant with the written English language, a much larger set of arbitrary symbols) and where the programmer learns the conventions of the language in order to write expressions that the Python interpreter can properly interpret. So, we see that computer literacy consists of learning a set of conventions arising out of the use of different kinds of signs. This applies not only to the use of computers, but also encompasses the programming of computers.

Three pillars of literacy

Andrea diSessa (2000), in *Changing Minds: Computers, Learning and Literacy* describes what he calls the three pillars of literacy. The first is the material pillar involving "external, materially based signs, symbols, depictions, or representations." The components of interacting with a computer, including computer code, certainly consist of these kinds of materials as we just saw in the discussion of signs. DiSessa also says that the material pillar of literacy has two important features: they are technologically dependent, and they are designed. Both of these features are true of computer artifacts and computer languages.

The second pillar of literacy is mental or cognitive, that is, the inscriptions of the material pillar are meaningless without a corresponding consciousness to bring meaning to them. "Clearly the material basis of literacy stands only in conjunction with what we think and do with our minds in the presence of inscriptions" (p. 8). Indeed, this cognitive pillar constitutes much of the effort of learning a programming language, as detailed in the five kinds of programming knowledge in the second chapter (the knowledge framework for teaching programming). But it also presupposes much auxiliary knowledge associated with computer literacy, that is, knowing how to use a computer.

The third pillar of literacy is social, that is, a literacy does not exist solely with one person. Its conventions are shared among a group of people who use it to communicate or calculate or perform some other function of importance to the group. Computer use is certainly social in that many people share the conventions of using their machine's operating system, and of the applications that run on it. Programming languages, such as Python, are also included in that social milieu by having their own set of conventions that are shared by the programmers who learn that language.

Having described these three pillars, diSessa (2000), defines a central hypothesis of literacy: "A literacy is the convergence of a large number of genres/social niches on a common, underlying representational form" (p. 24). We see this definition in action by considering the multitude of ways people use their computers. Thus, one can send and receive a variety of text-based messages such as e-mail, word processing documents, newsletters, etc. using a computer ('a common, underlying representational form'). One can read and create static graphic-based messages such as spreadsheet documents, graphing calculator equations, drawings, presentations etc. using a computer ('a common, underlying representational form'). One can read and create dynamic graphics such as, movies, slideshows, virtual reality scenes, etc. using a computer ('a common, underlying representational form'). Across all of these activities (genres, social niches), is a quickly evolving (not yet universal) set of conventions that enable users to 'read' the computer applications being employed for their tasks, 'write' the messages they wish to convey, and ultimately, create new message-creating tools themselves.

Increasingly, too, this 'body of knowledge' that computer literacy is comprised of is coming to resemble the cognitive structures that computer programmers use to construct their programs and applications, as if these artifacts that computer users create were 'programmed'. Harrell (2003) details this evolution:

> The tools used to present and create media art lie behind every media artwork. The theory of programming languages is a useful means by which to characterize these media. Formal languages offer broad insight into the nature of computational manipulation and specific organizational structures of imperative languages reveal reflections of these structures in media software. This is a natural reflection because the theory of languages expresses organized models for executing algorithms and structuring data, which are the types of manipulations human creators perform on media when treating it as computational data.

Here we see again an example of diSessa's definition of a literacy: "the convergence of a large number of genres/social niches on a common, underlying representational form," that is, a large number of media arts niches are converging on an underlying representational form that look and behave very much like programming languages.

Computer Fluency

Just as there are degrees of print literacy, ranging from barely literate to highly literate, there are also degrees of computer literacy. Lawrence Snyder, chairman of the Committee on Information Technology Literacy that produced “Being Fluent With Information Technology,” a report published in June 1999 by the National Research Council, in an interview with Florence Olsen (2000), discusses the differences between computer literacy and computer fluency:

> Think of fluency as having three kinds of knowledge—skills , concepts, and logical reasoning. Skills are knowing how to use e-mail, browse the Web, and so forth. The basic concepts that students need to know for fluency are such things as how does a computer work, what is a network, how do we represent information digitally, algorithmic thinking, things like that. The intellectual capabilities needed for fluency include logical reasoning, the ability to manage complexity, to troubleshoot and debug information systems.

One way to promote such computer fluency is to engage students in computer-based projects:

> The report proposed teaching fluency in the context of projects that require students to use those three kinds of knowledge. So let me give you an example: formulating an H.I.V.-tracking system for a hospital or doctor's office. It's a great project. It is a database project, because you need to record clients coming into the clinic, to keep track of the specimens they give, and where the specimens are sent out for testing. A project gives you a chance to learn and practice three or four skills, three or four concepts, and three or four capabilities.

One place to embed such computer projects is in programming classes. This wouldn't necessarily mean teaching C++ or Java or other 'heavy-duty' compiled languages. Guido van Rossum, originator of the Python programming language, initiated a general computer literacy research effort called Computer Programming for Everybody (CP4E) (1999) which purports 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." He goes on to describe the goals and motivation of this research effort:

> 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.
> …
> 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, 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.
> …
> 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.

By making computer literacy more commonplace, van Rossum hopes to answer the question, “What will happen if users can program their own computer?” In other words, what changes to individuals and society, analogous to the changes wrought by mass print literacy, will occur when programming computers is as ubiquitous as, say, using a word processor is now? Clark (1997) puts this process in the context of becoming more cyborg-like:

> We see some of the 'cognitive fossil trail' of the Cyborg [cybernetic organism] trait in the historical procession of potent Cognitive Technologies that begins with speech and counting, morphs first into written text and numerals, then into early printing (without moveable typefaces), on to the revolutions of moveable typefaces and the printing press, and most recently to the digital encodings that bring text, sound and image into a uniform and widely transmissible format. Such technologies, once up-and-running in the various appliances and institutions that surround us, do far more than merely allow for the external storage and transmission of ideas. They constitute, I want to say, a cascade of 'mindware upgrades': cognitive upheavals in which the effective architecture of the human mind is altered and transformed.

We understand the word ‘cyborg’ in the sense defined by Haraway (1992) where she says “the cyborg is the figure born of the interface of automaton and autonomy” (p. 139, in Aarseth, 1997, p. 54); we can think of cyborg as the symbiosis of mechanism and organism. This points to one possible direction that society may evolve as programming approaches the ubiquity mass print literacy has currently achieved.

Computer Literacy as the New Rhetoric

Why study computers? Is there any value to such study? One suggestion, which we might call the joy of proficiency, comes from Ong (1991):

> Technologies are artificial, but—paradox again—artificiality is natural to human beings. Technology, properly interiorized, does not degrade human life but on the contrary, enhances it. The modern orchestra, for example, is the result of high technology. A violin is an instrument, which is to say a tool. An organ is a huge machine, with sources of power – pumps, bellows, electric generators – totally outside its operator. Beethoven's score for his Fifth Symphony consists of very careful directions to highly trained technicians, specifying exactly how to use their tools. … The fact is that by using a mechanical contrivance, a violinist or an organist can express something poignantly human that cannot be expressed without the mechanical contrivance. To achieve such expression of course the violinist or organist has to have interiorized the technology, made the tool or machine a second nature, a psychological part of himself or herself. This calls for years of 'practice', learning how to make the tool do what it can do. Such shaping of a tool to oneself, learning a technological skill, is hardly dehumanizing. The use of a technology can enrich the human psyche, enlarge the human spirit, intensify its interior life. (p. 83)

Ong is comparing the technology of musical performance with the technology of writing and arguing that competence in either enhances our lives. To achieve a similar effect of enrichment, enlargement and intensification in our lives through the technology of computer literacy requires a similar process of "interiorization". This dissertation explores the practices of programming teachers that can guide us to imbue students with at least a taste of the satisfaction that comes from becoming computer literate.

To achieve a similar competency with the technology of traditional print literacy, we learned what was traditionally called rhetoric:

> We have in the West a venerable tradition of studying how human attention is created and allocated: the "art of persuasion" which the Greeks called rhetoric. A better definition of rhetoric, in fact, might be "the economics of human attention-structures," for whenever we "persuade" someone, we do so by getting that person to "look at things

> from our point of view," share our attention-structure. (Lanham, 1991, p. 227)

According to Michael Goldhaber, attention is the currency of cyberspace:

> So a key question arises: Is there something else that flows through cyberspace [besides information], something that is scarce and desirable? There is. No one would put anything on the Internet without the hope of obtaining some. It's called attention. And the economy of attention - not information - is the natural economy of cyberspace. … In his online book Virtual Community, Howard Rheingold lays out two guidelines: "Rule Number One is to pay attention. Rule Number Two might be: Attention is a limited resource, so pay attention to where you pay attention." (1997a; see also Goldhaber, 1997b)

Thus, some of these modern expressions of rhetoric, or "the science of human attention-structures" (Lanham, 1991, p. 134), such as web sites and various other activities emerging on the Internet, are growing in importance. Computer literacy will be essential if students are to succeed in the process of "interiorizing" web skills that enable competency in these rhetorical activities. We note that the term rhetoric has a somewhat negative connotation, and for Lanham, unjustifiably so:

> The intellectual structures of formal rhetoric have formed part of Western culture for so long, and yet we have for so long suspected and despised rhetoric as simply hypocrisy and deception, that it is very difficult to recognize it for what it is—an information *system*. Systems, at least for humanists, have never escaped from the Platonic orbit; they are closed patterns organized like human society in the *Republic*. Everyone has a single job; every element a fixed place; the aim is perfect stasis, with an emphasis on both "perfect" and "stasis." Our notion of systems is Platonic philosophy on the one hand and physics on the other. What Plato wanted above all to exile from his utopia, like Thomas More after him, was *style*, the unabridged range of ornament, of purposeless play. Rhetoric defines itself as a counter-system to the Platonic political order by admitting stylistic, ornamental behavior, by acknowledging that such behavior lies at the heart of human life, is what human politics is all about. If stylistic behavior is acknowledged as part of the complex human "reason," then rhetoric becomes the

> systematic attempt to account for this complex "reason," and find agreements within it. (Lanham, 1993; p. 57)

Lanham's major thesis is that philosophy (think 'science') and rhetoric (think 'humanities' or 'arts') are interestingly antagonistic, that philosophy has been ascendant for the past three hundred years or so, that rhetoric is making a comeback (through much of what Ong refers to as 'secondary orality'), and that digital technology is making these new rhetorical devices available to the masses:

> The quarrel between the philosophers and the rhetoricians constitutes *the* quarrel in Western culture. McLuhan's argument for electronic media reintroduced the rhetorician's conception of language, and of human self and society, after three hundred years dominated by the philosophers, with their strongly opposed conceptions of language and social reality. … The rhetorical/philosophical distinction, though it grows from the technological distinction between oral and literate cultures, concerns more than technology. It debates opposed theories of human motive, human selfhood, and human society. (pp. 202-203)

> Classical rhetoric, and hence all of classical education, was built on a single dominant exercise: modeling. The key form was the oration, and it was rehearsed again and again in every possible form and context. *Declamatio*, as the modeling of speeches came to be called, stood at the hub of Western education, just as computer modeling is coming to do today. The world of electronic text has reinstated this centrality of modeled reality. The computer has adopted once again, as the fundamental educational principle, the dramatizing of experience. (p. 47)

This 'dramatizing of experience' extends beyond language; we can also, says Richard Buchanan (1989), think of design as rhetoric:

> Communication is usually considered to be the way a speaker discovers arguments and presents them in suitable words and gestures to persuade an audience. The goal is to induce in the audience some belief about the past (as in legal rhetoric), the present (as in ceremonial rhetoric), or the future (as in deliberative or political rhetoric). The speaker seeks to provide the audience with the reasons for adopting a

> new attitude or taking a new course of action. In this sense, rhetoric is an art of shaping society, changing the course of individuals and communities, and setting patterns for new action. However, with the rise of technology in the twentieth century, the remarkable power of man-made objects to accomplish something very similar has been discovered. By presenting an audience of potential users with a new product—whether as simple as a plow or a new form of hybrid seed corn, or as complex as an electric light bulb or a computer—designers have directly influenced the actions of individuals and communities, changed attitudes and values, and shaped society in surprisingly fundamental ways. This is an avenue of persuasion not previously recognized, a mode of communication that has long existed but that has never been entirely understood or treated from a perspective of human control such as rhetoric provides for communication in language. (p. 93)

And just as Lanham seeks to reunite philosophy and rhetoric into a more dynamic oscillation than existed previously through the use of computer technology, Buchanan does also for science and design through the use of technology writ large:

> The primary obstacle to such understanding is the belief that technology is essentially part of science, following all of the same necessities as nature and scientific reasoning. If this is true, technology cannot be part of design rhetoric, except as a preformed message to be decorated and passively transmitted. Design then becomes an esthetically interesting but minor art that is easily degraded into a marketing tool for consumer culture. However, if technology is in some fundamental sense concerned with the probable rather than the necessary—with the contingencies of practical use and action, rather than the certainties of scientific principle—then it becomes rhetorical in a startling fashion. It becomes an art of deliberation about the issues of practical action, and its scientific aspect is, in a sense, only incidental, except as it forms part of an argument in favor of one or another solution to a specific practical problem. (p.93-94)

Buchanan's, (and by analogy, Lanham's) argument rests on the premise that (computer) technology is more concerned with the 'probable', with the 'contingencies of practical use and action', than with 'necessity', with the 'certainties of scientific principle'. If we accept this premise, then, for

Buchanan, design becomes strongly rhetorical (in a positive sense), able to persuade in ways akin to spoken and written rhetorical devices. And for Lanham, rhetoric itself becomes more of an equal partner with philosophy.

What this suggests is that computer literacy, becoming proficient with computer technology, can be situated closer to rhetoric than philosophy, closer to design than science. Becoming computer literate, therefore, means less of 'learning about how computers work' and more of 'using computers to express oneself.' The myriad variety of digital messages, *designed* artifacts that have been, and continue to be created to persuade an audience of some idea or action, can be studied in a manner analogous to the way traditional spoken and written communication artifacts were studied in classical rhetoric. And furthermore, achieving competency, proficiency, or even virtuosity in the study and creation of such digital artifacts can enrich, enlarge and intensify our lives in the way Ong suggests musicianship can.

Expressions and Representations

Finally, if we consider computer literacy as the modern rhetoric uniting philosophy/science with humanities/design/arts, we need to carefully consider the nature of the artifacts each domain creates.

Jerome Bruner (1986) claims that "there are two modes of cognitive functioning, two modes of thought, each providing distinctive ways of ordering experience, of constructing reality" (p. 11). He goes on to say:

> Each of the ways of knowing, moreover, has operating principles of its own and its own criteria of well-formedness. They differ radically in

> their procedures for verification. A good story and a well-formed argument are different natural kinds. Both can be used as means for convincing another. Yet what they convince *of* is fundamentally different: arguments convince one of their truth, stories of their lifelikeness. The one verifies by eventual appeal to procedures for establishing formal and empirical proof. The other establishes not truth but verisimilitude. (p. 11)

These two modes correspond to two traditional philosophical functions of language, namely, representation and expression. A useful way to distinguish these two functions is to examine Richard Rorty's (1989) analysis of metaphor[3] as seen by positivists and romantics:

> The Platonist and the positivist share a reductionist view of metaphor: They think metaphors are either paraphrasable or useless for the one serious purpose which language has, namely, representing reality. By contrast, the Romantic has an expansionist view: He thinks metaphor is strange, mystic, wonderful. Romantics attribute metaphor to a mysterious faculty called the "imagination," a faculty they suppose to be at the very center of the self, the deep heart's core. Whereas the metaphorical looks irrelevant to Platonists and positivists, the literal looks irrelevant to Romantics. For the former think that the point of language is to *represent* a hidden reality which lies outside us, and the latter thinks its purpose is to *express* a hidden reality which lies within us.
>
> Positive history of culture thus sees language as gradually shaping itself around the contours of the physical world. Romantic history of culture sees language as gradually bringing Spirit to self-consciousness. (p.19, my emphasis)

Thus, representation corresponds to Bruner's 'well-formed argument,' while expression corresponds to his 'good story' (where, of course, metaphors are welcome!) I do not subscribe to either the positivist or the romantic view entirely; it appears that both functions of language are valid, and that these

[3] In his discussion, Rorty is describing Donald Davidson's view of language, whose view is antithetical to either a positivist or romantic view.

functions interpenetrate each other, that is, representations of external reality are *also* used to express aspects of our interior lives, and that expressions of our interior lives are *also* used to represent exterior reality. The former establishes the ground for our story-telling expressions (but of course stories need not be constrained by objective reality), while the latter gives life to otherwise lifeless scientific representations (but such subjective appeals need not be considered when evaluating their validity.)

This split between the two modes of cognitive functioning runs deeply through our modern educational system. Kieran Egan (2001) discusses this split and how one function is more closely associated with oral language, while the other is more closely related to written language. He describes the 'expression' function as socialization, that is, the function of schools to socialize our students into the larger society in which our schools are embedded. This is largely accomplished through story-telling, the ancient art that emerged after the development of human language. This socialization function was developed by hunter-gatherer tribes to imbue their young with

> images of who 'we' are and what we are doing here—in this forest, on this plain, by this seashore, among these hills, alongside these animals, under these stars—and where we are going next. The stories typically told about gods or sacred ancestors who warranted the norms and values that constituted the culture of the particular hunter-gatherer society. (Egan, 2001, p. 924)

Here we see the appeal to rhetorician-artists and their affinity for expressions of verisimilitude.

Egan describes the other cognitive function, as embodied by representations or well-formed arguments, as being founded on the academic ideal that emerged after the development of literacy:

> Literacy has allowed generations of people to record their knowledge and experience. Further generations can compare that recorded knowledge with what they can see or discover and leave a more accurate record; and they can compare other's experience with their own, enlarging and enriching their experience in consequence. Today we have stored vast amounts of knowledge in written records and we have access to a vast array of varied human experience. These enable our minds to transcend our own time, place, and circumstances. … When the best accumulated knowledge coded in writing is learned, Plato taught, it transforms the mind of the learners and enables them to understand the world more accurately and truly. (p. 928-9)

This clearly shows the emphasis on representations and its appeal to the philosopher-scientist.

We see now that one mode of thought, that of the well-formed argument, is the goal of the philosopher/scientist, while the other, that of the good story, is the goal of the rhetorician/artist. Bruner says that the first mode, "the paradigmatic or logico-scientific one, attempts to fulfill the ideal of a formal, mathematical system of description and explanation. It employs categorization or conceptualization and the operations by which categories are established, instantiated, idealized, and related one to the other to form a system" (p. 12). Whereas the second, "the narrative mode leads instead to good stories, gripping drama, believable (though not necessarily 'true') historical accounts. It deals in human or human-like intention and action and the vicissitudes and consequences that mark their course" (p. 13). Each of these modes is essential to a full human life; additionally, each constitutes a world in which facility with computers and computing can assist in the creation of the computer literate user's messages and artifacts of representation, or messages and artifacts of expression. Later, we will see how these two cognitive modes are used in computer programming.

1.4 Literacy enables learning

One of the primary effects of *learning to read* is enabling students to *read to learn*. Indeed, reading is perhaps the primary method for reproducing knowledge in school, thus acquiring this skill is of paramount importance in schooling. There is the corollary maxim that one effect of *learning to write* is to *write to learn*; that is, to use the process of writing to explore what one knows and further and/or solidify one's understanding of a topic by expressing it to others textually. Ong (1988) reminds us:

> To say writing is artificial is not to condemn it but to praise it. Like other artificial creations and indeed more than any other, it is utterly invaluable and indeed essential for the realization of fuller, interior, human potentials. Technologies are not mere exterior aids but also interior transformations of consciousness, and never more than when they affect the word. Such transformations can be uplifting. Writing heightens consciousness. Alienation from a natural milieu can be good for us and indeed is in many ways essential for full human life. To live and to understand fully, we need not only proximity, but also distance. This writing provides for consciousness as nothing else does. (p. 82)

These effects are sometimes achieved in schools, and, by analogy, point to one reason why it may be important to become computer literate. Before exploring that, however, there is another notable effect that being literate enables: participation in communities. For example, Benedict Anderson

(1991) describes how reading the newspaper fosters a shared sense of community:

> In this perspective, the newpaper is merely an “extreme form” of the book, a book sold on a colossal scale, but of ephemeral popularity. Might we say: one-day best-sellers? The obsolescence of the newspaper on the morrow of its printing–curious that one of the earlier mass-produced commodities should so prefigure the inbuilt obsolescence of modern durables–nonetheless, for just this reason, creates this extraordinary mass ceremony: the almost precisely simultaneous consumption (‘imagining’) of the newspaper-as-fiction. We know that particular morning and evening editions will overwhelmingly be consumed between this hour and that, only on this day, not that. … The significance of this mass ceremony–Hegel observed that newspapers serve modern man as a substitute for morning prayers–is paradoxical. It is performed in silent privacy, in the lair of the skull. Yet each communicant is well aware that the ceremony he performs is being replicated simultaneously by thousands (or millions) of others of whose existence he is confident, yet of whose identity he has not the slightest notion. Furthermore, this ceremony is incessantly repeated at daily or half-daily intervals throughout the calendar. What more vivid figure for the secular, historically clocked, imagined community can be envisioned? At the same time, the newspaper reader, observing exact replicas of his own paper being consumed by his subway, barbershop, or residential neighbours, is continually reassured that the imagined world is visibly rooted in everyday life. (p. 35)

We see most eloquently how embedded in community literacy enables a modern newspaper reader to be. Of course, there are numerous other traditional communities literate citizens belong to: periodical subscribers, book clubs, letters-to-the-editor writers, groups informed via newsletters, to name but a few, including some not so obvious ones, such as the community of consumers who read labels at the grocery store.

But recently, a new class of community has arisen that depends not only on traditional print literacy for its existence, but also the ability to manipulate a computer. These computer-based communities, such as e-mail lists, instant messaging groups, weblog groups, usenet groups, and others bind people in ways that traditional print literacy communities could not. We will see in Chapter Two how the data for this dissertation originates from one of these 'discourse communities'. Moreover, the nature of participation in these communities is also altered. Whereas print literacy community members were largely readers, or consumers of information, computer literacy community members are much more likely (and able) to also be writers, or providers of information.

Granted, some of this so-called information is merely personal opinion, but some of it provides a wealth of services to individuals seeking specific information about specific questions currently impinging on their lives. These services may be anecdotes based on personal or second-hand experiences, or pointers to relevant information providers, or sympathetic responses from others who have shared similar experiences. These messages do not essentially differ from what might have been shared orally, but the *reach* of the messages differs significantly in two ways: The communicants may be widely dispersed geographically, and the audience of 'listener-ins' may be much larger than what would be possible using only oral means. Thus computer literacy, coupled with print literacy skills, enables participation in a more 'nuanced' set of communities than just print literacy communities, as written participation in such communities by its members stamps those communities with a 'character' that is qualitatively different than those led by a single individual.

We see a similar claim made by Lanham (1993):

> The digital computer both strengthens and weakens the oration as a compositional form and educational technique. On the one hand, what the computer does best, besides counting, is *modeling*. It has made learning through *rehearsal-reality* possible across the complete range of human thinking and planning. *Declamatio* was education by endless rehearsal-reality, and the computer has simply adopted and expanded this basic expressive technique. On the other hand, electronic text is clearly finding its way to a new, and a new *kind* of, paradigm for writing—interactive on-line conversation. Such a form represents a movement into nonlinear hypertextual space where the classic oration cannot follow. … In a classroom based on networked personal computers, the teacher no longer provides the authoritarian focus. Teacher is but one voice on-line, and other voices too timid to speak in class are often emboldened by the different and more protected role an on-line conversation provides (p. 78).

Lave and Wenger (1991) advanced the concept of 'legitimate peripheral participation' in which "learning is an integral part of generative social practice in the lived-in world. … Legitimate peripheral participation is proposed as a descriptor of engagement in social practice that entails learning as an integral constituent" (p. 35). This social practice can be situated in the classroom, but can also extend to the Internet. In an interview, John Seely Brown links this concept with 'lurking,' that is, reading online conversations without contributing to the discourse:

> Lurking is a prosocial activity?
>
> Absolutely. Lurk is the cognitive apprenticeship term for legitimate peripheral participation. The culture of the Internet allows you to link, lurk, and learn. Once you lurk you can pick up the genre of that community, and you can move from the periphery to the center safely asking a question – sometimes more safely virtually than physically -

> and then back out again. It has provided a platform for perhaps the most successful form of learning that civilization has ever seen. We may now be in a position to really leverage the community mind.
>
> Is that another way of saying open source?
>
> Open source is about creating "literacy." Successful open source creates communities that are literate in understanding the dynamics of what code can or cannot do, global communities that can create standards with minimal elegance. ... Open source may also give us a way to crack the robustness problems of really complex systems. In Linux, for example, you write code to be read by others as well as executed by the computer. Writing code to be read is a great form of community hygiene. And when code is meant to be read by others, it has its own social life—it gets picked up by the community and used in all kinds of new ways. Pretty soon the community mind becomes a new kind of platform for innovation. (Schrage, 2000)

Participating (and lurking) in such online communities is one example of an effect of computer literacy suggested earlier by print literacy, that is, it will be important to *learn computer literacy* because students will *use computer literacy to learn.* Like regular print literacy, this has two aspects corresponding to reading and writing. One aspect is as a user or consumer of computer applications, the other aspect is as a creator, or originator, or initiator of computer programs, in other words, as a programmer. Both aspects are involved when using computer literacy to learn, but it is likely that the first aspect will be the more heavily used mode, while the second, like writing itself, the more difficult and less used mode (and also the one contributing more to computer fluency as described above). Note that there is usually a blending of various literacy skills occurring while utilizing a computer in the learning process. For example, if a student is using a word processor to write a paper, that student is using (at least) two sets of literacy skills: not only is one *writing to learn* as was mentioned earlier with print literacy skills, but by using a computer word processor one is also employing computer literacy skills to facilitate that learning.

One simple example of the first aspect of computer literacy, namely as a user rather than a programmer, is the graphing calculator (which can be a regular desktop computer application or integrated into a special purpose calculating device). By using this application's ability to instantly display the graph of an equation and experimenting with a variety of equations, variables and coefficients, students gain a far richer understanding of the relationship between *symbolic* forms of equations and their *graphical* forms than would be possible by manually graphing those equations with pencil and paper (Dunham and Dick, 1994). Another example is the Arab-Israeli Conflict simulation hosted by the University of Michigan's Interactive Communications and Simulations group. Using computers connected to the Internet, students assume the character of a political or military leader in a simulation of the dynamics in the Middle East, gaining a perspective on the issues surrounding those dynamics that would not likely occur by a simple reading of an essay or textbook (Kupperman, 2002; Scott, 1997). Additionally, being informed by such readings is likely to enhance the student's performance in the simulation, thus such participation may also motivate traditional print-based learning methods.

Yasmin Kafai (1996) has investigated the second aspect of 'using computer literacy to learn', namely programming to learn, with children:

> There are currently few opportunities for children to go beyond button-pushing and mouse-clicking in their interaction with technology.
>
> By asking children to program software for other children, we are turning the tables and placing children in the active role of constructing their own programs—and constructing new relationships with knowledge in the process. … [T]he very process of programming game software to teach fractions (or any other subject topic, for that matter) to younger users allows children to engage in significant mathematical thinking and learning. But most importantly, through programming, children learn to express themselves in the technological domain. (p.38)

Kafai goes on to describe how this programming was taught:

> A software design project starts with a simple instruction: "Design a computer game that teaches something about fractions to younger students." Everything else is left open. A class of students transforms their classroom into a game design studio for six months. During that period, they are:
>
> - Learning programming;
> - Thinking about interface designs;
> - Designing graphical elements'
> - Conceiving story structures, dialogue, and characters;
> - Devising instructional strategies; and,
> - Creating fraction representations.

Another pioneer in using programming to learn is Stephan Wolfram (2001). In an interview preceding the publication of *A new kind of science* he says:

> Almost all the science that's been done for the past three hundred or so years has been based in the end on the idea that things in our universe somehow follow rules that can be represented by traditional mathematical equations. The basic idea that underlies *A New Kind of*

> *Science* is that that's much too restrictive, and that in fact one should consider the vastly more general kinds of rules that can be embodied, for example, in computer programs.
>
> What started my work on *A New Kind of Science* are the discoveries I made about what simple computer programs can do. One might have thought that if a program was simple it should only do simple things. But amazingly enough, that isn't even close to correct. And in fact what I've discovered is that some of the very simplest imaginable computer programs can do things as complex as anything in our whole universe. It's this point that seems to be the secret that's used all over nature to produce the complex and intricate things we see. And understanding this point seems to be the key to a whole new way of thinking about a lot of very fundamental questions in science and elsewhere.

The word 'programming' takes on a somewhat different and more approachable meaning here. Rather than devising complex applications for computers to execute, computer literates instead devise *simple* programs that, when run iteratively millions upon millions of times, results in a complexity that has a striking resemblance to the complexity we find in our scientifically studied world. The *meaning* of this similarity is certainly debatable, but the fact that it cannot be dismissed out of hand implies that perhaps the boundaries of scientific knowledge are increasing as simple computer programs devised by students and scientists open fields of inquiry not previously possible without them.

1.5 Computer literacy in schools

What has been established up to this point are the similarities between print and computer literacies (reading is like using a computer, writing is like programming a computer) and the primary difference between them: print literacy is focused on what one *says* (that is, what is *written*), whereas computer literacy is focused on what one *does* (that is, what is *done* within the context of a computing machine). Most people's familiarity with computers is based on using applications: word processing; web browsing; database entry; reading and sending e-mail; managing digital songs, digital photos, digital movies; perhaps using instant messaging services, perhaps creating presentations, and so forth. A kind of *ad-hoc* computer literacy emerges out of the repeated use of common computer applications, reading manuals and getting help from friends and colleagues.

So, if one is concerned about the promotion and development of computer literacy in school settings, one might conclude that it is sufficient for students to be given ample opportunity to use computers in order for computer literacy concerns to be addressed. One might think: as long as students are given plenty of assignments that require them to use computer applications during the completion of those assignments, then they will *de facto* graduate with a high enough degree of computer literacy to be able to function in the world beyond school. Although I believe this much competence is necessary, I also believe it is insufficient in at least a couple of ways.

First, consider if we applied the same reasoning to the teaching of print literacy. We would then have to say it would be sufficient to give students ample opportunities to read a variety of texts during the completion of their assignments (recitations? – no writing allowed!) and that no written artifacts would need to be completed to graduate with a high school diploma. Even if a few students have managed to actually accomplish this feat in certain districts across the country, I'm sure most educators would strongly feel that such an educational system was deficient. Many reasons might be given for this feeling, but the one most applicable in this context is that *writing completes reading*. That is, when we learn to read, we enter a world where we can learn, if not all, certainly much of what we need to learn during our schooling, and beyond, by reading. That is to say, we learn to read in order to *read to learn*. However, reading has what we might call an out-to-in vector, meaning, the source is 'out there' and through the reading process, enters into our consciousness. And this by itself results in an unbalanced, or at least incomplete, set of learning experiences. Writing, on the other hand, can be characterized as an in-to-out vector, meaning, the source is inside oneself and, through the writing process, appears 'out there' in the world. And through engaging in this process, another, different learning experience occurs which cannot be gained simply by reading. We may say that after one learns to write, one *writes to learn*. So when I say *writing completes reading*, I mean that the path of learning that reading enables is complemented and extended by the learning that occurs through writing. Therefore, if it is important for students to write while in school in order to complete their print literacy education, and if computer literacy is seen as becoming increasingly important in daily affairs, and if programming computers is the counterpart to writing in computer literacy, then it follows that computer programming is as important to computer literacy as writing is to print literacy. Although it isn’t quite as pithy as *writing completes reading*, we may by analogy say that *programming computers completes using computers* and thus constitutes an essential component of a computer literacy curriculum.

The second reason that simply using computer applications is an insufficient strategy for promoting computer literacy in schools grows, by analogy, out of two points made earlier with respect to print literacy, namely, that learning to read enables one to read to learn and learning to write enables one to write to learn. Likewise, *learning to program enables one to program to learn.*[4] What do we mean by ‘programming to learn’? First, what we *don’t* mean in this particular context is a kind of Deweyan learning (programming) by doing (programming). Certainly, in a formal programming course one *wants* students to learn programming by doing programming, lots of programming; however, eventually, learning to program may lead to a state of expertise in a student where he or she begins to use those programming skills in support of learning that is *outside* the context of the programming course. What the nature of that learning actually *is* is somewhat indistinct at the moment since the ubiquity of computers in schools is both rather new and rather ‘thin’ at present. We don’t have much historical precedent for the kinds of learning activities that ‘programming to learn’

[4] For completeness’ sake, there is also the idea that learning to use computers enables one to use computers to learn, analogous to ‘learning to read enables reading to learn.’ For an overview of this type of learning using the World Wide Web, see Reeves (1999).

might entail. What we do have, though, are clues from present occupational practices where professionals who have no specific training in computer science program their computers to accomplish tasks germane to their everyday work activities. And it is these kinds of programming-based learning/tasks that we want more and more students to experience to prepare them for the doing of those tasks, and more, when they enter the workforce.

The third reason has to do with school reform efforts. Computers *per se* in the classroom do not necessarily bring about any kind of real reform to the educational process. Semour Papert (Harel & Papert, 1991) discuss this in Constructionism:

> The need to distinguish between a first impact on education and a deeper meaning is as real in the case of computation as in the case of feminism. For example, one is looking at a clear case of first impact when "computer literacy" is conceptualized as adding new content material to a traditional curriculum. Computer-aided instruction may seem to refer to method rather than content, but what counts as a change in method depends on what one sees as the essential features of the existing methods. From my perspective, CAI amplifies the rote and authoritarian character that many critics see as manifestations of what is most characteristic of—and most wrong with—traditional school. Computer literacy and CAI, or indeed the use of word-processors, could conceivably set up waves that will change school, but in themselves they constitute very local innovations—fairly described as placing computers in a possibly improved but essentially unchanged school. The presence of computers begins to go beyond first impact when it alters the nature of the learning process; for example, if it shifts the balance between transfer of knowledge to students (whether via book, teacher, or tutorial program is essentially irrelevant) and the production of knowledge by students. It will have really gone beyond it if computers play a part in mediating a change in the criteria that govern what kinds of knowledge are valued in education.

A computer literacy that embraces computer programming is much more likely to set up such a positive reform dynamic in the classroom than simply learning to use a spreadsheet application or a computer-aided instruction program. By its very (constructionist) nature, learning to program leads to a 'production of knowledge by students' and will, eventually, lead to changes in the criteria that govern the kinds of knowledge that are valued in education.

Thus, I believe that simply using computer applications is *insufficient* for bringing about a degree of computer literacy that will optimally prepare high school students for the expectations that may greet them at home and in the workplace with respect to computer usage. Just as we have English courses that increase the print literacy skills of reading and *writing*, I believe we should increasingly provide computer classes that offer at least rudimentary computer *programming* instruction to complement and extend the regular computer usage instruction. Furthermore, just as those writing skills that are exercised and developed in the English classroom are then put to use in Social Studies, History, and Science (writing to learn), the programming skills that are developed and exercised in the Computer class should then be put to use in the Math, Science, Art and Humanities classes (programming to learn).

1.6 Issues surrounding the teaching of computer programming

Given that a school district (or state department of education) has identified the need to offer at least a minimal level of computer programming in their secondary curriculum, two basic questions arise: What language should be taught? How should it be taught?

What language should be taught?

There are many computer languages from which to choose. However, when one examines a fair number of them, one language stands out as being particularly appropriate for being taught as a beginning computer language: Python.[5]

Python is a high level programming language that excels in clarity and simplicity of expression allowing programmers to build their computing solutions using human modes of thought and logic. Python was created by Guido van Rossum in 1989 with the goal of developing a language that could be used to teach the most advanced concepts of programming to non-programmers. Python's mix of qualities combines flexibility with grace, logic with clarity, directness with power, speed of development with maintainability, and portability with scalability. Its simplicity, power, and portability make it ideal for a wide range of applications - from simple scripts to large and sophisticated systems. Clarity of expression makes it equally useful for software specification and rapid prototyping. Python can also be characterized as a hybrid language that blends procedural, object-oriented, and functional programming paradigms, offering programmers a choice of approachs for building each part of the solution. This enables teachers to

[5] <http://www.python.org>

choose the paradigm they wish to emphasize in the classroom. Python is also well documented. There are numerous books available and the Python website has an excellent documentation section to assist the student and teacher alike in resolving inevitable programming problems and issues.

Python is available under a non-restrictive open-source license, which allows a school or district to download, use, modify and deploy the language as needed. Python's large open-source community enhances the value of Python and allows the platform to grow, without being dependent on any one vendor. Python is also extremely portable. There are currently identical versions available for nearly every computing platform, major and obscure, from PDAs to mainframes. This combination of simplicity, power and portability, along with its open-source nature, has made Python extremely popular. Over the last decade, still under the guidance of van Rossum, Python has grown from a small teaching project, into a programming language that is used by major enterprises around the world, in mission-critical applications.[6]

How should Python be taught?

Given that a prospective teacher has chosen to teach Python, he or she might reasonably wonder how best to teach the language in a classroom setting. What issues might arise, what choices will need to be made? What sorts of programming exercises promote language acquisition? What *doesn't*

[6] See Python Business Forum: http://www.python-in-business.org/

work well? What is potentially confusing for beginning students? What are the common mistakes? How can I, as a teacher, prepare myself for teaching this subject matter?

The designer of the Python language, Guido van Rossum, wrote a DARPA proposal called ‘Computer Programming for Everybody’, which turned out to be under-funded and now survives primarily as a web site expressing the desire of its author and adherents to make computer programming much more accessible to everyone. Out of the initial enthusiasm for that proposal was born the Python in Education mailing list.

When I found out about this mailing list, I realized that those messages might be fruitfully analyzed for clues to an answer for the question posed at the beginning of this chapter: *what considerations are most important in teaching Python as a first programming language* in a secondary school setting? All the messages are archived at the group’s website and are publicly and freely available. In the next chapter, we take a look at what is involved in creating a computer program, and situate the Python language in a context of other programming languages.

CHAPTER 2

LEARNING PROGRAMMING

*Programming is hard. It's the process of telling a bunch of transistors to do something, where that something may be very clear to us fuzzy humans, with all our built-in pattern matching, language processing, and existing knowledge, but really, horrifically, tediously difficult to communicate to a bunch of dumb transistors. Python *is* hard, because programming is hard. On the other hand, Python is easier than (in my experience) C, C++, Objective C, Pascal, Postscript, Forth, Java, Javascript, Perl, etc. In some cases it is so much easier that it almost appears *easy* in comparison. But there is a huge difference between *easier*, even vastly easier, and *easy*.*

—Dethe Elza

2.0 Introduction

Since much of this dissertation concerns itself with computer programming, it will prove useful to discuss both the general process of writing a computer program and the general concerns of educators teaching computer programming. We will begin by taking a look at what the research literature has to say with respect to both of these in order to contextualize our research results. We then look at the Python language itself, and at the historical context of the newsgroup that served as the source of data.

2.1 Programming steps

Let's begin with a definition of programming. Gal-Ezer and Harel (1998) offer the following:

> First, we should state clearly that we take programming here in a rather broad sense, covering not only the coding act itself, but also the design of the algorithms underlying the programs and, to some extent, considerations of correctness and efficiency. To some, this interpretation of programming might be the obvious one to adopt, but experience shows the point ought to be made more explicitly. (p. 82)

We will see throughout this dissertation examples of how programming is more than just writing code. There are several other cognitive skills involved ranging from conceptualizing the problem, to creating a viable algorithm, to expressing thay algorithm to the computer and to humans, to eradicating the inevitable 'bugs' that occur in the code. We will see how programming is a form of writing with its own, related set of difficulties and challenges. In this section, we discuss the five major steps associated with creating a working computer program.

In a review of literature of teaching high school computer programming, Taylor (1991) identified five interrelated steps taken in the creation of a functional computer program:

1. Problem definition
2. Algorithm design
3. Code writing
4. Debugging
5. Documentation

It should be kept in mind that although there may be a general flow from first to last while creating a program, there is often much cycling back and forth between steps. This is especially true, for example, with the documentation step often occurs hand-in-hand with the code writing step, and even during the algorithm design step.

Problem Definition

Another way of expressing this concept is ‘product specification’. The student/programmer needs to understand what functionality the program is expected to have. Usually this means identifying the *inputs* to the problem, the likely *components* of the solution (if not explicitly specified), and the *outputs* that either solve the problem or meet the product specifications. In a beginning programming class, the initial problem statement is likely to come from the teacher, however, later on, students may be asked to submit their own natural language description of the proposed product.

Algorithm Design

Algorithm design results from algorithmic thinking. Algorithmic thinking includes:

> functional decomposition, repetition (iteration and/or recursion), basic data organizations (record, array, list), generalization and parameterization, algorithm vs. program, top-down design, and refinement. Note also that some types of algorithmic thinking do not necessarily require the use or understanding of sophisticated mathematics. The role of programming … is a specific instantiation of algorithmic thinking. (National Research Council, 1999)

Note that ‘algorithm’ does not necessarily mean the same as ‘program’! The same report emphasizes this:

> For example, one difference between an algorithm and a program is that the algorithm embodies the basic structural features of the computation

> independently of the details of implementation, whereas a program commits to a specific set of details to solve a particular problem. Understanding this interrelationship is basic programming, but the principle applies throughout life. … One observes that a solution technique (corresponding to the algorithm) can be used in different problem situations (corresponding to the programs). Inversely, one expects that a successful problem solution embodies a more general process that is independent of the situational specifics.

Using algorithms and doing algorithmic thinking is something we do everyday but not necessarily in the context of using a computer. Understanding algorithms and algorithmic thinking means realizing that the machine receiving the algorithm does not understand the way a human understands:

> An algorithm is a formula or set of steps for solving a particular problem. To be an algorithm, a set of rules must be unambiguous and have a clear stopping point. Algorithms can be expressed in any language, from natural languages like English or French to programming languages. …
>
> So the secret is to clearly spell out the rules. The act of breaking down a complex problem into small chunks and then breaking it further into very simple tasks is the essence of programming and is a crucial skill in math, science, and design. ...
>
> To think through the sequence of a task, you will need to identify the start point, what decisions will need to be made, what will influence those decisions, what will result from those decisions, and when you will consider the task complete. ...
>
> When communicating with people, most of the assumptions are clear but when you are programming, the computer doesn't have any "common sense". You will need to spell it all out in a very clear sequence.[7]

With this top-down design style the programmer divides a problem into smaller sub-tasks, describes a technique for accomplishing each sub-task, and

[7] http://www.roboeducators.org/downloads/prog%20inst/Algorithmic%20Think.doc

shows how each sub-task solution contributes to solving the overall problem. For large projects, this is often an iterative process where each sub-task can be further subdivided into sub-sub-tasks, and so on. Beginners usually have the most difficulty breaking the overall problem into specific sub-tasks (Ingram, 1988).

The top-down design style was more strongly emphasized in the past when the languages used were compiled rather than interpreted (these terms will be discussed more fully later this chapter), and interaction with the computer was both ‘scarce’ and ‘expensive’, both in terms of costs and time. However, with the rise of interpreted languages and the apparent ubiquity of personal computers, interactions with the computer become both plentiful and cheap. Thus, especially for smaller problems (and especially with certain languages), some researchers advocate a more freeform or interactive style of algorithm design. Thus bottom-up algorithm design involves rapid, iterative cycles of testing and refining hypotheses in the interpreter to find code that works as expected, and building up, procedure by procedure, module by module, a working program in the absence of any written algorithm specification. Alternatively, bottom-up and top-down can work together, sequentially:

> Early research has identified that students employ different strategies in programming tasks. Often two categories are used, usually reflecting variations on a top-down vs. bottom-up dichotomy. The top-down, analytic style is often associated with more proficiency. However, students often combine the styles and some benefit from being allowed

to "tinker" in a bottom-up manner at first before planning in a top-down manner. (Clements, 1999)

Whichever style of designing is used, the expectation is strong that each procedure and module created is documented (commented) in the body of the code so that human readers of the code can understand what the algorithm is and how it is done.

Templates and patterns

Beginning programmers have not developed any sort of algorithmic repertoire, that is, they don't yet have the experience that would enable them to quickly ascertain which algorithms to apply to a given problem. Students find it very helpful to be shown algorithmic 'templates' or patterns that they can use in their problem solving. Templates are stereotypical sequences of computer instructions represented as code, pseudocode, or natural language descriptions. As code, templates are short programs that express specific algorithms that the students can quickly use and modify for their own purposes. They are abstractions at a higher level than specific language features (syntax), and offer students reasonable suggestions for how to proceed in their programming efforts. Linn (1985) found that experienced, successful teachers explicitly taught good, well-organized templates to their students.

Working in teams

In the workplace, programmers rarely work alone, and researchers have found that students perform better in teams:

> Contrary to a popular view of programmers held by nonprofessionals, strong beneficial effects of computer programming have been reported in the area of social and emotional development. Teachers report that students exposed to computer programming are more likely to interact with their peers. They engage in group problem solving, sharing, and acknowledging expertise and creative thinking. Social isolates benefit the most. Children are eager to cooperate and share what they have learned with others. Thus, computer-programming environments can facilitate social interaction and positively focus that interaction on learning. (Clements, 1999)

Especially when they are expected to present their work to their peers, for example, in a structured walk-through, they spend more time planning their programs, write more eloquent programs, and generally have a more positive attitude towards programming (MacGregor, 1988). Better programs also resulted when students participated in group discussions during the planning phase (Webb, 1986). This is where real development of algorithmic thinking can develop as students verbally hash out the 'what, when, where, how and why' details of their code.

Importance of planning and describing the algorithm

Many writers expressed the importance of planning before coding. Leavens, et al (1998) are especially keen on this because of its consequences for evaluation. They stress that programming is much more like writing than calculating:

> To an outside observer (who is not a practicing mathematician or computer scientist), programming seems similar to solving a specific numerical problem. Memories of fundamental arithmetic may lead one to believe that the process of getting the answer is not as important as the answer. Nothing could be further from the truth.
>
> Writing a program is not at all like finding an answer to a specific instance of a numerical problem (e.g., multiplying 3 and 4). The

> programmer does not solve a specific problem instance (e.g., multiply 3 and 4 and produce 12). Instead, a programmer's task is to write a set of unambiguous instructions for a computer that can solve a whole class of problem instances (for example, acting as a calculator). If only one instance of a problem is to be solved, then it is not cost-effective to write a program. (Leavens, et al, 1998)

Consistent with this idea of writing, they go on to emphasize that the program is written as much for other humans as it is for the machine:

> Although programs are a means to instruct a computer, humans have to read them as well. Indeed, in the software industry, the human readers of a program are just as important as the machine. This is critical for several reasons:
>
> • Programs are often written in teams, and team members need to understand each other's code and procedures.
> • Debugging a program (correcting errors) also requires reading. Here the reader is often the program's author; the difficulty of finding bugs in a program shows the difficulty in reading programs carefully. Even the program's own author will have difficulty in reading a program that is unclear, poorly organized, or poorly documented.
> • Programs are often read during "code walkthroughs." Here the readers are programmers other than the program's author, who read the program carefully to validate its correctness.
> • Programs are read by "reusers," people who wish to use or adapt the code for another purpose.
> • Perhaps the most important reader of a program is the maintenance programmer. This is, very often, a different person than the program's author. The maintenance programmer has to understand a program to fix or enhance its functionality, in case the author of the program is not available to answer questions. (Leavens, et al, 1998)

And of course, in the classroom, the instructor must read the programs too:

> When students know that their programs will be read carefully by a TA, and graded on the quality factors that reflect advance planning (clarity, organization, documentation, and test or verification plans) they are pressured to think and plan before they begin to write code. In such a situation, correcting students who fall into the "generate and test" pattern is much easier, because the grading system emphasizes and reinforces the value of advance planning. In addition, because the students are more involved with the writing of their programs, they

> learn more and increase their understanding as the programming assignments become more complex. (Leavens, et al, 1998)

So programmers have the dual responsibility of writing both for the machine and for human readers who need to understand in natural language what the machine is being instructed to do. Careful planning can ensure that the proper communication occurs to both sets of ‘readers’ of the program.

Code Writing

At this point the programmer understands the problem, has divided it into subtasks and has written an algorithm describing how to accomplish each subtask. Now, each algorithm must be coded in the chosen programming language. What knowledge is needed for this to occur?

Three types of programming knowledge

Researchers have identified three types of programming knowledge: syntactic, conceptual and strategic. Syntactic knowledge refers to knowledge of the language’s grammar, for example, the use of white-space indentation for code blocks, how to call a ‘for’ loop (>>> for item in list:), or which brackets to use for lists and dictionaries ([] and {}, respectively) in the Python language. Syntactic knowledge alone is not sufficient to write a working program; it must be accompanied by a sound understanding of the programming principles necessary to develop logically correct programs.

Conceptual knowledge involves understanding the semantics of the language, and the development of design skills and mental models necessary to create working programs. This is knowledge that transfers to other languages when necessary. For example, almost all languages have some kind of looping mechanism. Understanding that a 'for' loop iterates over each item in a sequence, and how to use it in the expression of an algorithm is an example of conceptual knowledge. Conceptual knowledge also facilitates combining language features in program design; for example, one begins to understand when to use a list inside a dictionary to store program data. Conceptual knowledge entails a full understanding of the semantics of syntactic constructs and the ways they can be combined to solve a problem.

Strategic knowledge uses syntactic and conceptual knowledge in the most appropriate and effective way in the service of solving novel programming problems. It is essential for recognizing that a problem can be solved and for identifying appropriate techniques for doing so. For example, if one is programming a wiki server[8], strategic knowledge informs the programmer that a state machine[9] is necessary to process the text rather than, say, a regular expression.[10]

[8] The term WikiWiki (which means "quick" in Hawaiian) can be used to identify either a type of hypertext document or the software used to write it. Often called "wiki" for short, a wiki is a collaboratively written website. A user may contribute or alter content without those efforts being reviewed prior to its inclusion.
<http://www.infoanarchy.org/wiki/wiki.pl?Wiki>)

[9] "A finite state machine (FSM) or finite state automaton (FSA) is an abstract machine used in the study of computation and languages that has only a finite, constant amount of memory (the states). It can be conceptualized as a directed graph. There are a finite number of states, and each state has transitions to zero or more states."
<http://en.wikipedia.org/wiki/Finite_state_automaton>

[10] "A way of describing a pattern in text - for example, 'all the words that begin with the letter A' or 'every 10-digit phone number' or even 'Every sentence with two commas in it, and no capital letter Q.'"
<http://www.deserve-it.com/manual/glossary.html>

These programming knowledge distinctions are important for highlighting deficiencies in the types of knowledge acquired in many introductory programming courses. McGill and Volet (1997) report that:

> A close examination of the nature of the knowledge acquired by students who did not reach the desired level of achievement indicates that they did acquire some syntactic knowledge, but failed to develop conceptual and strategic knowledge (Linn, 1985). This outcome is quite consistent with the emphasis that has been typically placed on the acquisition of syntactic knowledge in introductory classes and many introductory textbooks. (p. 3)

Conversely, the same authors report that when conceptual models of a language are emphasized during instruction, problem solving performance (strategic knowledge) is enhanced, and also that problem solving performance is strongly related to measures of conceptual knowledge (Bayman & Mayer, 1988).

The importance of all three types of knowledge can be understood in another way. During the Middle Ages the foundation for higher learning, known as the *quadrivium* (arithmetic, astronomy, geometry, and music), was the *trivium* and consisted of three parts: grammar, dialectic (logic and disputation), and rhetoric, each of which was mastered in sequence according to one's developmental age. Grammar, of course, corresponds with syntactic knowledge, dialectic with procedural, and rhetoric with strategic. Dorothy Sayers (1948), in a wonderful essay called "The lost tools of learning" emphasizes that the purpose of the *trivium* was not learning subjects *per se*, but rather learning how to learn (so as to know how to learn the subjects in the quadrivium):

> Taken by and large, the great difference of emphasis between the two conceptions holds good: modern education concentrates on teaching subjects, leaving the method of thinking, arguing and expressing one's conclusions to be picked up by the scholar as he goes along; mediaeval education concentrated on first forging and learning to handle the tools of learning, using whatever subject came handy as a piece of material on which to doodle until the use of the tool became second nature. ...
>
> For the tools of learning are the same, in any and every subject; and the person who knows how to use them will, at any age, get the mastery of a new subject in half the time and with a quarter of the effort expended by the person who has not the tools at his command. To learn six subjects without remembering how they were learnt does nothing to ease the approach to a seventh; to have learnt and remembered the art of learning makes the approach to every subject an open door.

This maps quite well to our conception of learning computer programming: the purpose is not to program a specific application *per se*, but rather learning how to program for its utility in other endeavors.

Having identified these three types of programming knowledge, we turn to an examination of three types of cognitive knowledge, and then to a synthesis of the two.

Three types of cognitive knowledge

Cognitive psychology has long distinguished among different forms of knowledge. Usually, these fall into three broad areas: declarative, procedural, and conditional.

Declarative knowledge is knowledge about something, normally expressed as facts, concepts or principles. For example, knowing that the circumference of a circle is equal to twice its radius times pi is a form of declarative knowledge. One important aspect of declarative knowledge is that it is typically expressed through language.

Procedural knowledge is the active use of declarative knowledge. It is commonly defined as the knowledge of ‘how to’ do something. Procedural knowledge is demonstrated in action rather than being spoken about or written about. Nevertheless, even though we can refer to these two forms of knowledge separately, they are seldom found in isolation from each other.

Conditional knowledge is knowing when, where and why to apply procedural and declarative knowledge to a given situation (Paris et al., 1983). It is knowing under which conditions a particular strategy is important, when and where to use that strategy, and how to evaluate its effectiveness.

McGill and Volet (1997) note the similarity between the three types of programming knowledge and three kinds of cognitive knowledge and combine them into a two-dimensional conceptual framework of the various components of programming knowledge.

Knowledge framework for teaching programming

The first two components of each knowledge domain, syntactic and conceptual, and declarative and procedural, are not considered isomorphic but rather orthogonal. However, strategic knowledge and conditional knowledge are considered essentially equivalent:

> A two-dimensional model of the various components of programming knowledge distinguishes among four interrelated but conceptually distinct forms of programming knowledge. The four categories of knowledge are declarative-syntactic, declarative-conceptual, procedural-syntactic, and procedural-conceptual. It is assumed that together these knowledge categories form the basis of programming knowledge. The ability to know why, how, where, and when this knowledge can be used appropriately leads to a conceptually distinct higher form of knowledge

called strategic or conditional. Strategic/conditional knowledge refers to the ability to integrate and orchestrate the use of all other forms of knowledge. This higher level of knowledge development is achieved when an individual is able to use procedural knowledge (syntactic and conceptual) flexibly and appropriately across novel situations and tasks in a way that is syntactically correct and that reflects a sound understanding of the semantics of the actions executed by the program (declarative-conceptual) (McGill and Volet, 1997, p. 7).

Table 3 Components of programming knowledge

	Declarative Knowledge	Procedural Knowledge
Syntactic Knowledge	Knowledge of syntactic facts related to a particular language, such as: • knowing that a colon must follow a function definition • knowing the difference between a function and a method	Ability to apply rules of syntax when programming, such as: • the ability to write a syntactically correct class definition • the ability to read from and write to a text file on the local file system
Conceptual Knowledge	Understanding of and ability to explain the semantics of the actions that take place as a program executes, such as: • the ability to explain how a particular loop exits • the ability to explain how to import a module and call one of its functions	Ability to design solutions to programming problems, such as: • the ability to design a function that computes the mean, median and mode of some data • the ability to modify a program that returns a 1-D array to return a 2-D array

Their table is reproduced (modified with examples from Python instead of the original Pascal and Basic examples; see Table 3). It presents the conceptual framework of the various components of programming knowledge, with some examples within each category.

Declarative-syntactic knowledge

McGill and Volet describe this category of knowledge as facts about a particular programming language's syntax that are typically introduced at the beginning of an introductory programming course. Conceptual understanding of programming is not assumed, nor is the ability to use those facts in a program. This knowledge is usually presented in lectures or learned from books.

Declarative-conceptual knowledge

This category is described as representing an understanding of and ability to explain the meaning of the actions that take place as a program executes. It differs from both types of procedural knowledge in that knowing the semantics does not mean that a student can actually apply those actions in a working program. Students may be able to explain how a particular part of a program performs, but they do not necessarily access that knowledge when writing their own programs. This kind of knowledge can be taught in lectures or tutorials, or by observing programs in action.

Procedural-syntactic knowledge

This category of knowledge refers to the ability to apply rules of syntax when programming, according to McGill and Volet. This form of knowledge is usually emphasized while doing programming exercises during laboratories. However, having this kind of knowledge does not ensure semantic understanding, that is, a student may know that a particular form is correct, but may not know exactly what the form is doing.

Procedural-conceptual knowledge

This category refers to the ability to use semantic knowledge to write programs. Unfortunately, this type of knowledge is often not taught explicitly; instead students are expected to acquire it as a result of the declarative knowledge presented in lectures and tutorials and their experiences undertaking hands-on programming exercises.

Strategic/conditional knowledge

Finally, according to McGill and Volet, the synthesis of the four other kinds of knowledge results in the ability to use syntactic and conceptual knowledge effectively to design, code and test a program that solves a novel problem. Furthermore, the programmer is also able to explain the semantics of the actions executed by the program; hence, he or she possesses both declarative and procedural knowledge.

This framework of the components of programming knowledge is extremely useful when designing or evaluating a programming class. Usually introductory programming classes tend to emphasize syntactic constructs, and neglect explicitly developing procedural-conceptual knowledge. However, studies show that emphasizing procedural-conceptual and strategic/conditional knowledge results in increased levels of student understanding and achievement (Bayman & Mayer, 1988; McGill & Volet, 1997). For example, McGill and Volet studied the effect of an instructional package consisting of 1) an interactive teaching approach involving modeling, coaching, and collaborative learning, and 2) an emphasis on the student use of a planning strategy for algorithm development and programming:

> The results of this study suggest that an instructional approach that emphasizes the development and use of a planning strategy for algorithm development in conjunction with modeling, coaching, and collaborative-learning activities can have positive effects on students' development of introductory programming knowledge.
>
> ... Although control students were introduced to the same amount of declarative knowledge in lectures and given the same opportunities to develop procedural knowledge through the completion of practical exercises in tutorials, their development of strategic/conditional knowledge was not guided by structured interactive-modeling and cognitive-coaching instruction. These results support Oliver's (1993) assertion that computing educators should give attention not only to the mode of delivery but also to the use of appropriate rehearsal and consolidation learning activities. (p. 12)

One way to ensure that emphasis is the liberal use of annotated templates and patterns as canonical examples, as described earlier. Another way is by providing detailed feedback: "writing comments on completed assignments, returning assignments promptly, explaining to small groups or individuals how to improve programs, providing a solution to the assignments, and describing different ways the assignment could have been solved." (Linn, 1987, 486-87; in Taylor, 1991, 31)

Debugging

After a program is written, it is rare for it to run successfully the first time. Usually there are errors, called 'bugs' that need to be corrected. This process is called debugging, and is often tedious and difficult. Pea (1986) identified three classes of bugs, or misunderstandings that arose from students reverting to natural language rules when their understanding of computer language rules was faulty or incomplete:

> The 'parallelism bug' assumed that several lines of code in a program could be active at the same time, just like a human could process several lines of a conversation at one time. The 'intentionality bug' assumed that the computer could look ahead, or anticipate, what would happen in the program. The 'egocentrism bug' assumed the computer could fill in or supply parts of code that the student had left out, similar to the way a human could interpret what was meant from what was said. (in Taylor, 1991, 20)

As students become more proficient in their coding, these anthropomorphic mistakes become much less common, but are replaced with both syntactical errors and logical errors.

A programming environment refers to the software applications that are used to create other software applications. At minimum, this would consist of a text editor and the programming language itself. Most modern programming environments include specialized text editors that greatly reduce the occurrence of syntactical errors by highlighting them as they are typed (much as word processors highlight misspelled words). Language interpreters can also easily catch syntactical errors when code is run and report which line or which command is erroneous. However, it is much more difficult to eradicate bugs that result from logical errors where the code will run but the output is wrong. Again, most modern programming environments include debuggers which slow down code execution to run step by step so the programmer can see where the execution flow is occurring at any given instant, and what the values of the variables are at that instant. Debuggers help programmers visualize static code as a dynamic computational structure at execution time.

Particular sources of bugs for beginning programmers include using the equality symbol (=), knowing the value of a dynamic variable, controlling loops, using conditional constructs (if, then, else) and knowing the functional difference between ‘print’ and ‘return’. Suggestions for reducing the occurrence of these and other bugs include: 1) encourage analogical reasoning by presenting new concepts using a variety of examples so that students are less likely to associate a procedure with only one type of problem; 2) emphasize the strict flow of control the computer adheres to–only one line of code is executed at a time, only current values of variables are applicable at one instant, and the computer neither remembers what has occurred (unless explicitly told to) nor anticipates what is to come; and 3) stress the mechanical aspect of controlling a machine through language commands–although humans can interpret remarks, fill in gaps and repair ambiguities based on a shared social background, computers can only interpret instructions based on a mechanical process defined by the rigid rules of the language (Taylor, 1991).

Documentation

As has been mentioned several times already, programming is more than just issuing instructions to the computer. If code is to be maintained or extended or reused or modified or evaluated (say, by the teacher) in any way, what the program is doing needs to be documented in some way. Abelson and Sussman (1996) emphasize this in the preface to their *Structure and Interpretation of Computer Programs*:

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

This is echoed by Moglen (1999) in his discussion of open source software:

> The function of source code in relation to other human beings is not widely grasped by non-programmers, who tend to think of computer programs as incomprehensible. They would be surprised to learn that the bulk of information contained in most programs is, from the point of view of the compiler or other language processor, "comment," that is, non-functional material. The comments, of course, are addressed to others who may need to fix a problem or to alter or enhance the program's operation. In most programming languages, far more space is spent in telling people what the program does than in telling the computer how to do it.

There are several strategies for accomplishing this documenting function. Perhaps the most common is embedding comments in the code that are meant for human readers and ignored by the computer. These comments are meant to document what the code does (for example, the expected inputs and outputs), how it is to be used, and how it accomplishes what it does (that is, an explanation of how the algorithm works). The advantage of this strategy is that it encourages (nearly) simultaneous coding and documenting, since they coexist in the same source file. Also, comments can be placed exactly in the code where the explanation pertains.

The importance of documentation in large programming projects is emphasized on the 'literate programming' page;[11] several quotes here give a flavor of the thinking:

> The structure of a software program may be thought of as a "WEB" that is made up of many interconnected pieces. To document such a program we want to explain each individual part of the web and how it relates to its neighbors. –Donald Knuth

---

> A traditional computer program consists of a text file containing program code. Scattered in amongst the program code are comments which describe the various parts of the code.
>
> In literate programming the emphasis is reversed. Instead of writing code containing documentation, the literate programmer writes documentation containing code. –Ross Williams

---

> Of course, it is important to point out that comments are not an end unto themselves. As Kernighan and Plauger point out in their excellent book, The Elements of Program Style, good comments cannot substitute for bad code. However, it is not clear that good code can substitute for comments. That is, I do not agree that it is unnecessary for comments to accompany "good" code. The code obviously tells us what the program is doing, but the comments are often necessary for us to understand why the programmer has used those particular instructions. –Edward Yourdon

---

> Unit tests are also documentation. The unit tests show how to create the objects, how to exercise the objects, and what the objects will do. This documentation, like the acceptance tests belonging to the customer, has

[11] <http://www.literateprogramming.com/>

> the advantage that it is executable. The tests don't say what we think the code does: they show what the code actually does. –Ron Jefferies

> As I gradually improved my in-code documentation, I realized that English is a natural language, but computer languages, regardless of how well we use them, are still "code." Communication via natural language is a relatively quick and efficient process. Not so with computer languages: They must be "decoded" for efficient human understanding. –David Zokaities

However, becoming proficient at both coding and documenting is rather more challenging than simply coding alone:

> There are a few issues limiting its [literate programming's] popularity.
>
> One of the core ideas is to write a piece of documentation that is a cross between a well-written essay and concrete detail documentation. The problem here is that not many people write well enough to achieve even the essay and even fewer can effectively combine that with detail docs. The writing skills need to be augmented by knowledge of the markup language (usually TeX). And the writer needs to know the programming language being used. And the programmer needs to be skillful enough to write a program in disconnected fragments that still makes sense and can be debugged after re-assembly.
>
> Donald Knuth is one person who has all of these skills and can apply them all at the same time. In contrast, the average Joe is lucky to have even a few of the skills, much less being able to apply them all at the same time. It is rarer still to find teams where all of the programmers can read and write in a literate style.[12] –Raymond Hettinger

Thus, even if literate programming techniques *per se* are not adopted in the curriculum, it still remains beneficial for beginning programmers to be encouraged or required to include detailed comments in their code both for evaluation purposes and for future ease of maintenance purposes.

[12] <http://mail.python.org/pipermail/python-list/2003-June/166476.html>

2.2 Teaching Programming

In the previous section we explored the process of writing a working program, which consisted of defining the problem, designing an algorithm to solve the problem, converting the algorithm into code, debugging the code to run as error-free as possible, and throughout the process, documenting, in natural language, descriptions of what the program is doing and explanations of how it is doing it. In this section we address the question of how teachers can foster the learning of programming by their students, and explore the cognitive effects of learning programming beyond the learning of the language itself.

Programming not only for computer science majors

The foundational premise of the edu-sig newsgroup, from which this dissertation data is taken, is that everybody, not just students and practitioners of computer science, can do computer programming. (I will address certain caveats about the term 'everybody' later in this chapter.) This is analogous to the claim that everybody, not just professional writers, can write in his or her native language. Figgins (2000), in an interesting extended metaphor between hacking and tracking, puts it this way:

> The goal of literacy was not to make everyone a professional writer, but to allow people to understand the writings of others and enable them to write simple documents of their own.
>
> You may never write a novel, or even a short story, but with basic literacy, you can communicate your thoughts. The rise in literacy helped make the Renaissance possible. It changed how we view the world so much that people in the Middle Ages could not have predicted the Renaissance; they had no idea what the future would hold.

> We don't yet know what the result of general programming literacy would be, but van Rossum believes that it could be just as big of a change as was brought about by general literacy. Now that there's a computer on every desk, can we now make everyone literate in how to write simple programs? ... This is really about freedom. Just as the rise in general literacy brought about an emancipation of ideas in the Renaissance, a rise in programming literacy might also free us to explore a world where computers are becoming ubiquitous—embedded in many of the tools that surround us daily. Literacy in the Information Age will mean computing literacy, and knowing how to program.

The 'tracking' half of Figgins' metaphor implies knowing a vast amount of information about the physical environment; it is "a complex art that engages all the senses, ... a demanding skill that involves mental acuteness, craft and concentration." Because of the quality of their immersion in the natural world, tribal people have "a huge incentive to teach their children as much as they can about the plants and animals that surround them, how they interact, and how they can learn from them." So what might this have to do with programming?

> With the rise of the Internet, there is a new wilderness—a new ecology made from the interactions of billions of communications from billions of devices. They are being woven together in a web that is just as complicated as the web of life. And our success as individuals, as a culture, is going to depend on our ability to understand and manipulate that environment. ... Away from the tame, the known, the village, the city, hackers and trackers understand what makes things tick. They know how to read the environment, manipulate it, and stretch this awareness as far as they can, to adapt and use whatever is presented to them. ... A long unused area of our brains is beginning to come alive. It is the art of the tracker that is reemerging in the art of the hacker. In the network we once again have a rich medium for this art. And we have a compelling reason to learn, because those most successful at manipulating the medium will be the most successful at survival.

Thus, teachers are implicitly encouraged to intercalate computer programming into their curriculum, or offer classes specifically designed to teach beginning programming to all interested students, because it may increasingly become a necessary survival skill in our modern culture, similar to the way knowing how to read and write is now; the way tracking used to be (and still is for tribal cultures).

We will see that participants in the edu-sig newsgroup often noted the difference between teaching programming in a programming class, and using programming in another subject. This dichotomy appears to be ingrained in the computer science field. Gal-Ezer and Harel (1998), in an article concerning what material should be included in a course for computer science educators, note that:

> the unique nature of CS, with its special algorithmic way of thinking and extremely short history, has led to a diversity of opinions about its very substance. As an example, here are two strikingly conflicting quotes by two prominent computer scientists:
>
> > Computer science has such intimate relations with so many other subjects that it is hard to see it as a thing in itself.
> > –M.L. Minsky, 1979
>
> > Computer science differs from the known sciences so deeply that it has to be viewed as a new species among the sciences.
> > –J. Hartmanis, 1994
> > (p. 79)

We will see both viewpoints played out not only in the expressed views of the posters but also in the two primary contexts in which programming is incorporated into existing school curricula. This may also contribute to the dearth of formal curriculum materials for teaching Python programming: uncertainty about audience. An author of a programming lesson plan may well wonder if it is meant to be used specifically in a programming course, or in another subject area.

There are of course other reasons for this dearth: the relative newness of the Python language (vis-à-vis LOGO or BASIC), the low penetration of general programming courses in school curricula, the absence of general programming benchmarks in state standards, the focus of existing materials on AP computer science, and the perceived lack of market opportunity by large publishing houses. However, if we look back at the comparison of programming with writing, we can anticipate that curriculum materials will be more readily accepted if they target the beginning, non-CS programming course, and indeed, we see many references to such materials available on the web by the edu-sig posters; see Appendix B for some suggestions. However, programming materials targeting specific subjects, especially Mathematics, are also available and can be incorporated into a general programming course as well as the specific subject area. Also, the relative lack of curriculum materials represents an opportunity for a creative educator to develop his or her own materials, and to share them with other interested teachers via the Internet. Indeed, these are two of the foundational goals of CP4E (#1 & #3):

1. To develop a high school and college curriculum
2. To create better tools for program development
3. To build a user community around these tools to help with their development

Connectionist teaching

This lack of curriculum material represents an opportunity in another sense: a chance for a teacher to change his or her perspective on what it means to teach. Yuen (2000), in a study of computer programming instructors, described two different worldviews of the nature of the mind:

> Over the years, two predominant computational theories of mind have emerged from the study of artificial intelligence (AI), namely, the symbolic model fashioned after the digital computer, and the connectionist model modeled after the human brain (Korf, 1991). ...
>
> In the symbolic model, concepts are represented by symbols, knowledge is conceived of as objects in individual minds (Bereiter & Scardamalia, 1996). Learning in such mind-as-container metaphor is a linear problem-solving process, in which symbols are manipulated in a sequence of steps in two directions, forward from a set of givens or backward from a goal. This metaphor suggests that when we learn something, there is a sentence in our head representing that knowledge (Strauss & Quinn, 1997).
>
> In [connectionist] models, knowledge is not represented by symbols linked together in sentences, but by simple processing units arranged in layers. An artificial neural network consists of a very large number of individual elements (processing units), modeled after neurons, each connected to a large number of neighboring elements, and each computing a very simple function, such as a weighted sum of its inputs. Connectionist models presume that cognitive functioning can be explained by collections of processing units that operate in this way. Concepts to be learned in connectionist systems are represented as patterns of activity over a set of processing units. Propositions are represented by patterns of activation over many units.
>
> Most teachers take knowledge as objects and treat understanding as a characteristic of something in students' minds. These pedagogical assumptions are fully consistent with the mind-as-container metaphor.

Yuen goes on to explain how the connectionist model offers a more “enlightened” theory than the symbolic model:

> The connectionist view of understanding is based on commonsense criteria. First of all, there is no privileged way that ought to be expected

> of student. In judging a student's understanding of x, the following criteria are applied: (a) how intelligently the person acts with respect to x, (b) his or her ability to explain whatever is judged to be in need of explaining in x, and (c) awareness of and interest in doing something about shortcomings of the first two criteria. An immediate benefit of this approach to understanding is that `it permits liberality without total collapse into relativism' (Bereiter & Scardamalia, 1996, p. 499).

The "Connectionism" entry in the Stanford Encyclopedia of Philosophy[13] has an excellent discussion of the differences between connectionism and symbolic processing. It is important to stress that there is not necessarily an either/or choice between connectionism and symbolic processing. The Encyclopedia says:

> However many connectionists do not view their work as a challenge to classicism [symbolic processing] and some overtly support the classical picture. So-called implementational connectionists seek an accommodation between the two paradigms. They hold that the brain's net implements a symbolic processor. True, the mind is a neural net; but it is also a symbolic processor at a higher and more abstract level of description. So the role for connectionist research according to the implementationalist is to discover how the machinery needed for symbolic processing can be forged from neural network materials, so that classical processing can be reduced to the neural network account.

I believe it is important for all teachers, including those who teach computer programming, to keep this accommodation between connectionism and symbolic processing in mind, and to maintain a balance of the two views as they teach.

Yuen, after describing his research with computer science instructors, lists the deep, endemic problems they face, and offers four guidelines for how connectionist theory can help alleviate the situation. These suggestions are

[13] <http://plato.stanford.edu/entries/connectionism/>

primarily designed to change teachers' *perspectives* on what it is they are doing. His first recommendation concerns the teacher's focus:

> Given time, resource constraints, students' low ability and motivation, as well as complicated subject content, teachers fell into the dichotomies of whether to stick to the syllabus or to shift to the interesting and lively presentation materials; whether to focus on the public examination or to cultivate students' logical thinking ability; whether to adopt short-term survival teaching or to provide long-term quality instruction. All problems are deep-rooted. `In the connectionist view of mind, there is no mental content to talk about. There are only abilities and dispositions' (Bereiter & Scardamalia, 1996, p. 499). The first pedagogical change, supported by the research in the 1980s, is to adopt a specific teaching focus, a focus geared towards the cognitive development of students, for computer programming. The content should go beyond the public examination syllabus, and emphasize thinking processes and transfer of learning.

We will see this as an implicit characteristic of the postings in the edu-sig newsgroup. The "public examination syllabus" as such is left to the AP classes and the AP exam. Computer programming for everybody is still too new to have been codified into benchmarks, thus teachers so inclined can more easily adopt this connectionist teaching strategy.

Yuen's second recommendation concerns the shift from subject-centered to student-centered teaching:

> Knowledge cannot simply be transferred by means of words, and learning is the product of self-organization (von Glasersfeld, 1998). A connectionist mind is a self-organizing mind. Thus the second pedagogical change is to move the transmission-oriented pedagogy (teacher-dominated or subject-centered) to student-centered and self-organizing learning. Traditionally, teacher-dominated or subject-centered methods refer to the use of learning activities which typically involve direct teacher involvement, participation and interaction with pupils, in which, through the use of informing, describing, explaining, questioning, modeling, demonstrating, and coaching, the teacher transmits the knowledge, understanding, skills and attitudes that he or she wishes to develop (Kyriacou, 1995). This teaching-as-telling method

> is never, *on its own*, sufficient to ensure deep understanding. However, the student-centered approach builds on the idea that students learn best when engrossed in the topic and motivated to seek out new knowledge and skills because they need them in order to solve the problem at hand (Norman & Spohrer, 1996; emphasis added).

We will see near unanimous support of this pedagogical strategy by the newsgroup participants. Consistently and repeatedly the posters emphasized their recommendation that students be allowed to pursue programming problems that piqued their own interests. This does not necessarily obviate the need for teacher- or subject-centered instruction, but certainly sends a strong message not to ignore the benefits of self-organized learning derived from connectionist teaching. For a report on using a decentralized approach to computer programming, see Resnick (1996).

Yuen's next recommendation emphasizes the value of collaborative projects:

> Third, collaborative project work should be encouraged. Project work is a complex cognitive and metacognitive process that requires both hands-on and minds-on learning, that is, concrete subject-based knowledge and abstract high-order thinking skills. Project-based learning is action-oriented and focuses on doing something rather than learning about something (Moursund, 1999). In projects, students engage in a complex process of learning, inquiry and knowledge construction. The result is an artifact, a product of student knowledge that can be shared and critiqued. The resultant artifact becomes a product for review and reflection (Dede, 1998). Projects are contextualized in issues and topics related to the real world and, properly chosen, the context can situate learning to promote authentic learning and improve transfer. In project learning, students construct knowledge by manipulating and extending ideas and information. Moreover, in project work, students share ideas and exchange information as a group and improve knowledge collaboratively. This change is particularly important in learning and teaching computer programming.

We will also see several remarks by the edu-sig posters that note how much programming is done in teams, and many suggestions that programming exercises be done as projects in small groups. There is much research in support of the educational benefits of such project-based learning, and learning computer programming would especially seem to lend itself to such learning-by-teamwork techniques.

Finally, Yuen's fourth recommendation underscores the importance of guidance to ensure that the necessary connections are in fact occurring in the students' minds:

> Connectionist systems `always start with some initial constraints only, and gradually acquire the rest of their knowledge through exposure to a variety of specific examples and repeated correction of inferences about those instances' (Strauss & Quinn, 1997, p. 57). Thus, lastly, students need guided instruction and exploration to insure they acquire the concepts and strategies underlying the diverse activities of computer programming. This emphasis is in line with the research on teaching computer programming since the 1980s.

It is this recommendation that keeps the connectionist theory from becoming *too* open-minded, liberal, or relativistic. The key word here is 'guided' so that students stay on track and on task in their programming endeavors. We will see that this recommendation is in harmony with the goals and expectations of the computer programming for everybody charter. Phil Agre (1998), discussing his reorganized writing class, puts it like this:

> Vygotsky's theory predicts that every classroom becomes part of your mind, and so it's better to have a classroom that embodies positive values. Many liberals, reacting against the rigid and factory-like classrooms of authoritarian educators, have interpreted this idea as an argument against structure. But that's not right. You can't learn without structure, and the key is to provide a structure that adapts to

> each individual's needs. The structure should also provide what Winnicott called "holding", or that others call a "container"—a supportive emotional framework that pushes people to do their best without shaming mistakes.

And this, in turn, is reminiscent of Papert and constructionism:

> There are two basic ideas of education. One is instructionism; people who subscribe to that idea look for better ways to teach. The other is constructionism; we look for better things for children to do, and assume that they will learn by doing. (Pease, 1989)

We see here that constructionism supports connectionism. By 'better things for children to do', Papert means constructing numerous programs, so that gradually, the necessary conceptual connections are formed that lead to programming proficiency.

Evaluation strategies

Once a program has been written or programming course completed, the question of evaluation takes on two guises: How can the program itself be evaluated? And, having completed a course of programming instruction, how can the student/programmer be evaluated?

Program evaluation can be approached from a functional viewpoint, and from a formal viewpoint. A functional evaluation would simply run the program to see if it consistently generated expected results. Sometimes, especially with larger programs involving a graphical user interface, this becomes unwieldy at best due to the difficulty of trying all possible combinations of menu commands and other interactions in the application. One way to mitigate this is to require 'unit tests' with the program. Unit tests are programs that call the solution program and compare its output with what the unit test expects. A way to do this in Python is to use the `doctest` or `unittest` modules.[14]

A formal evaluation of a program means to actually read both the code and the comments; evaluation would be done much as an English teacher would evaluate an essay or other written assignment, preferably in conjunction with a functional evaluation to be sure the code actually runs. As mentioned earlier, projects will often be done in teams, so team presentations can also serve as a formal and functional evaluation tool. Members of the team would present the problem, the algorithmic strategy for solving the problem, the essential portions of code corresponding to program functions, as well as demonstrate the code in action. Instructors may still want to evaluate code comments separately to accentuate their importance to good programming.

Student evaluation can also be done using more traditional methods, if desired. For example, instructors can devise a comprehensive objective exam consisting of questions about syntax, flow control, and other language functions; recognition and interpretation of language statements and expressions; evaluation of simple to more complex programs, predicting their output; making changes to a given code fragment to effect a change in output; creating simple programs to solve a problem; finding the syntactical and

[14] See <http://www.python.org/doc/current/lib/module-doctest.html> and <http://www.python.org/doc/current/lib/module-unittest.html> for details.

logical errors in a given program; and perhaps cloze exercises where blank portions of a functioning program need to be filled in.

Cognitive effects of learning programming

Are there any beneficial effects to learning programming beyond programming itself? For one, programming conveys a kind of perspective about computers and their mechanistic nature. More and more people need to know something of how computing machines and the systems they support operate. According to Kaput, Noss and Hoyles (2002) our very dependence on these machines makes it imperative that we develop a 'sense of the mechanism':

> An accepted (but, as we shall see, fundamentally false) pedagogical corollary is that since mathematics is now performed by the computer, there is no need for 'users' to know any mathematics themselves. Like most conventional wisdoms, this argument contains a grain (but only a grain) of truth. Purely computational abilities beyond the trivial, for example, are increasingly anachronistic. Low-level programming is increasingly redundant for users, as the tools available for configuring systems become increasingly high-level. Taken together, one might be forgiven for believing that the devolution of executive power to the computer removes the necessity for human expression altogether (or at least, for all but those who program them).
>
> In one sense this is true. Precomputational infrastructures certainly make it necessary that individuals pay attention to calculation: and generations of 'successful' students can testify to the fact that calculational ability can be sufficient (e.g. for passing examinations) even at the expense of understanding how the symbols work. In fact, quite generally, the need to think creatively about representational forms arises less obviously in settings where things work transparently (cogs, levers, pulleys have their own phenomenology). Now the devolution of processing power to the computer has generated the need for a new intellectual infrastructure; people need to represent for themselves how things work, what makes systems fail and what would be needed to correct them. This kind of knowledge is increasingly

> important; it is knowledge that potentially unlocks the mathematics that is wrapped invisibly into the systems we now use, and yet understand so little of. Increasingly, we need—to put it bluntly—to make sense of mechanism. (p. 15, .pdf version)

For the most part humans reserve the power of judgment for themselves as more and more work-related practices devolve calculational expertise to the computer. But this power of judgment is strongly dependent upon having a well-developed sense of the mechanism:

> Judgment in the presence of intimate computational power requires new kinds of representational knowledge: distinguishing between what the computer is and is not doing; what can be easily modified in the model and what cannot; what has been incorporated into the model and why; and what kinds of model have been instantiated. (p. 17, .pdf version)

It is computer programming that offers the most straightforward way to achieving this necessary representational knowledge by developing a powerful sense of the mechanism. Recall the quote by Harrell (2003) in Chapter One where he describes the influence of programming languages on media software:

> However, the tools used to present and create media art lie behind every media artwork. The theory of programming languages is a useful means by which to characterize these media. Formal languages offer broad insight into the nature of computational manipulation, and specific organizational structures of imperative languages reveal reflections of these structures in media software. This is a natural reflection because the theory of languages expresses organized models for executing algorithms and structuring data, which are the types of manipulations human creators perform on media when treating it as computational data.

This effect of formal programming languages extends to the structure of the software programs that are created by it. Thus, having acquired a ‘sense of the mechanism’ through knowledge of computer programming, one has enabled at least rudimentary comprehension of other software that reflects 'organized models for executing algorithms and structuring data'.

In a survey of current research findings on the effects of computer programming, Clements (1999) discusses mathematics, including geometry, number, arithmetic, algebra, ratio and proportion; problem-solving and higher-order thinking; language arts; creativity; and social-emotional development:

> From an optimistic perspective, it could be claimed that few educational environments have shown consistent benefits of such a wide scope, from the development of academic knowledge and cognitive processes to the facilitation of positive social and emotional climates. Yet, somewhat paradoxically, realizing these multifarious benefits does not imply lack of focus: Integration into one or more subject matter areas maximizes positive effects.

Liao and Bright (1991) also found that the effects of learning a programming language included greater reasoning skills, logical thinking and planning skills and general problem solving skills. In addition, Mayer, Dyck and Vilberg (1986) discovered that learning to program results in increases in four specific higher-order thinking skills: word problem translation, word problem solution, following procedures, and following directions.

Finally, analogical thinking, the ability to utilize a well-understood problem to provide insight and structure for a less understood problem, is often required in programming where one often draws systematically on prior programs in the development and construction of new programs (Maheshwari, 1997). Lakoff and Núñez (2000) define 'cognitive metaphor' as a “grounded, inference-preserving, cross-domain mapping.” In this sense, most programs are, in fact, cognitive metaphors, where the problem the program is solving is ‘grounded’ outside of the computer, inferences based on the problem are preserved in the programmed solution, and the mapping occurs in the computed domain as the program is developed. Thus, writing programs uses and develops analogical thinking, as Maheshwari and others suggest.

2.3 Introduction to Python

Since the research data focuses almost exclusively on the Python programming language, it will be worthwhile to become acquainted with a bit of its history, its context with other languages, and how the data came to be. In this section, we discuss the advantages and disadvantages of Python, where it fits in the pantheon of computer languages in general, how it is a general programming language as opposed to a domain-specific one, and how it supports different programming styles.

The language’s designer, Guido van Rossum, describes his motivation for Python in an interview:

> In the early 1980s, I worked as an implementer on a team building a language called ABC at Centrum voor Wiskunde en Informatica (CWI). I don't know how well people know ABC's influence on Python. I try to mention ABC's influence because I'm indebted to everything I learned during that project and to the people who worked on it.
>
> ABC's design had a very clear, sharp focus. ABC was intended to be a programming language that could be taught to intelligent computer users who were not computer programmers or software developers in any sense. During the late 1970s, ABC's main designers taught

> traditional programming languages to such an audience. Their students included various scientists—from physicists to social scientists to linguists—who needed help using their very large computers. Although intelligent people in their own right, these students were surprised at certain limitations, restrictions, and arbitrary rules that programming languages had traditionally set out. Based on this user feedback, ABC's designers tried to develop a different language. …
>
> In 1986 I moved to a different project at CWI, the Amoeba project. Amoeba was a distributed operating system. By the late 1980s we found we needed a scripting language. I had a large degree of freedom on that project to start my own mini project within the scope of what we were doing. I remembered all my experience and some of my frustration with ABC. I decided to try to design a simple scripting language that possessed some of ABC's better properties, but without its problems.

The Python website describes the result of Guido’s design in its executive summary[15] as follows:

> Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed.

Python is part of the Open Source Initiative[16] which declares:

> The basic idea behind open source is very simple: When programmers can read, redistribute, and modify the source code for a piece of software, the software evolves. People improve it, people adapt it, people fix bugs. And this can happen at a speed that, if one is used to the slow pace of conventional software development, seems astonishing.

---

[15] <http://www.python.org/doc/essays/blurb.html>

[16] <http://www.opensource.org/>

> We in the open source community have learned that this rapid evolutionary process produces better software than the traditional closed model, in which only a very few programmers can see the source and everybody else must blindly use an opaque block of bits.

This isn't to say that Python cannot be used to create closed, proprietary software applications (it can); however, the spirit in which Python is offered to programmers is usually reciprocated by products written in Python, that is, many Python-based software packages are also open source. Python's open source nature and ability to run on almost any computer likely to be found in school classrooms are two reasons to favor it over other languages that may be proprietary or only run on a few types of computers.

The following two lists summarize the advantages and disadvantages of Python as a programming language and were compiled from the following sources: (Mitchell 1999; Forum 2003; Trauring 2003).

Advantages of Python:

- Portability
    - Python is available on a wide range of hardware and software platforms. This includes: Sun, Intel, IBM, Microsoft Windows variants, Macintosh OS variants and all Unices, as well as less well-known platforms, including PDAs and set-top boxes.
    - Python plays well with other languages. Python programs can be extended using C, C++, or Java, or can be embedded in programs written in these languages.
    - Python plays well with programming standards. Many high-quality Python extensions are available which support almost all Internet standards, CORBA, COM, SOAP, XML, XML-RPC and so on.
- Powerful Simplicity

- Python programs are far quicker to develop than other high-level languages. Because of the elegance and simplicity of the language, Python programs tend to be 3-5 times shorter than their equivalent in Java, and 5-10 times shorter than C++ equivalents.

- Python programs are easier to maintain. The simple, clean syntax not only allows the original developers to remember what they did, it also allows other developers to understand and change programs. This allows for much lower maintenance costs for Python programs.

- Python is an extremely versatile language. It can be used for the simplest scripting applications, as well as for the development of complex websites, and all the way up to the construction of complex distributed applications.

- Python's object-oriented paradigm is the most powerful and easiest to use of any commercial programming language.

- Convenient hassle-free strings, lists, and dictionaries that will store as many objects as you like with no need for you to know in advance how many you want to store.

• An extensive function library.

  - Python makes it almost impossible to write obfuscated code. Block structure is indicated by indentation, outline-style. The syntax is clean, with a consistent calling structure for modules and functions.

  - Python uses small, well crafted components, called modules. Modules are very easy to design and use, which encourages formal and informal code libraries.

  - Python is a byte-compiled language (conferring advantages of speed) as well as an interpreted language (conferring advantages of ease for the programmer).

• Free/Open Source Software (FOSS)

  - Thousands of programmers around the world contribute to the development of Python. These programmers are not being paid to develop Python, but are being paid to develop applications. They use Python in real world settings. This ensures that Python is robust, secure, relatively efficient, portable, scalable and has a feature set which meets real world needs, not what vendors think customers should have.

- Python like all FOSS software has three freedoms: It can be used freely, without paying exorbitant licensing fees. It ensures free development of software applications without any artificially imposed vendor restrictions. It ensures a free software market, eliminating enterprise dependence on any software vendor large or small.

- The online documentation that comes with it is thorough and good.

Disadvantages of Python:

- If you want other people to use your Python programs, you have to persuade them to install Python — and they have to install the whole development environment.

- There is no standard Python graphical user interface library. Instead, it borrows from elsewhere. The nearest thing to a standard is Tkinter, which is supplied with Python and seems to be adequate.

- Lists are indexed from zero (0, 1, 2, 3, etc.). This doesn't come naturally to me or to most other humans, who are accustomed to count things starting at 1.

- Python is case-sensitive. The variable names BOB, Bob, and bob are different and can have different values. (This is actually an advantage according to many programmers.)

Despite these minor disadvantages, I believe the preponderance of advantages, especially with respect to the clear, clean syntax of the Python language, makes it the overwhelming choice for a first programming language for middle and high school students. For a comparison of Python and other languages with respect to their applicability as a first programming language, see Georgatos (2002).

Kinds of computer languages

To understand where Python belongs in the pantheon of computer languages, we need to establish context by considering the three general ways source code (written by humans) can interact with a computer's central processing unit (executed by the machine). Much of the information in this section comes from "The Unix and Internet Fundamentals HOWTO" website written by Eric Raymond.[17]

The most conventional kind of language is a compiled language. The programmer writes code (called source code) that gets translated into binary machine code by a special program called a compiler. Once the binary has been generated, you can run it directly without looking at the source code again. This is the format in which most commercial software is delivered. Compiled languages tend to give excellent performance and have the most complete access to the underlying operating system, but are also considerably more difficult to program. The best known examples of compiled languages are C, C++, FORTRAN and COBOL.

The second kind of computer language is an interpreted language. Instead of a compiler translating source code into independent binary machine code, the language depends on an interpreter program to read the source code line by line and translate it directly. The source code has to be re-interpreted (and the interpreter present) each time the code is executed.

[17] <http://www.ibiblio.org/pub/Linux/docs/HOWTO/other-formats/html_single/Unix-and-Internet-Fundamentals-HOWTO.html>

Interpreted languages tend to be slower than compiled languages, and often have limited access to the underlying operating system and hardware. On the other hand, they tend to be easier to program and more forgiving of coding errors than compiled languages. Because of their general slowness, pure interpreted languages are not widely utilized, especially due to the rise of the third kind of language.

This third kind is known by at least two different appellations: P-code, and tokenized. With either term, it refers to a kind of hybrid language that uses both compilation and interpretation. When the language's source code is executed, the interpreter (also known as a parser) creates 'tokens' in language-specific binary files (rather than machine-specific binaries like a compiler would). It is these language-specific tokenized binaries that then get executed, and they do so independently of the source file. This tokenizing process causes the initial startup of an application to be slow, but then the program executes very fast: often it's hard to distinguish between a compiled program and a tokenized program in terms of speed. Thus, tokenized languages blend the flexibility and power of a good interpreter with the speed of a compiled language. Python, Perl and Java are the three best known such languages.

Alan Gauld offers a concise summary of the differences in his book *Learning to Program*[18]:

[18] <http://www.freenetpages.co.uk/hp/alan.gauld/>

> Basically a programmer writes a program in a high level language which is interpreted into the bytes that the computer understands. In technical speak the programmer generates source code and the interpreter generates object code. Sometimes object code has other names like: P-Code, binary code or machine code.
>
> The interpreter has a couple of names, one being the interpreter and the other being the compiler. These terms actually refer to two different techniques of generating object code from source code. It used to be the case that compilers produced object code that could be run on its own (an executable file - another term) whereas an interpreter had to be present to run its program as it went along. The difference between these terms is now blurring however since some compilers now require interpreters to be present to do a final conversion and some interpreters simply compile their source code into temporary object code and then execute it.

Thus Python benefits from not having an explicit compile step (the executable portions are automatically generated) and yet runs reasonably fast enough for most programming situations. For situations where greater speeds are warranted, a few different strategies are available, but these are outside the scope of this overview.

John Ousterhous (1998) has written of the ascendancy of scripting languages over system programming languages. In his terminology, system programming languages are the compiled languages and scripting languages are the interpreted and tokenized languages described above:

> Scripting languages and system programming languages are complementary, and most major computing platforms since the 1960's have provided both kinds of languages. The languages are typically used together in component frameworks, where components are created with system programming languages and glued together with scripting languages. However, several recent trends, such as faster machines, better scripting languages, the increasing importance of graphical user interfaces and component architectures, and the growth of the Internet, have greatly increased the applicability of scripting languages. These trends will continue over the next decade, with more and more new

> applications written entirely in scripting languages and system programming languages used primarily for creating components. ...
>
> Scripting languages represent a different set of tradeoffs than system programming languages. They give up execution speed and strength of typing[19] relative to system programming languages but provide significantly higher programmer productivity and software reuse. This tradeoff makes more and more sense as computers become faster and cheaper in comparison to programmers. System programming languages are well suited to building components where the complexity is in the data structures and algorithms, while scripting languages are well suited for gluing applications where the complexity is in the connections. Gluing tasks are becoming more and more prevalent, so scripting will become an even more important programming paradigm in the next century than it is today.

Python is an exemplar of the scripting paradigm Ousterhous describes.

General and domain-specific languages

Python is a general programming language, not a specialized one. The distinction is important as it relates to the consequences of learning how to program. In the Computer Programming for Everybody proposal (discussed more fully later in this chapter) the distinction was described as follows:

> 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[20] is the key concept here.

[19] "A type is a classification of data that tells the compiler or interpreter how the programmer intends to use it. For example, the process and result of adding two variables differs greatly according to whether they are integers, floating point numbers, or strings." <http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?type>

[20] "A Turing-complete system is one which has computational power equivalent to a universal Turing machine. The concept is named in honor of Alan Turing. In other words, the system and the universal Turing machine can emulate each other. No computers completely meet this requirement, as a Turing machine has unlimited storage capacity, impossible to

> 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.

Wrestling with this distinction came up significantly in a number of places in the data. For example, during a discussion of why girls seem to be much less interested in programming than boys, attention turned to the viability of using a MOO (multi-user domain, object-oriented), a virtual meeting place where participants interact, chat-like, and create and change the virtual environment through issuing special commands. (See Bruckman (1995) for more information.) The ‘special commands’ that can be expressed in a MOO constitutes a *domain-specific* language, and some participants in the list objected that focusing on such would detract from the business at hand,

emulate on a real device. With this proviso, however, all modern computers are Turing-complete, as are all general-purpose programming languages.” <http://en2.wikipedia.org/wiki/Turing-complete>

teaching a *general* language, Python. In this particular case, finding the right combination of using both the general aspects of the Python language and the domain-specific aspects of a MOO language proved to be too nettlesome and no consensus was reached. Some of the participants advocated a kind of augmented MOO where Python language constructs could be used directly as MOO commands to create a much richer environment than the traditional MOO environments. However others noted that there was already an established (domain-specific) syntax for online MOO interactions that ought to be respected; Python should simply be used to (re)create the environment.

This same issue arose in various other contexts, usually with respect to the advisability of using packages that facilitate the creation of two- and three-dimensional graphics. These packages often include a domain-specific macro language to interact with the package, possibly detracting from using and learning the Python language. Often the issue was resolved by noting that Python can be used as a control language issuing domain-specific commands to the package, thus combining the two types in a way advocated in the quote above. Nevertheless, we will see in Chapter Four a similar situation where no such resolution was achieved.

Styles of programming

There are three styles of programming in general use: procedural (also known as imperative), functional, and object-oriented. Different languages support these different styles to greater or lesser extents. Python supports all three of these styles, with perhaps a lesser emphasis on the functional style. The functional style is not to be confused with *functions*, which are chunks of reusable code. Creating and calling functions is fully supported in Python, and indeed, the ease with which one can do so is one reason for its popularity as a first programming language. Here is a posting by one high school Python teacher, Mr. Elkner:

> Another example of how Python aids in the teaching and learning of programming is in its syntax for functions. Of all the things that I learned by using Python this year, the way in which the right tool could help in explaining functions was the most exciting. My students have always had a great deal of difficulty understanding functions. The main problem centers around the difference between a function definition and a function call, and the related distinction between a parameter and an argument. Python comes to the rescue with syntax that is nothing short of beautiful. Function definitions begin with the key word 'def', so I simply tell my students, "when you define a function, begin with 'def', followed by the name of the function that you are defining, when you call a function, simply call (type) out its name." Parameters go with definitions, arguments go with calls. There are no return types or parameter types or reference and value parameters to get in the way, so I was able to teach functions this year in less then half the time that it usually took me, with what appears to be better comprehension.

The functional *style* of programming is something different, and is largely ignored in this study. However, for a useful introduction to functional programming in Python, see Gauld (2000).

In his introduction to object-oriented design in *Data Structures and Algorithms with Object-Oriented Design Patterns in Python*, Preiss (2004) gives a high-level overview of the differences between these three styles:

> Traditional approaches to the design of software have been either data oriented [procedural] or process oriented [functional]. Data-oriented methodologies emphasize the representation of information and the relationships between the parts of the whole. The actions which operate

on the data are of less significance. On the other hand, process-oriented design methodologies emphasize the actions performed by a software artifact; the data are of lesser importance.

It is now commonly held that object-oriented methodologies are more effective for managing the complexity which arises in the design of large and complex software artifacts than either data-oriented or process-oriented methodologies. This is because data and processes are given equal importance. Objects are used to combine data with the procedures that operate on that data. The main advantage of using objects is that they provide both abstraction and encapsulation.

We will see in Chapter Four some disagreement among the participants as to which style(s) should be taught to beginning programmers, and when. Fortunately, the Python language supports whatever combination of styles a teacher determines is best for his or her students. And, arching over all the styles is a kind of rubric governing the approach to programming that Python favors. If one goes to a Python prompt (>>>) in the interpreter and types 'import this' (without the quotes), the following poem will appear on the screen:

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one--and preferably only one--obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
```

```
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

These lines express half-whimsically, half-seriously the overall design principles of the Python language.

2.4 Computer Programming for 'Everybody'

In 1999, a proposal was submitted by Guido van Rossum to the Defense Advanced Research Projects Agency (DARPA) titled "Computer Programming for Everybody: A Scouting Expedition for the Programmers of Tomorrow".

The proposal was predicated on 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.

Of the first two points, the proposal goes on to say:

> 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. ...
>
> We plan to start by basing both components on Python, a popular free interpreted object-oriented language. ... 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.

The proposal also described the motivation driving it:

> 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. ...
>
> 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!

As we saw in Chapter One, the vision of the future was articulated as follows:

> 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, 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.

Despite the attractiveness of this vision, however, there are some who question the advisability of teaching everyone to program a computer. One such discussion[21] offers a sample objection with a few responses:

> I have a problem with this idea - it smacks of both a denigration of software development as a profession and seems to imply that everyone should be able to tell a machine what they want it to do. Look at the tax system - everyone with an engineering degree can do the math necessary to calculate their taxes, but many choose to have someone else do their taxes. Why? Because some things are better left to the specialists. A similar trend exists in auto maintenance - why should I buy and store a set of tools to change my engine oil, filters, brakes, etc, when for less that half the cost in materials, and much less in my time, I can drive the car to the shop, wait 30 minutes, and have it done for me? Software development is not trivial, and the average person without training in the field will produce poor solutions. –PH

> You can hire a CPA to do your taxes, but is it really worth it if you're using the short form and can get it done in a half hour? In the same way, I've found many uses for just firing up Python and typing a few lines to get a job done, or writing a little one-line Perl or Awk program to do a job for me. Who's to say that others wouldn't benefit? And even if the average person doesn't need to program, learning how to do so might make the computer itself seem like a less mysterious and cryptic device. –NB

> I like the analogy to writing. Folks with serious talent do it as their primary profession--maybe compare a novelist to a compiler hacker. Folks with other types of talent use writing as a tool--compare lawyers who write briefs to web designers who develop Javascript maybe. Almost every one else writes on an amateur basis, but can we say the same thing about programming? There seems to be discontinuity in the spectrum with programming. –SH

> Inspired by CP4E, Yorktown High School is now in its 4th year using Python in an intro CS course. My goal is to get as many students from as broad a range of interests and backgrounds as possible into the course. I do not aspire to make computer programmers out of everyone. I do want to empower all my students with a greater understanding of what a computing machine is and what it is capable of doing. I want to

[21] <http://c2.com/cgi/wiki?ComputerProgrammingForEverybody>

> “demystify” computer programming for all students in the class. Given how important these machines are becoming in the production of our world, such an understanding has important implications for democracy. It has been both fun and challenging working successfully with a very heterogeneous population of students. Using Python has played a big part in making the course as successful as it has been. –JE

The automotive repair analogy seems compelling on the surface: after all, cars and computers are both machines, and when something goes wrong, it appears that we need a professional mechanic or programmer to fix the problem rather than try to fix it ourselves. Also, we use the same word, ‘tools,’ to refer to devices that both use to perform their jobs. Furthermore, it seems easy to equate the experience of driving a car with using a computer, a car’s engine with a computer’s central processing unit, and a car’s auxiliary subsystems such as braking, electrical, cooling and steering with a computer’s auxiliary subsystems such as memory, hard drive, monitor, and operating system. But the analogy of mechanic to programmer is flawed primarily because of the nature of the material each role manipulates.

Notice the qualitative difference in the problem solving that a mechanic performs and the problem solving that a programmer performs. A mechanic is primarily concerned with *restoring* a car’s existing function, for example, replacing brake pads, changing the timing chain, or repairing a clutch. These functions all existed prior to an owner’s bringing the car to the mechanic, and the problem solving activity is designed to bring the function back to its preexisting state.

However, the nature of the problem solving that a programmer undertakes, especially in the context we are considering here, is to *create* a function that doesn't yet exist. This isn't to say that programmers don't ever fix problems like mechanics do (restore a prior function), but that isn't the main focus of their work. It also isn't to say that auto mechanics never create an artifact or new function, but that too is not the main focus of their work.

This qualitative difference is primarily due to the material that each works with: mechanics work with hardware, programmers work with software. Mechanics work with parts that deteriorate with vehicular use and eventually need to be restored. Programmers work with ideas and algorithms that do not rust, corrode or wear out. They may need to be improved, or replaced with better ideas and algorithms, or 'ported' to another language or machine, but they do not deteriorate.

We can now clearly see how the analogy is flawed by considering the purpose of the machines under consideration. The purpose of a computer is to compute, in the broadest sense of the word. Assuming a working computer, you can consult a programmer for a solution to the problem of computing, for example, a list of prime numbers. However, the purpose of a car is to transport. Assuming a working car, you would *not* consult a mechanic for a solution to the problem of transporting yourself, for example, to Glacier National Park: The mechanic helps you with the vehicle; the programmer helps you with the trip.

Finally, unlike owning a car, having a computer often means that the tools you need to do programming are already on your hard drive. (And if not, they are easy to obtain and install.) Unfortunately, those tools are often neglected by ordinary users because they believe programming is difficult to learn and therefore don't attempt it. The CP4E effort is designed to radically alter that belief by teaching people to use a particular set of tools centered on Python designed for beginners.

The more compelling analogy to learning programming is learning to write, which was covered in Chapter One. Computing and driving are much more *dis*similar than similar; computer programming is much more similar to writing than to fixing an automobile. Thus the goals of Computer Programming for Everybody are both laudable and plausible:

> 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.

Having said this, I would like to temper the apparent ambition of the goal (computer programming for *everybody*) with a couple of observations. Recalling Mihai Nadin's call for *multiple* literacies, we have to admit that programming for everybody is just as unfairly hegemonic as literacy for everybody:

> I am not proclaiming that tomorrow we should not teach language anymore. My major message is to teach together: traditional literacy, visual literacy, multimedia literacy. Let's give every individual the possibility to unfold according to his or her abilities. Some people are not, due to biological constitution, inclined toward a literate mode of

> expression. Others are more inclined. If you start now working toward a multitude of forms of expression, you are going to achieve the possibility of allowing each individual to reach his or her potential and that is something which literacy never allowed. Literacy is a very powerful instrument which says that the whole society is going to fit into a single model, a single mold. It cannot be done. There are many people given their biological condition who will never be able to write correctly. We keep telling them they have to, they have to. Why not give them something which corresponds to their biological precondition? That is what I meant by opportunity, multiple literacies.[22]

This echoes Howard Gardner's theory of multiple intelligences (1999) where he defines 'intelligence' as "the ability to solve problems and fashion products that are valued within one or more cultural settings." He identifies nine types of intelligences:

- Verbal-Linguistic Intelligence—well-developed verbal skills and sensitivity to the sounds, meanings and rhythms of words
- Mathematical-Logical Intelligence—ability to think conceptually and abstractly, and capacity to discern logical or numerical patterns
- Musical Intelligence—ability to produce and appreciate rhythm, pitch and timber
- Visual-Spatial Intelligence—capacity to think in images and pictures, to visualize accurately and abstractly
- Bodily-Kinesthetic Intelligence—ability to control one's body movements and to handle objects skillfully
- Interpersonal Intelligence—capacity to detect and respond appropriately to the moods, motivations and desires of others.
- Intrapersonal Intelligence—capacity to be self-aware and in tune with inner feelings, values, beliefs and thinking processes
- Naturalist Intelligence—ability to recognize and categorize plants, animals and other objects in nature
- Existential Intelligence—sensitivity and capacity to tackle deep questions about human existence, such as the meaning of life, why do we die, and how did we get here.[23]

[22] <http://www.mime.indiana.edu/mihaiNadin/>

[23] <http://www.thirteen.org/edonline/concept2class/month1/>

Programming aptitude is likely to be a combination of two or more of these intelligences, and it is highly unlikely that any combination will be applicable to *everybody*. 'Everybody' serves as a proxy for 'more than just those who make a living by programming'; analogous to those who are literate, yet do not make a living by writing. One poster put it like this: [24]

> And the same aforementioned marketing-python poster pointed out that brain surgery is not for everybody, and that programming is also not for everybody. And I agree. But the CP4E slogan is using hyperbole to make a memorable point: that programming can be usefully extended to a far greater audience. –Ron Stephens

This same person serves as an excellent example of the effect of learning Python programming by a non-Computer Scientist. This rather lengthy posting made to the computer.language.python newsgroup summarizes his involvement with Python inspired by the CP4E movement:[25]

> As most of you know, the founder and creator of Python has stated an interest in bringing the joys of computer programming to a wider audience, and he has coined the phrase "Computer Programming for Everybody" to sort of sum up this concept. I fit in this category, for prior to about 3 yrs ago, my programming experience and knowledge was next-to-none. After 3 yrs of enjoying Python as a hobby in my limited spare time, I have created a program, which I call askMerlin, that is simple enough for a newbie to understand, yet interesting and different.
>
> Having greatly cleaned up the original code, the basic program is now a class Decision() that allows one to simply analyze any decision one needs to make. The other modules are subclasses of Decision() and are sort of ultra-mini expert systems on how to decide what to eat for lunch, who to vote for in an election, how to predict who will win a given basketball game, football game, and how to answer yes or no type questions. The

[24] <http://groups.google.com/groups?th=26115a570df7854e>

[25] Ibid.

> last two modules apply the same technique but also utilize the Internet to gather data used to analyze and make decisions.
>
> I think it is a fun little program. Newbies can not only understand it, but can add simple modules, in any area of their expertise, by simply subclassing Decision(), or by simply creating an instance of Decision and over-riding a few key methods, or perhaps more simply by imitating the functional logic.
>
> OK, that's my spiel. The program can be found at my web site at http://www.awaretek.com/plf.html
>
> humbly-yet-fearlessly-treading-where-he-ought-not-ly-yours
>
> Ron Stephens
>
> P.S. of course I am still adding to this program, especially the internet-enabled part, and welcome ideas / contributions from any and all, especially relative newbies who need a place to start.
>
> P.S.S. to Newbies, the best way to understand this program is just to run it a few times. It is a command line program, and although I have cleaned up the code, I also removed for the time being the documentation and comments, that were becoming unwieldy after three years of fiddling. I think the code speaks for itself, especially after you run it a few times and see how it works. And yes, I do intend to add some simple comments back into the code ;-))) eventually. Also, I apologize in advance for any indentation errors anyone has after downloading or cutting and pasting the code; just run the program and let the Python error messages help you sort out any formatting or indentation errors you may encounter.
>
> The basic idea is to choose between a few options or alternatives, by basing your choice on a few good criteria, with each criteria having a weight, or relative-importance-factor, and then, by various means, determining a score for each option on each criteria. There, that's the documentation for now ;-)))
>
> And lastly, yes, I do realize it isn't much to show for three years of part time work, but hey, I give me A's for persistence.

One person questioned the connection of his software program to the CP4E slogan. Here was the gist of Ron Stephens' reply:

> Look, the main thing is this. My program is trivial; I could have done more than half of it by simply using an Excel spreadsheet way back 3 years ago. But I wouldn't have, and the Python program, trivially simple as it is at its roots, raises interesting ideas for extension and expansion that could just possibly lead to quite interesting uses. It will still be trivial in most ways, it won't ever earn a buck, but it could be interesting and useful; and I don't think I would have been motivated to follow up on any of those ideas in a lesser language in which the barrier to such exploration was higher than I would want to pay. Python seems to encourage and reward incremental effort, and it leads one to explore extensions and improvements to programs because the language makes it relatively easy to see how to do the extensions.
>
> And it gives me satisfaction; and, if Python continues to stimulate my interest and motivation for another 25 years (if I should live so long...I am 50 already!) then who knows, it just might be a nice program in 2028 when I pass on. Stranger things have happened.
>
> And how else could I have spent my precious leisure time? Watching TV? Reading more books? They would have earned me no more money, and perhaps simulated my mind less.
>
> But the real payoff, I am sure, is the community. Even reading the great classics would not have expanded my community of social contacts and interaction with intelligent, humorous people who help me with my programs, answer my questions, and enliven my life with discussions I would have not otherwise been privy to.
>
> And so if I feel bold enough to share my little program with the newsgroup, and risk the ridicule of showing off my Mickey Mouse side (as one of my beloved Physics professors used to say), then so be it.
>
> Watching Python grow is a good pastime. If even one other person benefits from seeing my program, even if it is to suddenly realize and say to themself "Hey, I could do that a lot better than that" then I am happy I shared it.
>
> If someone codes a mini-expert system to add to the program, I will be ecstatic. And if none of this happens, I am none the worse for wear.

2.5 Edu-sig newsgroup

A few months after the Computer Programming for Everybody proposal was made public, a new Python special interest group was formed which refers to itself as the ‘edu-sig’ newsgroup. We find the following description of the group on their home page:[26]

> Python continues to make inroads at all levels in education. As a first programming language, Python provides a lucid, interactive environment in which to explore procedural, functional and object oriented approaches to problem solving. Its high-level data structures and clear syntax keep the forest in view through the trees. …
>
> Edu-sig provides an informal venue for comparing notes and discussing future possibilities for Python in education.

Discussion in the group ranges over a wide variety of Python-related topics at all levels, from beginning programmers to advanced applications. The group constitutes a ‘global discourse community’, in the sense described by Killingsworth and Gilbertson (1992):

> *Global discourse communities*, by contrast [to local discourse communities], are groups of writers and readers defined exclusively by a commitment to particular kinds of action and discourse, regardless of where and with whom they work. (original emphasis) (Killingsworth and Gilbertson, 1992, p. 162; in Swales, 1998, p. 201)

The original (1992) sense of these ‘particular kinds of action and discourse’ tended to include practices such as conference participation and professional journal subscriptions; however, with the rise of the Internet, it can also include membership in online newsgroups such as the Python edu-sig.

Swales (1998) describes how Porter (1992) characterizes this kind of participation in conferences and journals as a forum:

> A forum is a “concrete, local manifestation of the operation of a discourse community” (1992, p. 107). For Porter these fora can range from being a defined place of assembly, to being an occupational

[26] <http://www.python.org/sigs/edu-sig/>

> location, and on to being a vehicle for discourse community connection, such as a conference or a journal.

Or a newsgroup. The ‘local manifestation’ that defines the edu-sig forum is the set of messages that are sent to each member of the group, and which are archived in a public repository. And these texts are unified by the group’s focus on promoting the use of Python as a first programming language in school settings, thus defining the group’s raison d’être.

In the next chapter I characterize the message data more fully, and we’ll see that much of the data was excluded from consideration for various reasons, but let me mention here a bit about the members of the group. Across all of the messages in the initial dataset there were 260 different posters. Of those, 146 posters had messages in the selected set of message threads. Of those, I looked at the 67 posters who had more than three postings in the data (which included Guido van Rossum). I looked to see what ‘roles’ they could be said to be playing while participating in the forum. In trying to ascertain those roles, I examined any self-introductions they might have made, the signatures they might have appended to their posts, their email domains, and any self-referential information they might have included in a post. Nevertheless, there were a few posters whose role I could not classify.

There were two primary roles identified: education, and computer business, with a third minor role of writer. The ‘education’ role included teachers and professors at the high school and college levels, a few students (primarily high school), parents in a teaching role with their children or other relatives, and others whose primary mission appeared to be actively bringing computer programming into a school's curriculum. Of these, relatively few appeared to be teachers in content areas other than programming. Of those few, the subject areas represented were Math and Physics, as well as one in Art.

The 'computer business' role included a wide variety of for-profit and non-profit endeavors based on whatever I could glean from references to web pages in their signature files or in their postings. The 'writer' roles were mainly identified by personal familiarity with their writings, for example, authors of Python-related books in bookstores, and articles in computer-related magazine and websites.

Some of the roles were 'blended' roles, for example, a person in an IT department at a college: education or computer business? A person selling educational training software: computer business or education? A published writer who teaches at the local community college: writer or educator? I had to read a sampling of these people's postings to get the flavor of which role seemed more appropriate for them. For a breakdown of the 62 posters I was able to classify, representing the most prolific posters in the edu-sig membership that contributed to the data I eventually analyzed, see Table 4.

Table 4 Newsgroup participant roles

No.	**Role**	**Percentage**
35	Educators	56%
22	Computer business	35%
5	Writers	8%

These participants contributed the bulk of the messages that constituted the selected data. In the next chapter, we take a look at some statistical data across all the messages in the archive, the procedures used to reduce the archive to a manageable size of relevant messages, and the qualitative framework within which the selected messages were analyzed.

CHAPTER 3

METHODS AND PROCEDURES

The so-called empirical science of nature is, as actually experienced, the highly contrived encounter with apparatus, measuring devices, pointer readings and numbers. Inquiry is made 'methodical,' through the imposition of order and schemes of measurement 'made systematic' under rules of a new mathematics expressly invented for this purpose. This mathematics orders an 'unnatural' world that has been intellectually 'objectified,' re-presented or projected, ripe for the mind's grasping—just as the world itself will be grasped by the techniques that science will later provide. Even the modern word 'concept' means 'a grasping-together,' implying that the mind itself, in its act of knowing, functions like the intervening hand—in contrast to its ancient counterpart, 'idea,' 'that which can be beheld,' which implies that the mind functions like the receiving eye. Science becomes not the representation and demonstration of truth, but an art: the art of finding the truth—or, rather, that portion of truth that lends itself to being artfully found.

—*Leon R. Kass,* Life, Liberty and the Defense of Dignity

3.0 Introduction

As described in the previous chapter, the source of data for this dissertation was a set of messages posted to a public newsgroup. This data had to be subjected to a set of procedures in order to extract the salient items before a content analysis could commence. This chapter describes those procedures as well as offering various statistical profiles of the messages. Along the way, we discuss two procedures, PhraseRate and TextGraphs, which provided two ways of abstracting the contents of the message threads. We end by discussing the type of analysis used on the data.

3.1 Data profile

I began my research with a single large mailbox file that came from downloading the message archive from the edu-sig website. The first message in the database was posted Jan. 29th, 2000; the last one was Apr. 18th, 2003 (my data cutoff point). In order to work with this data, I had to import it into a database. This process proved to be less straightforward than it perhaps should have been, but thanks to a key script found on the web, was readily accomplished. The steps I followed were:

- Open and save the mailbox archive in text format using BBEdit text editor.
- Import messages into Mailsmith mail reader: resulted in 2,782 messages.
- Export messages into Filemaker database using custom Applescript[27] (this didn't work the first time. It took a few hours of troubleshooting to discover the problem, which, fortunately, was easily solved: reformatting the date fields in the database)

I now had a 'messages' database where each record represented a single message posted to the edu-sig group. Resisting the temptation to begin at the beginning and read every message in chronological order, I determined that it would be best to randomly sample the database of messages to get a global feel for the variety of the topics of discussion and the range of the group's 'tenor'. It seemed as though 10% of the messages would constitute an

[27] http://home.knuut.de/paul/scripts/mailsmith/Export_to_FMPro3.html

adequate sampling, but this figure was arrived at intuitively rather than empirically. Also, there was another factor to consider. All messages occur in the context of a 'thread', that is, a series of responses to an initial posting, or responses to responses, etc. (There were many messages with no responses; these were considered threads with a length equal to one.) Operationally, I defined a thread as a set of messages ordered by date and time which all have the same Subject header.[28] I then constructed a related database table ('threads') that brought each message into its thread, and counted how many messages were in each thread (even if that thread was only 1 message in length). Initially there were 758 threads ranging in length from 1 to 42 (see Fig. 1). As we can see, almost half of the threads were only one message long.

To obtain my random sample, I wrote a short database script that randomly selected 5% of the messages for initial perusal. In addition, I wrote another script to select the second random 5% of the messages, but chosen at the thread level. That is, *threads* were selected at random, and a running total of the messages they constituted was kept until 5% of the messages were selected. (As it turned out, there was little overlap between these two samples.)

I then read these messages to acquaint myself with the data. As I read, I marked each message with one or more codes that indicated the content of

[28] In practice, posters occasionally continued an existing thread with a modified or completely new subject title, so some threads seem to begin in the middle of a conversation. These 'branches' were treated as separate threads.

the message. These codes were not used directly in the analysis process, rather, they constituted the groundwork as I created the categories into which the threads were sorted during the later thread selection process.

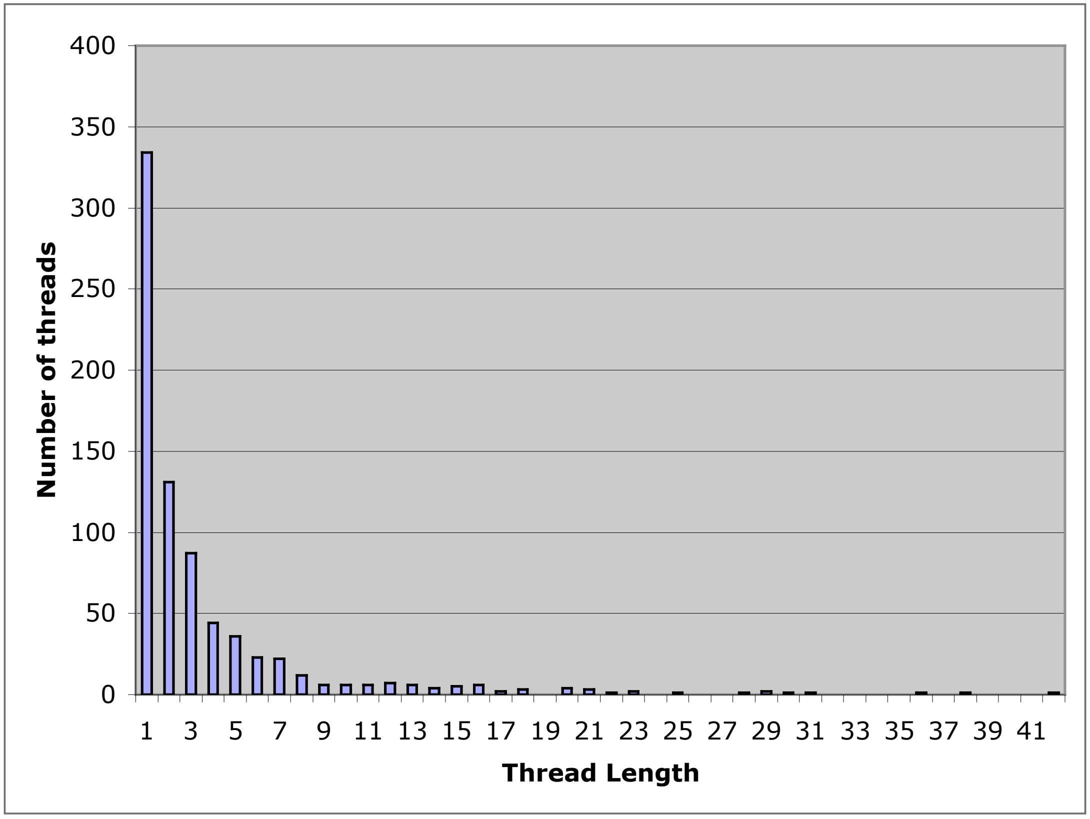

Figure 1 Distribution of thread lengths

Poster participation

We see in Figure 2 the ebb and flow of messages posted month by month over the 3-year time period. There is an initial burst of enthusiastic activity in the first month, February 2000, which is never subsequently equaled (although May and June 2001 come somewhat close).

In Figure 3 we see the distribution of messages by poster. Of the 265 total posters to the group, 112 posted a single message. However, the group also has a few prolific posters: shown at the far right is one who posted 110 messages during the 3-year period. In addition, not shown on the chart for reasons of scale, are two outliers, one who posted 325 messages, and another who posted 605 messages!

Thread Dynamics

We can also examine the distribution of messages through time within a thread. In Figures 4 – 12 we see the number of messages per day over the lifetime of the nine longest threads in this study. (These figures are ordered by date of first posting.) There didn't appear to be any consistent pattern to these distributions, but a larger sample might produce some similarities that would allow threads to be grouped accordingly.

In Figure 13, we see this group's thread persistence – the average number of days a thread of a given length persisted. So, for example, threads of length 10 lasted an average of 3 days. Keep in mind that the sample size of the longer threads is much smaller than the shorter ones; much more variation will be exhibited. As it turned out, the average duration of all threads of any length (greater than 1) is almost 6 days.

Another measure of thread dynamics is to consider the number of different posters that contributed to the thread. In Figure 14, we see that the number of different posters increases as thread length increases. As in the previous figure, the number of longer threads is much lower than the number of shorter ones. The average number of different posters over all thread lengths (excluding length 1) is 3.4.

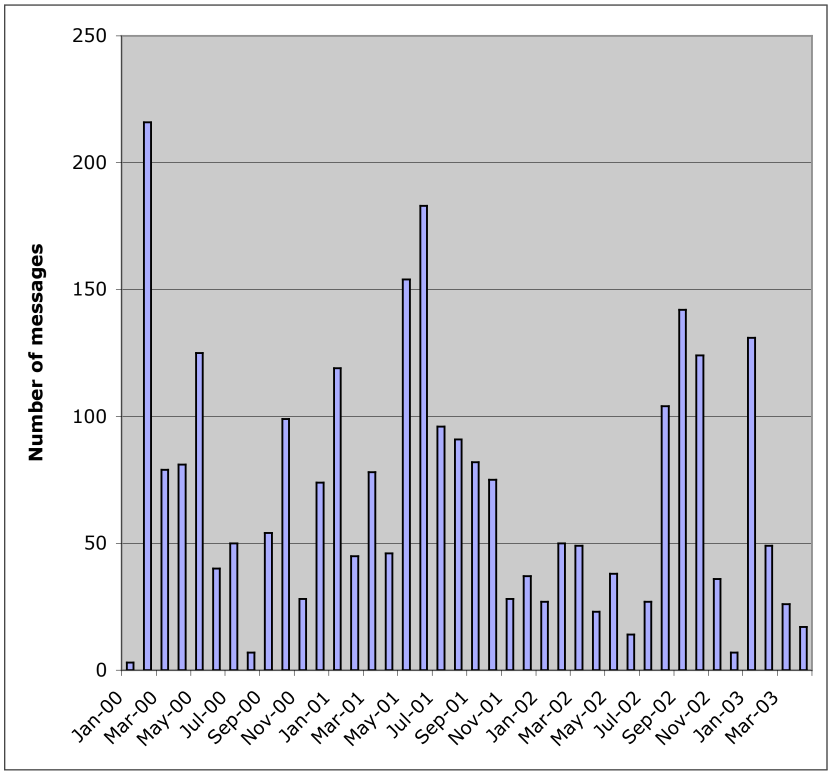

Figure 2 Total no. of messages posted each month

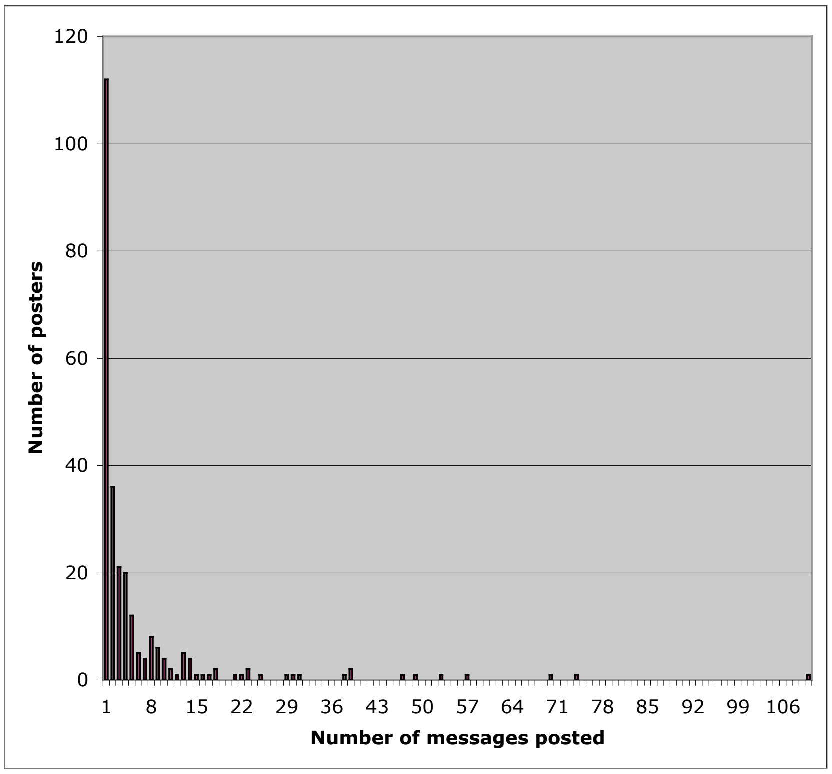

Figure 3 Most participants posted few messages

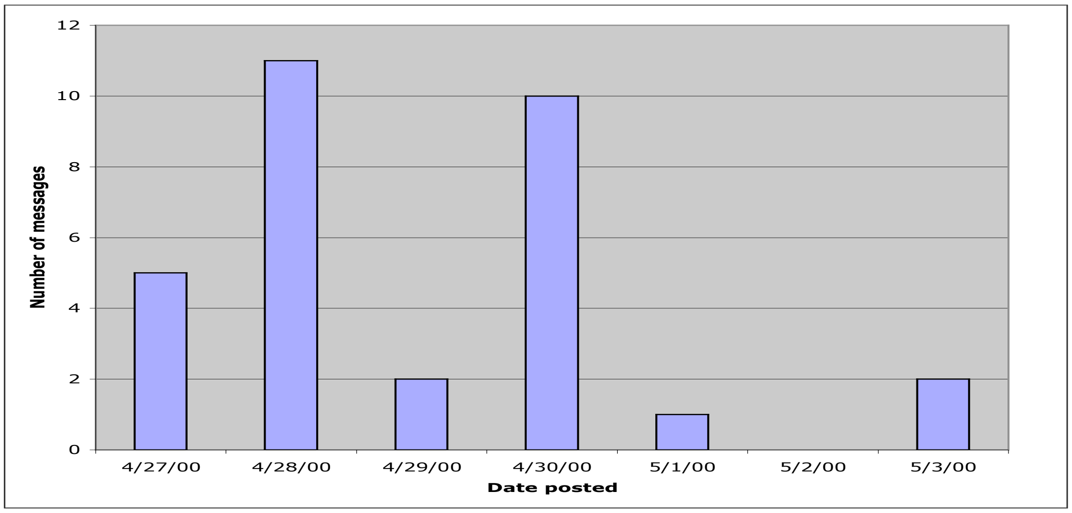

Figure 4 Messages per day for thread length 31

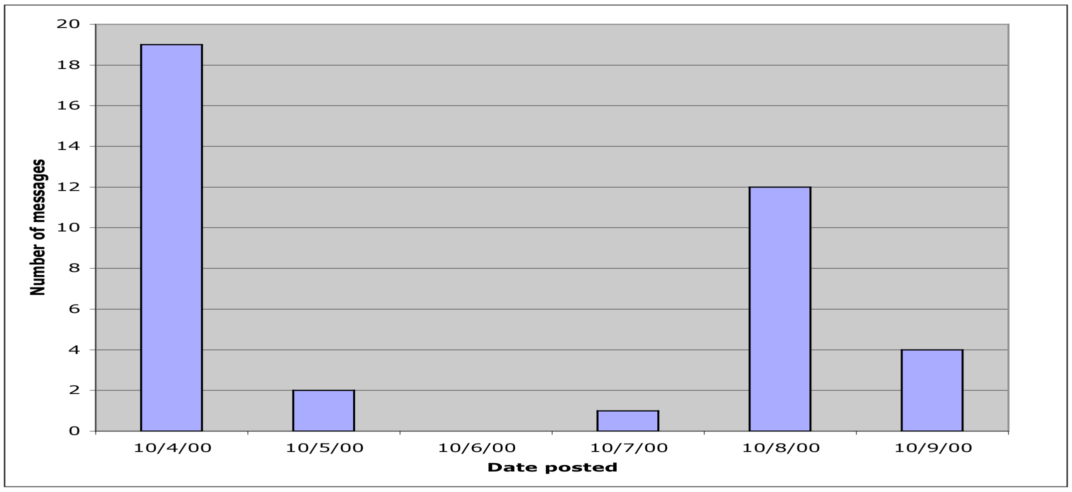

Figure 5 Messages per day for thread length 38

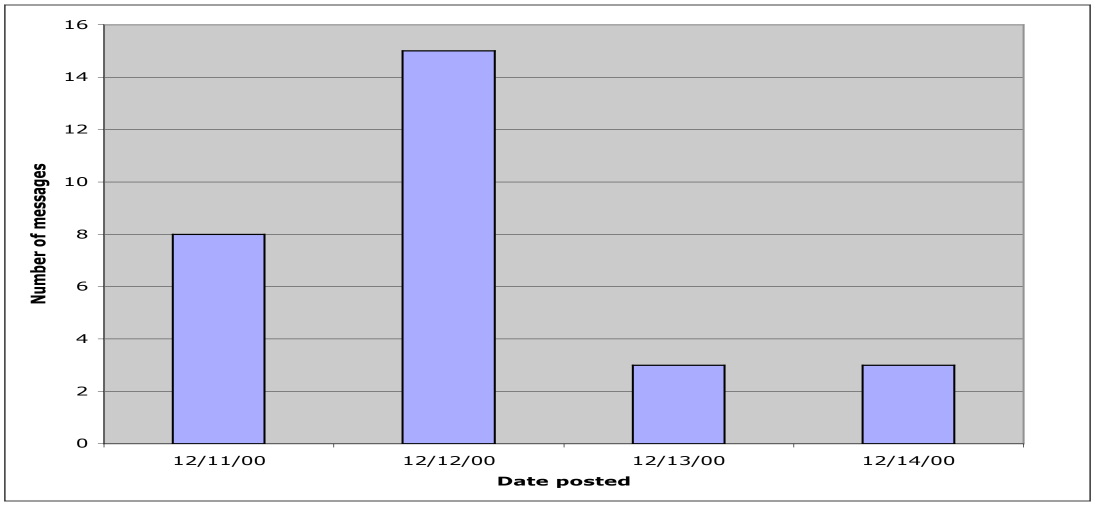

Figure 6 Messages per day for thread length 29

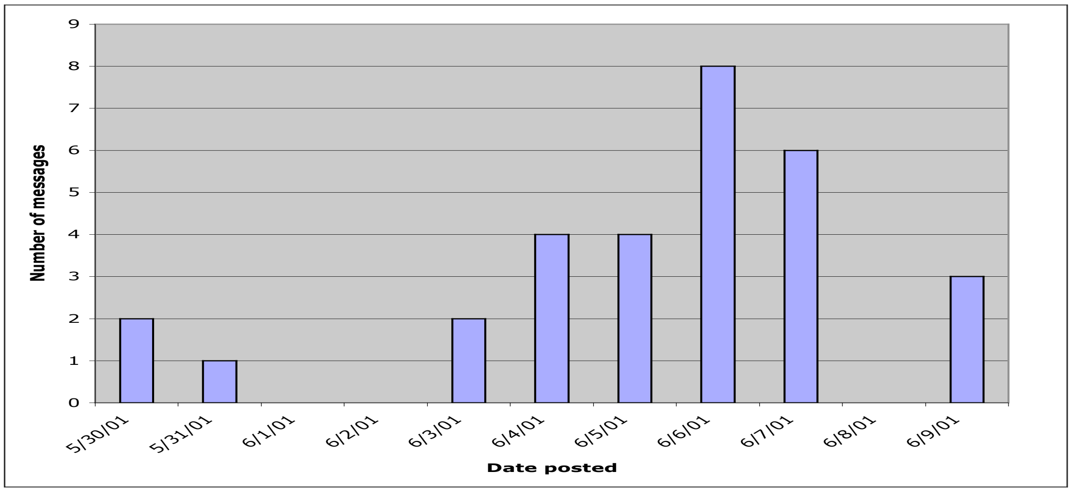

Figure 7 Messages per day for thread length 30

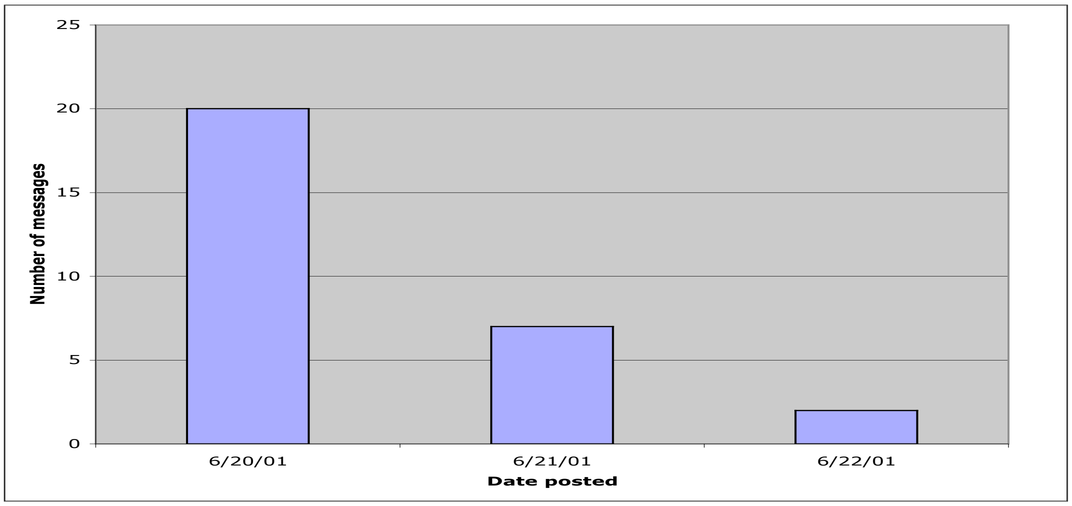

Figure 8 Messages per day for thread length 29

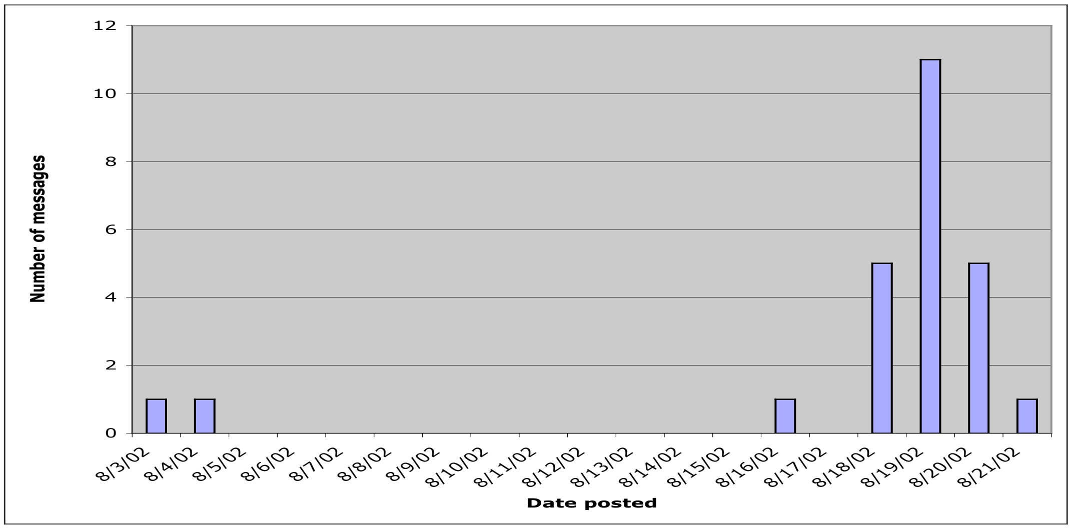

Figure 9 Messages per day for thread length 25

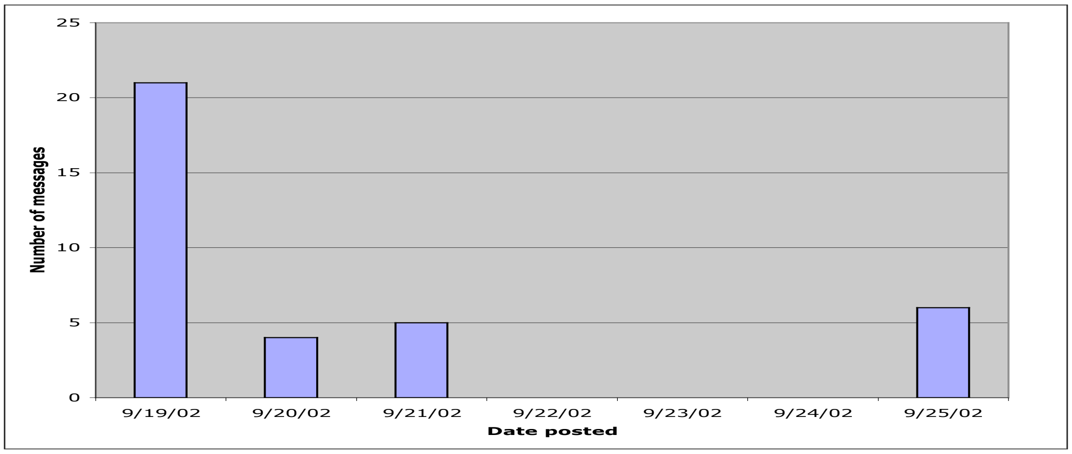

Figure 10 Messages per day for thread length 36

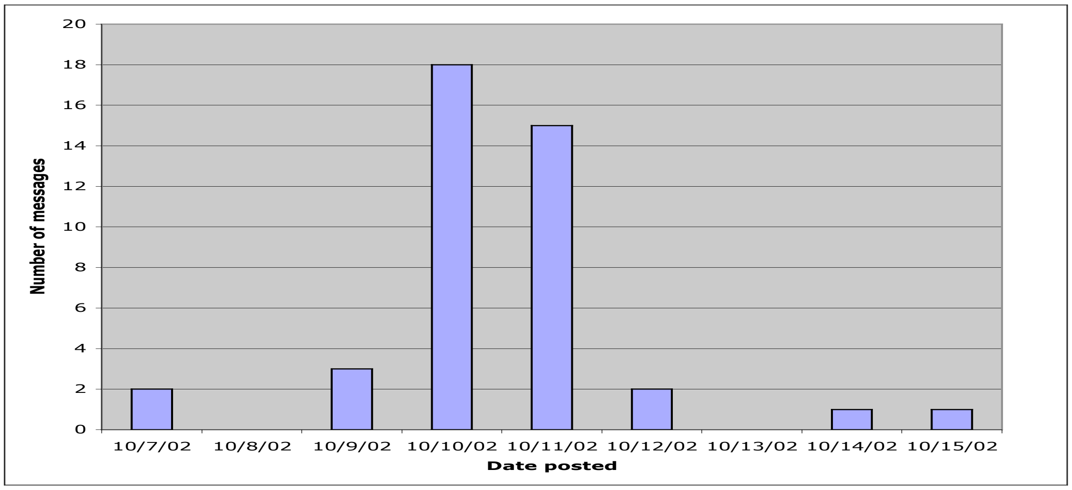

Figure 11 Messages per day for thread length 42

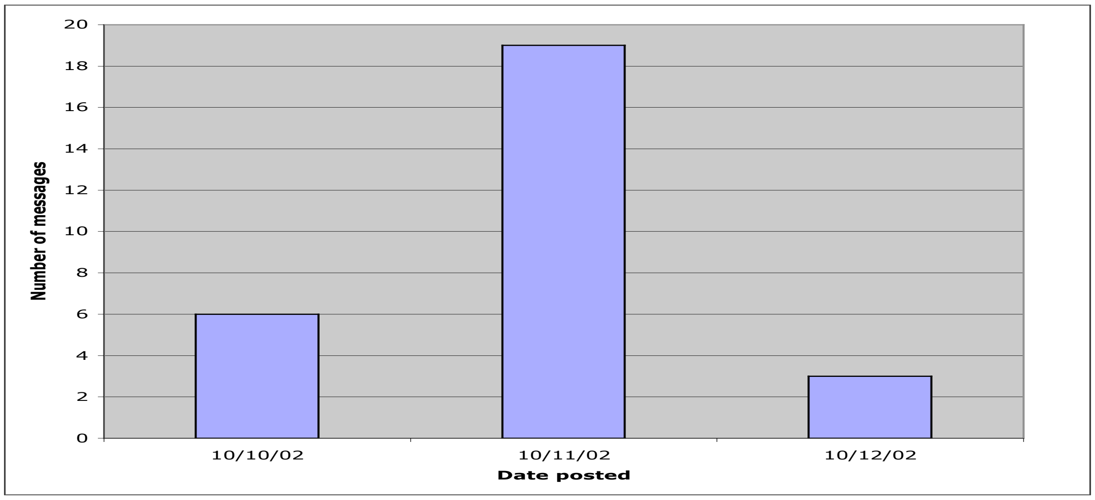

Figure 12 Messages per day for thread length 28

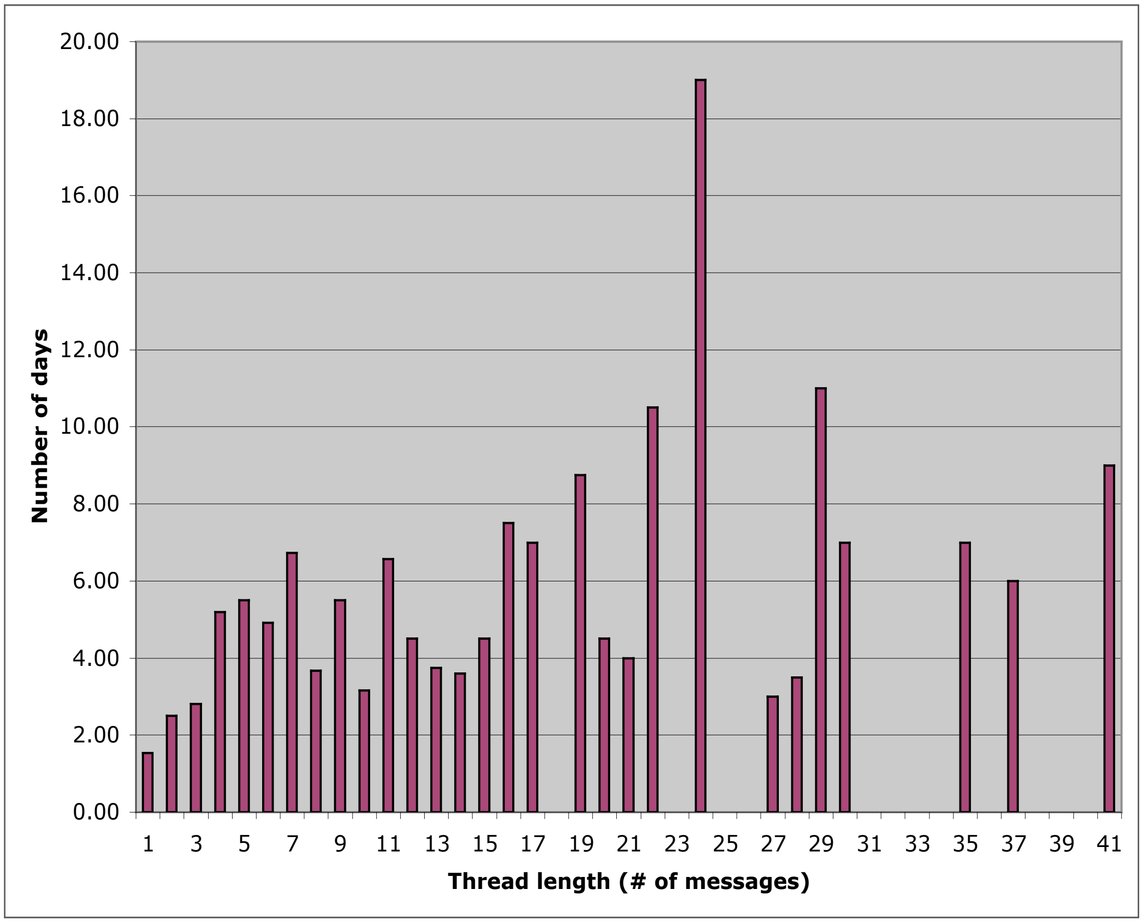

Figure 13 Thread persistence

3.2 Thread Selection Procedure

It was clear from the initial reading of the 10% data sample that there were many threads that would not contribute any meaningful information for the present study. I had to winnow the data somehow, so the question became, which threads constituted ‘wheat’ and which ‘chaff’? I needed to discard the threads that held little likelihood of containing useful contributions to the topic. To do so, I developed a heuristic based on two measures: one based on thread length and the other based on poster variety.

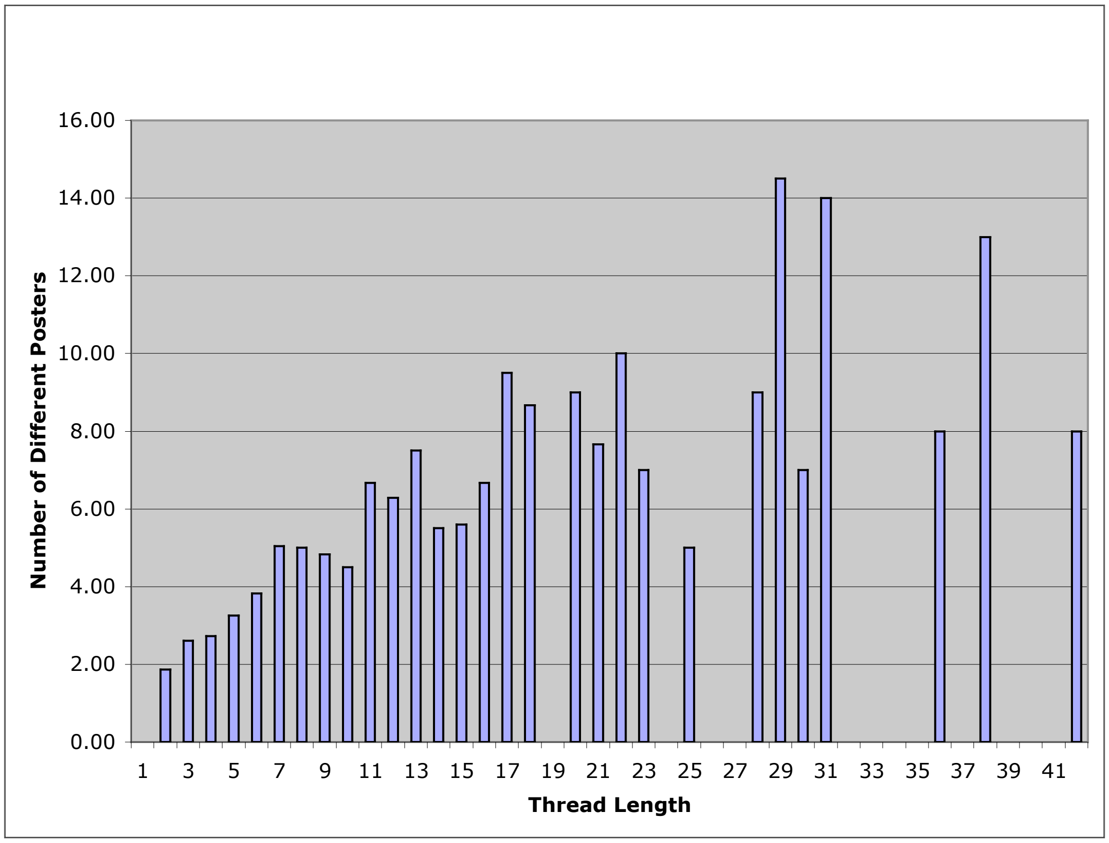

Figure 14 Average number of different posters by thread length

I began by assuming that threads of length one would automatically be considered ‘chaff’ since these engendered no responses and were evidently of little interest to the group. After removing these from consideration, I found that the average length of all threads was 5.6 messages. Next, I considered the number of *different* posters for each thread. We see in Figure 14 that in general, unsurprisingly, the number of different posters increases as the thread length increases. The average over all threads greater than length one is 3.4 different posters per thread. Combining these two measures, I narrowed the list of threads to those that were five or more messages long having at least three different posters. Out of the original 758 threads, this heuristic left 150 potentially ‘interesting’ threads (averaging almost 11 messages in length each).

My next step was to look at the subject headings of these 150 threads, and group them into categories (see Table 5). I operationalized these categories by grouping headings that had related terms. This process was remarkably straightforward since I had already familiarized myself with the range of topics in the threads from having read the initial random sample. The thread subject headings fell into place with little effort. In some cases, a heading would seem to belong to two groups, so I had to weigh the probable intent of the thread and place it somewhat arbitrarily. For example, one thread heading was “text compression as a learning tool”; did that belong in the Education grouping (‘learning tool’) or the Python/Computer Science grouping (‘text compression’)? In this case, I reasoned that the original poster was more likely to be discussing the pedagogical ramifications of the technique rather than the specifics of the technique itself, so I put it in the Education category. The Miscellaneous category consisted of topics that only a single thread addressed while the Unknown category consisted of topics that were not reflected in the language of the heading at all.

Eleven categories emerged. As I studied the headings, it became obvious that some of the categories clearly *were* appropriate to the present study (Education, CP4E, Math-related, Science-related, and Programming for fun),

Table 5 Thread categories

Thread Category	**# of threads**	**Sample Subject Heading**
Education	33	• My experience teaching Python • Socratic methods • Assigning homework
CP4E	6	• Python for non-programmers • Jpython and CP4E • CP4E VideoPython learning to teach/teaching to learn
Python/C.S.	28	• Python for AI • Analyzing algorithms • How to improve code performance
Math-related	15	• Where mathematics comes from • Algebra + Python • Calculating area of a surface plane on a spherical body
Science-related	5	• Periodic table • Modelling • Scientific Python 2.2
Editors	9	• In praise of VPython • Python Shell
Graphics-related	9	• Simple frame buffer module • Steps for making a fractal with Python
Programming for fun	7	• Programming for the fun of it • Cards n stuff • Python programming for kids
Python & other languages	8	• Lisp vs Scheme vs Python • Emulating Pascal input
Miscellaneous	8	• Would you use PythonCard • Database for a small network • Beyond 3D
Unknown	22	• And now for something completely different • Which way did the chicken cross the road • A fact on the ground

some clearly were *not* appropriate (Editors, Graphics, Python & other languages) and some were *possibly* appropriate (Python/Computer Science, Miscellaneous, and Unknown).

‘Appropriateness’ was defined pragmatically. For the purposes of the present study, I had to limit myself to a manageable number of topics. The topics I excluded certainly are important in their own right, and certainly have relevance to the issue of programming Python in an educational setting; however, these particular topics are also either too transient, too marginal, or at too low a level to be usefully considered in this study. For example, I decided not to consider comparisons of Python with other languages vis-à-vis their applicability as a first programming language. Many of the posters had experience with and opinions of, for example, Java, C/C++, Pascal, Perl, Scheme or Lisp, Logo, Basic, APL, and numerous other languages, but to adequately review and compare all of those languages would necessitate a separate project.

Another popular topic was that of graphics: how to handle, how to manipulate, how to usefully deploy in an educational environment. Again, the variety of modules and approaches and projects seemed too distracting from the primary focus of the present study. Besides, it felt as though the topic was straying away from computer literacy and into what we might call media literacy. Using analog or digital media, there is a technological skill, a literacy, a ‘media-savvyness’ to be acquired by creating and receiving such messages; e.g. being able to critically examine a graphic artifact to see how it achieves its effect on an audience. Susan Langer (1942) points toward this type of skill, at least in relation to visual graphics, when she writes:

> Visual forms—lines, colors, proportions, etc.—are just as capable of *articulation*, i.e. of complex combination, as words. But the laws that govern this sort of articulation are altogether different from the laws of syntax that govern language. The most radical difference is that *visual forms are not discursive*. They do not present their constituents successively, but simultaneously, so the relations determining a visual structure are grasped in one act of vision.

Such concerns are valid and important for a programming teacher who teaches how to create applications that also include graphical user interfaces, and there are many paths that can be followed to achieve this. I didn't feel that I could resolve the data into a group consensus, nor adequately present the arguments for and against different positions. I also felt resonance with the thought expressed by Python's creator, Guido van Rossum:

> I believe that programming itself, i.e. expressing the intended operation of a computer program, is best done through text. I guess that means that I believe that text (symbols) is more powerful than images. While a picture may be worth a thousand words, in my experience, most meaningful collections of a thousand words are hard to capture in a picture. (You can also see a program as a two-dimensional picture made up out of symbols.) (see also McCloud, 1999, p. 8)

A closely related issue was that of using an integrated development environment (IDE). There are an abundance of choices, and other sources of information would be better than trying to review or comment on the ones mentioned in the threads. Plus, Python comes with IDLE, an IDE that serves as a baseline environment from which to begin.

Finally, there were quite a few threads that concerned themselves with questions of Python syntax. Certainly syntax issues are important for anyone teaching or learning a computer language, but for the most part, the issues raised in these threads are either moot, due to changes in later versions of the language, or are simply out of the user's control – they are part of the language's design. I saw these discussions as interesting in their own right, but of little value to the goals of the present study.

So, of the eleven groupings based on the subject headings, threads from five were definitely in, threads from three were definitely out, and threads from three were possibly in, possibly out. However, I felt that examining the subject headings alone was insufficient, especially the Miscellaneous and Unknown category threads. I needed some way to 'glance' inside each of these threads to distinguish content that was likely to be salient from that which was superfluous. There were two techniques that I was aware of: PhraseRate and Pathfinder text graphs. Describing each will take us on a bit of a 'scenic ride' before we come back to the data itself.

3.3 PhraseRate

PhraseRate[29] is a keyphrase extraction tool for use with webpages. Unlike other similar tools, it doesn't rely on an existing corpus to calibrate its results, but applies bottom-up heuristics to determine the key phrases of a text. Humphreys (2000) says: "PhraseRate…introduces a novel keyphrase

[29] http://infomine.ucr.edu/download/PhraseRate/

extraction heuristic for web pages which requires no training, but instead is based on the assumption that most well written webpages 'suggest' keyphrases based on their internal structure." He further explicates how that 'internal structure' is evaluated:

> 1. The program should not depend on training: There is a large variety of topics and writing styles among webpages as well as emerging topics, words, and phrases in research. It would be difficult to provide an accurate and flexible phrase extractor in an open ended environment derived from a closed set of training examples. So a design based more on the intrinsic properties of the webpages was desired. (If training was to be used, it should be for higher level parameter tuning only.)
>
> 2. Candidates for phrases should be from 2 to 5 significant words long and should not cross various blocking structures such as punctuation and HTML formating blocks. Longer phrases supporting keywords are rather rare and were considered to be an excessive computational expenditure.
>
> 3. Extraction should be accomplished by rating and filtering.
>
> 4. The rating of a phrase should be in part dependent on the number of instances in the document: After evaluating individual phrase instances, we fold in their weights to form a global evaluation. We want to emphasize repeated phrases subject to the other conditions.
>
> 5. The rating of a phrase instance should be in part dependent on the strength (weights) of its constituent words.
>
> - Emphasize uniformly good sequences of words: A highly rated word accompanied by a pair of weak words does not form a distinguished sequence. For the group as a whole to be considered, all the individuals should be distinguished to a fair degree.
> - Phrase rating length coordination: We don't want to consider long phrases of trivial words to be somehow better than a good pair of words. On the other hand, a string of uniformly emphasized words is probably a good candidate, so we want to encourage longer strings somewhat.
>
> 6. The weight of an instance of a word was to be determined by:

- o HTML mark up, in a nested fashion,
- o location from the start, up to an "introduction-limit" distance. That is, an emphasis function that decayed up to a fixed level was to magnify the importance of introductory words.
- o capitalization (modestly),

7. The global (document-wise) weight of a word was taken to be the accumulation of the weight of all its instances, and hence be additive.

8. Last, we consider the following reasonable phrase ranking scale invariance conditions:

Given the additive nature of rated word occurrences, if the word weights are rescaled by a constant, then the ordering of the rated phrases should remain unchanged.

This is satisfied by the natural *Homogeneity property*: If all the weighting values of the component words are rescaled by a constant c > 0, then the rating of the phrase are likewise scaled by c.

For each webpage that is submitted to PhraseRate, it returns a short list of 10-20 keyphrases, rank ordered by importance, representing, according to its heuristic, the 'essence' of the text. I installed PhraseRate and then wrote two scripts:

- A Filemaker script that exported each thread as a single webpage
- A Python script that captured and saved the PhraseRate results for each thread/webpage

These results were then imported into the Threads database so that by quick inspection I might determine the topics covered within a thread. However, before doing this, I felt it would be worthwhile to check PhraseRate results against another technique I had employed elsewhere in my research (Miller, 1995). Instead of a short linear list of keyphrases, Text Graphs produce a two-dimensional graph akin to a concept map using the Pathfinder algorithm.

3.4 Pathfinder text graphs

A brief summary of the Pathfinder algorithm is offered by Johnson (1991):

> The Pathfinder scaling algorithm transforms a proximity matrix into a network structure in which each object is represented by a node in the network and the relatedness between objects is depicted by how closely they are linked. The method searches through the nodes of the network to find the closest indirect path between objects. A direct link between two nodes is added only if the closest indirect path between the two nodes is greater than the proximity value for that pair of objects.

For a more complete discussion, see Schvaneveldt (1990). For an example of using the algorithm in educational research, see Berger and Jones (1995). Prof. Berger is also the person who introduced me to the Pathfinder algorithm and encouraged me to explore its use for creating text graphs.

In a text graph each node is a unique term (word or recurring phrase) of a given text. The proximity matrix is generated according to the algorithm described below.

There are two stages to processing a given text: data preprocessing, and relationship matrix generation.

Data Preprocessing

1. First, remove all stop words
2. Then, stem all remaining words
3. Next, combine frequent word sequences into phrases

This leaves an ordered list of terms (words and phrases) from which we generate a few basic statistics to work with while generating the matrix:

4. Generate a list (called 'uniqlist') of all terms that occur at least once in the document (after the first three processing steps)
5. Associate with each term in the uniqlist a list of (numerical) positions corresponding to its occurrences in the modified source text (e.g. 'teach' → [34,98,156,332,450])
6. If desired, limit the number of nodes in the graph to the top 25, 50, 75 or 100 or so terms having the greatest number of occurrences in the text.

Relationship Matrix Generation

Each cell of the relationship matrix holds a value representing the strength of the relationship between two terms. This value is the sum of two measures: one of frequency, and one of proximity. Support for the use of proximity and frequency comes from Osgood's (1963) 'semantic differential' where we can say that proximity serves as a proxy for 'potency,' and frequency for 'activity.' (The third differential, 'evaluation,' appears not to be measured in the present version of text graphs.) Remarkably, Osgood's method for measuring the semantic differential is quite similar to the Pathfinder procedure for measuring node closeness.

These values are arrived at by two sets of calculations on the numbers in the list associated with each term. The first measure, frequency, is, ostensibly, a simple product of the length of the list of positions for each term. We shall see that this needs to be normalized, but the product is a first approximation. The second measure, proximity, is the sum of reciprocals of all the pair-wise differences between the positions of one term with the positions of the other.

Let's consider an illustrative example. If one of the terms is ('teach', [34,98,156,332,450]) and the other is ('learn', [32,76,211,408]) where the numbers represent positions in the text where the word occurs we see that there are 5 x 4 = 20 occurrences of the 'teach – learn' pair since 'teach' occurs 5 times and learn occurs 4 times in the text. Therefore, the measure of frequency will contribute a score of 20 towards the final value in the corresponding cell of the relationship matrix. However, we can modify this value by considering how closely together the two terms occur to each other (proximity measure). If the two terms are close, then the difference in their positions will be small, and the reciprocal of that difference will be (relatively) large. If the two terms are far apart, then the difference in the positions will be large, and the reciprocal of that difference will be (relatively) small. Thus the sum of all possible reciprocal differences between the two terms is a measure of proximity and is added to the frequency score to arrive at a final value for that relationship. In this example, the proximity measure would be:

1/(|34-32|) + 1/(|34-76|) + 1/(|34-211|) + 1/(|34-408|) + 1/(|98-32|) + 1/(|98-76|) + 1/(|98-211|) + 1/(|98-408|) + 1/(|156-32|) + 1/(|156-76|) + 1/(|156-211|) + 1/(|156-408|) + 1/(|332-32|) + 1/(|332-76|) + 1/(|332-211|) + 1/(|332-408|) + 1/(|450-32|) + 1/(|450-76|) + 1/(|450-211|) + 1/(|450-408|) =

1/2 + 1/42 + 1/177 + 1/374 + 1/66 + 1/22 + 1/77 + 1/310 + 1/124 + 1/80 + 1/55 + 1/252 + 1/300 + 1/256 + 1/121 + 1/76 + 1/418 + 1/374 + 1/239 + 1/42 =

.5 + .024 + .006 + .003 + .015 + .045 + .013 + .003 + .008 + .013 + .018 + .004 + .003 + .004 + .008 + .013 + .002 + .003 + .004 + .024 =

.709

This results in a total value of 20 + .709 = 20.709. Unfortunately, the contribution made by the frequency measure is disproportionately large relative to the contribution made by the proximity measure. Suppose we would like each measure to contribute equally to the final values of the relationship matrix. We have two choices: either magnify the effect of the proximity measure, or reduce the effect of the frequency measure. We shall choose to reduce the effect of the frequency measure.

What is the contribution made by each of the measures? We can calculate this value for each measure from the data. Our data structure is a Python dictionary where the keys are the terms that will become nodes in the graph, and the values are a list of the positions in the document where the term is found. Let us construct a simple example using three terms, 'a', 'b', 'c' randomly distributed in positions 1-9:

```
B A C B A C A B A
DATA = {'A':[2,5,7,9], 'B':[1,4,8], 'C':[3,6]}
```

We can calculate the total frequency measure in this data set by finding all the possible pairs of 'a', 'b' and 'c' (excluding self pairings). To do this we simply multiply the number of positions of each term by the number of positions of the other term (see Table 6).

Table 6 Frequency measure for example data

	a	**b**	**c**
a	0	4•3=**12**	4•2=**8**
b		0	3•2=**6**
c			0

This give us a total contribution of 12 + 8 + 6 = 26 by the frequency measure. To do this in Python, we can write:

```
terms = ['a','b','c']
pos = {'a': [2, 5, 7, 9], 'b': [1, 4, 8], 'c': [3, 6]}
freq_sum = 0
num_nodes = len(terms)
for eachterm in terms:
    next_node = terms.index(eachterm) + 1
    if next_node < num_nodes:
        rest_nodes = num_nodes - next_node
        for i in range(rest_nodes):
            freq_sum += len(pos[term]) * len(pos[terms[next_node]])
            next_node += 1
    else:
        break
print freq_sum
```

```
→ 26
```

We can calculate the total proximity measure in this data set by a different algorithm. This time, for each term pair occurrence, we calculate the sum of the reciprocals of the differences. So, for example, to get the proximity measure for the 'bc' pair, we would sum all the values bolded in Table 7:

Table 7 Proximity measures for example data

	1(b)	2(a)	3(c)	4(b)	5(a)	6(c)	7(a)	8(b)	9(a)
1(b)	0	1	**1/2**	~~1/3~~	1/4	**1/5**	1/6	~~1/7~~	1/8
2(a)		0	1	1/2	~~1/3~~	1/4	~~1/5~~	1/6	~~1/7~~
3(c)			0	**1**	1/2	~~1/3~~	1/4	**1/5**	1/6
4(b)				0	1	**1/2**	1/3	~~1/4~~	1/5
5(a)					0	1	~~1/2~~	1/3	~~1/4~~
6(c)						0	1	**1/2**	1/3
7(a)							0	1	~~1/2~~
8(b)								0	1
9(a)									0

The total proximity measure is the sum of all values in the table above that aren't crossed out (crossed out values measure the proximity of self-same terms). One way to implement the code is as follows:

```
terms = ['a','b','c']
pos = {'a': [2, 5, 7, 9], 'b': [1, 4, 8], 'c': [3, 6]}
prox_sum = 0
num_nodes = len(terms)
for term in terms:
    next_node = terms.index(term) + 1
    if next_node < num_nodes:
        rest_nodes = num_nodes - next_node
        for i in range(rest_nodes):
            for j in pos[term]:
                for k in pos[terms[next_node]]:
                    prox_sum += 1.0 / abs(j-k)
            next_node += 1
    else:
        break
print prox_sum
```

```
→ 13.45
```

In this simple and contrived example, the discrepancy between the frequency (26) and proximity (13.45) measures isn't nearly as pronounced as in the earlier example (20 v. .709); however, this is due simply to the short length of the sample: 9 positions. With normal length passages of hundreds or thousands of positions, the frequency measure quickly overwhelms the proximity one.

So, to normalize the contributions each measure makes to each relationship matrix value, we simply calculate the proportion between the two measures and multiply each corresponding frequency value to reduce its effect. In this case, 13.45/26 = .517, so multiplying each frequency value by this amount yields the values in Table 8:

Table 8 Normalized frequency measures for example data

	a	**b**	**c**
a	0	6.208	4.138
b		0	3.104
c			0

This yields a sum of 13.45, which shows that the proximity and frequency measures contribute equally to the final values of the relationship matrix.

We can now combine these algorithms and generate a script to perform all of the specified steps on an input file, in this case the text of a thread. This results in a file containing the values of a Pathfinder proximity matrix that can then be fed to a commercial program (KNOT, or Knowledge Network Organizing Tool, or its [now discontinued] Macintosh version, MacKNOT) which will generate a network from the data provided.

3.5 Thread Selection Results

So now there are two ways to evaluate a thread to get a summary of its contents: a PhraseRate list of keyphrases, and a Pathfinder-based text graph. As an example of the results from the two techniques, the text graph of one thread, 'Programming for fun', is shown in Figure 15 while the PhraseRate results of the same thread are shown in Table 9.

We see by inspection that one of the two central nodes in the graph, 'mathematics' is the same as the top ranked keyphrase. And one of the links

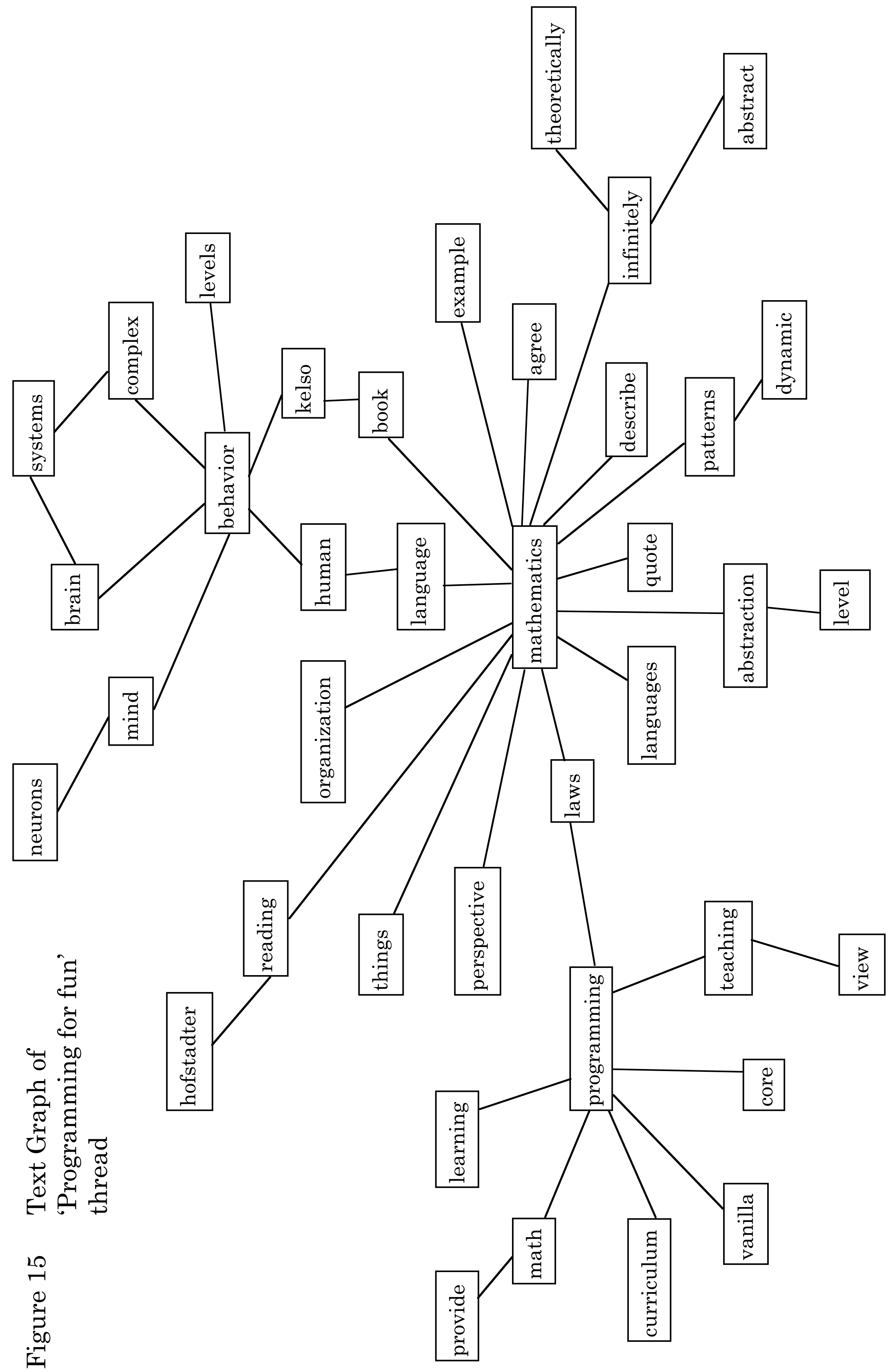

Figure 15 Text Graph of 'Programming for fun' thread

Table 9   PhraseRate keyphrases for sample thread

* mathematics
* infinitely abstract
quote by chris langreiter
nice quote by chris langreiter
* brain and behavior
* programming to math
* human behavior
* theoretically infinite
applying programming to math
joys of applying programming to math
* dynamic patterns
* mathematics and language
quote except
vanilla author
subsequent abstraction

(*phrases found as nodes or connected nodes in Fig. 15)

off of ‘mathematics’ is ‘infinitely’ $\rightarrow$ ‘abstract’, corresponding to the second highest ranked keyphrase. Other similarities can also be found, for example, ‘brain and behavior’ and ‘human behavior’ can be found as links in the text graph, as can ‘dynamic patterns’ and ‘mathematics and language’. This process of comparing the two result sets was repeated for a total of ten threads, with comparable correlations. Thus, it appears that the results of the two techniques begin to converge in their summarizing function. Since I already had confidence in the text graphing technique (Walberg, 2002), I felt that I could use the PhraseRate results to assist in evaluating the 150 candidate threads. (Unfortunately, the text graphing technique is significantly more labor-intensive than the PhraseRate technique at present. Even though the text graph results are somewhat richer and more visually appealing, time constraints precluded their extensive use in the present study.)

Given that the PhraseRate-generated key phrase results appeared to offer a usable summary of the content of each of the threads, I re-examined the 150 threads chosen earlier that were at least five messages long consisting of at least three different posters. This included the threads from the provisionally excluded categories (Editors, Graphics, Python & other languages) to verify that none of the threads was particularly appropriate, the included categories (Education, CP4E, Math-related, Science-related, and Programming for fun) to verify that all of the threads *were* appropriate, and especially the *possible* categories (Python/Computer Science, Miscellaneous, and Unknown) where some threads were likely to be appropriate and some not, but where the subject heading alone was insufficient. Perhaps the most interesting category with respect to this procedure was the Python/Computer Science one. It originally had 28 threads, but after consulting the key phrases, I was able to discard almost half of them. This was primarily due to the fact that the key phrases indicated whether or not the thread was likely to concern itself with low-level syntax issues, or graphics issues, or esoteric computer science issues or other issues that I had already determined fell outside my purview. Likewise, the key phrases for the other two categories, Miscellaneous and Unknown, allowed me to keep most of those threads, as they indicated that their contents *were* likely relevant. This final winnowing process resulted in the selection of 108 threads consisting of 1228 messages; their subject headings are shown in Appendix A.

3.6 Data Analysis

At this point, I would like to describe how the kind of research I conducted on the data is situated with respect to similar research methods. The messages and threads were subjected to a computer-mediated content analysis (CMDA). CMDA is not a theory or single method, but an approach to analyzing the data:

> CMDA applies methods adapted from language-focused disciplines such as linguistics, communication, and rhetoric to the analysis of computer-mediated communication (Herring, 2001). It may be supplemented by surveys, interviews, ethnographic observation, or other methods; it may involve qualitative or quantitative analysis; but what defines CMDA at its core is the analysis of logs of verbal interaction (characters, words, utterances, messages, exchanges, threads, archives, etc.). In the broadest sense, any analysis of online behavior that is grounded in empirical, textual observations is computer-mediated discourse analysis. (Herring, 2004)

Herring goes on to emphasize that CMDA "is not a single method but rather a set of methods from which the researcher selects those best suited to her data and research questions." CMDA can apply to any of four different levels of language: structure, meaning, interaction, and social behavior. This study focuses on the 'meaning' level of language, including "words, utterances … and larger functional units (e.g., 'macrosegments')" (Herring, 2003).

Herring says that there are three theoretical assumptions underlying CMDA:

- discourse exhibits recurrent patterns ("…patterns in discourse that are demonstrably present, but that may not be immediately obvious to the casual observer or to the discourse participants themselves.")
- discourse involves speaker choices ("These choices are not conditioned by purely linguistic considerations, but rather reflect cognitive and social factors.")
- computer-mediated discourse may be, but is not inevitably, shaped by the technological features of computer-mediated communication systems

Of these, the first two assumptions are the most relevant for the present study since I am not focusing on the effect of the medium on the message but simply the messages themselves.

CMDA encompasses many research paradigms, including conversation analysis, interactional sociolinguistics, critical discourse analysis, semantics, pragmatics, text analysis, and others. For the purposes of the present study a simple semantic content analysis was chosen. Lemke (1998) emphasizes these 'recurrent patterns' in his discussion of semantic content analysis:

> How can we characterise what a text says about its topics, or even what its topics are, better or more concisely than the text does itself? This is possible only to the extent that the text repeats the same basic semantic patterns, makes the same basic kinds of connections among the same basic processes and entities again and again. It happens, in our culture, and probably in most, that not only do we repeat these thematic patterns, or formations, again and again in each text, merely embroidering on the details, we also do so from one text or discourse event to another. … Thematic analysis, correspondingly, must be done by hand, but it enables you to see that the same concept or relationship may be expressed by many different verbal forms and grammatical constructions, and to exclude cases where the form is right but the meaning in context is not.

Each set of threads in a category was given a holistic reading where I noted the salient issues as they arose. 'Salience' was subjectively determined by a combination of my teaching experience (in English as a Second

Language) and my programming experience (including Python). As the readings progressed, recurrent patterns germane to the topic were identified and assiduously cultivated as they occurred in any of the messages of the threads in a given category I then reread the threads as I addressed these patterns searching for evidence in support of, and contrary to, each issue's essence. Next, we turn to the harvest of that cultivation.

CHAPTER 4

RESULTS

Most important of all, however, was the lesson that I learnt from the study of people who create something in their lives out of nothing—we termed them alchemists. They proved to me that you can learn anything if you really want to. Passion was what drove these people, passion for their product or their cause. If you care enough you will find out what you need to know and chase the source of the knowledge or the skill. Or you will experiment and not worry if the experiment goes wrong. The alchemists never spoke of failures or mistakes but only of learning experiments. Passion as the secret of learning is an odd solution to propose, but I believe that it works at all levels and all ages.

—*Charles Handy*, The Elephant and the Flea

4.0 Introduction

The sections of this chapter correspond to the categories identified in the previous chapter: CP4E, Education, Python and Computer Science, Math-related, Science-related, and Programming for fun; the two categories Miscellaneous and Unknown are grouped together last. Each category is introduced with the key phrases drawn from the subject lines and the PhraseRate results that characterize the category. Subsections within each section correspond to the recurrent patterns identified throughout that category. A conclusions section is found at the end of each category.

Online discussions are usually informal, and posters are not usually punctilious about their textual communications. In the many quotes in this chapter, I have taken the liberty of 'lightly cleaning' the messages with respect to spelling, grammar and punctuation in order to enhance readability while retaining the author's style and tone. I have occasionally inserted an explanatory phrase in square brackets: [ ]. Each posting has two capital initials at the end to indicate the first and last name of the poster, followed by a thread code (for example, 'ED-5') that refers to the thread the message comes from and which is indexed in Appendix A, and finally an ordinal number (like '3rd') indicating which position in the thread the message occupies. Where I refer to the 'OP', it stands for the thread's Original Poster, the person who started the thread.

Finally, at the end of this chapter, I review the results of the methods and procedures used in processing the data.

4.1 Computer Programming for Everybody

This category concerns itself with threads that had either 'CP4E' or 'non-programmer' in their subject line. Computer programming for everybody is an ambitious goal in the context of mass education. The term should probably be read as 'computer programming for everybody (who interacts with a computing machine)' but even this should be taken with a grain of salt. (Recall the discussion of 'multiple literacies' and 'multiple intelligences' from Chapter Two.) If programming is to computer literacy what writing is to print literacy, then we know that not everyone who is ‘literate’ can write and not all who write can write well. Similarly, we should expect that many who learn to program will not program well, or at all, after their efforts. Nevertheless, the goal of CP4E is long-term rather than short-term, and its adherents expect that programming will become much more mainstream (like writing) than esoteric.

Programming and school subjects

The only major, recurring topic in these threads was the relationship between learning to program and school subject matter. Many posters emphasized that it was highly beneficial to teach Python in the context of another subject:

> I think the difference between ‘using Python to do X’ and ‘doing X to teach Python’ really needs to remain front and center in this SIG. –KU, CPE-1, 4th

> I think CP4E needs to instead emphasize the interaction, i.e. you teach programming by allowing the student to explore a subject area with a programming language. –SM, CPE-1, 13th

The latter poster offered a linguistic analogy along with a short observation that is worth repeating:

> To me the language/subject dichotomy is a noun/verb, object/action kind of thing. Python is the verb that acts on the subject being explored (the noun.) You need to start with simple nouns and verbs and work up to complex ones. The complexities of both need to go in parallel, especially when you are trying to draw in the student in a single course or at a single point of contact.
>
> My late father-in-law was an impressive self-taught linguist. He started with German, Latin and Greek in school and then taught himself to be fluent in French, Spanish, Russian and Romanian and then to be able to

> read technical work and poetry (!) in many other languages. He usually learned a new language because there was something he wanted to read in that language and he didn't trust the translators.
>
> He was obviously gifted (his rote memory was phenomenal) but his methodology was interesting. He said that instead of finding a language tutorial/course etc. and learning to say simple but useless things he started by trying to read a subject that he knew well, the more complex the better as long as he already understood the complexities of the subject. He said that his understanding of the subject gave him clues that help him learn the language as it is used.
>
> The same distinction shows up in trying to teach English writing skills. It is much easier to teach students to write about something when they have something to say. –SM, CPE-1, 13th

We see here a strong analogy with the traditional English class where it not only is taught in a class itself but is also employed in some fashion in most other academic classes. That is, students are expected to write not only in English class, but also in Social Studies, Science, History, and Humanities classes. Indeed, this is often one way teachers evaluate the degree to which their students have assimilated the subject's material. Similarly, programming can certainly be taught in a Programming class (like English is taught in an English class), but it should be employed in the other academic areas as well: Mathematics and Science, of course, but also History, Humanities, Social Studies and English. And then programming assignments could be used by teachers as another, probably optional, method of evaluating students' knowledge of the subject matter.

The question of how to 'bootstrap' the learning to program process appears in several threads. One, for example, begins by noting the value of learning from 'the masters':

> Personally I would love to spend a couple of hours sitting next to an experienced Python programmer who could show for example the process of putting together a program, testing it, shaping, how they react to error messages, how they sculpt the code, how they might look at the problem a couple of ways, how they use the tools, how many windows [are] open, what they look at; the real-world rhythm of a few hours in the life of a python program; even better if this was intelligently structured and re-playable with examples. It would be nice to have a human being talk and type me through some code.
>
> Yes there is no substitute for hands-on learning/doing. But there is no substitute for great teachers either! And since we don't have the joys of python class in school or at our corner adult-education center, we go to the web and read and download and hack/explore till hopefully it clicks. Some experienced folks can transfer their previous learning fast, but for others who may have no experience or come from another background, the first steps are very important. …
>
> Lord I would love a Python video from the masters at work and play. In any field there are rare few people who _really_ understand it, and even rarer are the ones who can teach it. –JC, CPE-4, 1st

Of course, this presupposes that the masters are coding an application or program in *some* domain, not *in vacuo*:

> Sometimes I think what bogs people down is forsaking any knowledge domain and trying to learn the 'programming language' as the generic 'thing to know'.
>
> More useful, I think, is to bring in a knowledge domain (some topic in mathematics, a graphical challenge, some real problem needing a solution), and then learn the language in tandem with that knowledge. –KC, CPE-4, 3rd

Quite a few other posters echoed this sentiment, that the value of learning to program is not intrinsic to programming itself, but that its value is revealed in the context of learning other subjects. Even the subject line of the thread these messages come from contains a related mantra: "learning to teach/teaching to learn"! Or, for our purposes, we can say: "learning to program/programming to learn”. The OP of this thread posted again several messages later and offered an analogy with music education:

> But as part of the big experiment where I am part observer, part guinea pig, part instigator, I do feel that Python is very good language for learning computer programming hands-on. Learning *through* python, rather than learning python, if you see what I mean. Of course I want to learn python. But I am interested to learn it in a way which goes beyond that. …
>
> Getting one's head around abstract structures is easier for many if there are tangible metaphors like sounds or images which provide rapid feedback. So many computer programming tasks or examples are based on real-world problem solving, but in my opinion are not so good for beginners: piping files around, making list boxes and yet another boring little application is ok, but not very inspiring. I suspect music and graphics for many are more motivating and encourage both abstraction and real-world problem solving. This was part of the insightful premise of LOGO after all.
>
> [H]mm…Let's see, I am looking for metaphor... ok. You can learn a lot about music sitting with a good piano teacher; you will learn some piano, but the right teacher will use the piano to teach music structure, composition, improvisation, style, etc. As I understand it this is what underlies the CP4E idea and others. –JC, CPE-4, 6th (emphasis added)

Conclusion of computer programming for everybody category

So, learning the Python language *per se* is more of a means than a goal for many of the posters on the list. I draw this conclusion based on reading several assertions of similar ilk, with no dissenting opinions proffered. By identifying goals in other classes, programming can be taught as a means of attaining those goals. By attaining those goals, programming is learned and can be used to attain further goals, and, at the same time, furthering one’s programming expertise. This dynamic is similar to way English works in both English classes and other academic classes. It is also remarkably similar to the core notion of the *trivium* that was introduced in Chapter Two: "education concentrated on first forging and learning to handle the tools of learning, using whatever subject came handy as a piece of material on which to doodle until the use of the tool became second nature" (Sayers, 1948). The collective agreement of these posters seemed to be that learning to program as a means of enhancing the attainment of academic goals in other subject areas was a recommended way of beginning to learn programming.

4.2 Education

The threads in this category were placed here by virtue of the following list of terms, one or more of which occurred either in the subject line or in the PhraseRate results:

- Education, elementary education, middle-school, high school, K-12
- Assignments, assigning homework, programming assignments
- Teacher, teaching
- Curriculum, lesson plan
- Pedagogy
- Socratic
- Beginner
- Course
- Learning
- Tutorial
- Students

In many ways, the threads concerned with education touched the heart of this dissertation's topic. Many issues were raised, and many of those were simply raised without coming to any definite conclusion. Still other issues generated opposing viewpoints, thoughtfully presented. The perspectives of the posters ranged from "voice-of-experience" pragmatic to "flight-of-fancy" idealistic, but all appeared united by both the desire to bring programming to the masses, and overall admiration for the syntactic adroitness of the Python language.

This category was by far the largest, both in terms of the number of threads (33) and the number of messages (317). The issues covered here are, in order of presentation, a) where and how programming can fit into a school's curriculum, b) infrastructural concerns such as curriculum materials and Python's modes of operation, and c) an extensive section on teaching methodologies. Each of these subsections has its own set of sub-subsections, illustrating the range of educational issues that were raised by the posters.

4.2a Where does programming 'fit' into a curriculum?

One of the more persistent concerns of the group was integrating Python programming into a school's curriculum. There appear to be three (two major and one minor) paths to follow, with one of them proving to be exceedingly difficult to navigate. The first path is to integrate Python into existing classes. The second path is to create a Python programming class of its own. And the third path is to create an after-school computer club. These paths are not meant to be mutually exclusive, but rather, synergistically coexisting to maximize the opportunity of promoting computer literacy among the student populace.

Integrating Python into existing classrooms

One of the most visible obstacles to widespread use of Python in the classroom is the Advanced Placement test for Computer Science since it is only offered in the Java language. Thus, all AP classes use that language either exclusively or primarily. However, it was generally agreed on the list that educators were *not* seeking to convert the College Board to (at least) allow the Python language into its AP test; they conceded, for the time being, that Computer Science majors needed to know the language of the test. Instead, since the audience of CP4E was 'everyone' other than CS majors, it was deemed wise to introduce Python in a different context, for example, Math class.

In many ways, this would seem to be logical, since computing depends so strongly on mathematical foundations. Nevertheless, it is not easy to accomplish. Let me illustrate with an admittedly long exchange from one of the threads. There was an unusually sobering posting made by a high school math teacher who, although she was strongly in favor of the goals of the group and had made a valiant effort to implement reform in both her classes and in her department, reported the difficulties she faced:

> But what really shocked me was the experience I had today with my colleagues when I tried to show it [using Python in the math classroom] to them as something with great potential for help with understanding algebra. I was just showing them how you could use it as something better than a hand calculator for doing such things as solving equations by searching, which I think is a really good idea for keeping students in touch with the base meaning of solving equations. And one of my colleagues practically expressed horror and said that this would totally put him off of mathematics. And others expressed similar opinions. I

> remember the first time I saw you [KU] write about how you could define a function in the console mode, e.g. ‘def f(x): return x**2’, and then proceed to evaluate it from a composition function, I immediately thought that was just such a great way for students to see such things right in front of their eyes, for it to no longer be abstract. But he seemed to think it would take him hours to master the syntax of it and for the students it would be just one more thing to learn when they were already afraid of the subject. And partly from some of the reactions I have gotten from students, it seems that he is likely to be right. For him the fact that it there is a ‘:’ and a ‘return’ instead of just an equal sign was totally daunting and the ‘**’ [exponentiation] makes it even worse.
>
> So my question for you is have you found this kind of Python anxiety, and if so how have you dealt with it? –SW, ED-27, 1st

This perfectly illustrates the resistance many math faculty (and faculty from other subjects too) exhibit when faced with the prospect of integrating computing into their curricula. I imagine this poster may have been hoping for some simple solution to counteract these opinions, but there was only one substantive reply to her concerns:

> Re your specific questions, I have to confess up front that I have very limited personal experience trying to phase in Python in the ways I suggest. I would *like* to have more of these opportunities, but the fact is that I am not now a classroom teacher (I used to be, but that was many years ago). When I do get together in a room to present Python or Python-related topics, chances are they're already sold on the program -- I'm mostly just preaching to the choir as it were. …
>
> But with the math teachers, I think the reaction is coming from a different place. If they're horrified by the colon and the return keyword in Python, they'll be even more horrified by all the syntactical clutter of *any* computer language -- even Mathematica, which has gone a long way to accommodate traditional math notation. But as Wolfram points out, traditional notation is ambiguous. Does s(x-1) mean the function s, applied to x-1, or does it mean s times x-1? … Whereas humans can tolerate a lot of ambiguity, owing to sensitivity to context, computers cannot. And so in a lot of ways, computers force *more* precision on a notation.

With math educators, it does no good to talk about Python's power and sophistication vis-a-vis C++ and Java. Their beef is with the whole idea of diluting the purity of their discipline with material from an alien discipline, i.e. computer science and/or engineering. To start using a computer language in an early math curriculum looks like the harbinger of nothing good: it means math will become mixed up with all kinds incompatible grammars which come and go, vs. the staying power of a more stable, core notation. Plus if computer languages invade the math classroom, then teachers will be forced to learn programming, which many are loathe to take up. The hand held graphing calculator is as far into computing technology as these teachers want to go, and even there, their programmability is often ignored.

But not all math educators are on the same page here. Many recognize the advantage of having an "executable math notation" vs. one that just sits there on the printed page, doing nothing (except crying out to be deciphered). Kenneth Iverson makes this point very clearly when he writes:

> It might be argued that mathematical notation (to be referred to as MN) is adequate as it is, and could not benefit from the infusion of ideas from programming languages. However, MN suffers an important defect: it is not executable on a computer, and cannot be used for rapid and accurate exploration of mathematical notions.
> –Kenneth E. Iverson, Computers and Mathematical Notation, available from jsoftware.com

It's this ability of computer languages to promote the "rapid and accurate exploration of mathematical notions" which some of us find exciting and empowering. …

Furthermore, one might argue that imparting numeracy is *not* limited to teaching just those topics and notations most traditionally favored within mathematics. It's about imparting some familiarity and comprehension around *whatever* happen to be the culture's primary symbolic notations (music notation included) beyond those which we group under the heading of literacy.

We need to provide some exposure to computer languages because our industrial society is completely dependent upon them, because they've become ubiquitous.

We have the choice of segregating these topics from mathematics, just as we've divorced mathematics from the physical sciences. But some educators in every generation advocate curriculum integration through

> cross-pollination, and to such educators, it makes perfect sense to meld computer topics with math topics, with science and even humanities topics (cryptography is a good example of where all of these converge). In my view, the benefits to be obtained through synergy outweigh the arguments of turf-protectors who would keep their respective disciplines "pure".[30] –KU, ED-27, 2nd

Kaput, Noss and Hoyles (2002) agree with KU's and Iverson's notions of an executable notation for learning math:

> There are two key developments in a computational era: first that human participation is no longer required for the *execution of a process* and second, access to the symbolism *is no longer restricted to a privileged minority*. (p. 12, .pdf version; original emphasis)

Doron Swade (2000) characterizes this separation of human participation from the 'execution of a process' as an 'ingression of machinery into psychology' when he describes Charles Babbage's initial success at building an automated 'difference engine' in 1832:

> But with textile machines, trains, and all the other wondrous contrivances that poured off the drawing boards of inventors and from the manufactories, the human activity they relieved or replaced was physical. Babbage's engine was a landmark in respect of the human activity it replaced. It was the first ingression of machinery into psychology. (p. 84)

Furthermore, this separation of execution and human participation accomplishes two functions: one, it overcomes the barrier to learning the "coupling" between physical phenomena and their static, symbolic mathematical representations; and two, it redefines what mathematical

[30] This thread was continued by the group several months later, outside the scope of the data. However, since the exchange continued to be so thoughtful, it is reproduced in Appendix D.

knowledge becomes necessary to learn now that machines can execute the symbolic manipulations:

> The close relationship of knowledge and its culturally-shared preferred representations, precisely the coupling that has produced such a powerful synergy for developing scientific ideas since the Renaissance, became an obstacle to learning, and even a barrier which prevented whole classes from accessing the ideas which the representations were so finely tuned to express.
>
> While the execution of processes was necessarily subsumed within the individual mind, decoupling knowledge from its preferred representation was difficult. But as we have seen, this situation has now changed. The emergence of a virtual culture has had far-reaching implications for what it is that people need to know, as well as how they can express that knowledge. We may, in fact, have to reevaluate what knowledge itself is, now that knowledge *and the means to act on it* can reside inside circuits that are fired by electrons rather than neurons. Key among these implications is the recognition that algorithms, and their instantiation in computer programs, are now a ubiquitous form of knowledge, and that they—or at least the outcomes of their execution—are fundamental to the working and recreational experiences of all individuals within the developed world. (Kaput, Noss and Hoyles, 2002, p. 14, .pdf version; original emphasis)

Through their SimCalc Project,[31] they offer a glimpse at what the new knowledge representations to be taught might consist of:

> Over the past two decades, the character string approaches to the mathematics of change and variation have been extended to include and to link to tabular and graphical approaches, yielding the "the Big Three" representation systems, algebra, tables, and graphs frequently advocated in mathematics education. However, almost all functions in school mathematics continue to be defined and identified as character-string algebraic objects, especially as closed form definitions of functions—built into the technology via keyboard hardware. In the SimCalc Project, we have identified five representational innovations, all of which require a computational medium for their realization but which do not require the algebraic infrastructure for their use and comprehension. The aim in introducing these facilities is to put

[31] http://www.simcalc.umassd.edu/

> phenomena at the center of the representation experience, so children can see the results, in observable phenomena, of their actions on representations of the phenomenon, and vice versa. (p. 18, .pdf version)

Finally, Kaput, Noss and Hoyles describe how the power of calculus is embedded in their new representational systems, and how that power is a natural extension of the power conferred by arithmetic and algebraic systems:

> A key aspect of the above representational infrastructure is revealed when we compare how the knowledge and skill embodied in the system relates to the knowledge and skill embodied in the usual curriculum leading to and including Calculus. At the heart of the Calculus is the Fundamental Theorem of Calculus, the bidirectional relationship between the rate of change and the accumulation of varying quantities. This core relationship is built into the infrastructure at the ground level. Recall that the hierarchical placeholder representation system for arithmetic and the rules built upon it embody an enormously efficient structure for representing quantities (especially when extended to rational numbers) which in turn supports an extremely efficient calculation system for use by those who master the rules built upon it. This is true of the highly refined algebraic system as well. Similarly, this new system embodies the enormously powerful idea of the Fundamental Theorem in an extremely efficient graphically manipulable structure that confers upon those who master it an extraordinary ability to relate rates of change of variable quantities and their accumulation. In a deep sense, the new system amounts to the same kind of consolidation into a manipulable representational infrastructure an important set of achievements of the prior culture that occurred with arithmetic and algebra. (p. 25, .pdf version)

It may well be that these arguments and ideas would be best applied in teacher education classrooms. Yet the software Kaput, Noss and Hoyes describe is freely available for download by any Mathematics instructor, along with several units to get started with. Nevertheless, as the posters' messages illustrate, there is considerable resistance to attempts to persuade high school faculty to begin using machine-executable notation in their classrooms. Postrel (1998) brings this resistance into sharp relief on a larger stage than education:

> How we feel about the evolving future tells us who we are as individuals and as a civilization: Do we search for "stasis"—a regulated, engineered world? Or do we embrace "dynamism" —a world of constant creation, discovery, and competition? Do we value stability and control, or evolution and learning? Do we declare that "we're scared of the future," decrying technology as "a killing thing"? Or do we see technology as an expression of human creativity and the future as inviting? Do we think that progress required a central blueprint, or do we see it as a decentralized, evolutionary process? Do we consider mistakes permanent disasters, or the correctable by-products of experimentation? Do we crave predictability, or relish surprise? These two poles, stasis and dynamism, increasingly define our political, intellectual, and cultural landscape. The central question of our time is what to do about the future. And that question creates a deep divide.

This divide is evident in the willingness teachers have of adopting programming in their classrooms and changing the way they teach their subjects to their students.

Creating new Python programming classes

If Python cannot be readily introduced into typical AP classes, and if teachers of other subjects have concerns about adopting a programming language in their curricula, another alternative is to create an independent (non-AP or pre-AP) programming course centered on Python. The resistance here comes primarily from school administrators, but several posters have reported success at initiating and sustaining such courses in their schools, and the evident enthusiasm of both students and teachers help to grow the number of sections. A list of such schools is posted at the PyBibliotheca[32]

[32] http://www.ibiblio.org/obp/pyBiblio/

website. One in particular, Yorktown High School, Arlington, VA, deserves mention because the teacher who spearheaded Python's adoption in the curriculum, Jeffrey Elkner, has presented his experiences at two Python conferences (Elkner 2000, 2001). Elkner, in collaboration, has also ported a textbook, *How to Think Like a Computer Scientist*, from its original language, C++, to Python so that beginning Python programming courses now have at least one textbook to choose. These and other resources are listed in Appendix B.

After-school computer club

A few posters related their experiences of running after-school computer clubs. These seemed especially to cluster in the middle schools, where AP classes don't exist, nor any official programming classes (at least, none that were mentioned). These clubs appear to fulfill a felt need for exercising and developing young adolescents' programming abilities. Many posters underscored the importance of learning Python as a *first* language, so getting in on the ground floor at this age level seems especially appropriate.

Therefore, the path of least resistance to integrating Python in the school curriculum appears to be: begin with computer clubs in the middle schools that use Python as the primary language for doing projects of interest to the club members. That can then lead to Programming classes in the middle school curriculum. Alternatively, a teacher or group of teachers might institute a Programming class in their high school to build on the programming club experiences of the middle-schoolers, as well as to introduce programming to the newcomers. Recalling Postrel's distinction between those who search for stasis and those who embrace dynamism, educators who are concerned about promoting programming skills in their students must not be "scared of the future" but instead "see technology as an expression of human creativity" while simultaneously attempting to persuade their more fearful colleagues to learn how to "relish surprise," the surprising programming accomplishments their students can provide.

4.2b Infrastructure

Having gotten approval for such a class (or club), however, a teacher is still faced with challenges. We assume that hardware and software are not part of these challenges (although minor issues, of course, may occur). Most schools have some computers available, and the freely available Python language (and likewise the Linux OS) is almost certainly capable of running on whatever machines a school has. These requirements are modest and are assumed in the discussion that follows.

It is also assumed that the schools have some sort of networking system in place with (at minimum) a file server and perhaps a web server, and that the teachers involved with teaching programming feel comfortable using the school's network and can readily integrate it into the flow of creating, running and storing scripts that their students work with. One of the more advanced questions the group considered was whether or not to use a CVS

(Concurrent Versions System), or equivalent, with the class (this would require a CVS server) and such capabilities are not assumed, but may be worth supporting in a productive, learning programming, environment.

In this section we look at two aspects of a programming class' infrastructure: the available curriculum materials, and the distinction between the interactive shell and standalone scripts within the Python programming environment.

Curriculum materials

One striking need that emerged from several discussion threads was for texts and lesson plans. Since the idea of teaching programming using Python is relatively new (compared with, say, LOGO), there is a dearth of teaching materials available. There is plenty of Python-related and programming-related material, but very little specifically targeted for classroom use. Many available texts assume prior programming experience, and the ones that don't are either reference-type compilations or tutorials meant for self-paced study. During the course of these conversations, a small group of interested teachers formed an auxiliary website (the aforementioned PyBibliotheca) which aims to be a repository specifically for teaching materials such as lesson plans and online textbooks. The site also contains a list of schools that teach Python and includes a fun, 24 minute MPEG video called "Introducing Python" (Python Software Foundation, 2001). Nevertheless, teaching Python in schools is currently a niche activity, so Programming teachers still largely have to gather materials piecemeal, or create their own. For an example where this was successfully accomplished at the undergraduate level, see Shannon (2003).

Shells and scripts

Once a teacher is ready to begin teaching, an important distinction to make from the very beginning is one that is embedded into the very design of the Python language itself, and one that was strongly emphasized by many members of the group. There are two ways to 'talk' to the Python interpreter: through the shell (aka the interactive interpreter), and through scripts (text files stored on the computer's file system). Using the shell is a bit like talking directly to Python: you enter a statement or an expression at the prompt (>>>) and Python responds, for example:

```
>>> print 'The equator is about 25,000 miles long.'
The equator is about 25,000 miles long.
>>>
```

The shell ends with another prompt when it finishes responding. This experience is like a conversation in another way—no record of the interaction is stored. If you have built up a set of commands that does what you seek, you must save those commands in a separate file. This file is then a script that can be run by the interpreter (either by double-clicking it, or calling it from the shell).

The shell is an invaluable tool for exploring the Python language and for debugging smaller sections of larger scripts. A student's initial exposure to

Python, it was argued, should be through the shell; one gets immediate feedback as to the success or failure of what one has written:

What I don't like are teaching approaches heavily influenced by C or Java which pretend you can only write scripts/programs, and don't explore the interactive potential of a command line environment [shell]. Yet the latter is a wonderful scratch pad in which to test/learn the basics of the language, with immediate feedback. To bypass the shell is to make a huge pedagogical error, IMO. –KU, ED-26, 14th

I consider the Python shell to be the primary interface (in terms of importance) for the Python code I'm developing. ... To me, the interactive Python shell is just as important and significant a feature of Python as the clear, elegant syntax. –PO, MISC-4, 24th

The shell is also the place the official Python tutorial begins. Since using the shell is considerably easier to do than to explain, perhaps a quick example would be beneficial. The following transcript comes from a short shell session; '>>>' is the shell prompt that means "it's your turn; I'm listening" followed by commands typed by the user. The Python interpreter's replies are on lines following the commands:

```
>>> 2+3
5
>>> 15*8
120
>>> 8-80
-72
>>> print "Even 'quoted text' can be displayed!"
Even 'quoted text' can be displayed!
>>>
```

One can have shell sessions where the 'commands' stretch over more than one line; in that case the interpreter offers a different prompt ('…') to indicate "Yes, I hear you; I'm waiting for you to finish". This example constructs a loop that squares each number in a list, puts it into another list and prints the result:

```
>>> list_of_numbers = [20, 21, 22, 23, 24, 25]
>>> list_of_squares = []
>>> for each_number in list_of_numbers:
...     square_number = each_number**2
...     list_of_squares.append(square_number)
...
>>> print list_of_squares
[400, 441, 484, 529, 576, 625]
>>>
```

Using the interactive interpreter is a great way to introduce the Python language to beginners because there is so little 'mechanism' between the user and the interpreter; it is quite easy to get successful results virtually immediately. And having learned to use the shell, it becomes indispensable once you start writing longer programs where debugging a small but crucial area is unwieldy in the context of the whole script, but is quite manageable in the shell.

Of course, the downside to the shell is that once one quits the session, all the typing that was done has been lost. If you have created a set of commands that you expect to run at a later time, those commands need to be saved as a script, which is simply a text file with a name that ends with the .py extension. This requires a text editor of one sort or another; there are many appropriate ones available that work well with Python. Choosing the right one is usually a matter of taste and experience.[33] Once the script has

[33] Python comes with IDLE (an Integrated DeveLopment Environment) which serves both functions, shell and editor.

been saved, the Python interpreter can execute that file either by double clicking it or calling it from the shell. So, much of one's day-to-day programming experience becomes that of editing a script and executing it to see what changes need to be made, and also going to the shell to work out the details of some portion of the larger script. However, to emphasize, before introducing the notion of scripts, it is important to become facile with the shell so that this valuable way of interacting with the interpreter is not ignored by the student programmers.

4.2c Teaching methodologies

Many discussion threads revolved around notions of 'how' and 'what' to teach in the classroom. Many issues were raised, and experiences were shared that were meant to shed some light on those issues. What follows is certainly not an exhaustive listing of all the issues a programming teacher might face, nor are all the issues discussed in the forum included; however, those that are included seem to have qualified by virtue of the passion exhibited by the posters or the logic evident in their stories. The following subsections include: involving the student's interests, different programming styles, learning by doing, the Socratic method, encouraging planning, and homework and grading.

Involving the student's interests

True to the spirit of CP4E, there was a marked emphasis on involving the student's own interests in programming exercises. At the same time, it was recognized that when one is beginning to program, one doesn't know *how* to use programming to further one's interests in a (non-programming) topic. One needs to 'grow' into it, and there was some discussion as to what can be done to sustain students' interest in programming during that awkward beginning growth phase. This is partially a question of motivation, and partially one of seduction. To a certain extent, one can rely on motivation by assuming a certain degree of voluntary participation in an elective class by students who are intrinsically motivated to learn how to program. For those whose intrinsic motivation is not sufficient to get through the initial growth phase, teachers need to involve them in activities which have 'extrinsic' motivation, that is, objectives that are interesting in their own right. The domains of these objectives could be, for example, textual manipulation, mathematical calculation, scientific simulations or even designing graphical games.

One technique to maintain interest is to develop a very simple program that everyone can do and understand, and then gradually add features to the program as new programming constructs are introduced:

> I'm interested in doing some kind of iterative instruction using one project to illustrate a lot of the characteristics and functionality of the language. For example (after Lutz):
> - start with a simple, hard-coded dictionary of names/addresses (an address book). Print them out.
> - add a simple, text-based UI, allowing user to add, delete, edit entries (input/output/UI)
> - complexify the data stored and move to file based storage (file io, data stores)
> - add ability to sort (without built-in methods, to learn the algorithms)

> - add ability to find (more algorithms)
> - then, move to a GUI?
> and then....
> (I've got lots of ideas - net-based sharing, file input/output, move to a web-based CGI system, etc - just wondering what people think. Also, would this excite students?) –RR, ED-19, 1st

This theme was echoed with other similar designs based on, for example, an employee database and building up a game. However, one poster warned:

> I think this is okay, and students like it, but I'd caution to not have it be your /only/ set of examples. It seems to work for me if you occasionally say “okay, now let's come back to our XXX” but it's too much if the single “theme” is the only thing you're showing. –MW, ED-19, 2nd

Alternatively, a teacher can begin with numerous small programs and when the students are savvy enough, introduce a larger project that ties various concepts together. One poster raised the issue as follows:

> Looking through some of the edu-sig posts … I'm seeing that assigning students a large project seems to be more popular than assigning a number of smaller, more pedantic problems. What is the rationale behind the larger projects as opposed to smaller projects? What are the benefits and drawbacks of asking students to spend a week on one large project instead of that same week on three or four smaller tasks, then giving a large task once a month or so? I've been using what I consider to be smaller projects … and am wondering if fewer, larger projects would be better for my kids. –BE, ED-23, 1st

The responses were quite consistent:

> I did many more small assignments at the beginning of the year--little functions for programs that could be completed in twenty minutes or so. Now that my students have been exposed to most of the Python syntax, I have them working on larger assignments that are really just a bunch of little ones glued together. Plus, the larger projects tend to be more interesting for the students. –TW, ED-23, 2nd

---

> What I do first is assign small exercises from each chapter, and then assign a big project. ... There are two very good reasons for doing bigger

> projects:
> 1. The ability to write meaningful programs should be a primary teaching goal. Academic programs have often come under criticism by industry for not preparing students for “real world” programming. There is a natural tendency in academia to focus on small examples that make the concept transparent without confusing learners with extraneous details. The problem is students come away with no understanding of the software development process itself. Their learning doesn't scale up to bigger problems. This is also the motivation for the case study now used in the Advanced Placement program.
> 2. Big projects are Cool! My experience is that students like them because they solve more interesting problems. –JE, ED-23, 3rd

---

> Since I believe that programming is a craft (not an art or science) we call our after-school programming club a Guild. Apprentices learn what the tools are and how to use them. Journeymen work on small projects at my suggestion or as part of a larger project. At this stage they are developing their craft and learning the why of their tools. Once they demonstrate an understanding of programming they become a Master. As a Master they write their own programs and ask teachers if there is something that they could develop for them and their classes, such as a physics simulator or the animal game for understanding classification. All this apprentice, journeyman, master stuff is informal, I don't give tests or allow hazing, the students seem to know what level they are. –JS, ED-23, 6th

---

Yet another technique suggested, which might help bridge the gap between small and large projects, was to learn from already functioning code:

> Don't ask them to face a blank page.
> If it’s Game of Life, start them with functioning code and challenge them to extend it in some fairly specific way.
> If it’s graphics, start with functioning code that draws lines, ask them to draw boxes and circles.
> Even at high levels of programming, few people are asked to face the blank page. One is extending, optimizing, debugging, porting, etc.
> Give them something broken to fix. Plant a few bugs.
> But don't ask them to face the blank page. –AS, ED-14, 8th

This sentiment was echoed by:

> I think when it comes to teaching Python, you need to have some source code that already does some fairly interesting stuff which you then dissect, i.e. students learn to read working source and understand what

> it's doing before they try to tackle coding a program of similar complexity *ab initio*. Reading well-written code and understanding why it works is an important aspect of learning to program. –KU, ED-11, 2nd

In addition, some posters mentioned that virtually all real-world programmers use modules in their projects. These are simply a collection of functions bundled into a file that offer useful procedures for accomplishing a necessary but (usually) tedious task. Learning how to import and use modules is a necessary skill for any Python programmer, and once mastered, can lead directly to larger projects without having to code all the functionality in the modules oneself.

This latter poster also advocated finding that 'extrinsic' motivation in the coursework of the students' other classes:

> What I would suggest is you find out what these same students are learning in other course work, and try to play off that. Allude to their other subjects or tackle them directly. Whatever they're doing in math these days, do something in the same ballpark in Python. But you needn't limit yourself to math. There should be some angles on other subjects as well, e.g. if they're reading a novel, copy a paragraph from it and encrypt it (simple letter substitution), or if they're learning some history, try to figure out how many days have passed since some event. … The idea is to have what's going on in programming reinforce whatever content is already filtering through their minds anyway. That definitely includes whatever they're learning in math, but there's no need to limit your scope to that exclusively. –KU, ED-11, 5th

Thus, when structuring the curriculum of a programming class, these educators recommend beginning with small programming tasks that gradually increase in size and complexity and lead to a larger project that has greater interest for the students. If possible, many of the smaller tasks should center on the same application, which then ramifies as it matures.

Alternatively, these projects should relate to the students' coursework in other classes, so that their 'learning to program' efforts lead to 'programming to learn' results.

Different programming styles

As discussed in Chapter 2, there are three broad styles of programming: procedural (also called imperative), functional, and object-oriented (OO). There was some discussion as to when these different styles ought best to be introduced to students, but these discussions primarily centered on when to introduce OO programming v. procedural programming; functional programming was not, as far as I could tell, advocated as a beginner's style. Python is somewhat special as a computer language in that it handles all three styles adroitly; it doesn't coerce the user into preferring one style over another.

There were two clear positions with respect to teaching OO programming. One position suggested teaching OOP from the beginning; while the other position preferred beginning with a procedural approach and then later introducing OOP. Those who advocated OOP from the beginning had this to say:

> I've been trudging through Yourdon & Coad's *Object-Oriented Analysis* of late, and they had an interesting point on teaching OO: their experience is that old-style programmers brought up in a COBOL world have a really hard time "getting" OO, but non-programmers—in their context, it was marketing and sales staff, but it could be anyone—"get it" immediately.

Another point they drive home again and again is that "generalization/specialization" and "whole/part structures" are not just pie-in-the-sky abstractions dreamed up by CS professors, but two of the fundamental techniques used in human thought. Duh! Of *course* they are, but I never really saw it that way before, since I grew up in a pre-OO world (Pascal and BASIC) and had to learn it through the same artificial examples (biological taxonomy and a graphics library) as everyone else.

I think this argues that OO concepts should be built in from the start, if possible. –GW, ED-1, 3rd

---

> > Don't you think classes are as easy as variables?
> No, they're not. …
That's funny. My experience was quite different. When I first saw objects and classes it was like a revelation from God. They seemed intuitively obvious and the answer to my prayers. I wonder if this is a right brain/left brain thing or something. –SM, ED-1, 5th

---

I was suggesting that both the OOP paradigm, and the functional paradigm, as well as the more classic procedural paradigm, all intersect in Python, and it's worthwhile to not just seize on the procedural alone.

One should "get in on the ground floor" with OOP and functional concepts as well. Primitives which take functions as arguments, like map, reduce and apply, are good examples of functional concepts.

I think there's a danger in having people most comfortable with procedural programming inadvertently depriving newbies from accessing alternative philosophies that are just as basic in their own way.

I think an intro course that focuses on basics might do well to actually spend some time on these broad brush stroke concepts: procedural, functional and object oriented, using Python examples to illustrate what's meant. –KU, ED-32, 6th

---

From these and other postings it is clear that the normal way to introduce programming is to use a procedural style. So, introducing OO programming early on is a slightly radical departure from how things are normally done. For some advocates, this was a philosophical position, for others a psychological one. The philosophical reasoning was based on a holistic viewpoint that one begins with an entirety and then creates subclasses of special case instances, akin to the way OO programming itself proceeds. The psychological advocates felt an immediate kinship with OO programming when they learned it, and didn't want to deny others the same pleasure. However, not everyone was convinced:

> \> Don't you think classes are as easy as variables?
> No, they're not. Maybe the theory is as easy as a variable, but the practice isn't. It took quite long for me to find out when to use classes. And multiple inheritance still struggles me, and private methods (__aaa changes to whatever but still isn't private?). I think we shouldn't be too fast with classes. We _can_ explain how to use them (jan = Turtle(100, 200)), but subclassing is not easy to explain. –GH, ED-1, 4th

And, in fact, my experience has been that OO programming *is* rather more difficult to learn than procedural, although learning it in Python is vastly more approachable than in other languages. Having weighed the various arguments presented by the posters as to when to introduce OOP, it seems most prudent to begin a programming course by teaching procedural programming and get a solid grounding in that. However, it seems critical to then include OO programming during, say, the second semester (of a year's class). Since much of the inner workings of the Python language is grounded in the OO programming style, it behooves the curriculum designer and classroom teacher to make that style explicit. Furthermore, it seems that a sizeable percentage of the students will gravitate towards OOP (for philosophical or psychological reasons), and those that prefer the procedural style will at least be aware of OOP if it becomes necessary to use it later.

Bruno Preiss (2004) reinforces the need to learn OOP early on:

> I have observed that the advent of object-oriented methods and the emergence of object-oriented design patterns has led to a profound change in the pedagogy of data structures and algorithms. The successful application of these techniques gives rise to a kind of cognitive unification: Ideas that are disparate and apparently unrelated seem to come together when the appropriate design patterns and abstractions are used.
>
> This paradigm shift is both evolutionary and revolutionary. On the one hand, the knowledge base grows incrementally as programmers and researchers invent new algorithms and data structures. On the other hand, the proper use of object-oriented techniques requires a fundamental change in the way the programs are designed and implemented. Programmers who are well schooled in the procedural ways often find the leap to objects to be a difficult one.

It appears that one reason OOP is so appealing to some programmers is that it exercises a kind of cognitive-artistic talent. It takes a certain inner perspective to see where the natural boundaries of a problem are and to construct objects that reflect those edges. What we need to keep in mind is that programming *can* appeal to artistic types; perhaps we could call them 'cognitive artists'!

> I find the suggestion interesting that experienced programmers often have more difficulties with objects than beginners. That isn't my personal experience but I have heard it said many times.
>
> BTW this topic may seem a little off point except that we need to teach people with different learning strategies. There is a tendency of programmer types to use left-brain teaching techniques and thus lose half their audience out of the gate. Anyone who has ever sat in front of a room full of young students knows that there are as many learning strategies as there are students. CP4E needs to be sensitive to this or it will be one more also-ran strategy. –SM, ED-1, 10th

And for those fortunate enough to 'get it', perhaps nirvana is within reach:

> I'd studied objects before Python, but Python transformed my perception of them. Now they make sense. Only I think that objects and classes are as pure an expression of the Buddha nature underlying all existence as you can find outside of a Tibetan monastery. –IL, ED-1, 8th

Not all students take well to a formal, analytic approach to programming. Seymour Papert addresses another difference in programming styles:

> The simplest definition of constructionism evokes the idea of learning-by-making and this is what was taking place when the students worked on their soap sculptures. But there is also a line of descent from the style idea. The metaphor of a painter I used in describing one of the styles of programmer observed at the Lamplighter school is developed … in two perspectives. One ("bricolage") takes its starting point in strategies for the organization of work: The painter-programmer is guided by the work as it proceeds rather than staying with a pre-established plan. The other takes off from a more subtle idea which we call "closeness to objects"—that is, some people prefer ways of thinking that keep them close to physical things, while others use abstract and formal means to distance themselves from concrete material. Both of these aspects of style are very relevant to the idea of constructionism. The example of children building a snake [using LEGO/Logo] suggests ways of working in which those who like bricolage and staying close to the object can do as well as those who prefer a more analytic formal style. (Harel and Papert, 1991)

The distinction here is between top-down and bottom-up programming, where a bottom-up style of programming is favored by the *bricoleur* and those wanting to stay 'close to objects', while top-down is the 'more analytic formal style'. So not only does the teacher need to consider how to sequence procedural and object-oriented styles of programming, but also how to accommodate both top-down and bottom-up styles of development.

Before leaving this topic, I would like to offer an exchange that occurred on a different, but related, mailing list (python-tutor) that illustrates the relationship of OOP and procedural programming in practice. Although we aren’t directly considering creating graphical user interfaces, keep in mind that OOP is the preferred style for such programming:

Hi, I have been studying python for a month or so now and have written a few small apps the main being a ~300 line application. I am finding as I work through things that I am coding in a procedural way, and then going back over the code and recoding parts into an OO model. Is this a normal progression, or should I be trying to force myself into an initial OO model, my brain just doesn't seem to think like that, and I have work through a procedural method to actually solve the problem. Do experienced programmers find this sometimes also? –R

My non-guru opinion is that some small things lend themselves better to procedural solutions than OOP ones, even if it were just because it requires less typing. Big programs however are better as OOP than procedural. Exactly where the boundary is, depends on the application. –A

I found something similar, at first. I was never formally taught OO, and my programming experience was mainly fortran, pascal, perl, IDL, and assembler; the whole OO thing was a mind twister. But increasingly it feels more natural, and python made the transition reasonably painless.

Several factors helped me make the jump: a big-ish project, a GUI interface using wxPython, and several weeks of uninterrupted time to work on it. The GUI bit was both the most frustrating and the most helpful (wxPython is big, but is easier if you think OOP).

I started doing things procedurally, but the GUI bit slowly helped me understand the OO paradigm and forced me to begin making my own classes. I rewrote my procedural code into some classes, making it much easier to fit it into the wxPython framework.

I still think procedurally, but I no longer hesitate to create my own classes. Lots left to learn, but in three weeks I've come a long ways. I couldn't have come this far in C++ or Java. –KF

Here, too, we see support for beginning with a procedural style for smaller applications and then segueing into an object-oriented style for larger projects.

The functional programming style was not much discussed. Although Python supports it, there are other languages that support it better (e.g. Scheme or Haskell). One highly regarded poster emphasized that functional programming was largely the domain of computer science and that:

> If the goal [of CP4E] is primarily to teach practical programming skills, best to forget them [functional idioms] entirely! They add no power to Python, and the testimony of "almost all" posters to c.l.py [the python newsgroup] over the years who have addressed this issue is that a straight Python loop [procedural style] is easier to write, and to understand later, and *much* easier to modify later. –TP, ED-1, 13th

Nevertheless, it is also claimed that there are some computing problems which are *much* more difficult to solve procedurally instead of functionally, so it may be worth learning at least a little about it.

Learning by doing

Writing software falls squarely in the domain of constructionism. Students are constructing actual, real-world artifacts that can be shared with others. Harel and Papert (1991) distinguish constructionism from constructivism:

> Constructionism--the N word as opposed to the V word--shares constructivism's connotation of learning as "building knowledge structures" irrespective of the circumstances of the learning. It then adds the idea that this happens especially felicitously in a context where the learner is consciously engaged in constructing a public entity, whether it's a sand castle on the beach or a theory of the universe.

As was mentioned in Chapter Two, Papert goes on to distinguish between *constructionism* and *instructionism*. Each embodies a different theory for the transmission of knowledge: whereas instructionism favors a pipeline model of transmission, constructionism favors, shall we say, a self-organizing model of transmission (knowledge self-organizes as the learner engages with construction activities). These are not meant to be seen in opposition, rather, the inclusion of constructionist methods is meant to bring about real reform in our educational processes:

> I do not mean to imply that construction kits see instruction as bad. That would be silly. The question at issue is on a different level: I am asking what kinds of innovation are liable to produce radical change in how children learn. Take mathematics as an extreme example. It seems obvious that as a society we are mathematical underperformers. It is also obvious that instruction in mathematics is on the average very poor. But it does not follow that the route to better performance is necessarily the invention by researchers of more powerful and effective means of instruction (with or without computers).
>
> The diffusion of cybernetic construction kits into the lives of children could in principle change the context of the learning of mathematics. Children might come to want to learn it because they would use it in building these models. And if they did want to learn it they would, even if teaching were poor or possibly nonexistent. Moreover, since one of the reasons for poor teaching is that teachers do not enjoy teaching reluctant children, it is not implausible that teaching would become better as well as becoming less necessary. So changes in the opportunities for construction could in principle lead to deeper changes in the learning of mathematics than changes in knowledge about instruction or any amount of "teacher-proof" computer-aided instruction.

We will see some examples and suggestions along these lines in the sections on Math and Science.

It is curious that the word instruction also means a command issued by a software program to the underlying hardware processor that carries out the instruction. With this sort of connotation, it may well be prudent, if we wish to avoid turning our students into automatons, to gravitate more towards a constructionist style of teaching, to which learning computer programming, indeed any language learning, lends itself. Instead of teachers issuing instructions, they can offer construction environments that are more conducive to their students' personal growth.

The spirit of this distinction is also made by Goodman (1995) in his discussion of metaphoric games:

> The distinction between being instructive and being provocative lies at the heart of Cornell West's (1989) argument in *The American Evasion of Philosophy* in which he makes the claim that it is Ralph Waldo Emerson who ushers in that bastion of American philosophy known as pragmatism, the kind of thinking pursued by William James, John Dewey, and, of late, Richard Rorty. "The primary aim of Emerson's life and discourse is to provoke; the principle means by which he lived, spoke, and wrote is provocation. At the 'center' of his project is activity, flux, movement, and energy" (West, 1989, p. 25). Nothing could be more important than making this distinction between instruction and provocation when it comes time to make claims about the instructional value of this kind of game under consideration here. In much the same way that a successful metaphor provokes new ways of thinking about familiar matters, the games that I have designed that I think of as being metaphoric in character are intended to provoke thought, not to instruct their participants to think in the way the designer thinks. (p. 185-186)

Constructionism, like metaphor, is intended to provoke thought rather than invoke instruction. We also recall the value of metaphoric thinking for creating the algorithms used in programming.

Socratic method

Given the previous Papert quote, one might conclude that all CAI (computer-aided instruction) was simply instructionist, but this isn't necessarily the case. Consider an interesting thread simply titled ‘Socratic methods’. The original poster suggested developing a system of iconic programming for teachers (or curriculum developers) that would lead students through a question and response sequence designed to elicit agreement with some end state:

> The interesting thing is that you can also think of it as a teaching tool. You can map out trees of current and desired thinking habits, and build a transition tree that leads logically from one way of thinking to another.
>
> Now it takes a great deal of human thought to come up with these trees, but once they've been mapped out, just about anyone can follow them. …
>
> I think it would be really cool if a computer could be programmed with a tree like this to teach people using the Socratic method.
>
> That is, it asks the user questions, and based on the answer either helps the user to understand things better, or progresses along the tree to the next step… –MW, ED-8, 1st

There was quite a bit of discussion of software tools that might be utilized to build such a system, along with references to tools that already performed certain aspects of the idea, and then another poster came through with this wonderful story and speculation that I repeat in its entirety:

> \> I think it would be really cool if a computer could be programmed
> \> with a tree like this to teach people using the Socratic method.
>
> I have been thinking about this, while playing around with a Palm program called CyberTracker. It is used for data entry. This particular database interface was designed for illiterate Bushmen trackers. Equipped with Palms, the Bushmen have been producing lots of data on rhinos in South Africa and other animals as well. More data than we had collected on the animals in the last hundred years was gathered in the first few weeks of this program.

> Rather than the usual database interface of filling out a form, the developer of this interface uses icons that follow the taxonomy used by the trackers to group related tracks and signs. They use a lot of icons mixed with text (teaching the bushmen a bit of word recognition as a side effect.) Because the interface is so closely designed with the Bushmen in mind, it takes them about 15 minutes to learn how to use it.
>
> Switch to North America, a group of trackers here are using a modified version of the program, not only to collect data, but also to hone their skills. As they enter data, they learn the taxonomic system of the master tracker that tailored the interface; they also train their minds to look for things in a particular sequence, and with great detail. One learning tracker claimed it was like have a master tracker right there looking over his shoulder, pointing out what he should look at next.
>
> This starts to flow into the notion that learning is all about patterning the brain. Working with this entry program with its words and icons does just that, running the mind through learning sequences, like a martial artist practicing their kata.
>
> Learning patterns can be used in lots of ways in teaching. MonArt art classes for children use pattern recognition as a way of increasing art skills, with really dramatic effect. Once kids begin to recognize angles, circles, dots, squares, etc. in the world around them, and working through some basic copying exercise where they copy and mirror line patterns, the kids' artwork improves remarkably. The instructors are using a taxonomy of shape components and hand movements to sharpen the mind's kinesthetic memory in the students.
>
> What is the Socratic method, but a set of questions that focus the mind on the topic at hand? It is a kind of taxonomy for mental inquiry. This isn't entirely new, we have used decision trees and flow charts in all kinds of diagnostics....
>
> I am not sure exactly where to go with this yet, but I am intrigued by the overlap here between data collection, decision trees, Socratic (maieutic) teaching and the patterning of the mind. –SF, ED-8, 6th

In his next post he explains the unusual term, maieutic:

> From the root maia, meaning midwife. It's another term for Socratic method. The teacher helps in giving birth to the knowledge in the student, drawing the knowledge out of them. –SF, ED-8, 8th

The Socratic method externalized as a CAI application might have real utility for teaching specific techniques. The OP of this thread later posted an educational example of the Socratic method in action at a website, Ms. Lundquist: The Tutor.[34]

For another example, one topic that consistently proves troublesome for beginning programmers is recursion,[35] which is often a more succinct way of generating a solution than looping (iteration). A simple problem that can be solved both ways (for illustration) is the factorial function: 'n!' in mathematical notation. A simple loop to calculate the factorial of a number, might be written as:

```
def factorial(n):
      product = 1
      for i in range(n):
            product = product * (i+1)
      return product
```

If we type 'print factorial(5)' at the Python prompt (after typing in this function) we get the response '120'. Even if one doesn't know Python syntax, the general outline of this procedural logic may be discerned in the 'for' loop above. However, another way to define the factorial function is:

```
def factorial(n):
      if n == 1:
            return 1
      else:
            return n * factorial(n-1)
```

[34] <http://www.algebratutor.org/>

[35] "Recursion is a way of specifying a process by means of itself. More precisely (and to dispel the appearance of circularity in the definition), "complicated" instances of the process are defined in terms of "simpler" instances, and the "simplest" instances are given explicitly." <http://en.wikipedia.org/wiki/Recursion>

Here too, if we type 'print factorial(5)' after the function definition, we get the response '120'. However, the underlying logic of this recursive code is not at all obvious, especially to a beginner, even if one knows Python syntax. In this kind of situation learning recursion via a Socratic-based CAI application might be a boon for many learners. In fact, this has already been done. Chang et al. (1999) developed such an application (which they called 'Web-Soc') specifically for teaching recursion. They tested three learning modes, computer-based collaborative learning, computer-based individualized learning, and learning without computer assistance, and found that

> the three learning modes—computer-based collaborative learning, computer-based individualized learning and learning without assistance—produced different results on students with different levels of prior achievement. For students with higher achievement, learning recursion with or without the assistance of Web-Soc did not seem to make significant difference for their improvement. But for students with lower achievement, the students using Web-Soc in either collaborative or individualized mode made impressive progress when compared with those who studied without the assistance of Web-Soc. (p. 5)

Of course, a master teacher using the same methodology would be even better, but such master teachers, like master trackers, appear to be in short supply; therefore, cybertracker-like, maieutic software for learning programming could well assist in the teaching of such necessary patterns.

Encouraging planning

One question that came up a couple of times was how to encourage students to plan their projects. A few teachers bemoaned the fact that their students would jump into coding without an overall plan of attack. The result was usually brittle or inelegant and required significantly more work to get to a working state. There were three primary suggestions made to help alleviate the situation:

> I'm teaching in a programming course for biologists. What we are doing, to encourage the students to think about the problem before implementing it, is to write the algorithm in a natural language pseudocode. We do not really introduce a pseudocode, which is even a formal language, but we ask them to describe their solution in their mother natural language (for us French) and discuss it before they start the implementation. The technique not only helps them to design an algorithm, but also helps us very often to understand what the difficulties are for them. –KS, ED-31, 3rd

> I would encourage a lightweight planning process, such as writing up a bunch of index cards with tasks and then ordering the index cards according to when tasks should be completed. –SH, ED-25, 2nd

> What I've taken to doing in my classes (teaching OOP in Java) is to force test-driven development with JUnit. Students don't buy it at first, so they'll write all the code, then write the test cases. Once I show them (projecting my laptop screen) how much easier it is, and how much better the design ends up being, the light starts to go on. Also, with test-driven development, they're forced to “evolve” the code (and sometimes the interface) as more test cases are added. Thinking of test cases will cause at least some forethought about the algorithm. –TW, ED-31, 2nd

There was no feedback on the efficacy of any of these methods by the OP, but there was one other technique brought up in these threads that bears mentioning: embedding the programming requirements in stories.

> I had expected more real “stories”—i.e. “So, a user walks into a brokerage house and asks...” with the eye on the stories being one to engage the imagination and from the story to distill the requirements.
>
> That's kind of what I have done for middle school students with respect to a Mars Voyage Simulation written in Perl though it's not quite the same thing because I was writing the program to fit some requirements of a Mars Space Camp that the students were taking. Still, I tried (for middle school students) to weave a “simulation story” around what I was doing so there would be a natural visualizable way that the

> programming aspects would fall out and be meaningful to the students rather than seeming to have just been pulled out of thin air.
>
> Your students are several years older than mine so it may be that they don't need (as I felt) the more concrete story telling to get their heads into the problem. –DR, ED-25, 6th

This could also tie in nicely with a Socratic approach where 'the story' is a Socratic question and answer sequence that also specifies the problem and points to a programming solution.

Homework and grading

A typical concern of teachers is assignments and evaluation. There was some discussion about this on the list, from posters representing a spectrum of levels, from middle school to graduate courses, making these issues difficult to generalize. Still some suggestions and experiences were offered:

> Our course is being taught at the graduate level, so somewhat different considerations apply. However, we do assign homework (on a weekly basis). The homework is submitted to the instructors as text or .py files and is required to be extensively commented. The function and strategy of each section of code must be described in the student's own words. Thus the homework assignments are sort of a combination of programming projects and discussion questions. –JH, ED-21, 2nd

---

> On the subject of students swapping code—I'd see that as inevitable and structure the homework around that assumption. Sharing code is a good thing, encourage it.
>
> One other thing to encourage is not to create from scratch but take a working program and extend it.
> - Make it menu driven
> - Add an extra function to the menu
> - Improve error handling etc... –AG, ED-21, 4th

> I cut the number of computers in my lab in half this year, so that two students share each computer rather than having each student have their own.

> I did this to make collaboration a necessity. I plan to use peer-programming methods (extreme programming) for projects throughout the year. In the beginning I will be the client and the students will be the programmers. Later in the year I would like to try setting up situations where the students can play both client and programmer roles for each other. –JE, ED-21, 5th

---

> The training outfit I teach for (and developed the Python course for) has used this model [peer-programming] successfully for many years. Our environment is somewhat different - four or five intensive days, and you're hitting up against the limits of what people can absorb. Learning with a partner, if managed a bit (making sure both students type, and that they really /do/ collaborate: we keep instructing them to READ the problem out loud and talk about it) takes some of the focus off of syntax and typing mechanics and puts it onto understanding the concepts. And it's just plain more fun. In effect, the partners teach each other, as they "get" different things at different times. There are a number of studies that back this up; my comments are based on my own observations, however. –MW, ED-21, 7th

---

We see here a shift away from an *instructionist* model where individual students are asked to memorize and recite material by following instructions and towards a *constructionist* model where teams of students work on projects that exercise their understanding of developing software applications. Evaluation is still tricky, though, and the OP of this thread lamented that none of the responses addressed that issue directly. Implicit in one suggestion, though, is a kind of evaluative method: commenting. Programmers are expected to heavily comment their code so that future programmers will have some idea of what each code section does. These comments can also be used by teachers to evaluate the student/programmer's understanding of what is happening with the code. Furthermore, since programming projects are very often done in teams using lots of borrowed code, comments can go a long way towards facilitating team interaction and productivity. Classroom dynamics ought to reflect that reality by encouraging well-commented code.

Conclusion of education category

We have seen the difficulties educators face trying to incorporate programming into their school's curriculum. Those who try to adopt it into their subject-specific classes face resistance from other department members who may be reluctant to change the knowledge being taught in their classes, and the way it is taught. A more successful approach is to establish a non-AP course in programming, or to sponsor an after-school computer club.

We have also seen that those teachers who do employ programming in their classes have a few formal textbooks tailored to the teaching of Python in a classroom setting, but that many resources exist in cyberspace that can be utilized in the creation of a customized syllabus. These posters also emphasized that there are two modes of interacting with Python, shell mode and script mode, where students obtain quick results using the interactive interpreter, and can consolidate successful results in scripts saved on their filesystem.

And, in the final section on teaching methodologies, we saw that the posters advocated retaining student interest in programming by involving them with projects that aligned either with the students' own personal goals, or with their other subject matter goals. We also saw that the projects should begin small, and gradually get larger as skills and syntax familiarity increase. While introducing programming assignments, teachers need to manage two aspects of programming style: procedural v. object-oriented, and top-down v. bottom-up. We also saw the importance placed on encouraging good planning by the budding programmers, and also on peer collaboration while working on the projects/assignments in a constructionist rather than instructionist environment.

4.3 Python and Computer Science

This set of threads was defined by the following keyword phrases:

- Case sensitivity
- Indexing
- Programming
- Accessibility
- Computer science
- Measuring Python
- Algorithms
- Notation
- AI (artificial intelligence)
- Sequences by reference
- Novice obstacles
- Interactive interpreter
- Lines of code

The set of threads included under this category contained one useful theme that I'd like to explicate here. Although there were several other themes of considerable interest discussed within these threads, they were excluded from this dissertation due to reasons explained in the previous chapter.

Insiders and outsiders

The primary theme to highlight from this category is that any language, human or computer, creates insiders and outsiders. The advantage of Python as a first computer language is that the linguistic barrier to becoming an insider is very low, relative to other computer languages. It is low largely due to its use of simple, ordinary words. Here's how one poster put it:

> Python: I love for its readability on two levels:
> 1. - consistent syntax and mostly very clear logical flow top to bottom.
> 2. - friendly visual syntax mainly derived from the obligatory but liberating white-space indentation
>
> Scrolling down and scanning large chunks of python code, I detect a well-tempered rhythm. The blocks, indentations, naming and repetition reveal a similar construct pattern across myriad examples. This is a very 'un-scientific' comment I realize, but one perhaps worthwhile investigating..
>
> I love Python's writeability for two main reasons also:
> a. - executable pseudo-code
> to me Python is very sculptural. You start by naming and imagining things and relationships and then proceed to sketch them in. With time and skill you just keep going, implementing details and functionality. Play and exploration are encouraged, as is throwing things away when one has a better insight. I think this stems from the fact that because everything in Python is an object, one must name it, and doing so makes it exist. You might argue that is true for all languages, and I would agree, but Python seems very direct in the workflow from imagine, to model to further implementation and naming and objects are present in the same manner every step of the way.
>
> b. - dictionaries and named arguments
> yes thank you Guido! I argue named arguments are a major reason why Python is readable and makes a great learning language. You take your semantic token map with you and share it liberally with all who come after. Dictionaries are the engine behind this. What you call something or assign a value/meaning to is more important than where it comes in the sequence, and we people should not be persecuted by having to interpret lists of unidentified symbols.
>
> Once you know how to 'read' a little Python, you know how to read most Python. Ditto writing. There is evident a clear design pattern here, which apart from a few characteristic idiosyncrasies, works very well.
>
> This I think makes it especially suitable for teaching programming. And I agree with KU, it is a very cosmopolitan language. I would stress also

> that any language for people to learn is only as good as its community. Even if a small tribe, is it a living growing language? I believe the mood and quality of its culture is especially crucial for learning computer programming. –JC, PCS-6, 4th

However, in another thread, this ‘readability’ was compared with two other attributes, succinctness and power:

> In “Patterns of Software,” Richard Gabriel makes an important corollary point. He refers to Compression rather than Succinctness, and compares it to the way language is used in poetry to generate multiple meanings. But he warns that compression without habitability (another important feature he discusses) leads to write-only languages (I'm paraphrasing liberally). In other words, taking succinctness as the only measure leads to Perl %-)
>
> Graham's article is a response to something Paul Prescod wrote, namely, “Python's goal is regularity and readability, not succinctness,” which Graham translates as “Python's goal is regularity and readability, not power.” My take on it would be “Python's goals emphasize regularity and readability (habitability) over succinctness (compression).” Python is extremely succinct compared to Java, C, or C++, but verbose compared to Perl or Lisp. On the other hand, I've found Python to be easier to read and comprehend than any of those languages. –DE, PCS-15, 3rd

In terms of trade-offs, Python sacrifices succinctness (or power) for ‘habitability’ (familiarity and comfort). That is, Python does not require excessive sophistication or abstraction to begin using it and thus is accessible to the average programmer. But it doesn’t trade away too much succinctness; just enough to be readable, familiar and comfortable, and no more. Nevertheless, despite its habitability, there is still a barrier to entry, which when crossed, creates a sense of community:

> Finally, as AS points out, we over time develop a Community that speaks a similar shop talk, so in gaining entrée to a Philosophy (or software package, or language), it's not just a set of skills you're mastering, but a set of contacts, relationships, peers, cronies, colleagues

> (living and dead). … So if you want to talk shop with the computational geometers, for example, it's probably a good idea to start learning the lingo, which includes translate, scale, vector, matrix, lattice, symmetry, rotation, symmetry group, polyhedron, vertex, edge, face, Euler, Gauss, Coxeter, Fuller... –KU, PCS-3, 5th

Herein lies the problem (and opportunity) of integrating computing with the traditional academic subjects. Learning the 'lingo' means belonging to the Python club, which is necessarily different from the Mathematics club, or the Science club, or the Humanities club, etc. And there is often some antipathy between members of certain clubs towards members of other clubs. But as computing becomes more and more integrated with each of these other disciplines, some computing *lingua franca* is sure to arise that allows communication between the clubs, somewhat the way Mathematics does currently with respect to the various branches of Science. But computing has the possibility of extending its reach to a far greater number of disciplines than Mathematics has, thus subsuming Mathematics within its embrace.

Conclusion of Python and computer science category

The main point here is that learning Python, in order to 'program to learn,' by necessity introduces one to a computing community that shares a common language, but that community should not be confused with the one known as Computer Science despite the fact that both primarily use computers and share some of the same terms and concepts. That is, Computer Science constitutes a field of study distinct from the goals of Computer Programming for Everybody:

> In its most general sense, computer science (CS) is the study of computation, both in hardware and in software. In practice, CS includes a variety of topics relating to computers, which range from the abstract analysis of algorithms to more concrete subjects like programming languages, software, and computer hardware. As a scientific discipline, it is a very different activity from computer programming and computer engineering, although the three are often confused.[36]

There are certainly areas of overlap between CS and CP4E, especially with respect to vocabulary and concepts, and certainly considerable dependency on the fruits of CS by CP4E, but the two are not synonymous and should be kept separate as they have different goals and different audiences. The primary skill being learned, computation, across all the disciplines involves designing artifacts in which the machine absorbs the costs of repetition. As the posters of this newsgroup have claimed, the beauty of Python as a vehicle for this design work is that its simple data structures and flow control syntax allows the programmer to readily express his or her concepts across all these disciplines. One's focus is not on Computer Science; one's focus is on expressing ideas, designs and artifacts in Mathematics or Physics or Art or Social Studies or English using a computing language that gets in one's way as little as possible.

4.4 Math-related

This category of thread was grouped by the following keyword phrases:

- Division
- Exponentiation
- Fractions
- Prime

[36] http://en2.wikipedia.org/wiki/Computer_science

- Precalculus
- Math, mathematics
- Algebra
- Calculating
- Rationals

There was an interesting struggle that occurred in these Math-related threads. That struggle is perhaps best characterized as one of hegemony, both of computing over math and math over computing. We will see this struggle played out over a few illustrative examples. In this section, we will discuss the Sieve of Eratosthenes, polynomials, division, and the difference between assignment and equality.

Sieve of Eratosthenes

The first case where the struggle occurs concerns an ancient algorithm for finding prime numbers, known as the Sieve of Eratosthenes. The algorithm can be expressed in English as follows:

```
1. Write down the numbers 1, 2, 3, ..., n. We will eliminate
   composites by marking them. Initially all numbers are
   unmarked.

2. Mark the number 1 as special (it is neither prime nor
   composite).

3. Set k=1. Until k exceeds or equals the square root of n do
   this:

   a. Find the first number in the list greater than k that
      has not been identified as composite. (The very first
      number so found is 2.) Call it m. Mark the numbers

      2m, 3m, 4m, ...

      as composite. (Thus in the first run we mark all even
      numbers greater than 2. In the second run we mark all
      multiples of 3 greater than 3.)

   b. m is a prime number. Put it on your list.
```

c. Set k=m and repeat.

4. Put the remaining unmarked numbers in the sequence on your list of prime numbers. (Alfeld, 2000)

One of the posters submitted a short program for finding primes using this algorithm. The initial version of it was straightforward, and followed the directions specified above faithfully. Subsequent posters offered suggestions for improving the code, explaining the rationale for each improvement or optimization. In the end, 17 lines of code were reduced to 8. Both versions are shown below:

```
def eratosthenes(n):
                             # a prime number sieve, thanks to Eratosthenes
                             # returns list of primes <= n
   cutoff = n ** 0.5         # 2nd root of n is cutoff
   sieve = [0, 0]+[1]*(n-1)  # [0 0 1 1 1...] with 1st 1 in position 2
   results = []              # empty list, will contain primes when done
   prime = 2                 # initial prime
   while prime <= cutoff:    # done when prime > 2nd root of n
      j = 2*prime            # jump to 2*prime (first multiple)
                             # turn sieve elements corresponding
                             # to multiples of prime from 1 to 0, up to n.
      while j <= n:
         sieve[j]=0
         j = j + prime
                             # scan for a 1 in sieve beyond highest prime so far
      prime = prime + sieve[prime+1:].index(1) + 1
                             # now translate remaining 1s into actual integers
   i=0
   for entry in sieve:       # step through all entries...
      if entry:              # if entry is 1 (i.e. true)
         results.append(i)   # add prime to list
      i=i+1
   return results            # return list of primes
```

```
def eratosthenes(n):
   """A prime number sieve, returns list of primes <= n
   Thanks to Eratosthenes for the algorithm, with
   streamlining by KY, JP and TP"""
   sieve = [0, 0] + [1] * n   # [0 0 1 1 1...]
   prime = 2                  # initial prime
   while prime**2 <= n:
       for i in range(prime**2, n+1, prime):
            sieve[i] = 0      # step through sieve by prime
       prime = prime+1 + sieve[prime+1:].index(1) # get next prime
                              # filter grabs corresponding range members
                              # only when sieve = 1
   return filter(lambda i, sieve=sieve: sieve[i], range(n+1))
```

(both methods submitted by KU, MTH-4, 1st & 13th)

What we do with either of these two code samples is strongly dependent upon the context in which they are being generated or used. If we are in a Mathematics class and we are studying the Sieve of Eratosthenes, we might initially be learning the algorithm with paper and pencil, or with other manipulatives representing integers from, say, 1 to 100 on a 10 by 10 grid. Once that appears to be understood, the teacher may want to reinforce the understanding of the algorithm by asking the students to write a program that can generate prime numbers far beyond what they can do with paper and pencil or manipulatives. And they are likely to come up with something closer to the first code listed above than the second one. Either way, now the students have generated a method that could go into a module for use later when a set of prime numbers is required for some other task. And, they likely have a deeper understanding of how Eratosthenes' Sieve actually works. Alternatively, they could explore other ways of generating a list of primes that is more efficient than this Sieve.

However, if the context is a Programming class, coming up with the first code as a way of generating primes is only the first step. The code performs the required steps, but we are interested in how to make the code 'sing'; that is, how to remove redundancies, inefficiencies, and bloat, in short, how to make it elegant. Having first established the connection between the English expression of the algorithm and the initial Python expression, we can now introduce new terms of the language and see how they work because they generate the same results in a more concise form. For example, the last line of the second program above uses ‘filter’ and ‘lambda’ to generate the final list of primes. Having built an understanding of the algorithm with the earlier, more naïve code, students are in position to understand the use of these two new terms since they accomplish the same result as the earlier code. In this context, the focus is not on the mathematics of the algorithm (although that is certainly foundational in this case) but on the programmatic constructs that accomplish the goal most elegantly. And, as in the Mathematics class, the topic could well shift to other algorithms that accomplish the same result of a list of prime numbers.

So we see the tension that ensues as programming is introduced into the Mathematics classroom. The focus could begin to shift away from purely mathematical considerations and into rhetorical concerns (how well an algorithm is expressed), taking time away from other mathematical topics. Let’s consider a few other examples of how this tension manifested itself.

Polynomials

One poster, KU, remarked at one point near the beginning of a thread, “A link between programming and algebra is in this concept of variables.” His goal was to create a module such that students could enter the coefficients of a polynomial, and a range over which the function could be evaluated. For example, one might enter:

```
>>> f = poly(2,3,1)
>>> f(4)
45
```

where the first line assigns the expression “$2x^2 + 3x + 1$” to the variable ‘f’. Then ‘f’ is given the value ‘4’ and the equation resolves to ‘45’. If one gave the equation ‘f’ a series of values, a list of results would be returned, which could then, presumably, be graphed. As the thread progressed, the function turned into a class that not only returned a numerical result if given a numeric argument, but could also return a symbolic expression, if given another function as an argument. For example, the ‘Poly’ class could be used as follows:

```
>>> from polynomial import *
>>> f = Poly([1,-3,2])
>>> g = Poly([1,3,0])
>>> f
x**2 - 3*x + 2
>>> g
x**2 + 3*x
>>> f(g)  # <--- symbolic composition
x**4 + 6*x**3 + 6*x**2 - 9*x + 2
>>> g(f)  # <--- symbolic composition
x**4 - 6*x**3 + 16*x**2 - 21*x + 10
>>> f(g(10))  # <--- or you can do numeric evaluation
16512
>>> g(f(10))
5400              # –KU, MTH-8, 15th
```

On one level, the code for the ‘polynomial’ module[37] can be used as a calculator, like it is in this example, much as a graphing calculator can be used, where the module ‘polynomial’ is as much a black box as a calculator. However, on a deeper level, the students might be asked to create such a

[37] See http://www.4dsolutions.net/ocn/

module themselves, rendering the box's opacity more translucent. Of course, the code for working with polynomials is more involved than the code for the Sieve of Eratosthenes. However, writing it represents an opportunity to really get to know how polynomials work (especially if the code graphed the results) as well as getting to know how Python works.

Both of these examples highlight the tradeoffs involved with introducing programming into the Mathematics classroom. Using prewritten code students can quickly explore a range of polynomials and discover the resultant curves when more powers are added or coefficients are altered in controlled patterns. They can do the same with graphing calculators. But either way, they are still dependent on a black box. However, by writing their own module, they not only gain greater mastery over the programming language used, they also come to understand the subject matter more deeply. But of course, such a journey takes time and the Mathematics instructor may be loathe to take it, perhaps perceiving it as an unnecessary detour (or dead end). On the other hand, the Programming instructor may well see this mathematic subject matter as fertile material for instruction (construction), seeing not a detour, but rather an opportunity for synergy with the Math classes the students are already taking.

Division

The struggle between Math and Programming exhibited itself prominently in another sense that manifested itself in a couple of different guises. These issues came up for beginners because we all (who have gone through school without any programming experience) have certain expectations based on the Mathematics training we received. These expectations clashed with Python most loudly in the area of division. If we divide, say, '6' by '2', we get '3', and this causes no problems for us or for the computer because '3' is in the same set of numbers (integers) as '6' and '2'. The same harmony prevails with '6 + 2', '6 – 2' and '6 * 2' because all of the results belong to the set of integers. However, if we divide '3' by '2' in Math class, we get '1.5' which causes *us* no surprise (anymore; maybe it did when we first encountered it). But Python behaves differently:

```
>>> 3/2
1
```

What Python is doing is returning the integer part of the answer (known as the 'floor'). This was done with the principle of least surprise in mind: operations involving integers should return integers. This always happens for addition, subtraction and multiplication, so it causes no surprise, but not always for division, so returning an integer when we expect a real number (because of what we learned in Mathematics) does cause a surprise. Python discards the decimal portion of the answer in order to return an integer[38]. If you want the decimal portion, then the division has to involve a decimal to begin with:

[38] Python includes a function which returns both the integer and remainder of a division:
```
>>> divmod(7,2)
(3, 1)
```

```
>>> 3./2
1.5
```

Notice the decimal point in the numerator; this tells Python that it isn't an integer (it's a real) and then the result can be a real (known as a floating point number in programming).

However, this original decision in the Python language was flawed because it violated the principle of least surprise in a different way:

> I personally don't think the primary reason for changing the behavior
> of / was that it confused newbies. –KU

Correct, it has nothing to do with that (even though newbie confusion led me to first see the problem). It has to do with substitutability of equal values with different types. When a==A and b==B, then a+b should be == A+B, at least within reasonable precision. –GvR, MTH-13, 4th

So a quick session with the interactive interpreter shows us the problem:

```
>>> a = 9/10          #assignment with integer values
>>> a                 #what is a?
0                     #returns the floor
>>> b = 7/10          #assignment with an integer value
>>> b                 #what is b?
0                     #returns the floor
>>> a + b             #what is the sum?
0
>>> A = 9./10         #assignment with a floating point value
>>> A                 #what is A?
0.90000000000000002
>>> B = 7./10         #assignment with a floating point value
>>> B                 #what is B?
0.69999999999999996
>>> A + B             #what is the sum?
1.6000000000000001
>>> a + b == A + B    #are the sums the same?
False
```

This illustrates the real problem with division as it was originally implemented in Python.

During the time of the messages in this study, the Python language was altered to accommodate a different behavior. But it had to be altered in such a way that it would be backwardly compatible with existing code that might rely on integer division returning integers instead of reals. So it was declared that a future version of Python will in fact return a real number when two integers are divided, and a new operator '//' was defined to perform the previous behavior (returning an integer, the floor). But how does that help the current Math teacher who wants division to return reals now? By importing a special module at the beginning of a script or interactive session, the new behavior may be invoked:

```
>>> from __future__ import division
>>> 3/2
1.5
>>> 3//2
1
```

And in a future version of Python, this will be the default behavior, so the import statement will be unnecessary.

Division was contentious for another reason:

```
>>> from __future__ import division
>>> 7/3
2.3333333333333335
```

Notice the '5' at the end. Doing this calculation by hand would generate a series of 3's for as long as we cared to calculate; no 5's should ever appear in the quotient. The problem arises because when we do it by hand, we are doing the calculation in base 10. The computer, however, is doing its calculations in base 2, and when it converts its answer to a decimal representation, there are ‘rounding errors’. This is often a source of consternation and confusion for beginners, but in this case, it is a property of computers themselves and there is no ‘from __future__ import exact’ to be had:

> Although there are infinitely many real numbers, a computer can represent only a finite number of them. Computers represent real numbers as binary floating-point numbers. Binary floating-point numbers can represent real numbers exactly in relatively few cases; in all other cases the representation is approximate. For example, 1/2 (0.5 in decimal) can be represented exactly in binary as 0.1. Other real numbers that can be represented exactly in decimal have repeating digits in binary and hence cannot be represented exactly. For example, 1/10, or decimal 0.1 exactly, is 0.000110011… in binary. Errors of this kind are unavoidable in any computer approximation of real numbers. Because of these errors, sums of fractions are often slightly incorrect. For example, 4/3 – 5/6 is not exactly equal to 1/2 on any computer, even on computers that use IEEE standard arithmetic. (Apple Computer, 1994)

```
>>> from __future__ import division
>>> (4/3)-(5/6)
0.49999999999999989
```

(See also the Python Tutorial for a thorough explanation.[39]) So, division in Python creates issues with respect to the mathematic expectations we form in school. But these are small issues that programmers easily learn to deal with.

Assignment and equality

The last main math-related issue that posters talked about concerned the equals (=) sign. Python, as well as almost all other programming languages, distinguishes between two different operations, assignment and

[39] <http://www.python.org/doc/current/tut/node14.html>

equality, through notation. Assignment is the process of assigning an expression to a variable, as in:

```
Circumference = 2 * 3.14 * radius
```

Here, the variable ‘Circumference’ is assigned the product of 2 times 3.14 times whatever the ‘radius’ is, and this assigning is performed via the equals sign. However, sometimes we need to test whether an expression on the left hand side is equal to an expression on the right hand side. Normally, we would expect to use the equals sign, but that is already being used for assigning, so Python (and many other programming languages) use a double equals (==) sign. We can see the difference in action:

```
>>> x=3*4       #assignment
>>> x           #what is x?
12
>>> y=2*5       #assignment
>>> y           #what is y?
10
>>> x==y        #is x equal to y?
False
>>> x==y+2      #now are they equal?
True
>>> x=y         #now x is assigned whatever has been assigned to y
>>> x           #what is x?
10
>>> y           #what is y?
10
>>> x==y        #now are they the same?
True
```

This may not be too confusing while reading the above snippet, but for beginners who are used to the equals sign meaning equality, the double equals sign notation takes some getting used to. Two of the posters described the situation like this:

Note that math books sometimes use the equal sign for assignment, and sometimes to assert equivalence. We're supposed to know the difference from context. Parsers don't like ‘context’ so much—better to be explicit. –KU, PCS-8, 4th

---

There is no logic to it.

When we started coding programming languages, “=” was in the character set and it was used for assignment. It started with Fortran and earlier efforts at programming languages. Since Fortran didn't have relational operators in the first version, there was no problem. In later versions, .EQ. was used for the relational operator, allowing Fortran to remain expressible in the original 48-character set.

When there were relational operators and they needed to be different, the conventions of “:=” and “=” (Algol family) and “=” and “==” (the Ratfor family) arose. The use of := (and =:) to indicate a composed arrow just didn't set well with some people. I've always liked it myself. There may be some odd connection with “:” not being available in all character sets at the time these practices were being worked out.

In some languages where there is no possible confusion of assignment and the equality relational operator, “=” is used both ways.

It is arbitrary. Simply arbitrary. Both operations are needed, and their symbols usually need to be distinct. That's the whole deal. Basically, “==” is now the prevailing custom for the equality relational operator in programming languages.

Since mathematics doesn't have an assignment operator, and mathematicians are willing to use other symbols (e.g., arrows) when needed, we are stuck with this odd cross-over dissonance. –DH, PCS-8, 5th

---

Conclusion of math-related category

We have seen two sources of confusion for the beginning programmer specifically derived from mathematic notational expectations: ‘/’ and ‘=’. Teachers need to be aware of these and make sure that students understand the difference between the mathematical meaning and the programmatic meaning of these symbols. Furthermore, we have seen how using programming in a Mathematics class introduces new decisions for the teacher as to where to direct the class' attention. What algorithms should be studied? How concise should programs that implement those algorithms be? What traditional topics need to be sacrificed to make room for programming concepts? Is it more important to learn how to do, for example, polynomials by hand on paper, or to program a polynomial function that might take considerably more class time to debug? These hegemonic questions illustrate some of the issues that the posters on the list grappled with when considering the intersection of Mathematics and Programming.

4.5 Science-related

These threads were characterized by the following keyword phrases:

- Robotics
- Scientific
- Modeling
- Periodic table
- Physics

There were many fewer threads on using Python with science subjects than math subjects. There also didn't seem to be any reports of a struggle between programming and science, as far as overlapping notation, or nomenclature or concepts were concerned, as we saw with Mathematics. Among the few ideas discussed were creating and using a machine-readable version of the Periodic Table of Elements, creating a Tree of Life, and designing an Educational Robotics platform (ala Lego® MindStorms™).[40]

[40] <http://mindstorms.lego.com>

However, I'd like to discuss one of the most valuable reasons for including computer programming in the sciences, even though this reason wasn't specifically mentioned directly, but only alluded to indirectly by one of the posters.

Dynamic representations

We can think of a computer program as a representation. In the domain of science, this might be a representation of a cell or an ecosystem, a molecule or a biochemical pathway, a cloud or a climate, a species or a kingdom, but whatever the object being represented, it is going to be a different representation than one generated by an essay, or a cardboard poster, or a video. And having multiple representations of the same object is highly valuable with respect to expertise:

> Scientists are very skilled at flexibly and fluidly moving across multiple representations based on underlying principles. They use the features of various representations, individually and together, to think about the goals and strategies of their investigations and to negotiate a shared understanding of underlying entities and processes. (Kozma, 2003, p. 224)

The context for this quote was multiple representations (primarily visualizations) generated by science-specific computer applications, but the point still holds for a representation that is created by the students themselves through programming. Such results may not be quite as sophisticated as the ones obtained by professional visualization software, but being able to express the principles of the system being modeled in software certainly contributes to a student's overall understanding of the subject. DiSessa (2000) echoes this importance:

> Representations and externalizations have always been part of scientific thinking. We need to pay more attention to the fact that computation has now brought the possibility of a hugely expanded repertoire of media constructions that may express scientific ideas or become part of scientific thinking. The proof of the four-color theorem—that any map can be colored by at most four colors (with no two bordering countries having the same color)—required a computer program. Science used to be thought of as a scientist making a discovery (possibly in the presence of some technology) and explaining his or her results to others. Science should be thought of now as a scientist coming to think differently in the presence of his or her representational technology and putting others in contact with that technology so that they may also think differently with it. (p. 116-117)

And finally, Bruce Sherin (1996) examined in great detail the question of learning Physics using two different symbol systems, one by using algebraic notation and the other by using a programming language:

> Central to this endeavor is the notion that programming languages can be elevated to the status of bona fide representational systems for physics. ... A conclusion of this work is that algebra-physics can be characterized as a physics of balance and equilibrium, and programming-physics a physics of processes and causation. More generally, this work provides a theoretical and empirical basis for understanding how the use of particular symbol systems affects students' conceptualization. (p. 3)

Since math and physics are so closely related, it may well be that this same conclusion could be reached for Mathematics, that is, traditional-notation-mathematics characterized as a mathematics of balance and equilibrium, and programming-mathematics as a mathematics of processes and causation. However, reaching this conclusion must wait for a different dissertation!

Conclusion of science-related category

It is likely that the posters of this group addressed the issue of representations in science subjects, but such issues may have been raised in the context of threads dealing with graphics, which were excluded from the studied data. Or, this issue might have been raised in connection with a discussion of Wolfram's book, *A new kind of science*, which was released during the time covered by the data, but was excluded during the winnowing process. Whatever the cause, I feel that most posters would agree that programming computer representations of scientific phenomenon would constitute an appropriate example of learning programming in order to facilitate programming to learn.

4.6 Programming for Fun

This category consisted of the following keyword phrases:

- Kids
- Fun
- Playing cards
- Science expo
- Artists

In this section, we look at two main topics. The first one centers on using programming with multimedia artifacts. In the second one we explore the group's thoughts on 'why program?'

Despite the apparent levity in the title of this category, there was a surprising streak of seriousness that went to the heart of this dissertation's topic. There certainly was some fun in evidence too, most notably the thread called 'Cards 'n stuff' which demonstrated how elements of card playing, such as cards, suits, ranks, decks, shuffling, dealing and even a game of blackjack, could all be modeled (without a graphical user interface) using simple object oriented programming. There was also fun in some of the suggested small projects that students might undertake to gain confidence in their programming skills. One example was a simple guessing game where the computer randomly chooses a number between, say, 1 and 99 and the user tries to guess the number using only the feedback ‘too high’ or ‘too low’ from the program. Another suggestion was to take a phrase (‘hippopotamus anthropologist’ for example) and see how many words could be generated, either by contiguous letters in the phrase, or by jumping from letter to letter in the left-to-right direction.

Multimedia

One of the more enthusiastic threads began as a request for ways to teach and use Python programming in the service of creating multimedia artifacts:

> I teach multimedia and web design at an art school as part of a curriculum that includes neon, kinetics, holography, microcontrollers, digital imaging and video, and pretty much anything else that involves technology. A lot of the problems our students run across require programming of one kind or another, and this trend seems to be increasing. My courses concentrate on programming a good deal, using javascript, flash/actionscript, or director/lingo for websites, cd-roms, gallery installations, etc. However, a good part of every semester is taken up with basic programming concepts before we can get to the “good stuff”. I'd like to introduce a solid programming class at the foundation level, and python seems like an interesting possibility. I haven't learned it yet, but have been working in c++, java, perl, etc. for a while so hopefully I can get a handle on it by September <g>. The problem with the approach we've been using (javascript, lingo, or

> actionscript as an introduction) is that each of these languages has a lot of features and quirks which are very unique, which gets in the way of teaching general programming concepts. I'd like to use something that will let me teach core concepts quickly without getting stuck on too many language-specific details. At the same time, whatever I use will have to be something that students can use to produce interesting results quickly as well - since this is an art school, we're really interested in artistic applications of programming rather than, say, calculating compound interest or the traveling salesman problem. So, my question is - has anyone used python for teaching art students? Does this sound like a reasonable thing to do? –BC, PF-6, 1st

This generated numerous responses, including many suggestions of modules and packages that generate aural and visual output and can be run by Python. This quickly led to the realization that there was a collaborative book /CD-ROM opportunity here, variously titled "Programming for the Fun of It" and "Programming for Artists" which would detail the interactions needed to work with these various packages. An issue arose, however, centering on how much math would be necessary in such a project. A couple of the posters, having had considerable experience generating visual models of geometric objects, claimed that programming in multimedia art required a fairly sophisticated degree of mathematical understanding:

> Can one use the Python interface to Blender [a 3D graphics creation suite] in any significant way without a decent grasp of certain mathematical concepts - trig is basic to defining paths for animation, for example. Or defining shapes. Or, I would argue, in talking to Blender in any significant way at all.
>
> Given that one's interests are artistic at their core, why bother to learn programming in pursuit of those interests if not to have more control, to interface with available tools at a lower level, to remove barriers and limits set by others.
>
> At a very practical level, math proficiency becomes key and fundamental to removing such barriers. Can one do good ray tracing - true artwork -

> without being armed with the right mathematical concepts? One might argue yes, but a good artist it seems to me won't settle for work-arounds and canned tools to achieve effects he might want to achieve. He'll go straight at it, acquiring the full range of skills necessary.
>
> In about everything I've seen with computer generated graphics, if one wants to work with freedom, at a low level of interface to one's tools - that means a good degree of math proficiency in addition to the ability to write code, scripts, whatever.
>
> I for one both learned math to draw the pictures and drew pictures to learn the math, never really knowing or particularly caring which goal was primary. –AS, PF-6, 11th

However, other posters were much less enthusiastic about having to learn math to do art:

> I would like to extract the patterns within multimedia just as one might do for 'Math'. I am far from convinced that Math is central. Essential yes, but I follow the line that it is pattern recognition and structural [=clear] thinking which are the heart of CP4E. …
>
> I guess what I am trying to get at is that the basic constructs and habits of programming "for everyone" may not need to be particularly related to obviously Mathematical stuff.
>
> Such as:
>
> • opening and closing files,
> • navigating dicts, lists and data structures
> • parsing and formatting results
> • formats and protocols
> • passing parameters for API class methods
> • reading + writing code
> • piping, processing and linking
> • looping and conditionals
> • interfaces: what are they, how do they work?
> etc
>
> Literacy in these and more I consider almost completely un-mathematical, depending on your context. (If you want to get very mathematical with them you can.) ...
>
> Most 3D modeling and animation and graphics software offer sophisticated user interfaces, because most of the time those interfaces

> are much more efficient, 'readable' and better than miles of numbers in brackets and quotes. –JC, PF-6, 10th

As I read these and the other messages in the thread, it seemed that the real struggle was not about math and art, but about what really constituted programming. At what point does interacting with a computer application switch from *using* (reading-like) to *programming* (writing-like)? In the case of these multimedia applications, the separation is not so clear. The issue here is the same as was discussed in the second chapter: the distinction between 'procedural' and 'declarative' languages. Some posters were arguing that to be true to the spirit of 'programming for everybody', the product being discussed should fully utilize a Turing-complete, procedural language, like Python, rather than simply describing how to use various products' declarative APIs (application programming interfaces). Others were arguing that using such APIs *was* in fact, a form of programming, and had the additional benefit of being approachable to a larger number of people who might otherwise be put off by the low-level math needed to construct multimedia artifacts only using a procedural language without benefit of the slick graphical user interfaces the declarative language packages provided.

As it turned out, there was no agreement reached on the matter, but the lesson to be learned is that if one joins a collaborative effort such as the one described in this thread, it should be made clear to each of the collaborators which programming approach (procedural or declarative) is being followed.

A related issue came into focus with another thread notable for its depth of thoughtfulness.

The 'why' of programming

The thread began with a simple question:

> So, I've been working for awhile on the how of teaching programming to non-geeks. What I think is a more important question, is Why? Why should someone be interested in learning to program? I'm not talking about convincing them to give up their day job, or learn higher math, just to have an interest in occasional programming. Here's my first cut at some possible motivations, but I'm very interested in finding more:
>
> - Build problem-solving skills
> - Learn math
> - Create toys
> - Fix the software you use
> - Understand computers and the digital world better
> - Extend the software you use
> - Create tools you need. –DE, PF-2, 1st

An unexpected post that fits remarkably well with the multimedia thread discussed above soon followed:

> G.H. Hardy's book, "A Mathematician's Apology", covers some of the issues of teaching abstract concepts. Why do people do crossword problems? Crossword problems surely don't have any direct application, yet people derive satisfaction from solving a puzzle. Hardy says that people do mathematics, not because it's practical, but because it's beautiful --- that's his primary justification for mathematics.
>
> Likewise, I think people program, not only because it's useful, but because it's intellectually stimulating; there's something wonderfully _neat_ about seeing these processes run under our fingertips. All these other perks: improving one's employability, gaining problem-solving skills, are all secondary to the idea that programming is fun. –DY, PF-2, 3rd

The OP supported this idea by saying:

> I like this. David Gelertner writes about advances in computer science happening as a pursuit of beauty, and the whole pattern movement

> came about from a theory of architecture which strives for the "Quality without a Name," i.e., what distinguishes [code|buildings] that are *alive* from those which are not.
>
> Wonderful! –DE, PF-2, 5th

Later, in response to this post, 'truth' is brought into the conversation:

> I was in a conversation a while back where it was pointed out that the pursuit of science, generally, is one of beauty, and that it is primarily religion which pursues truth.
>
> That was very interesting for me. It gave me a new perspective on a famous quote from Albert Einstein and Leopold Infield (The Evolution of Physics, 1938): "Physical concepts are free creations of the human mind, and are not, however it may seem, uniquely determined by the external world." (p.31).
>
> It also clarified for me how often techies like myself lapse into religious debates, the tip-off being claims about the "best" programming language, development methodology, or operating-system model without any grounding in empirically confirmable values. It is useful to remind myself that it is all made up and some of it can be beautiful in its conceptual harmony and the utility that becomes available. It's just not the truth.[41] –DH, PF-2, 16th

The thread also took an interesting educational turn by pointing out how few teachers are *able* to use computers in their teaching:

> One reason people might take up programming:
>
> Because nothing else in the whole world will reliably do what you tell it to do, when you tell it to, if you tell it correctly. (This one's for parents and teachers.) Which leads to my next point.
>
> There is still (look, look! at how many schools and classrooms still are NOT using the web for publishing, even when they are well endowed with computing resources and internet access) an awful lot of resistance to using any technology (= hardware/software combination) in the classroom unless the teacher in charge of that classroom has mastered that particular technology. It is a control issue.

[41] As I read this I was reminded of the end of Keats' 'Ode on a Grecian Urn' where he writes 'Beauty is truth, truth beauty, that is all Ye know on earth, and all ye need to know.'

> Even with all sorts of resources out there on the web for the curious student/parent/teacher, you won't have programming in the classroom on a widespread level unless you turn the teachers into programmers. Or, to borrow Neal Stephenson's metaphor from In The Beginning Was the Command Line[42], you have to turn the teachers from Eloi (passive consumers of "we'll give you what we think you want" technology) into Morlocks (the ones who make the technology/write the code). –JC, PF-2, 7th

Which was quickly developed by the next poster:

> And it's not just that teachers are control freaks and won't let technology they haven't mastered take over for silly reasons. On the contrary, I think especially young students are in need of role model adults who aren't fazed by computers and know how to put them through their paces.
>
> You wouldn't trust a horse-riding academy where the trainers were afraid of the beasts. So we definitely want Morlocks, not Eloi, showing off computers in the classroom -- lest we get another generation of passive "you do it for me" types. Teachers are correct not to push computers into the limelight until they achieve Morlock status. As professional teachers, that's their job (to stand for mastery and competence in a particular field).
>
> So when I imagine a healthy use of computers in a classroom, it's not necessarily this situation where a teacher gets a CDROM going, and then walks away (computer as babysitter, a kind of boob tube, freeing the teacher to give his or her attention to other students).
>
> No, my image of healthy and proper Morlock role modeling is a teacher with a projected computer screen, writing interactive command lines in a kind of stream of consciousness way, talking and interacting, explaining, popping source code (some of it written by the teacher, some of it by students, some downloaded etc.)
>
> Of course too much sustained droning and watching someone else code is no fun -- that's just a mode the head Morlock should have mastered. Then the students get to turn to their own machines and teach it similar tricks, emboldened by the model of a teacher who has no fear. –KU, PF-2, 8th

[42] <http://www.cryptonomicon.com/beginning.html>

This was then followed up with a wonderful story by a high school teacher of Python exemplifying an answer to the question “why program?”:

> I like all of DE’s reasons for non-geeks learning to program, but I particularly like:
>
> 1. Build problem-solving skills
> 2. Create tools you need
>
> These two go really well together also. While I have no doubt that the logical reasoning skills that come from learning basic programming will be of general benefit to the programming student, the benefit is abstract and not immediate enough to be the *only* reason for doing it. I think Python's clear syntax, extensive libraries, and rapid development capabilities make it wonderfully suited to the other key reason: creating tools you need. This both gives immediacy to the learning and makes it fun!
>
> Last weekend my 5th grade son came to me and asked if I would randomly call out the five notes he was learning to play on the trumpet. I told him that I had a better idea. I would write a little program for him that he could use anytime he wanted. A few short minutes later I had the following:
>
> ```
> #!/usr/bin/python
> import time
> import random
>
> def play():
>     #[includes an improvement suggested by KU]
>     notes = ['C', 'D', 'E', 'F', 'G']
>     while 1:
>         print "Now play " + random.choice(notes) + " . . . ."
>         time.sleep(2)
> play()
> ```
>
> My son really liked this, and for the first time showed some interest in learning about Python. –JE, PF-2, 11th

Finally, there were a few other reasons given for ‘why learn to program’:

• Work more effectively with computer support personnel, system managers and programmers

• Understand the strengths and weaknesses of computers and software for supporting real-world tasks

People who do not become programmers benefit from programming experience because it gives them a more realistic view of how computers and “computer people” work. There are relatively few tasks today that don't involve computers in at least a peripheral way, so this experience is highly generalizable. –JH, PF-2, 6th

---

Actually, if this were a class I think the target audience would be at-risk teens, women re-entering the workforce and/or retraining for the tech world, and the economically disadvantaged (a group whose constituents change with geography). These are the folks with the most to gain by adding programming skills to their repertoire, and often the ones with the least inclination to learn programming having been told that it is hard, that you must be a genius, that it's boring, etc. –DE, PF-2, 13th

---

Because computers have changed everything we do, being able to program can make you a better photographer, teacher, artist, student, architect, broker, etc.

Note, the reverse is true also, that the real world is far more satisfying and stimulating, and by mixing previously unrelated ideas, often from widely disparate fields, we get true innovation. Which is an argument for programmers to learn painting, architecture, medicine, economics, education, etc. –DE, PF-2, 15th

---

Conclusion of programming for fun category

Altogether, I found this to be one of the most stimulating and satisfying threads of all the ones I read. I, too, find a kind of poetic/mathematic beauty to a well-written program which, in many ways, is reason enough to learn how to program. But these posters also found real-world grounds for non-professional programmers to learn programming, for both practical and aesthetic reasons.

Let me end this ‘Programming for fun’ section with two quotes that point to one potential challenge awaiting the computer game designer. One comes from an edu-sig poster, the other from an interview of Seymour Papert by Dan Schwartz (1999):

> I've written a functioning Reversi game that provides a framework for defining new computer player strategies. It's meant as a general programming project and an introduction to AI [artificial intelligence]. Reversi is one of the first games I programmed (in high school, in BASIC on an Apple II+). I rewrote this game several times in various languages on various computers as I learned more about programming and AI. Most high school students would probably rather learn to program arcade games and graphics. I think that's fine and I'm interested in making some materials that could be used to that end. However, for the right student, I think AI programming can be a more interesting programming problem since it causes you to think about how humans think, how computers think, how they are related and how they are different. –BB, ED-14, 11th

> It's one thing for a child to play a computer game; it's another thing altogether for a child to build his or her own game. And this, according to Papert, is where the computer's true power as an educational medium lies—in the ability to facilitate and extend children's awesome natural ability and drive to construct, hypothesize, explore, experiment, evaluate, draw conclusions—in short to learn—all by themselves.

Finally, for teachers who would like to incorporate game design into their programming classroom, one site dedicated to using Python to create games is Pygame.[43]

4.7 Miscellaneous and Unknown

The miscellaneous and unknown categories contained a variety of subject headings and key phrases. The miscellaneous category consisted of threads where the subject line was indicative of its contents, but didn't fit into any of the other existing categories, nor did they form their own

[43] http://pygame.org

categories if considered with any of the other miscellaneous threads. In other words, they were single-thread categories. The unknown category simply consisted of threads where the subject line was *not* indicative of its contents. In both cases they are not defined by any list of specific keyword phrases; a listing of the subject lines of these threads (along with all the other categories) can be found in Appendix A.

We examine two topics in this section. Having just finished 'the why of programming,' we now take a look at 'the how of programming'. And, closely related, we explore the group's thoughts on motivation in programming.

The 'how' of programming

To a large extent the threads in these two categories were, in fact, irrelevant to this dissertation. In a few cases, the threads contributed to topics covered earlier in this chapter, and were included there. However, there was one topic discussed that complements the rest of this chapter and serves a fitting ending. In the words of KO, who distinguished how to code from how to program:

> I see programming and thus to some extent software engineering as more of an art field than a science—after all, you are simply expressing your ideas to a computer. The "science" part is that you are given a solid, specific set of ways of expressing your ideas, and that it resembles a mathematical language to some extent, but that really hides the fact that you are still expressing ideas and can be as creative as you want in devising a solution to a problem. …
>
> What is taught in class for most computer science programs is a whole lot of math, a lot of programming syntax, and all sorts of discussions of abstract programming concepts like objects, recursion, etc. While that's all nice and good, learning to code is not learning how to program. … I

> think what people are asking for are the “laws” of how to program - those few core ideas that make one master the 'art' of programming. –KO, UNK-16, 12th

So, if learning to program is not synonymous with learning to code (learning a language’s syntax), then what is it? For some, it is a kind of problem solving:

> It has been of interest to me that in various readings that I have started over the last few months, there have been several references from different sources about how computer programming is an extension of the cognitive psychology of problem solving. I would extend this and suggest that computer programming is similar to, or a branch of, epistemology: it concerns the construction and negotiation of problem frames and solutions to those problems, and really underscores the processes by which we organize the world, its data, and - to top it off - how we organize our thinking processes (analysis, hypothesis, antithesis, synthesis - the usual Aristotelian process). From this perspective, I think that teaching school kids how to program is not only great vis-à-vis the development of computer/digital savvy (as was suggested in the edu-sig docs), but also a significant step forward into advancing structured cognitive training for kids (and adults too) to assist them in the analysis of problems, the proposition and testing of solutions and a meta-analytic perspective in terms of the construction, relevance, and flow of data/cognitive constructs: computer science meets George Kelly in education. –AH, UNK-20, 4th

For others, a way of seeing:

> They [computer programs] are all representations of something in the real world, communicated in a form that best fits a particular forum. I would expect a GUT [Grand Unified Theory] of Programming to explain how one takes an arbitrary phenomena and represents it in an arbitrary computing language. ...
>
> The problem is the translation from real to representation, it depends on what aspects of the phenomena are deemed important enough to translate, which depends on the forum the translation is targeted at, and those involve value judgments. –BS, UNK-16, 9th

And of course, what is seen affects the seer, in a quantum-mechanical understanding of the ‘life imitates art’ argument:

> There's a question here as to whether programming mirrors our thought process or whether, after we program for awhile, our thought process starts to take on some features of programming. Certainly it's a great source of metaphors.
>
> The books usually say “objects” (in the programmed sense) are metaphors for objects in the real world i.e. the problem space is modeled by the solution space in terms of objects. But it works the other way too: getting used to thinking of composition and inheritance affects the way you see the real world (suddenly, that cell phone “really is” a subclass of the more generic telephone class, and so on).
>
> Good thing my dog here overrides some of those wolf methods, with more domesticated versions. –KU, UNK-20, 5th

> I come from a psych and philosophy background, and this taps into a whole bunch of issues from those fields - the notions of 'reality' and the constructions of reality, the idea/image or simulacra and the whole question of representation. ... The lens of language (non-computing) informs the way that we perceive and interact with our worlds (introducing the ideas of discourse and interpretive frames, for example), so it makes good sense that when one learns computing language as a means of representing 'reality' (putting to one side the important questions about that particular concept!), the 'world' begins to 'resemble' (or perhaps more germanely - becomes 're-assembled'!) according to the codes of reference of that language. In philosophy, one of the issues has been the extent of interleaving between concepts and that which they are said to represent: when we have a concept for microscopic particles (e.g. viruses or bacteria) it is easier to 'see' them with a microscope. The beauty of a well-written program does not become apparent unless one knows what one is looking at. Before Object Oriented Programming, the notion of using objects was foreign to programmers; once upon a time, it was unheard of not to use 'goto' in a program; now it is almost bad manners to use 'goto'. –AH, UNK-20, 13th

And it is the perceptions and concepts of the liberally educated (who also happen to be computer-savvy) and who have the skills to represent those perceptions and concepts in code that become prized in the job market:

> I heard a presentation by a former Disney recruiter on the radio recently and he was talking about how his industry couldn't get enough

> of the “hybrid artist” type -- those with good grounding in one or more of the arts, but with enough technical background to take to a state-of-the-art studio like fish to water. He stressed that he was talking about the humanities, about well-rounded individuals with a lot of appreciation for culture, contemporary as well as past. His industry wants comprehensivists, in other words. I consider this a helpful cue. –KU, UNK-4, 3rd

So learning to program is more than learning to code. A perceptual shift occurs when the budding programmer begins to see patterns in the world, and starts to understand how an algorithm might represent those patterns. Initially these patterns are explicit, both spatial ones, like checkerboards and picket fences, and temporal ones, like rhythms and seasons. However, learning to program also means learning to see the implicit patterns of constancy and changeability in whatever phenomenon is being modeled, whether linguistic probabilities of Markov chains,[44] or annual travel routes of migratory species, or workflows in the workplace. Of course, this change in perception is what education is about, not just programming; it’s just that the mode of expression is different. Instead of essays or equations, students express their perceptions in computer programs, and the greater that expressivity, the deeper the perception of patterns becomes.

Motivation

But how do educators help their students to ‘see’ these patterns in the world around them? One suggestion came in the context of integrating programming into Mathematics classes:

[44] A Markov chain is a process that consists of a finite number of states and some known probabilities of moving from one state to another.

I was a real math head when I was younger, but a few years of high school education educated that love right out of me. Later on in college I revisited Calculus and had a grand time. The book's method was pretty much the same. It didn't really work for me. But the instructor was great. He told us stories about the formulas, where they came from, who the people were that discovered them. He walked us through how the formulas were discovered. He set things in a historical context.

Well, anyway, inspired by what I might be able to do with NumPy [a Python package], I checked out a couple dozen books from the library on Math, looking to understand Matrices and Linear Algebra. Ugh! What a lot of boring material there is out there! Pretentious too, loaded with terms that they don't really bother to define. There is a horrible barrier to learning this stuff. But I have found a couple good books in there, but the point is probably that I am inspired to learn this. And because I am inspired, I will learn it.

Ah, but what inspires me, may not be what inspires you. Inspiring a love of math or a love of programming is not something that can be placed in any curriculum. It is a rare gift that some instructors have, not nearly enough. Some writers have it too, but they are even rarer. If you want children interested in math you can't just teach them formulas and tricks. You have to find what interests them, what they want and need to do, and encourage that growth. Assist them in their learning.

Inspired people with access to knowledge and people to assist them when they ask cannot be stopped from learning. They will seek out the maximum cognitive load. They will learn. Those with no interest in the material will learn what they need to get by, and no more. Dumbing math down from where it is only helps them get by with less. The problem is not that math is too hard, it is that most see it as uninspiring drill work, a lot of useless memorization of formulas they will never need to know. ...

I think there should be an immense initiative to train instructors in story telling, and give them access to great stories to tell. There should be summer story workshops for them. Let a good quarter of their time be spent telling stories, another part in constructing math labs, and math contests, I bet you would see an incredible increase in the math abilities of our children.

You want children to learn programming, learn to tell programming stories. Give them good programming tools and tutorials and set them loose to create their own projects. Have open source style competitions. Have them work not just in the schools, but in the real world too. If it is

> going to be a common literacy, it has to be very visible, and the benefits of knowing it have to be obvious, everything around us should remind us of those benefits. If not, it will remain a minority interest. –SF, UNK-4, 4th

This poster reminds us of the importance of motivation in learning, and one of the key strategies for inspiring students is telling stories. Since programming is so new relative to other school subjects, the pool from which stories might be drawn is much smaller; nevertheless, seeking out the ones that do exist and bringing them to life can be a central technique to teaching students how to program. Even better would be stories that come from personal experience. To gain those, teachers need to engage themselves in actual programming events situated in a context that is more meaningful than textbook exercises. Elsewhere, this same poster discusses motivation again:

> Finding what motivates can be the hard part.
>
> One of the things we discussed in an Art of Mentoring class I took from the Wilderness Awareness School was how we learn best when we are excited, particularly when our adrenalin is flowing. This is one reason memories of frightening events are so vivid. But you don't have to frighten your students into learning, any excitement will do.
>
> One key to mentoring is to teach to a student's passions, whatever they are. What we learned in Art of Mentoring was how to profile students, and how to use those profiles to manipulate them into learning. My teachers called this Coyote Teaching, because it involves some trickery. After you find what hooks them, you draw that out, don't give them what they want right away, but dangle it before them some, dragging them through things they might otherwise have avoided. Excite their passions and manipulate them into the right situations and they will pretty much learn on their own. My teachers called this “creating a vacuum,” the empty gap between what they know and what they want to know.

> Coyote teaching works best one on one, but you could also use it in a small classroom if you can figure out some common passions your students have. Individual projects will also give you an opportunity to tailor things to each student. –SF, UNK-15, 7th

This really addresses the art of teaching, where teachers discern each student's passion and use that to motivate them, perhaps through story-telling, perhaps through other inspirational techniques. Teachers mustn't assume that students are necessarily motivated to learn how to program *per se*, but should instead assume that something unique makes each student tick, and find out what that something is and tap it in the service of a programming project.

Conclusion of miscellaneous and unknown categories

A couple of interesting issues arose in these threads. First, we saw that programming is more than just learning how to code; it also involves a way of seeing patterns in the world that are amenable to computations. This was earlier characterized as algorithmic thinking and involves creating the algorithms by which a computation can be rendered. We also saw that teachers need to be sensitive to individual differences in interests and passions and should try to use those differences as a motivating factor in designing programming projects and assignments for their students.

4.8 Summary

In this chapter we discussed the topics under each category that contribute to our understanding of teaching Python in the secondary classroom. We began by considering the premise that computer programming is a component of computer literacy that ‘everybody’ should know, and discovered three suggested contexts for learning programming: in a programming course, in other subject matter courses where programming is integrated into the curriculum, and after-school clubs. This led us to consider a variety of factors in the education category beginning with the struggle educators face trying to achieve any sort of integration of programming in an existing curriculum. We also noted the need for curriculum materials, and how that need was being met by some of the posters on the list, and the emphasis placed on distinguishing and using both the interactive shell and saved scripts. And great attention was paid to teaching methodologies and concerns that posters found appropriate when teaching programming such as involving the student’s interests, various programming styles, encouraging planning, and working in groups.

We then discussed the issues encountered when considering computer programming for everybody in the context of Computer Science, Mathematics and Science. We discovered that despite much overlap among these subject areas, computer programming for everybody constitutes a distinct subject area in its own right, with its own set of vocabulary terms and concepts, and that these can occasionally conflict with terms and concepts in other subjects, especially Mathematics, creating confusion in, and resistance from established teachers. We also determined that posters generally agreed that these obstacles were not insurmountable but could be overcome by demonstrating the advantages of fostering programming skills for achieving curriculum benchmarks.

Finally, we positioned computer programming as being primarily a creative activity by emphasizing its utility in artistic endeavors. We also considered both the question of ‘why program’ and issues of motivation, and the question of ‘how to program’ beyond simply learning a language’s syntax by connecting perception of patterns with algorithmic expressivity.

Having presented the results from the data analysis, we turn now to the concluding chapter to consider the results of using the methods and procedures described in the previous chapter, and conclusions drawn from the results presented in this chapter.

CHAPTER 5

CONCLUSION

It has often been said that a person does not really understand something until he teaches it to someone else. Actually a person does not really understand something until he can teach it to a computer, i.e., express it as an algorithm.

–Donald E. Knuth

5.0 Introduction

This, the last chapter, provides us with the opportunity to reflect on the preceding research and consider what has been learned, as well as offer a few possible trajectories for further research, and end with how these results might be seen to fit into a larger context of education beyond educational computer programming. We turn first to the findings on the methods and procedures used in this dissertation beginning with a summary of those procedures and followed by my thoughts on them. We then turn to three main conclusions drawn from the content results, followed by suggestions for future research. We then end this dissertation with a few thoughts centered on the notion of convivial programming.

5.1 Method Findings

The data used in this research was in a format known as unstructured text. Each data record originated as an email message, so it was accompanied by limited meta-information such as name, date, time, subject and email address, but the body of the message had no additional information other than the text itself (and any quoted text that might have been included to establish context for the reader). Educational research data is often in an unstructured text format, for example, interview transcripts, talk-aloud transcripts, instant messaging transcripts, online news articles, weblog entries, etc. So, let us review the procedures that were followed in working with this newsgroup archive and discuss those aspects that proved beneficial, and those that were problematic in the hope that these procedures might be usefully employed by future researchers having to deal with large, unstructured corpora.

Use a database

The most prominent feature of the data downloaded from the edu-sig website was its bulk. There were more than 2700 messages archived over the 3+ years. Thus, my most immediate concern was to select those messages that were most likely to contribute to the topic. Before that, however, the data had to be assembled into a manageable form.

The data's original incarnation was one large, single file, in standard mailbox (.mbox) format. (Other researchers working with unstructured text might begin with a folder full of text files, or perhaps a large server logfile, or some other format.) To work with the data, it is virtually imperative that it be stored in a database, preferably a relational one. This enables several different tasks that will prove invaluable when working with and manipulating the data:

- Sorting the data
- Searching the data
- Selecting the data
- Counting the data
- Importing and exporting the data
- Randomizing the data

And if the database supports relationships, you can generate tables that isolate specific features of the data. For example, with the present data, I generated both a ‘threads’ table and a ‘posters’ table that brought the data together in specific configurations (by thread and by poster) that facilitated analysis. Sorting, searching and counting database records is important for generating aggregate statistics that profile the data. Importing and exporting is especially important if you work with more than one database; for example a relational database for storing the data, and a qualitative analysis package for fine-grained coding of the data. And finally, randomizing the data proved important for the next step in data reduction.

Get to know the data

Having gotten the data into a database, two steps awaited, but it was not clear which should go first. One step was to reduce the data, since there was too much to read in its entirety, and the other step was to get to know the data by reading a random sample. My preference was to get to know the data first, since I thought that knowing a bit about it might inform the subsequent data reduction step. So, of the original 2757 messages in the database, I read a 10% random sample.

In this particular case, there was an additional consideration: the messages needed to be grouped by subject, that is, threaded. Other data would use different grouping criteria depending on the research topic, such as poster id, date range, messages containing particular keywords, or other message classification criteria such as thread initiator, thread response, type of response (for example, supportive, alternative, hostile, rejoinder or other), or containing other kinds of speech acts. I split my 10% sampling into two groups of 5%: one sample drawn from the set of unthreaded messages, and the other sample drawn from the set of threaded messages. I then read those messages, beginning with the unthreaded samples, and finishing with the threaded samples, which were read thread by thread in chronological order to maintain continuity and context. While reading, I noted the topics and themes that each message addressed, irrespective of my dissertation's topic. This way I had a global overview of the range of topics being discussed and developed an understanding of what to expect in the data. This familiarity proved crucial during the later categorizing step.

Discard irrelevancies

This step would be difficult to generalize to other research data, so I'll simply recount what heuristic was followed in the present case. It was clear that I would be excluding data at the thread level, not the message level. I was searching for discussions (threads) that involved more than a few posters lasting more than a few messages, so excluding by thread seemed most appropriate for this study.

I then made two measures on each thread: message length and poster diversity. After calculating the mean for each measure, I excluded those threads with a message length less than the mean (5 in this study) or less than the mean number of different posters (3) assuming that whatever threads were left would have a reasonable chance of containing interesting discussions. This reduced the number of threads by 80%, from 757 to 150.

Classify the threads

My next step was to classify the 150 threads based on the subject lines. Keep in mind that I had already given a sample of the data a close reading and thus had an expectation of what a subject line might imply as to the contents of the threads. So, I was able to group the threads into readily identifiable categories based on the subject lines leaving me with 150 categorized 'black boxes' of text with only a subject line visible to (possibly) indicate each of their contents. At this point I felt that simply excluding certain categories (graphics, editors, etc.) would be sufficient to weed out most of the remaining irrelevancies, but I also felt that having more of a glimpse into the contents of each thread (beyond the subject line) would be welcome for a couple of reasons: One, it would allow me to choose which threads in each category to keep or discard, and two, I could feel more certain that I wasn't inadvertently discarding an important thread by discarding a whole category of threads.

Characterize the data

What I needed was some kind of automatic text classifier system. There are a fair number of such systems available, but most require a training period where human classified text samples are given to the classifier which can then proceed to classify new material. For some research purposes, this is probably preferable, especially if there is prior, already classified data available. However, I was more interested in finding a technique that wouldn't require prior knowledge of the textual domain, could be used on unstructured text data, was freely available and could run on my machine. PhraseRate fulfilled those conditions (and was the only application, other than text graphs, I found that did). With it, I was able to generate a list of 10-20 keyword phrases that characterized each thread, allowing a dimly lit glimpse into what the posters might be discussing.

I also informally tested the reliability of the PhraseRate results by comparing its output with that of another technique, text graphing. I found enough of the PhraseRate keyphrases in the nodes of the corresponding text graphs to feel confidence in its results. (I discuss areas of future research for these two methods shortly.) Thus, using the subject line of the thread, the list of key phrases generated by PhraseRate, and the category in which the thread had been placed, I was able to reduce the number of relevant threads about 30%, from 150 to 108, by discarding threads on topics not central to this study.

Thoughts on the procedures

Qualitative researchers who have an abundance of unstructured text data to process have to find ways of reducing their datasets to manageable and germane levels. The procedures developed and described in this study may well be applicable to other researchers in similar situations. However, a few caveats are in order.

Procedure steps

First, I believe that two of the steps were done in the reverse order of what would be optimal. Specifically, it makes more sense to first *characterize* the data with an automatic text classifier, and remove the irrelevant threads, and then *classify* the remaining threads. The main reason for this is that the text characterization data (e.g. keyword phrases) can then be used while classifying the remaining data. So a better sequence would be:

1. Put the data into a database
2. Read a random, representative sample of the data
3. Develop and apply heuristics to remove low value data
4. Apply automatic text classification algorithms to further extract the high value data
5. Categorize the high value data into analyzable chunks

It may be possible to automate the last step (for content-based analyses rather than communicative-based ones), that is, given a set of keyword phrases for each thread, one might do a statistical cluster analysis on that metadata to clump similar threads into categories. This might prove especially attractive for larger datasets. However, in the present study, grouping by hand provided satisfactory results.

Initial heuristic

Second, I was especially pleased with the initial data reduction step, that is, selecting threads with message lengths and poster diversity greater than the mean. This particular heuristic proved remarkably effective at isolating the thoughtful and sustained discussions in the data that brought out a variety of viewpoints and opinions among the posters. In retrospect, I was, perhaps, a bit too conservative in its use by erring on the side of including threads on the shorter end of the spectrum. As it turned out, most of the really interesting data was located in the longer threads, presumably because posters had time to develop their arguments through more give and take.

Subject headings

Third, using subject headings to categorize the threads proved to be fairly reliable. Although I occasionally came across a thread (say, in the education category) that really belonged in a different category, for the most part, thread categories seemed to ‘hang together’ rather nicely. At least some of this can be attributed to gaining a ‘feel’ for the data from the initial random sample reading, but how much this contributed, I could only guess. Subject lines, even when categorized correctly, can also be misleading in another sense: sometimes an initial posting was followed up by an off-topic remark that then became the main focus of the thread. Roughly half the time this would be fortuitous, as the new topic would prove to be more pertinent to the research, while the other half the new topic would unfortunately wander into relative irrelevancy.

PhraseRate and TextGraphs

Fourth, although the PhraseRate and TextGraph techniques afforded a view of the contents of unread threads and assisted in the winnowing of the data, I felt there was room for improvement in each method. One limitation that was visually obvious in the TextGraph technique (and presumably applies to the PhraseRate results, although this wasn't tested) was that if the topics of the text being graphed were too spread out, or had too little focus, the resulting graph lost all meaningful structure. This was a surprising discovery because my earlier research with Text Graphs (Miller, 1996) only explored text that had well-defined foci, and I was unaware of this limitation.

During the course of this investigation, I generated text graphs for each of the eight *categories*, (each of which consisted of from 5 to 33 *threads*, each of which consisted of from 5 to 42 *messages*.) Most of these graphs had an extremely simple starburst pattern where one or two central nodes were connected to all or most of the other nodes. There was simply too much text focused on a multitude of topics to render a meaningful structure to the category. The smaller categories did not exhibit the starburst pattern, but instead, formed a network of haphazard links among the nodes that had no discernable structure (where 'structure' can be seen as central nodes surrounded by peripheral nodes). On the other hand, text graphs of individual threads had remarkably well-defined structures, presumably because the text was focused on only a few defined topics. Thus, the Text Graph technique as it currently stands, is mostly useful for generating a profile of text that has inherent internal structure.

One way to overcome this limitation would be to allow the researcher to choose which terms were to be graphed. As it is now, the top 50 or 75 most-often occurring terms are automatically chosen for graphing; these tend to overshadow the more interesting research terms as the amount of text increases.

It also isn't clear to me which method, PhraseRate or TextGraphs is the more efficacious for classifying text. I would like to see a study where the same set of threads was classified four different ways: one by people who actually read the threads before classifying them, another by people who only had the subject line and the PhraseRate results, third, by people who only had the subject line and the text graphs, and fourth, by people who only had the subject lines. Then we might see how much those particular algorithms contribute to classifying unread, unstructured textual data. My feeling is that an algorithmic blend of the two methods might be ideal, where the two-dimensionality of the text graph coupled with the succinctness of the keyword phrase results might generate the most robust ‘snapshot’ of the data. This echoes the direction of research being pursued elsewhere as reported in The Economist (2003).

Newsgroups as a data source

Finally, the question arises as to the value of the source of the data, that is, what are the strengths and limitations of analyzing data that originates with online newsgroups such as the Python edu-sig for educational research purposes? One major strength of this type of data is that many of the participants are well-versed both in computing issues and classroom issues, thus the perspectives they bring are informed by both sets of expertise. Not all participants are so informed; however, the diversity of experiences provided by other participants further adds to the richness and complexity of the ensuing discussions. Thus the potential, in such groups, for discussions actually containing data that proves useful to the researcher, I believe, is high. The main challenge in dealing with such data is sifting through the ‘message-silt’ in order to retrieve the useful nuggets of information. Also, these nuggets, instead of being used directly as I did, could instead be used by researchers to familiarize themselves with a group’s ideas and opinions in order to construct more complete alternative instruments for data collection, such as interview protocols, questionnaires, survey instruments, or analytical frameworks for analyzing related data.

Another strength that researching relevant newsgroups provides is access to motivated educators. That is, one can fairly readily identify the posters who are actively teaching an area of interest, or using a method of interest, and can use email to contact those educators to ascertain their interest in participating in further research, or simply initiate discussion to obtain more detailed information than what is being provided in the newsgroup.

Nevertheless, there are some limitations to be aware of. Perhaps the most prominent is the 'noise' that many newsgroups generate, that is, I found there were some social interactions that did not materially contribute to the group's ostensible charter. Personal feelings inevitably get hurt, especially in an unmoderated forum, and reactions to that hurt sometimes became sources of new hurts, perpetuating an unfortunate cycle of recriminations. For some researchers, such interactions might actually *be* the richest source of data, but for the present purposes, these data were considered noise.

Another constraint to be aware of is that the distribution of participants necessarily limits the topics that can reasonably be discussed. That is, a researcher interested in exploring educational conditions within a specific school district, or state, would not turn to a newsgroup like edu-sig, which is more global in nature. A newsgroup set up just for the educators in that district or state would be a more reasonable source of data since the topics covered would more likely be focused on local concerns.

We now turn to the findings derived from the results of the content analysis of the data.

5.2 Content Findings

This dissertation began, in Chapter One, by considering the nature of computer literacy and claiming that computer literacy is becoming important in education in a way analogous to the way print literacy is currently important; and that therefore, it would soon become as important to learn how to program a computer as to know how to write. I showed how reading and writing enable us to learn, and how computer literacy, including the ability to program, will increasingly be needed to enable our learning.

I then, in Chapter Two, considered the steps involved with creating a computer program and what the issues were for teaching those steps. This was followed by a discussion of the Python programming language and the genesis of the Python edu-sig newsgroup as part of the Computer Programming for Everybody effort. Then, in the fourth chapter, I looked at their online discussions surrounding the teaching of Python, and drew from them heuristics and rubrics for such teaching.

It is time to revisit the original thesis topic of this dissertation: What considerations are most important in teaching Python as a first programming language in a secondary school setting? We are now in a position to place these considerations into three major findings: programming as writing, programming to learn, and executable notations.

Programming as a literate activity

Perhaps the most significant finding from this research is that not only is programming *like* writing, as the analogy:

```
reading : writing :: using computers : programming computers
```

suggests, but additionally, programming *is* a form of writing, as we have seen in numerous ways. For one, the actual code written to instruct the computer what to do is expressed using English terms and familiar punctuation symbols, and with the Python language, using a syntax that is more user-friendly than most, if not all, other computer languages. More importantly, this code needs to be supplemented by extensive comments that describe and explain to human readers what the code is doing, and what the algorithm accomplishes. Recall Donald Knuth's comment regarding programming:

> Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do. (Knuth, 1992, p. 99)

This, in turn, presupposes considerable thoughtfulness, or a kind of pattern analysis, by the programmer as to the most strategic approach towards accomplishing the programming goal and explaining that approach to others; which, in turn, presupposes an apperceptive intuition of what it is that needs accomplishing, and can be accomplished by computing.

We can also consider computer programming as writing in another sense. In the first chapter we talked about the difference between representations and expressions, corresponding to "two modes of cognitive functioning, two modes of thought, each providing distinctive ways of ordering experience, of constructing reality" (Bruner, 1986, p. 11). As a reminder, I repeat the rest of Jerome Bruner's quote here:

> Each of the ways of knowing, moreover, has operating principles of its own and its own criteria of well-formedness. They differ radically in their procedures for verification. A good story and a well-formed argument are different natural kinds. Both can be used as means for convincing another. Yet what they convince *of* is fundamentally different: arguments convince one of their truth, stories of their lifelikeness. The one verifies by eventual appeal to procedures for establishing formal and empirical proof. The other establishes not truth but verisimilitude. (Bruner, 1986, p. 11)

We called (via Rorty) the 'well-formed arguments' *representations*, and the 'good stories' *expressions* and then grouped the philosopher-scientists with those who value true representations while the rhetorician-artists were grouped with those who value lifelike expressions. It might initially appear from Bruner's description that programming lies more in the domain of creating representations. We represent a logical argument, in code, to the computer as a means of "fulfilling the ideal of a formal, mathematical system of description and explanation" (Bruner, 1986, p. 12). However, we also saw in the second chapter the importance of viewing programming as being concerned with creating cognitive metaphors, where the programmer's task is to express a real-world phenomenon as an algorithm with as much 'cognitive lifelikeness' as possible.

What this suggests, then, is that programming computations is a form of writing that expresses both forms of knowing, where representations (in code, meant for the machine) and expressions (in comments, meant for the human reader) appear in tandem, convincing us of both truthfulness and lifelikeness. This, perhaps, is one reason computer programming is considered 'hard'; it demands both kinds of cognitive functioning, both kinds of knowing. To create a functioning program, we are required to *express* an algorithm that simulates a state or process in the world to other readers, and to *represent* that algorithm within the formal constraints of the programming language to the computer.

Thus, the first important consideration for teachers of Python as a first computer language is to think of programming as *writing*. And as such, it is more than just learning how to write correct code, it also develops creativity and imagination, and requires aspects of representations (well-formed arguments) and expressions (good stories) to be convincing, or well-written.

Programming to learn

In the first chapter we discussed how learning to read enabled reading to learn, how learning to write enabled writing to learn, and that by analogy, learning to program enabled programming to learn. Then, in the second chapter, we briefly used another analogy, comparing the learning of programming with the classical *trivium*. The *trivium* taught Latin as the means of learning how to learn, focusing on grammar, dialectic (logic and disputation), and rhetoric. (The oral and written aspects of each were emphasized differently through history.) The result was *not* an accretion of disparate facts and figures, but rather a *means* of learning. The student entered the succeeding *quadrivium* armed with knowing how to learn the higher subjects. In other words, he or she had learned how to learn.

We see that there are tantalizing parallels to be made by substituting ‘Python’ for ‘Latin’ in a computational version of the *trivium*. We see that learning to program also involves learning a language’s syntax. But beyond that, we see a dialectic occurring as the programmer interacts with the machine (via the interactive shell and running scripts), learns the effects and results of various programmatic constructs, and comes to know how the machine operates (gaining a ‘sense of the mechanism’). And we also see that learning to program involves learning a kind of ‘rhetoric’, that is, the algorithms and patterns that ‘persuade’ the machine to yield the results we seek.

Recall the quotes cited by Gal-Ezer and Harel (1998, p. 79) where they identified a dual nature to Computer Science:

> Computer science has such intimate relations with so many other subjects that it is hard to see it as a thing in itself.
> –M.L. Minsky, 1979

> Computer science differs from the known sciences so deeply that it has to be viewed as a new species among the sciences.
> –J. Hartmanis, 1994

The quote by Hartmanis represents the *trivium* stage of learning programming; while the quote by Minsky represents a computational *quadrivium* stage where programming plays an increasingly important role in the practice of so many disciplines.

Thus, the second important consideration for teachers of Python as a first computing language is to think of computing more as a means of learning than as an end in itself. The purpose of this computational *trivium* is *not* an accumulation of unrelated facts and figures, but is instead the acquisition of the *means* of knowing how to learn something that is computationally knowable; this then enables the learner to master the subjects in the modern version of the *quadrivium*. This approach to teaching computer programming also suggests a solution to the question of where to begin integrating it into the curriculum. Following this classical model, we ought first to teach the language until it is mastered, and then use the acquired tool manipulation skills in the service of learning the higher-level subject matter.

Executable mathematical notation and regular expressions

The classical *quadrivium* consisted of geometry, astronomy, arithmetic and music. These were the subjects learned after the *trivium* (using Latin) had been mastered. We have also seen in this dissertation the suggestion that learning to program (a computational *trivium*) can lead to learning other subjects (say, a computational *quadrivium*) in ways that differ significantly from current practice. The most obvious example from our results is Mathematics. Programming languages demand unambiguous notation in their expressions and statements. An equals sign (=), for example, cannot serve two functions, signifying both equality and assignment, as it does in printed mathematical notation (where context enables the human reader to disambiguate the meaning). Furthermore, machines can now easily process the necessary computations to arrive at a solution or solve an equation, thus radically changing what knowledge is deemed mathematically important (Kaput, 2002). Instead of learning how to *do* the computations, it now becomes imperative to learn how to *set up* the computations, to know how to precisely express a situation using computational tools, in short, how to become fluent with *executable* mathematical notation.

A related notion of 'executable notation' exists as a subset of many programming languages and deals primarily with textual manipulation rather than numerical calculation: that of the 'regular expression'. This is a notation where one can express the process of going through a (sometimes large) body of text and selectively choosing (and perhaps changing) those portions that match a given pattern. Use of this notation appears not only in programming languages but also in more and more software applications where users with some programming experience have expressed an interest in such functionality being incorporated. Here, too, we move away from *doing* the manipulation (manually finding all occurrences of the pattern in question) to *setting up* the manipulation (expressing the pattern as a regular expression) and thus becoming increasingly computer literate, letting the machine do what it excels at and letting humans do what they excel at.

We also know from other reports how using executable notation and creating simulations in science classes leads to different understandings of the subject matter. To recall Sherin's (1996) conclusion:

> Central to this endeavor is the notion that programming languages can be elevated to the status of bona fide representational systems for physics. ... A conclusion of this work is that algebra-physics can be characterized as a physics of balance and equilibrium, and programming-physics a physics of processes and causation. (p. 3)

We also recall Wolfram's advances in scientific simulations using relatively simple programs:

> But the crucial point that was missed is that computers are not just limited to working out consequences of mathematical equations. And indeed, what we have seen in this chapter is that there are fundamental discoveries that can be made if one just studies directly the behavior of even some of the very simplest computer programs. (p. 45)

Thus, the third important consideration for teachers of Python as a first computing language is to understand that integrating programming into curricular activities may significantly alter *what* knowledge becomes important to learn in many of the traditional subject areas, as well as *how* that knowledge is learned. Recall also that this is what Papert was pointing to in his discussion of constructionism:

> The presence of computers begins to go beyond first impact when it alters the nature of the learning process; for example, if it shifts the balance between transfer of knowledge to students (whether via book, teacher, or tutorial program is essentially irrelevant) and the production of knowledge by students. It will have really gone beyond it if computers play a part in mediating a change in the criteria that govern what kinds of knowledge are valued in education.

5.3 Summing Up

We have explored in this dissertation a range of suggestions for teaching a particular computer language, Python, in academic settings. We have seen how many of the suggestions follow a connectionist and constructionist philosophy, favoring a learning-by-doing approach. We have also explored a range of issues that are likely to arise in such settings, and discovered suggestions for dealing with those issues. In this last section of the dissertation, I make a few suggestions for where further research is indicated, and end with a few remarks about convivial programming.

Suggestions for future research

As noted in Chapter Four, there are several secondary classrooms around the United States where Python programming is taught. It would be interesting to compare the different teaching strategies used in each classroom, both as self-reported by the teachers themselves, and through classroom observation. It might also be interesting to compare those classes with a sampling of Advanced Placement Computer Science classes, perhaps by measuring programming proficiency, if fair criteria for comparison could be established.

However, one implicit goal of Computer Programming for Everybody is to enable students to use programming in their other classes. For those that are learning Python, do such opportunities arise, and are they able to utilize their programming skills to advance their learning in other areas? This, too, could be compared with students in AP computing classes to test the hypothesis that Python favors such ‘programming to learn’ activities due to its relative user-friendliness compared to C++.

Although the issue was not discussed in this dissertation, there are graphical interactive development environments (not necessarily Python-based) that offer a graphical user interface to aid in the creation of computer programs. These are usually meant to assist with the development of another graphical user interface for the new application being coded. It would be quite interesting to explore the differences in student learning between using these graphical interfaces and the more traditional command-line interfaces that we considered here.

Probably the greatest impediment to wider adoption of computer programming in school settings is the lack of adequate teacher education. Few teachers take it upon themselves to learn programming, and fewer still master it well enough to incorporate it into either their existing classes, or new classes specifically designed to teach programming. Thus, research is needed to provide guidance as to how to rectify this situation: Require programming in teacher education programs? Offer continuing education credits? Seek out computer professionals that are willing to teach? Develop student mentors to assist teachers with programming activities? I doubt that any solution will result in revolutionary changes, but if promising evolutionary approaches can be identified, then eventually there should be enough teachers comfortable with computer programming that students will not have difficulty using their programming knowledge in classroom learning activities.

Convivial programming

One aspect of programming with Python that was not strongly emphasized in this dissertation is the fact that it is part of the Open Software movement, that is, not only is the compiled product available for free, the *source* is freely available too. One result of this, which applies to other popular open source products such as the Apache web server, the Linux operating system, and the MySQL relational database server, is that there is an extensive community of fellow programmers and software users available to promote the use and development of those products. One consequence of this community is that the 'Woes of the Craft' are significantly mitigated. The frontispiece to this dissertation highlighted the 'Joys of the Craft' by Frederick Brooks (1975); however, he also described a complementary aspect to programming that bears repeating:

> The Woes of the Craft
>
> Not all is delight, however, and knowing the inherent woes makes it easier to bear them when they appear.
>
> First, one must perform perfectly. The computer resembles the magic of legend in this respect, too. If one character, one pause, of the incantation is not strictly in proper form, the magic doesn't work. Human beings are not accustomed to being perfect, and few areas of human activity demand it. Adjusting to the requirement for perfection is, I think, the most difficult part of learning to program.
>
> Next, other people set one's objectives, provide one's resources, and furnish one's information. One rarely controls the circumstances of his work, or even its goal. In management terms, one's authority is not

sufficient for his responsibility. It seems that in all fields, however, the jobs where things get done never have formal authority commensurate with responsibility. In practice, actual (as opposed to formal) authority is acquired from the very momentum of accomplishment.

The dependence upon others has a particular case that is especially painful for the system programmer. He depends upon other people's programs. These are often maldesigned, poorly implemented, incompletely delivered (no source code or test cases), and poorly documented. So he must spend hours studying and fixing things that in an ideal world would be complete, available, and usable.

The next woe is that designing grand concepts is fun; finding nitty little bugs is just work. With any creative activity come dreary hours of tedious, painstaking labor, and programming is no exception.

Next, one finds that debugging has a linear convergence, or worse, where one somehow expects a quadratic sort of approach to the end. So testing drags on and on, the last difficult bugs taking more time to find than the first.

The last woe, and sometimes the last straw, is that the product over which one has labored so long appears to be obsolete upon (or before) completion. Already colleagues and competitors are in hot pursuit of new and better ideas. Already the displacement of one's thought-child is not only conceived, but scheduled.

This always seems worse than it really is. The new and better product is generally not available when one completes his own; it is only talked about. It, too, will require months of development. The real tiger is never a match for the paper one, unless actual use is wanted. Then the virtues of reality have a satisfaction all their own.

Of course the technological base on which one builds is always advancing. As soon as one freezes a design, it becomes obsolete in terms of its concepts. But implementation of real products demands phasing and quantizing. The obsolescence of an implementation must be measured against other existing implementations, not against unrealized concepts. The challenge and the mission are to find real solutions to real problems on actual schedules with available resources.

This then is programming, both a tar pit in which many efforts have floundered and a creative activity with joys and woes all its own. (pp. 8-9)

Keep in mind that Brooks was discussing professional programmers in a commercial setting, thus some of his concerns are less applicable in an open-source setting. Nevertheless, access to an extensive community of like-minded programmers eases these woes and goes a long way towards forming what we might call ‘convivial programming’.

The term ‘convivial’ echoes Ivan Illich’s book, *Tools for conviviality* (1973) where he called for “autonomous and creative intercourse among persons, and the intercourse of persons with their environment” (p. 24). Open source software embodies this notion of convivial programming, as does programming with Python.

I bring up this notion of conviviality for a reason. Most of the research in this dissertation focused on what Illich called ‘curricular’ learning, which is learning that takes place in the context of mainstream schooling institutions. We have seen the difficulties faced by educators willing and wanting to incorporate computer programming into such classrooms. Although I hope that the research in this dissertation might help with the growth of programming classes in curricular settings, I don’t want to suggest that this is the only avenue by which computer programming with Python can be promoted. The open-source community is a prime example of a ‘tool for conviviality’ where the interested individual can engage not in the ‘curricular’ learning of programming, but in a ‘planned’ or ‘self-directed’ learning effort.

Elsewhere, Illich describes the question that a planned learning effort should begin with:

> The planning of new educational institutions ought not to begin with the administrative goals of a principal or president, or with the teaching goals of a professional educator, or with the learning goals of any hypothetical class of people. It must not start with the question, "What should someone learn?" but with the question, "What kinds of things and people might learners want to be in contact with in order to learn?" (Illich, 1971, pp. 77-78)

Again, the open-source community constitutes a wealth of convivial 'things' (for example, software projects, tutorials, websites and documentation) and convivial people (online newsgroups, for example) that learners might want to be in contact with in order to engage in the planned effort of learning programming for themselves. (Appendix B points to many resources that can aid such an effort.)

In *Deschooling Society*, Illich (1971) quotes Aristotle in distinguishing between two kinds of activity that relate to curricular and planned learning:

> Aristotle had already discovered that "making and acting" are different, so different, in fact, that one never includes the other. "For neither is acting a way of making—nor making a way of truly acting. Architecture [techne] is a way of making—of bringing something into being whose origin is in the maker and not in the thing. Making has always an end other than itself, action not; for good action itself is its end. Perfection in making is an art, perfection in acting is a virtue."[45] The word which Aristotle employed for making was "poesis," and the word he employed for doing, "praxis." A move to the right implies that an institution is being restructured to increase its ability to "make," while as it moves to the left, it is being restructured to allow increased "doing" or "praxis." Modern technology has increased the ability of man to relinquish the "making" of things to machines, and his potential time for "acting" has increased. (p. 62)

[45] Nichomachean Ethics, 1 140.

Another way to understand this difference between a 'move to the right' and a 'move to the left' is to think of the move to the right as being teleological, while the move to the left as being ontological. That is, as Aristotle says, "Making has always an end other than itself," which is teleological in nature, while "good action itself is its end" suggests an ontological nature.

Illich intends that the "making" in this passage be understood as "the making of curricular objectives in students' minds" by schools, which manifests as curricular learning artifacts such as test scores or book reports. However, Illich is arguing instead for a 'move to the left' where individuals can more fully engage in planned learning activities to occupy their leisure time (for example, after-school computer clubs, or online tutorials, or volunteering for a non-profit organization). Rather than viewing knowledge teleologically as a possession (of things 'made' by the curriculum), he is advocating the viewing of knowledge ontologically as an aspect of being in the world, as immanent in an individual's actions. This is fully consonant with the objectives of the *trivium* as to learning how to learn, and convivial programming as a means of programming to learn.

Thus, I have tried to show in this dissertation the value of learning programming, not as a specialized, professional skill (although it can be that), but as an essential component of a liberal arts curriculum. Programming is a literate activity that engages one's creative and imaginative faculties, and is a new mode of expressing thoughts, ideas, patterns, algorithms and concepts in a way that was largely unavailable to mass education until very recently (historically speaking). I expect that the adoption of programming into the curriculum will take time, time for teachers to learn and assimilate the value of programming into their lives and teaching methods, time for administrators to see and understand the value of programming as a means of increasing students' computer literacy, and time for Python (or some other language) to garner sufficient mindshare as the language of choice for 'everybody' learning how to program a computer. In the meantime, in the absence of curricular opportunities to learn programming, a convivial alternative exists in the open source community for individuals to invest their leisure time and to begin savoring the joys of the craft.

APPENDIX A

CHOSEN THREAD SUBJECT HEADINGS BY CATEGORY

Thread Category (# of threads)	**(Reference Code) Thread Subject Header (# msgs / # different posters)**
Education (33)	▪ (ED-1) Age groups (20/13) ▪ (ED-2) Pedagogy, programming environments, and readings (6/4) ▪ (ED-3) Hello from a CS teacher (16/9) ▪ (ED-4) My experience teaching Python (13/7) ▪ (ED-5) OO in K-12 (6/3) ▪ (ED-6) Teaching Python at a Junior College level (5/5) ▪ (ED-7) Teaching Middle-School Math with Python (21/8) ▪ (ED-8) Socratic methods (8/5) ▪ (ED-9) Number-line graphics for teaching arithmetic (6/3) ▪ (ED-10) Proposing/defending Python in a curriculum (7/7) ▪ (ED-11) Beginner programs (8/6) ▪ (ED-12) Python comes to Henry Sibley H.S (5/4) ▪ (ED-13) College CS courses (8/5) ▪ (ED-14) Intro and question: assignments/projects for year end (11/10) ▪ (ED-15) Text compression as a learning tool (6/4) ▪ (ED-16) Interactive tutorial (12/7) ▪ (ED-17) Disney learning (5/3) ▪ (ED-18) Lesson plan collection (9/5) ▪ (ED-19) Python Comp Sci course (5/5) ▪ (ED-20) Teaching python (12/5) ▪ (ED-21) Assigning homework (7/6) ▪ (ED-22) Teaching students to use CVS (5/4) ▪ (ED-23) Student assignment styles (6/5) ▪ (ED-24) Who is teaching Python (8/7) ▪ (ED-25) Encouraging students to plan effectively (6/4) ▪ (ED-26) The right learning environment (20/7) ▪ (ED-27) Python anxiety (18/10) ▪ (ED-28) Python @ education: what are your problems (16/10) ▪ (ED-29) Girls, women programming and Python (20/12) ▪ (ED-30) BBC NEWS UK Education GCSE 'gender gap' sparks concern (12/6) ▪ (ED-31) Techniques to force students to plan (5/4) ▪ (ED-32) Does any such tutorial exist (10/4) ▪ (ED-33) An outline I'm using (6/4)

CP4E (6)	▪ (CPE-1) Python for non-programmers (16/7) ▪ (CPE-2) 'Killer Apps' vs. CP4E (7/4) ▪ (CPE-3) Future of CP4E after CNRI (23/8) ▪ (CPE-4) CP4E VideoPython learning to teach/teaching to learn (10/4) ▪ (CPE-5) JPython and CP4E (7/5) ▪ (CPE-6) CP4E-2002 (11/6)
Python/Computer Science (15)	▪ (PCS-1) Stop the insanity -- no more case sensitivity discussion (5/4) ▪ (PCS-2) Beginner trouble, indexing starting w/'0' (7/7) ▪ (PCS-3) Natural language programming (15/6) ▪ (PCS-4) Python accessibility (8/7) ▪ (PCS-5) Computer science without all that 'heavy math' stuff (11/8) ▪ (PCS-6) Measuring python (6/4) ▪ (PCS-7) Analyzing algorithms (12/6) ▪ (PCS-8) Equality and assignment notation (6/6) ▪ (PCS-9) Python for algorithms and data structures (12/7) ▪ (PCS-10) Python for AI (9/6) ▪ (PCS-11) Python sequences by reference - how to make clear (36/8) ▪ (PCS-12) Top 5 All Time Novice Obstacles => #2 Helping help (6/4) ▪ (PCS-13) Top 5 All Time Novice Obstacles => #3 Where am I (15/6) ▪ (PCS-14) Interactive interpreter: expressions vs. statements (5/5) ▪ (PCS-15) Lines of code and programmer time (7/6)
Math-related (15)	▪ (MTH-1) Rational division (10/7) ▪ (MTH-2) Exponentiation should float (5/4) ▪ (MTH-3) Long integer fractions (14/6) ▪ (MTH-4) Sieve of Eratosthenes (14/5) ▪ (MTH-5) More Pythonic precalculus (9/4) ▪ (MTH-6) Long floats (6/3) ▪ (MTH-7) Goofing with groups (7/4) ▪ (MTH-8) Algebra + Python (23/6) ▪ (MTH-9) More spillover re the division PEP (16/5) ▪ (MTH-10) Calculating area of a surface plane on a spherical body (7/5) ▪ (MTH-11) Math weirdness (6/5) ▪ (MTH-12) Why 0**0 == 1? (5/4) ▪ (MTH-13) Types and true division (42/8) ▪ (MTH-14) Rationals (26/9) ▪ (MTH-15) Where mathematics comes from (20/4)
Science-related (5)	▪ (SCI-1) The Educational Robotics Platform (17/6) ▪ (SCI-2) Scientific Python 2.2 (5/3) ▪ (SCI-3) Modeling (6/3) ▪ (SCI-4) Periodic table (6/5) ▪ (SCI-5) Python promises a revolution (8/4)
Programming for fun (7)	▪ (PF-1) Python programming for kids (16/6) ▪ (PF-2) Programming for the fun of it (29/15) ▪ (PF-3) Cards n stuff (10/4) ▪ (PF-4) Observations from the Northwest Science Expo (7/6) ▪ (PF-5) Python for fun (10/5) ▪ (PF-6) Programming for artists (30/7) ▪ (PF-7) Programming for fun quote (7/4)

Miscellaneous (7)	▪ (MISC-1) Beyond 3D (5/4) ▪ (MISC-2) Question about a programming system (11/5) ▪ (MISC-3) PEP0238 lament (13/8) ▪ (MISC-4) Does edu-sig extend to Jython (25/5) ▪ (MISC-5) Database for a small network (5/3) ▪ (MISC-6) On and off-topic [was elegant copy-by-value] (11/5) ▪ (MISC-7) On-topic: LinuxFormat magazine (6/3)
Unknown (20)	▪ (UNK-1) Getting it going (17/13) ▪ (UNK-2) New game in town (6/3) ▪ (UNK-3) (no subject) (8/6) ▪ (UNK-4) On the front page (14/5) ▪ (UNK-5) Articles of possible interest (31/14) ▪ (UNK-6) My opinion (5/4) ▪ (UNK-7) Things to come (18/7) ▪ (UNK-8) Around again (5/3) ▪ (UNK-9) Now I went and did it (38/13) ▪ (UNK-10) A fact on the ground (18/10) ▪ (UNK-11) Active essays (5/3) ▪ (UNK-12) Switching gears (8/4) ▪ (UNK-13) And now for something completely different (8/3) ▪ (UNK-14) About myself (5/3) ▪ (UNK-15) Brainstorming and a neat link (8/5) ▪ (UNK-16) Off topic musings (22/10) ▪ (UNK-17) Thanks for the tip (5/3) ▪ (UNK-18) Which way did the chicken cross the road (7/5) ▪ (UNK-19) Slightly OT: O'Reilly article (21/9) ▪ (UNK-20) Losing the plot (15/5)

APPENDIX B

RESOURCES FOR LEARNING & TEACHING PYTHON

Beginner's Guide to Python
http://www.python.org/topics/learn/

Python Bibliotheca
http://www.ibiblio.org/obp/pyBiblio/
A library of educational materials using Python to teach computer programming, and a virtual meeting place for teachers and students engaged in learning and teaching using Python.

How to Think Like a Computer Scientist: Learning with Python
Allen B. Downey, Jeffrey Elkner and Chris Meyers
http://www.ibiblio.org/obp/thinkCS/python/english/
A collaborative translation of the original from Java into Python.

The LiveWires Python Course
http://www.livewires.org.uk/python/
Intends to teach the Python programming language to people who have never programmed before.

Guido van Robot
http://gvr.sourceforge.net/
Guido van Robot is a minimalistic programming language providing just enough syntax to help students learn the concepts of sequencing, conditional branching, looping and procedural abstraction.

Useless Python
http://www.uselesspython.com/
Some Python scripts you can play with on your own. If you make changes to someone's script, or do something in a different way, you can email the script along with anything you have to say about it, and it will be added to the collection.

Python Cookbook
http://aspn.activestate.com/ASPN/Cookbook/Python
A collaborative collection of Python coding techniques. Very useful!

Learning to Program
Alan Gauld
http://www.freenetpages.co.uk/hp/alan.gauld/
A tutorial for and anyone who wants to learn the art of programming.

A Mathematical Canvas
Kirby Urner
http://www.4dsolutions.net/ocn/
An excellent site for using computing in the Mathematics classroom.

Handbook of the Physics Computing Course
Michael Williams
http://users.ox.ac.uk/~sann1276/python/handbook/
It assumes no programming experience, although it does assume you are familiar with some high school level math (sine and cosine). It's a self-contained course with exercises included.

10 Python Pitfalls
Hans Nowak
http://zephyrfalcon.org/labs/python_pitfalls.html
A guideline to those who are new to Python. It documents language features that often confuse newcomers, and sometimes experienced programmers.

Python Quick Reference
Richard Gruet
http://rgruet.free.fr/

Python Eggs
http://www.python-eggs.org/links.html

Dive into Python
http://diveintopython.org/toc/index.html

Non-Programmers Tutorial For Python
Josh Cogliati
http://honors.montana.edu/~jjc/easytut/easytut/
A tutorial designed to be a introduction to the Python programming language. This guide is for someone with no programming experience.

PythonCard
http://pythoncard.sourceforge.net/
PythonCard is a GUI construction kit for building cross-platform desktop applications on Windows, Mac OS X, and Linux, using the Python language.

Thinking in Python
http://www.mindview.net/Books/TIPython

O'Reilly Python Center
http://python.oreilly.com/

Pygame
http://pygame.org
Pygame is a set of Python modules designed for writing games.

ScientificPython
http://starship.python.net/~hinsen/ScientificPython/
ScientificPython is a collection of Python modules that are useful for scientific computing.

Data Structures and Algorithms with Object-Oriented Design Patterns in Python
Bruno Preiss
http://www.brpreiss.com/books/opus7/
The primary goal of this book is to promote object-oriented design using Python and to illustrate the use of the emerging object-oriented design patterns.

Project DUPLEX (Drexel University Programming Learning EXperience)
http://duplex.mcs.drexel.edu/
Investigates the use of technological advances to enhance the quality and delivery of large computer programming classes, while reducing costs of course administration.

APPENDIX C

STORIES FROM THE EDU-SIG

The following three stories each illustrate ways of approaching the question of teaching Python to beginners:

> For what it's worth, I gave my 8-year-old son his first taste of Python with the proven “steal this code” approach. We started with a working Tkinter script with buttons calling a function that displayed text in a widget. He already could read and do basic on-screen editing (typing, copy-and-paste). We didn't discuss concepts like scope and parameter passing, but just started copying and modifying code. Soon he was changing the widget colors and printing his own messages to the screen.
>
> I think the lesson is that Piaget was right: make the material conform to the student's cognitive developmental level. For 8-10 year olds, that means keeping things concrete and emphasizing the “how” over the more abstract “why.” You might say they start off more as language users than programmers (in the same way that I'm a computer user and not a chip designer). Children at this age are not ready for abstractions, but they learn well by seeing things work. Treat the program as a machine, and let the students adjust the input and watch the output change. The quick code-and-run cycle with Python helps quite a lot in this regard. –DD, ED-1, 18th

---

> I'll take this as an opportunity to discuss my own limited experience with teaching Python to new programmers.
>
> I've taken a look at Learning Python and it doesn't seem to have the right structure if you're completely new to programming. My friend bought it and hasn't been too much use to her so far, though I expect it probably will be later on.

I'll describe how I explained things to my friend (her 12 year old daughter was also present during some sessions). Note that she only just got started and therefore I can't promise this approach will work. I'll get a chance to teach a number of other people starting next month, however, so I'll gather more data then.

I described interactive mode first. Since my friend is moderately familiar with command line environments this did not present too much of a problem with her. I described the basic idea of some various types; strings and integers, floats and longs. I didn't expect her to remember it all right away, but at least it'd give her an idea of what's possible. I also briefly mentioned lists.

I also introduced variables and assignments in the interactive mode. I paid special attention to the fact that you can basically use any word (or number of words) as a variable. I also described some built in functions such as int(), string() and float(). They are always there and aren't too difficult to explain, so that was useful.

As I'm not much of an interactive mode user myself (though certainly it's handy at times), I moved on to describe the editing environment. In my case it was Emacs, which can be horrendously complicated. The basics are pretty simple tough, and the python-mode is very nice (syntax highlighting and easy way to run the code in the editor to see immediate results). The menus under X helped a lot too.

I showed how you can print stuff, such as the result of expressions. After that, I introduced boolean expressions (perhaps I should've done so already in the interactive mode). Then I moved on to 'if' statements, and block based indentation. All that was still pretty clear.

'for' loops presented more problems, though! I had quite a bit of difficulty to explain that the for block gets executed for each element in the sequence, and that the variable (for variable in foolist) refers to something else each time in the execution. I need to think on a better way to express things apparently-tricky thing.

I explained how to write a program that adds all numbers from 1 to 9 (or any sequence of numbers, really), using the 'for' loop, and range(). This was also pretty difficult to get across. The concept of an extra variable 'sum' that keeps the current sum while doing a loop is not something people hit on by themselves!

After that I had them change the function to multiply instead of doing additions (so you'd do factorials). Due to my explanation that you can use basically any word for a variable, and also focusing on the idea that you should use descriptive variables, my friend and her daughter automatically started to change variables like 'sum' to 'product', and not only changed the + to *. This was a rather nice thing to see, though I hope it wasn't because they taught the computer actually understands what they call these variables (I don't think they did think that though). Also tricky was to change the starting value of '0' to '1' (otherwise the result is '0'), but they both could figure this out for themselves.

I then turned the factorial program they had produced into a function. In retrospect, I think I should've used the 'sum' program instead to produce a summing function, as it's probably conceptually easier to talk about. Like in my explanation of the 'why' of looping (you don't want to type 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9), I paid attention to the 'laziness' issue: good programmers are lazy (in the right way), and don't want to do boring stuff like lots of typing, and they definitely don't want to do the same thing twice. This seemed to work well.

Still, I had some problems in making clear the concept of functions. I started too difficultly by using the 'factorial' approach; I fell back to a very simple function soon enough:

```
def simple(): pass
```

I explained the difference between side effect and return value:

```
# return value
def simple1(a): return a
# side effect
def simple2(a): print a
# both
def simple3(a): print a; return a
```

This turned out to be fairly tricky at first. The difference between things like:

```
print simple1("hoi")
```

and

```
simple2("hoi")
```

was not immediately clear. It got especially tricky when we saw:

```
print simple2("hoi")
```

The idea that Python returns 'None' when there is no return statement came as a confusing surprise; I should probably have introduced 'None' earlier, but it's hard to motivate the existence of 'None' to a newbie, which is an argument against discussing None too early.

The problem with the approach with 'simple' versus that of 'factorial' is that 'simple' does demonstrate the mechanism of functions, but not the *why* of it. 'factorial' is much more useful here. I still need to figure out the right way to introduce the motivation for functions along with the mechanism for functions without causing confusion on either.

What was useful (again in the lazy programming context) was the concept of 'recipe'. A function does not get executed just by it being there; it is a recipe which you give to the computer. If you (daughter) give your mother a cookie recipe that doesn't mean immediately your mom will go and bake the cookies; you need to ask her to do so first. Unlike a mom, a computer will always obey (as long as the recipe is right), unfortunately, a computer is also very stupid so you have to be more detailed. :)

This analogy seemed to work pretty well.

That's about as far as I got this time. As you see you can spend lots of time with the basics. This is not because the mechanisms of the basics are generally hard to understand; they picked those up pretty well in general, though there were some problems. The problem is more one of *why* these mechanisms are useful. What you can do with them, and how you use them to do useful things. That's the real art of programming (along with the importance of a well readable and maintainable program structure). –MF, ED-4, 2nd

---

Using screen painter tools to design a front end is an old idea that didn't start with Microsoft, nor is this approach limited to VB. All the so-called Visual products (Visual C++, Visual FoxPro, and yes, the emerging Visual Python) are characterized by the floating toolbar palette of surfaces and controls. But then, so are the Java IDEs from Borland and Sun. I have nothing against these for what they are. They can save a lot of time.

I also think we need to get away from the "high level versus low level" to some degree. Projecting images to a monitor and polling for keyboard and mouse events is low-level interfacing to the human through the human's API. We're working to bridge the cell-silicon circuitries with interfaces of sufficient bandwidth to keep both operating with some efficiency. The biological is "low level" too, just that it's not a product of conscious engineering and we're not sure how it works exactly. But we still need to interface with it.

I don't regard using eyeball ports, which are optic-nerve endings of the brain, as ipso facto “higher level” than talking to a chip through its various pins. We're dealing with a primitive biological interface devices, highly suited to their functions (devices I don't see a real need to bypass, unless they're broken i.e. a lot of this fantasizing about hard-wiring components directly to the brain, Borg-style, seems a misguided effort to bypass what billions of years of refinement already give us “for free” -- like trying to drill holes in a Pentium chip in order to bypass its pins (why??)).

So when it comes to a highly graphical GUI, that's just a way of encoding frequencies in ways our brains have evolved to interpret, over the aeons. Electromagnetic radiation in a narrow band of the spectrum (visible light) is patterned with information that changes at human-sensible rates (vs. supra or infra-tunable speeds e.g. propeller blades when spinning at high speed, or glacial movements, are not directly perceivable, because too fast or too slow respectively -- GUI events need fit between these extremes, although sometimes a Windows progress bar will approach the glacial limit).

What we need to be clear about are “walks of life”. Some people have no slack for learning the intricacies of something that others groove on full time. We are all “disabled” in one way or another, in the sense that we lack capabilities, skills, capacities, which others have perfected and take for granted. In a lot of ways, this is exactly what interfaces are all about: making allowances for users who do NOT have the time and/or inclination and/or need and/or ability (for whatever reason) to penetrate the layers by other means and appreciate what's inside the black box.

I think it's this concealment and encapsulation of functionality that you are temperamentally disposed to challenge, and I think that's healthy. People have this natural curiosity to know “how things work” and like to consult those picture books, exploded diagrams, blueprints, cutaways, showing the innards, the guts, the “what makes it tick” mechanisms. Python is a great language for exposing a layer of mechanism handled by higher level programming languages. Even if you never learn C/C++ or LISP or Assembler (or MMIX), you get a strong dose of what a programming language is all about, how it feels to use one, how it mediates between a GUI front end, and internalized logic or routines. This is fun to find out about, and should be part of the generic K-12 curriculum IMO.

You pick up on the idea of an event loop, that's always listening, polling, waiting for changes, and triggered events, which lead to chains of programmed responses, perhaps along various threads each with a namespace, relative priority, and memory allocation. And you learn what people are talking about when they speak of class hierarchies, polymorphism and instantiated objects. This is all good general knowledge to have, even if your walk of life is not programming, not computer science. Because of your exposure to Python "cave paintings" which sketch the gist of a web server, or whatever it is, you have a feel for the role software plays in the world, even if you don't spend a lot of time writing it yourself. You've acquired a kind of generic numeracy a.k.a. literacy that'll help you read and understand technical literatures of various kinds. That's a worthy goal, in and of itself, and is what I aim for in my 'Numeracy + Computer Literacy' materials.

What I want to see develop from familiarity with Python, is a common short-hand for exploring various objects. For example, now that this 8th grader knows some Python, and now that we're studying physics, we might write something like:

```
class Atom:
    def __init__(self,protons,neutrons,electrons):
         self.protons = protons
         self.neutrons = neutrons
         self.fillorbitals(electrons)
    def valence(self):
         ...
         return ion_number
```

Then you'd have carbon, hydrogen and oxygen objects that instantiated this template. Compounds could be defined as assemblies of such.

Knowing Python already, kids would see this as a way of capturing some of the essential qualities of an atom. Furthermore, electrons might be a list rather than a number, consisting of Electron objects (same with neutrons and protons), such that we have class composition going on, with objects inside other objects (not in a class hierarchy sense, but in an agglomerative/associational sense).

And in a class hierarchy sense, what are Electrons a subclass of? We could have Fermion and Boson classes, from which our various subclasses with descend, their inherent differences manifest at this parent class level).

To me, this is more generic OO thinking, taking objects in the real world and expressing their common traits and behaviors in the form of a class template, then maybe subclassing to hardwire various attributes.

This kind of thinking about classes and objects should encompass the GUI widgets as a special case. Screen controls may be treated as a elements in a class hierarchy (or instantiations thereof), no question about it, but that's not where to *start* with the idea of objects. That's just one more example to consider.

To me, the idea of “modeling” really gets interesting to the extent we free ourselves from just thinking about GUIs in the narrow sense. –KU, SCI-3, 2nd

APPENDIX D

COMPUTER ANXIETY

This thread was in many ways the most interesting one that I came across during the study. The problems discussed were not really Python-specific, as you'll see, but certainly cover similar issues with teachers who may be asked to use programming in their classes. Also, the students in the group being referred to are adults rather than adolescents, and attitudes towards computing may be even more strongly resistant to change. The mathematical approach that the original poster refers to can be found at http://www.4dsolutions.net/ocn/. Some messages that strayed from the main topic have been elided, but the messages retained are intact.[46]

> I don't know whether you remember me, but I have seen your posts many times in the various math education groups, and it was through you that I discovered Python. The first time I saw it, I was absolutely delighted with its potential for helping with math understanding. But lately I have become disillusioned, because I am constantly finding that students don't like Python, and I can't at all figure out why. I thought at first it was because I was teaching students who were very computer illiterate, but then recently I have had the opportunity to teach a Discrete Mathematics for Computing distance education class, and

[46] http://aspn.activestate.com/ASPN/Mail/Message/1815054

again I saw great potential for using Python to make the abstract ideas more concrete, but again it has fallen flat. Many of these students have programmed in C++ and Java, so you would think they could learn enough Python for what I was suggesting in 5 minutes. I don't know C++ and Java and it only took me a slight bit longer. I have given them so many opportunities for extra credit projects having to do with it that they could all have perfect scores for the class if they wanted, but nobody has taken the bait. But partly this doesn't surprise me, because these distance education students are very lazy. There are a lot of people that do DE for a free ride, and the familiar is always more comforting to such people.

But what really shocked me was the experience I had today with my colleagues when I tried to show it to them as something with great potential for help with understanding algebra. I was just showing them how you could use it as something better than a hand calculator for doing such things as solving equations by searching, which I think is a really good idea for keeping students in touch with the base meaning of solving equations. And one of my colleagues practically expressed horror and said that this would totally put him off of mathematics. And others express similar opinions. I remember the first time I saw you write about how you could define a function in the console mode def f(x): return x**2, and then proceed to evaluate it on form a composition function, I immediately thought that was just such a great way for students to see such things right in front of their eyes, for them to no longer be abstract. But he seemed to think it would take him hours to master the syntax of it and for the students it would be just one more thing to learn when they were already afraid of the subject. And partly from some of the reactions I have gotten from students, it seems that he is likely to be right. For him the fact that it there is a : and a return instead of just an equal sign was totally daunting and the ** makes it even worse.

So my question for you is have you found this kind of Python anxiety, and if so how have you dealt with it? –SW

Re your specific questions, I have to confess up front that I have very limited personal experience trying to phase in Python in the ways I suggest. I would *like* to have more of these opportunities, but the fact is that I am not now a classroom teacher (I used to be, but that was many years ago). When I do get together in a room to present Python or Python-related topics, chances are they're already sold on the program -- I'm mostly just preaching to the choir as it were.

With that confession out of the way, I will attempt to give you some feedback.

I think you're encountering two different reactions here, when you talk about (a) trying to teach Python to students who may already have some C++ or Java experience versus (b) showing off Python's potential utility as a math-teaching aid to faculty members in a mathematics department.

In the former case, there's some chauvinism in the various language communities. C++ and Java both take longer to become productive in than Python, and are better established in the commercial sector. Python does enjoy a growing following on many fronts, but it's not atypical to encounter dismissive attitudes amidst those who've already made considerable investment in another language. My riposte is that anyone serious about programming needs to keep an open mind and appreciation for multiple languages. The idea of a “monolingual professional programmer” is something of an oxymoron.

More specifically, to C/C++ people I'll point out that Python is open source, written in C, and extensible in C/C++, so if you have C/C++ skills, you have lots of opportunities in the Python community, which is inhabited by a great many accomplished and sophisticated C programmers (Python being a wonderful example of C's capabilities). To Java people, I'd point out Jython, a version of Python implemented entirely in Java, and through which one has interactive access to the complete Java class hierarchy. Serious Java programmers needn't leave Java behind in order to avail themselves of Jython's power, and should realize that schools using Python and Jython are not necessarily competing with the Java community -- on the contrary, they're helping to train a next generation of Java programmer.

Also in this context, I might direct these skeptics to Bruce Eckel's web site: http://www.mindview.net/. Here's an excerpt from an interview at this site, which you might find interesting, given your own experience with distance education:

> One of the things I'm working on now is a distance-learning program for people who want to learn to program using Python. I think it will be a much faster and more efficient way for people to come up the learning curve. This is still in the formative stages; as you might have guessed by now I generally think about something for awhile before the right approach comes to me.

> Once you've had success with programming and are comfortable with objects, then you're ready to tackle a language like C++ or Java, which is heavier weight and has more arbitrary details for the programmer to master (or become confused by).
>
> [http://www.mindview.net/Etc/About/InformITRaw_html]

So here's a guy with some very deep and meticulous books on both C++ and Java -- who now advocates Python as his current language of choice. Again, Python is not just some toy academic language, nor just a “scripting language” beneath the dignity of serious programmers. It's a very high level general purpose language, and learning it first can make Java and C++ a lot more accessible -- as second or third languages.

But with the math teachers, I think the reaction is coming from a different place. If they're horrified by the colon and the return keyword in Python, they'll be even more horrified by all the syntactical clutter of *any* computer language -- even Mathematica, which has gone a long way to accommodate traditional math notation. But as Wolfram points out, traditional notation is ambiguous. Does s(x-1) mean the function s, applied to x-1, or does it mean s times x-1? In Mathematica, when a function is being applied, we use square brackets exclusively, while curved parentheses serve to indicate the order of operations. Whereas humans can tolerate a lot of ambiguity, owing to sensitivity to context, computers cannot. And so in a lot of ways, computers force *more* precision on a notation.

With math educators, it does no good to talk about Python's power and sophistication vis-a-vis C++ and Java. Their beef is with the whole idea of diluting the purity of their discipline with material from an alien discipline, i.e. computer science and/or engineering. To start using a computer language in an early math curriculum looks like the harbinger of nothing good: it means math will become mixed up with all kinds incompatible grammars which come and go, vs. the staying power of a more stable, core notation. Plus if computer languages invade the math classroom, then teachers will be forced to learn programming, which many are loathe to take up. The hand held graphing calculator is as far into computing technology as these teachers want to go, and even there, their programmability is often ignored.

But not all math educators are on the same page here. Many recognize the advantage of having an “executable math notation” vs. one that just sits there on the printed page, doing nothing (except crying out to be deciphered). Kenneth Iverson makes this point very clearly when he writes:

> It might be argued that mathematical notation (to be referred to as MN) is adequate as it is, and could not benefit from the infusion of ideas from programming languages. However, MN suffers an important defect: it is not executable on a computer, and cannot be used for rapid and accurate exploration of mathematical notions.
>
> Kenneth E. Iverson, Computers and Mathematical Notation, available from jsoftware.com

It's this ability of computer languages to promote the "rapid and accurate exploration of mathematical notions" which some of us find exciting and empowering (and if you want to impress math teachers with a wholly alien notation, which nevertheless expresses a lot of the same ideas (sometimes as generally and formally as any traditional notation), have them look at APL or J).

Furthermore, one might argue that imparting numeracy is *not* limited to teaching just those topics and notations most traditionally favored within mathematics. It's about imparting some familiarity and comprehension around *whatever* happen to be the culture's primary symbolic notations (music notation included) beyond those which we group under the heading of literacy.

We need to provide some exposure to computer languages because our industrial society is completely dependent upon them, because they've become ubiquitous.

We have the choice of segregating these topics from mathematics, just as we've divorced mathematics from the physical sciences. But some educators in every generation advocate curriculum integration through cross-pollination, and to such educators, it makes perfect sense to meld computer topics with math topics, with science and even humanities topics (cryptography is a good example of where all of these converge). In my view, the benefits to be obtained through synergy outweigh the arguments of turf-protectors who would keep their respective disciplines "pure".

That's the kind of overview debate I think is going on here. Then come the more specific points about the advantages of a computer language such as Python, versus the calculators, which are already pretty well established. Python's abilities with big integers are another selling point. Even though some calculators can work with 100-digit integers, they have a lot more trouble displaying them (more an argument for a big screen over a tiny one).

The ability to scroll back through a work session is another advantage.

And I think algorithms which involve alphanumeric processing, not just numeric processing, should get more central treatment, including in mathematics. For example, it's fine to express permutations using integers only, but for applications purposes, it's best if we can map 1-27 to the 26 letters and a space. Then you can see how a permutation results in a scrambling of the alphabet -- back to cryptography again (I recommend Sarah Flannery's 'In Code' by the way -- she's very clear about how important Mathematica was to her conceptual development -- which isn't the same as Python of course, but some of the same synergies apply).

Thanks again for your interesting letter. If you give your permission, I'll post it, along with my response, to edu-sig, in hopes of generating more discussion along these lines. Or again, perhaps you'd like to subscribe and make a fresh start. I think the attitudes you're running up against are not unique to your experience and it might benefit a lot of us to take part in more of such discussion. –KU

I while back I wrote to KU about something that happened to me when I tried to use Python to help with understanding of mathematical concepts. At that time I didn't have time to participate in this group when KU suggested it, but he told me he posted what I wrote to the group.

Basically the discovery that I am making over and over is that students that have trouble with just the kinds of mathematical topics that I would have thought some experience with Python would help with, are even more terrified of computers than they are of mathematics.

It isn't just Python, either, for all the talk about use of computers in the mathematics classroom as dumbing down, my recent experience is that students find it harder with computers rather than easier. I had statistics students who even in a distance education class where they were supposed to submit their assignments on Excel spreadsheets, would go so far as to submit something that was in tabular form in a textbox carefully using the space bar to get things to line up correctly. I had several others who would type without the = sign almost exactly the calculations that Excel would have done for them in a cell, and then repeat the same keystrokes in their calculators, and then type the answer displayed in the calculator. Most recently I have been teaching a “Nature of Mathematics” sort of survey course where use of spreadsheets is even part of the course, and I have a student who will do exercises from a section specifically about use of spreadsheet, and do the exercises perfectly, but refuse to actually put them in an Excel sheet, saying she can't deal with Excel.

Anybody have a clue about what is going on with such computer phobics or what to do about it?

I am ideally very attracted to the ideas that KU has about integrating mathematics and programming, but my recent experience is suggesting caution. I am very much interested in your opinions about this. –SW

Some of what you are seeing is the normal reaction to VERY BAD USER INTERFACES. I.e. they aren't _afraid_, and its not a _phobia_ -- they just think that the tools stink and are a tremendous waste of time.

They are (or think they are) more productive without the tools you suggest, and they resent having to learn something when they have a much better way to do it on their own. It's clearly not laziness, either. Part of their problem, if they are like the people I have known, is a basic issue about 'control' and 'flexibility'. They want to do the job their own way, under their own control, and the system they have is so inflexible that they have to learn a completely different way to get even the tiniest of correct results. But the people I am used to dealing with already know how to get correct results with a calculator or a pen, and they heartily dislike being re-trained to suit some computer's idea of how the job should be done. Telling them that after training they would be more productive is not the issue. Poor or Inflexible user interfaces make them feel slow, stupid, and _wasting their lives_ -- so they quit in disgust, resentment, and anger. And I don't blame them. I needed to make a GANTT chart for a grant proposal yesterday, and after 2 hours with Microsoft Project, it is back to pen and paper for me as well. I'll program PyGame to make pretty boxes. :-)

So, for your immediate need, you want something to do statistics designed by a somebody with a clue about Human Factors. Fortunately, we have such a beast, Salstat, written in Python by Alan Salmoni who just got his PhD in Cardiff University in Wales, cc'd on this note. http://salstat.sunsite.dk/ I hand it to undergraduates who used to be told to use SAS or SPSS, and they don't pester me with questions about how to use it. I'd try it and see what happens. If they hate SalStat, they can send mail to Alan, who is really interested in such bug reports and _really cares_ about their interactions.

He also knows more about exactly why people have troubles like you describe than anybody I have ever spoken with, so I hope his account works. –LC

I think the question you need to ask is 'what makes a calculator easy to use'. It may simply be that people are trained to use a calculator, long before you meet them. (How old are these students, anyway?) Plenty of things are called 'intuitive' when all they actually are is 'familiar'. This makes measuring whether something is actually 'easy to use' v. 'you just know how to do it' a difficult problem for Human Computer Interaction.

Perhaps your user community is distinctly different from mine, but the 12 year old children in the computer club don't like '**' for power because they are unfamiliar with that notation. (Some of them are unfamiliar with the notion of exponentiation as well.) Giving them a page of 'pen and paper' math problems to solve, where they were asked to use the ** notation made them familiar with it, which fixed that problem. The first year, I thought that 8 questions would be sufficient for familiarity, but that proved not to be the case. Next year I used 20, and had no trouble. It may be that your students need more hands on training with computers to become more familiar with loading and using programs in general, or certain programs in particular. –LC

> I am trying to discover such basic things as why it is so much easier to press 2+4= on a calculator than =2+4 in a spreadsheet. Or for that matter why typing 2+4 at a >>> prompt is so much harder than pressing 2+4= on a calculator.

An additional element that may contribute to this is the perceived risk of inadvertent error. In a setting like Excel where a large number of options appear to be available, you not only need to know how to accomplish a task, you also need to know how you could mess it up in ways that aren't obvious (such as a proportional font space in the wrong place). From the user's perspective the large, unknown (and hence essentially unlimited) number of possible errors means that if you care about accuracy you're going to have to check the results using a method you trust anyway. So why not save the time and aggravation, and use the trusted method first? –JH

Thank you, this is a very interesting post about an important topic..

Please can you tell us more context and who are your students -- school, socio-economic background, age, experience etc.

It sounds like they were so poorly educated and ill-prepared. Must have been terrifying and humiliating for some of them.

When you asked them to use Excel did you ever check if they know how to use it? Give any live demos/example yourself of what you wanted? Is access to the software itself an issue? Is it even installed? etc.

Was this shock in the first lesson and homework? Had you already covered some ground successfully? How much variation is there in the students? How many own their own or have good access to computers in their homes? Are they in any other classes which use computers? If so have you discussed with those teachers?

How did you deal with this situation? What do the students themselves say about their efforts to simulate the expected result?

If you knew then what you know now, how differently would you have approach this class?

I am very curious about many aspects of the story. –JC

> > I'm not looking for a statistical package. I am trying to discover such basic things as why it is so much easier to press 2+4= on a calculator than =2+4 in a spreadsheet. Or for that matter why typing 2+4 at a >>> prompt is so much harder than pressing 2+4= on a calculator. My experience was that students were even more violently against Python than they were against Excel. The ** drives for powers scares them even more than the ^ does.
>
> I think the question you need to ask is 'what makes a calculator easy to use'. It may simply be that people are trained to use a calculator, long before you meet them. [...]

Probably, but even if you've never used a calculator, *learning* to use a calculator is a lot easier than learning to use a computer. The reason is simple: affordabilities. A basic calculator (I'm not talking about those TI models) has a small number of buttons, each with a single function clearly labeled. “+” means plus, and so on. Compare that to a computer: the keyboard has over 100 keys, there's a mouse, there are buttons on the screen, etc., etc. Between all that, ** for power seems a minor issue (and easily solved, as Laura showed). –GvR

I hate to admit it, but I used to do first-tier telephone tech support. :-)

One of the things I found really interesting was the degree to which things that I as a frequent computer user take *completely* for granted would be serious obstacles for new users. Particularly older users who were trying to use a computer for the first time.

To give you an idea of what I'm talking about, find a computer running an unfamiliar operating system and window environment (e.g. if you're a Microsoft Windows user, use Mac or Linux+X). Now try to do something fairly straightforward, like open a word processor and write a paragraph of text.

WITHOUT USING THE MOUSE.

First time users frequently do NOT find the “Windows - Icon - Mouse - Pointer” model completely intuitive. I once spent about a half-hour trying to explain to a guy over the phone how to *resize a window* so that he could see something that was “behind” it. Remember also that there really *isn't* anything behind the windows on your screen, and the idea of the window environment as a bunch of layered images is a carefully preserved fiction of the interface. And if you don't buy into that fiction, the screen will be severely confusing.

And this is one of many, many “intuitive” elements of working with GUI environments that new users may not find intuitive at all.

As for the relevance to topic, what I'm saying is that you are taking for granted that “typing =2+4 in Excel” and “typing 2+4= in a calculator” are very similar experiences. They are for you and me, but this is partly because we have a large body of shared knowledge which we don't even acknowledge, because it's “intuitive”.

There's another factor, too. And this may be even more relevant to your case: FEEDBACK. Conceptually, both spreadsheets and programming languages record a formula for later use, rather than providing an immediate response. The time between these two phases can be quite short, but it is there. And a person who types “2+4=” on a calculator *never actually sees* the formula “2+4=”, but only experiences it by feel. So seeing it written out on the screen as they work, instead of immediate feedback, may seem odd.

I have to admit that I find calculation by hand to sometimes feel more natural, too. One of the things about Python that I really liked was the interactive command line which allows me to have *nearly* this experience on the computer, while conceiving of a program or just doing calculations. When I realized that the functionality of “calc” was basically a complete subset of the functionality of python -- I realized I should just stop using calc and use python instead. After all, if I ever found I needed more than a few calculations, I can always write a few lines of python code interactively to do the job.

And before I developed any familiarity with spreadsheets in general, I would do calculations just like that, and then paste over the results to a word-processor or other program. I eventually did learn to use kspread, but it's not always the easiest way, especially if you're picky about layout, which I often am. –TH

After reading your e-mail message, I think a better subject would read: Computer illiteracy.

> Basically the discovery that I am making over and over is that students that have trouble with just the kinds of mathematical topics that I would have thought some experience with Python would help with, are even more terrified of computers than they are of mathematics.
>
> It isn't just Python, either, for all the talk about use of computers in the mathematics classroom as dumbing down, my recent experience is that students find it harder with computers rather than easier.

Since 6 weeks, I am a physics student and I have a lot of Calculus. Part of my Calculus course is an introduction to Maple. I quickly saw that Maple would really be able to help me a lot, but even for me (being experienced with Python and computers in general), it is a large barrier to type some integral from my Calculus textbook into Maple in order to check whether it gives the same result as I calculated. This is because it is so new: I am not used to it. At the start, it is hard indeed, because it is not intuitive. Instead of using Maple to solve the Maple exercises, some of my co-students used their TI-83 calculator to do so. This illustrates the same reflex exists at a 'higher' level. I think it is a temporary reflex, solvable by doing more exercises, and maybe doing a few steps back.

> I had statistics students who even in a distance education class where they were supposed to submit their assignments on Excel spreadsheets, would go so far as to submit something that was in tabular form in a textbox carefully using the space bar to get things to line up correctly. I had several others who would type without the = sign almost exactly the calculations that Excel would have done for them in a cell, and then repeat the same keystrokes in their calculators, and then type the answer displayed in the calculator.

I don't think this as anything to do with computer hatred or computer fear. It is simpler than that: it is computer ignorance.

> I am ideally very attracted to the ideas that KU has about integrating mathematics and programming, but my recent experience is suggesting caution. I am very much interested in your opinions about this.

A problem with using computers and programming in education is that typically, a lot of differences exist between the students pre-knowledge. This is especially true in poorer socio-economic areas of society. For people (for me at least) grown up in the “West”, it is almost unbelievable that an adult person does not know about computers, but a few years ago the former Dutch prime minister turned out not to know how to use the computer mouse. If students who do not have a computer at home, nor have ever used any, are told to go programming, it is a natural reaction to have fear for this. It is simply 10 steps too far. –GH

> Thank you, this is a very interesting post about an important topic..
>
> Please can you tell us more context and who are your students -- school, socio-economic background, age, experience etc.

They are adults, mainly in their twenties, and in the US military stationed in Europe. The backgrounds are fairly mixed. As to math level, they all knew some algebra, but not a lot beyond that.

> It sounds like they were so poorly educated and ill-prepared. Must have been terrifying and humiliating for some of them.

It was not as bad as all that. I think their main problem was lack of patience.

> When you asked them to use Excel did you ever check if they know how to use it? Give any live demos/example yourself of what you wanted? Is access to the software itself an issue? Is it even installed? etc.

Basically they got *everything*, they got what plenty pay big bucks for, and yet many still acted like it was a punishment.

My first experience specifically with Excel hatred was when I decided to simply make homework count as more of their grade, because it seemed like such a waste to make people go through the tedium required to do statistics all by hand. The students in this class were mainly business majors who I thought worked with Excel all the time, and already knew how to use it better than I did, because previous classes had seemed to want to be able to use it. But as it turned out, this class was considerably weaker, so then I spend a whole lot of extra time giving them demonstrations in the computer lab and working with them on their homework. I also made up templates on everything we did so that they really could have got away with murder, because most of the time they didn't even need to type anything in, just cut and paste and change the numbers. And most of them did do all right with it in the end, and appreciated it. But still all the way through it always seemed like there was a lot of pulling teeth, and there were two in particular that simply up and refused to have anything to do with it … I continually offered them help, but they wouldn't take it.

My next experience was in a distance education class again mainly business students, like the other class, a class with really far too much and too advanced material for the level of students where it seemed that giving them a lot of Excel templates was the only way to make it do-able. As it turned out these students were actually even less prepared than they were supposed to be due to inadequate enforcement of pre-requisites for distance education class, and this was certainly at least part of the problem. And again, it wasn't a total failure, but the main thing that surprised me was the timidity that they still had even after the whole 14 weeks of the class, things like using my templates to check their calculator work, but presenting the less accurate (due to round-off error) calculator answer when the answers differed.

Then the next term I got to teach a class that can be used as a pre-requisite for the statistics class. It is a kind of a substitute for a an intermediate algebra class, and also one that actually has spreadsheet use as part of its content. It's also a kind of a kitchen sink class that has a lot of places where use of a spreadsheet can make the concepts clearer, and I would have thought more fun. Now this time I was a lot more aware of the difficulties students have with Excel, so I thought if I devoted more time to it, it would really help them when they got to statistics. So I found room to dedicate the whole first week to it, and gave them a confidence building first assignment, and succeeded with it. And many people found what I did with Excel in the class very interesting, and they all succeeded in doing pretty much what I expected of them with it. But still now as we are nearing the end of the class, I get a couple of students saying they hate Excel, and not telling me what they hate about it. And I'm inclined to think, was all the time I spent posting things using it to explain things from the course a total waste of time for them?

> Was this shock in the first lesson and homework? Had you already covered some ground successfully? How much variation is there in the students?

I think I've probably covered these. First assignment shock wouldn't surprise me. Refusal to write a final exam on a spreadsheet after seeing solutions to homework on the all term is what really surprises me. 13th homework assignments using my templates, but with shown work calculations where the simple insertion of = before the written out work would have computed it, but instead they opted to press all the buttons a second time on the calculator, that is the kind of thing that surprises me. And then there were the face to face students who were allowed to use Excel on my computer to check their work, but none did that made me think, "Was all my time spent in the computer lab a total waste?"

> How many own their own or have good access to computers in their homes?

They all have computers at home, and also ones at work and in the computer lab. I think many even work with computers at work, which in some cases is part of the problem. When they go to class they want to get away from work. Computers are in many ways becoming a bad word in our society as a whole. They are nothing but spam, porn, and bad ergonomics.

I think my shock is not so much that all of the students had problems, but that there are some who react even more violently against computers than they do against math.

> Are they in any other classes which use computers?

I don't know.

> If so have you discussed with those teachers?

No.

> How did you deal with this situation?

Mainly I spent a massive amount of my own time both in computer labs, and online, and in some cases it still wasn't enough. But partly I know that at least with the DE students, for many the problem is that these people simply don't belong in DE classes, and our DE program is just having teething problems that hopefully will eventually get worked out.

> What do the students themselves say about their efforts to simulate the expected result?

The DE students do what bad DE students do about all their other problems, which is they tell the instructor they are lost, but refuse to give specifics. They do the same thing as they do about other material they have trouble, scream that the book is written in Greek, Chinese, and Swahili the day before the homework is due after having not participated all week. These are students that as a whole never pestered me enough, and I am not the only one to have this experience with DE students. Good students can do well in DE, but there are also a lot who seem to be practically trying to fail.

Most of the f2f students work with me on it, and really did do some nice things with their homework. But it was just always more of a struggle than it seemed it should have been.

> If you knew then what you know now, how differently would you have approach this class?

I'm not really sure. I think mainly I am facing that change can't come too quickly, and also that students need to be more hungry before they can learn. They need to discover for themselves the need for computers. Let them complain about pushing the same buttons on the calculator over and over, and beg to be allowed to use a computer. No, that's just me feeling bitter and unappreciated. But there is still some truth in it. I don't know the answer. That's why I wrote about it.

In my present College Algebra class I show them some things on the computer, but they do everything by hand, and it is going a *lot* better. I even come up with sequences of calculator keystrokes for things that I would never use a calculator for, and to me that seems like a waste. Personally I hate hand held calculators, no record, and they breed errors. I would far rather make a computation with Excel, Python, or my Pacific Tech software Graphing Calculator. But they are happier being given keystroke sequences, and my work is so much less that I finally have time to write something like this.

> I am very curious about many aspects of the story.

But maybe computer use needs to be taught from ground up integrated with math, like teaching laboratory technique in science. But math is not thought of that way. It is thought of as a kind of intelligence to have, a kind of virtuosity where using computers is cheating and unnatural. Some people still see calculators as cheating, but they have now for the most part made the running shoes level of cheating when computers are still at the steroids level. –SW

REFERENCES

The American heritage dictionary of the English language. (3rd ed.)(1992). Boston: Houghton Mifflin.

Aarseth, E. J. (1997). Cybertext: perspectives on ergodic literature. Baltimore, MD: Johns Hopkins University Press.

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

Agre, P. (1998). Notes and recommendations, from http://commons.somewhere.com/rre/1998/RRE.notes.and.recommenda2.html

Alfeld, P. (2000). Eratosthenes of Cyrene, from http://www.math.utah.edu/~alfeld/Eratosthenes.html

Anderson, B. R. (1991). Imagined communities: reflections on the origin and spread of nationalism (Rev. and extended, 2nd ed.). London; New York: Verso.

Apple Computer Inc. (1994). Inside Macintosh. PowerPC Numerics. Reading, MA: Addison-Wesley Pub. Co.

Barthes, R. (1993). Mythologies (A. Lavers, Trans.). London: Vintage.

Bayman, P., & Mayer, R. (1988). Using conceptual models to teach BASIC computer programming. Journal of Educational Psychology, 80(3), 291-298.

Bereiter, C., & Scardamalia, M. (1996). Rethinking learning. In D. R. Olson & N. Torrance (Eds.), The handbook of education and human development: New models of learning, teaching and schooling (pp. 485-513). Cambridge, MA: Basil Blackwell.

Berger, C., & Jones, T. (1995). Analyzing sequence files of instructional events using multiple representations, from http://www-personal.umich.edu/~cberger/aera95bjfolder/aera95bj.html

Boyarin, J. (1993). The Ethnography of reading. Berkeley: University of California Press.

Brooks, F. P. (1975). The mythical man-month: essays on software engineering. Reading, MA: Addison-Wesley Pub. Co.

Bruckman, A., & Resnick, M. (1995). The MediaMOO Project: Constructionism and Professional Community. Convergence, 1(1).

Bruner, J. S. (1986). Actual minds, possible worlds. Cambridge, MA: Harvard University Press.

Buchanan, R. (1989). Design discourse: history, theory, criticism. In V. Margolin (Ed.), Design Discourse (pp. 91-109). Chicago: University of Chicago Press.

Chang, K.-e. (1999). Learning recursion through a collaborative Socratic dialectic process. The Journal of Computers in Mathematics and Science Teaching, 18(3), 303-315.

Clark, A. (1997). Being there: putting brain, body, and world together again. Cambridge, MA: MIT Press.

Clements, D. H. (1999). The future of educational computing research: The case of computer programming. Information Technology in Childhood Education, 1999, 147-179.

Committee on Information Technology Literacy. (1999). Being Fluent with Information Technology. Washington, D.C.: National Academy Press.

Dede, C. (Ed.). (1998). Association for supervision and curriculum development 1998 yearbook: Learning with technology. Alexandria, VA: ASCD.

Denning, P. J. (2002). The Invisible future: the seamless integration of technology into everyday life. New York: McGraw-Hill.

DiSessa, A. A. (2000). Changing minds: computers, learning, and literacy. Cambridge, MA: MIT Press.

Downey, A., Jeffrey Elkner, Chris Meyers. (2002). How to think like a computer scientist: learning with python (1st ed.). Wellesley, MA: Green Tea Press.

Dunham, P. H., & Dick, T. P. (1994). Research of graphing calculators. The Mathematics Teacher, 87(6), 440-445.

Economist, The. (2003). Grokking the infoviz, from http://www.economist.com/science/tq/displayStory.cfm?story_id=1841120

Egan, K. (2001). Why education is so difficult and contentious. Teachers College Record, 103(6), 923-941.

Elkner, J. (2000). Using Python in a High School Computer Science Program, from http://www.python.org/workshops/2000-01/proceedings/papers/elkner/pyYHS.html

Elkner, J. (2001). Python Bibliotheca, from http://www.ibiblio.org/obp/pyBiblio/

Elkner, J. (2002). Using Python in a High School Computer Science Program - Year 2, from http://www.elkner.net/jeff/pyYHS/year02/pyYHS2.html

Figgins, S. (2000). Hackers and trackers: CP4E, from http://www.oreillynet.com/pub/a/network/2000/01/31/hacktrack/index.html

Gardner, H. (1999). Intelligence reframed: multiple intelligences for the 21st century. New York, NY: Basic Books.

Gauld, A. (2001). Learn to program using Python: a tutorial for hobbyists, self-starters, and all who want to learn the art of computer programming. Reading, MA: Addison-Wesley.

Georgatos, F. (2002). How applicable is Python as first computer language for teaching programming in a pre-university educational environment, from a teacher's point of view? Unpublished Master's, University of Amsterdam, Amsterdam, NL.

Glaser, B. G., & Strauss, A. L. (1967). The discovery of grounded theory; strategies for qualitative research. Chicago: Aldine Pub. Co.

Goldhaber, M. H. (1997a). The Attention Economy and the Net, from http://www.firstmonday.dk/issues/issue2_4/goldhaber/index.html

Goldhaber, M. H. (1997b). Attention Shoppers! Wired, 5(12).

Goldsmith, T. E., Johnson, P. J., & Acton, W. H. (1991). Assessing Structural Knowledge. Journal of Educational Psychology, 83(1), 88-96.

Goodman, F. L. (1995). Practice in theory. Simulation & Gaming, 25(3), 178-190.

Harel, I., Papert, S., & Massachusetts Institute of Technology Epistemology & Learning Research Group. (1991). Constructionism: Research reports and essays, 1985-1990. Norwood, N.J.: Ablex Pub. Corp.

Harrell, D. F. (2003). Speaking in Djinni: Media arts and the computational language of expression, from http://www.ctheory.net/text_file.asp?pick=388

Herring, S. C. (2001). Computer-mediated discourse. In D. Tannen, D. Schiffrin, & H. Hamilton (Eds.), Handbook of Discourse Analysis (pp. 612-634). Oxford: Blackwell.

Herring, S. C. (2004). Computer-mediated discourse analysis: An Approach to Researching Online Behavior. In S. A. Barab, R. Kling & J. H. Gray (Eds.), Designing virtual communities in the service of learning. New York: Cambridge University Press, see http://ella.slis.indiana.edu/~herring/cmda.html

Humphreys, K. (2002). PhraseRate: An HTML Keyphrase Extractor, from http://infomine.ucr.edu/projects/Keith_Humphrey/PhraseRate/phraserate.pdf

Ingram, A. L. (1988). Instructional design for heuristic-based problem solving. Educational Communication and Technology Journal, 36(4), 211-230.

Illich, I. (1971). Deschooling society. New York: Harper & Row.

Illich, I. (1973). Tools for conviviality. New York: Harper & Row.

Interlink. (2003). KNOT (Version 4.3). Gilbert, AZ: Interlink Inc.

Iverson, K. E. (1999). Computers and Mathematical Notation, from http://jsoftware.com/books/help/camn/camn.htm

Kafai, Y. B. (1996). Software *by* Kids *for* Kids. Communications of the ACM, 39(4), 38-39.

Kaput, J. J., Noss, R., & Hoyles, C. (2002). Developing new notations for a learnable mathematics in the computational era. In L. D. English (Ed.), Handbook of international research in mathematics education (pp. xi, 835 p.). Mahwah, N.J.: Lawrence Erlbaum.
(see http://tango.mth.umassd.edu/downloads/NewNotations.pdf)

Killingsworth, M. J., & Gilbertson, M. K. (1992). Signs, genres, and communities in technical communication. Amityville, N.Y.: Baywood Pub. Co.

Knuth, D. E. (1992). Literate programming. Stanford, CA: Center for the Study of Language and Information.

Korf, R. E. (1991). Artificial intelligence as information science. Information Sciences, 57-58, 131-134.

Koza, J. R., Keane, M. A., & Streeter, M. J. (2003). Evolving inventions. Scientific American, 288(2), 52-59.

Kozma, R. (2003). Material and Social Affordances of Multiple Representations for Science Understanding. Learning and Instruction, 13(2), 205-226.

Kupperman, J. P. (2002). Making meaningful experiences through an on-line character-playing simulation. University of Michigan, Ann Arbor. Dissertation Abstracts International - A, 63(10), 3524. (University Microfilms No. AAT 3068908)

Kyriacou, C. (1995). Direct teaching. In C. Desforges (Ed.), An introduction to teaching (pp. 118-131). Oxford: Basil Blackwell.

Lakoff, G., & Núñez, R. E. (2000). Where mathematics comes from: how the embodied mind brings mathematics into being. New York: Basic Books.

Langer, S. K. K. (1942). Philosophy in a new key; a study in the symbolism of reason, rite and art. Cambridge, MA: Harvard University Press.

Lanham, R. A. (1991). A handlist of rhetorical terms (2nd ed.). Berkeley: University of California Press.

Lanham, R. A. (1993). The electronic word: democracy, technology, and the arts. Chicago: University of Chicago Press.

Lave, J., & Wenger, E. (1991). Situated learning: legitimate peripheral participation. Cambridge, England; New York: Cambridge University Press.

Leavens, G. T. (1998). Programming is writing: why programs must be carefully evaluated. Mathematics and Computer Education, 32(3), 284-295.

Lemke, J. (1998). Analyzing Verbal Data: Principles, Methods, and Problems. In B. J. Fraser & K. G. Tobin (Eds.), Kluwer international handbooks of education; v. 2. Dordrecht; Boston: Kluwer Academic.

Liao, Y.-K. C., & Bright, G. W. (1991). Effects of computer programming on cognitive outcomes: A meta-analysis. Journal of Educational Computing Research, 7(3), 251-268.

Linn, M. (1985). The cognitive consequences of programming instruction in classrooms. Educational Researcher, 14(5), 14-16.

Linn, M. (1987). Ideal and actual outcomes from precollege Pascal instruction. Journal of Research in Science Teaching, 24(5), 467-490.

MacGregor, K. S. (1988). The Structured Walk-Through. Computing Teacher, 15(9), 7-10.

Maheshwari, P. (1997). Improving the learning environment in first-year programming: integrating lectures, tutorials, and laboratories. The Journal of Computers in Mathematics and Science Teaching, 16(1), 111-131.

Mayer, R. E., Dyck, J. L., & Vilberg, W. (1986). Learning to program and learning to think: What's the connection? Communications of the ACM, 29(7), 605-610.

McCloud, S. (1999). Understanding comics: the invisible art. New York, NY: Paradox Press.

McGill, T. J., & Volet, S. E. (1997). A conceptual framework for analyzing students' knowledge of programming. Journal of Research on Computing in Education, 29, 276-297.

Miller, J. (1996). Text Graphs, from http://www.umich.edu/~jmillr/tg/

Mitchell, E. (1999). Python 101, from http://www-106.ibm.com/developerworks/linux/library/l-python101.html

Moglen, E. (1999). Anarchism Triumphant: Free Software and the Death of Copyright, from http://emoglen.law.columbia.edu/publications/anarchism.html

Moursund, D. (1999). Project-based learning using information technology. Eugene, OR: International Society for Technology in Education.

Nadin, M. (1997). The civilization of illiteracy. Dresden: Dresden University Press.

National Research Council (U.S.) Committee on Information Technology Literacy. (1999). Being fluent with information technology. Washington, DC: National Academy Press.

Norman, D. A., & Spohrer, J. C. (1996). Learner-centered education. Communications of the ACM, 39(4), 24-27.

Oliver, R. (1993). Measuring hierarchical levels of programming knowledge. Journal of Educational Computing Research, 9(3), 299-312.

Olsen, F. (2000). Computer Scientist Says All Students Should Learn to Think 'Algorithmically', from http://chronicle.com/free/2000/03/2000032201t.htm

Olson, D. R. (1985). Computers as tools of the intellect. Educational Researcher, 14(5), 5-8.

Ong, W. J. (1991). Orality and literacy: the technologizing of the word. London; New York: Routledge.

Osgood, C. E. (1963). An Exploration of Semantic Space. In W. L. Schramm (Ed.), The science of human communication; new directions and new findings in communication research (p. 158). New York: Basic Books.

Ousterhous, J. K. (1998). Scripting: Higher Level Programming for the 21st Century. Computer, 31(3), 23-30.

Paris, S. G., Lipson, M. Y., & Wixson, K. K. (1983). Becoming a strategic reader. Contemporary Educational Psychology, 8, 293-316.

Pea, R. D. (1986). Language independent conceptual 'bugs' in novice programming. Journal of Educational Computing Research, 2(1), 25-36.

Pease, T. M. (1989). Papert Describes His Philosophy of Education. Spectrum, 1(1). See http://web.mit.edu/giving/spectrum/

Peirce, C. S. (1958). Values in a universe of chance; selected writings of Charles S. Peirce. Stanford, CA: Stanford University Press.

Pickering, A. (1995). Cyborg History and the WWII Regime. Perspectives on Science, 3(1), 1-45.

Porter, J. E. (1992). Audience and rhetoric: an archaeological composition of the discourse community. Englewood Cliffs, N.J.: Prentice Hall.

Postrel, V. I. (1998). The future and its enemies: the growing conflict over creativity, enterprise, and progress. New York: Free Press.

Preiss, B. R. (2004). Data Structures and Algorithms with Object-Oriented Design Patterns in Python. New York: Wiley.

Python Business Forum (2003). What is Python?, from http://www.python-in-business.org/python

Python Software Foundation. (2001). Introducing Python, from http://www.ibiblio.org/pub/multimedia/video/obp/IntroducingPython.mpg

Reeves, T. C. (1999). A model to guide the integration of the WWW as a cognitive tool in K-12 education, from http://it.coe.uga.edu/~treeves/AERA99Web.pdf

Resnick, M. (1996). Beyond the centralized mindset. Journal of the Learning Sciences, 5(1), 1-22.

Rorty, R. (1989). Contingency, irony, and solidarity. Cambridge ; New York: Cambridge University Press.

Sayers, D. L. (1948). The lost tools of learning. (paper read at a vacation course in education.) Oxford, 1947. London: Methuen.

Schrage, M. (2000, August). The Debriefing: John Seely Brown. WIRED, 8.

Schvaneveldt, R. W. (Ed.). (1990). Pathfinder associative networks: Studies in Knowledge Organization. Norwood, NJ: Ablex Publishing Corporation.

Schwartz, D. (1999). Ghost in the machine: Seymour Papert on how computers fundamentally change the way kids learn, from http://www.papert.org/articles/GhostInTheMachine.html

Scott, D. J. (1997). The human dimension of computer-mediated communications: a case study of preservice teachers' use of a computer conference. University of Michigan, Ann Arbor. Dissertation Abstracts International - A, 58(10), 3898. (University Microfilms No. AAT 9811190)

Shannon, C. (2003). Another Breadth-first Approach to CS I using Python. In Proceedings of the Thirty-Fourth SIGCSE Technical Symposium on Computer Science Education (pp. 248-251). Reno, Nevada.

Sherin, B. L. (1996). The symbolic basis of physical intuition: A study of two symbol systems in physics instruction. Unpublished Dissertation, University of California, Berkeley, CA.

Strauss, C., & Quinn, N. (1997). A cognitive theory of cultural meaning. Cambridge, MA: Cambridge University Press.

Swade, D. (2000). The difference engine: Charles Babbage and the quest to build the first computer (1st American ed.). New York: Viking Penguin.

Swales, J. (1998). Other floors, other voices: a textography of a small university building. Mahwah, N.J.: Lawrence Erlbaum Associates.

Taylor, K. A. (1991). An Annotated Bibliography of Current Literature Dealing with the Effective Teaching of Computer Programming in High Schools. Unpublished Masters, Indiana University, South Bend.

Trauring, A. (2003). Python: Language of Choice for EAI, from http://www.eaijournal.com/Article.asp?ArticleID=649&DepartmentID=5

van Rossum, G. (1999). Computer programming for everybody: A scouting expedition for the programmers of tomorrow, from http://www.python.org/doc/essays/cp4e.html

von Glasersfeld, E. (1998). Cognition, construction of knowledge, and teaching. In M. R. Matthews (Ed.), Constructivism in science education: A philosophical examination (pp. 11-30). Dordrecht: Kluwer Academic Publishers.

Walberg, H. J., Arian, G. W., Paik, S. J., & Miller, J. (2001). New methods of content analysis in education, evaluation, and psychology. In M. D. West (Ed.), Theory, method, and practice in computer content analysis (pp. 143-158). Westport, CT: Ablex Pub.

Webb, N. (1986). Problem solving strategies and group processes in small groups learning computer programming. American Educational Research Journal, 23(2), 257.

West, C. (1989). The American evasion of philosophy: a genealogy of pragmatism. Madison, Wis.: University of Wisconsin Press.

Wolfram, S. (2001). Q & A with Stephen Wolfram about A new kind of science, from http://www.wolframscience.com/qanda/

Wolfram, S. (2002). A new kind of science. Champaign, IL: Wolfram Media.

Yuen, A. H. K. (2000). Teaching computer programming: A connectionist view of pedagogical change. Australian Journal of Education, 44(3), 239.

Zelle, J. M. (1999). Python as a First Language, from http://mcsp.wartburg.edu/zelle/python/python-first.html