57

Why everyone should know how to program a computer

Jeffrey L. Popyack
Nira Herrmann

Drexel University
Philadelphia
USA

ABSTRACT

The notion of programming a computer usually connotes the idea of writing computer programs in general purpose, block structured languages such as Pascal, C, FORTRAN, etc. The need for people to become proficient in such languages, even for scientists and engineers, is perceived to have diminished as powerful software tools have become available. In this paper we argue that the need for students to be familiar with the basic fundamentals of programming a computer are stronger than ever, regardless of whether they intend to become computer programmers. We discuss a software based approach developed for students in all curricula which addresses this need by introducing the concepts of programming a computer in more intuitive and friendly environments than those afforded by traditional programming languages.

Main conference themes:	integration, methodologies, software
Educational areas:	higher education, secondary education
Study topics:	computer literacy
Secondary keywords:	basic skills, computing, literacy, programming, teaching materials

INTRODUCTION

Computer literacy and computer programming

There seems to be little disagreement that computer literacy is a skill one needs in order to function competently in today's and tomorrow's increasingly technological society. Such literacy usually has as its goals the ability to use a computer to perform common tasks such as word processing or information retrieval, an understanding of commonly used computer terminology and knowledge of the computer's role in society. This stands in stark contrast to the early days of computing when anyone wanting to use a computer needed to program it for the designated task themselves, either in an assembly language or a higher level general purpose language such as COBOL or FORTRAN. Indeed, most college curricula in technical areas have at some time required at least one 'computer' or 'computer science' course which has been an introduction to computer programming in a block structured language.

The late 1980s saw a trend in many disciplines away from computer programming courses and toward computer literacy or 'software tools' courses. There are many reasons for this, including:

The power of today's software. It used to be the case that a short, simple computer program produced an outcome which could not be produced through other means and was therefore nontrivial. This quickly provided enough positive feedback to motivate the introductory student to learn to write computer programs. The proliferation of powerful user friendly software for such tasks as word processing, database and spreadsheet applications has raised the level of expectation for students and removed much of the motivation for learning to write their own computer programs in a traditional block structured language. Computer games with elaborate graphics and easy to use/learn formats have also increased student expectations on what should be produced for a certain amount of effort.

Demands of the job market. There are considerably more jobs available today for persons having experience with particular software (say, Lotus 1-2-3) than for persons experienced in programming with a particular high level language (such as Pascal). Students are more likely to expect to see an immediate benefit for material they are taught.

Few programming requirements in upper level classes. For students not majoring in computer science there is a great chance they will never be required to apply their programming skills in any other class they take. They may never realize that their programming skills should transcend the particular language they learned.

Clearly, these reasons have in common a perception that computer programming is no longer relevant. In many cases this message is conveyed from faculty and curriculum designers to their students. In the next section we present reasons which show why this argument is incorrect.

WHY PROGRAM?

Computer programming concepts are more important than ever

A popular myth is that software has become so powerful that there is no longer a need to write one's own programs. This argument ignores the true source of the software's power. It is the user's ability to tell the software what to do—in essence, to program it—which gives much software the generality which makes it so powerful.

We cite the following examples as evidence of the need to understand computer programming concepts:

- Today's major spreadsheet packages provide macro languages which allow the user to custom design functions and procedures, set up nested iterations, perform conditional branching and much more.
- Word processors provide 'mail merge' capabilities which allow the user to specify a 'form letter' template which when merged with a separate data file, produces outputs with the data values substituted for variables in the template.
- Database packages allow the user to construct complex queries using AND, OR, NOT and nested levels of parentheses.
- System software often allows the user to define scripts or macros to allow storing and playing of sequences of commands.
- A telecommunications package which comes with a modem is likely to allow the user to design scripts which remember how to configure the modem for each remote machine it connects to, branch to another phone number if the first one called is busy, etc.
- Users of network services such as ftp or World Wide Web (WWW) browsers find their capabilities greatly enhanced by a knowledge of more advanced features. For instance ftp users may define scripts which will allow sequences of ftp commands to be executed. Files used for WWW browsing are written in HTML (HyperText Markup Language) or HTML+ which include variables, comments, delimiters and pointers among other features.

Each of these examples concerns concepts generally taught in a traditional computer programming class albeit with a more traditional language such as

Pascal or C. While these examples are hardly exhaustive, these are representative of the major uses of computer software in the workplace today.

In the past we have professed the argument that students would be better able to understand and use such advanced features of their software once they understood how to program. Indeed our computer science majors routinely confirm this. We have gradually come to realize that the idea of making students program in a general purpose language before learning to use the advanced features of their software is exactly backwards. Software packages provide a rich set of resources for teaching programming concepts in a framework students see as relevant to their future professional needs. Furthermore our experience indicates that students are able to successfully transfer the knowledge gained about concepts from the software environment to a general purpose language in a way which makes the general purpose language easier to learn.

This phenomenon is not restricted to software—devices such as Video Cam Recorders (VCR), microwave ovens and Compact Disc players are common fixtures in today's western society which can be used to greatest advantage if the user knows how to program them. (Jokes about people who do not know how to program their VCR abound.) We note here that the essentials of programming a VCR are the same as those taught in an introductory computer programming class—namely, storing a sequence of commands in the machine's memory which will be executed when a certain set of conditions is met; expecting input provided via external sources and producing output saved on a secondary storage device.

It is because the user interface is so drastically different that one does not often think of the command set for such a device as being an actual programming language. Yet this is exactly what it is—a means for conveying instructions to the machine in a manner it understands. Because such devices are familiar to most students, they become another meaningful context for comparison of programming features which transfer to other devices and to software environments discussed above.

We expect this phenomenon to continue for the following reason: the designers of today's software and hardware devices are experienced programmers who also intend to use the products they create. As programmers they recognize the desirability of programmable features embedded in their products and are in a position to include them as enhancements. There is a large market of potential customers to whom these enhancements are also desirable. As a result we recommend strongly that computer literacy and software tools courses contain a substantial programming component covering variables, assignment statements, conditional branching, looping, arrays, subprograms, files, absolute and relative addressing, modular structure, comments, etc. In the next section we describe our efforts to accomplish this at Drexel University.

OUR APPROACH

Since 1989 we have been developing an approach for teaching 'introduction to computing' courses tailored to the needs of students in different curricula. This approach is based on a recognition that students in all curricula need to understand the concepts of programming a computer although not necessarily how to write programs in traditional, block structured languages. These concepts are introduced through the specialized built-in macro languages available in software used by the students and through common programmable devices such as those described above.

This approach allows the student to write programs immediately which produce nontrivial results. It provides stronger motivation to learn programming than traditional approaches, such as use of pseudocode or immersion directly into a general purpose language which requires a considerable start up effort to produce a program which displays 'Hello, World!' on the screen. (Many students find this result is not in the least commensurate with the amount of effort needed to learn the material to produce it.) Our experience has shown that this concept oriented approach prepares students to learn a general purpose programming language more easily by teaching them the kinds of functionality common to all programming languages. Our approach is based on teaching programming concepts and techniques, not simply how to use software, and differs fundamentally from a 'computer literacy' or 'software tools' approach.

The authors have personally developed variants of this course for majors in bioscience and biotechnology, and for majors in mathematics, computer science, and information systems we judged at risk of not being prepared for a traditional computer programming course. We have also worked closely with others in developing this approach for majors in science, engineering, business and design arts.

We have previously reported this approach in [1, 2, 3, 4, 5]. In [4] we provided a course overview, syllabus and examples for the use of macro languages in the Microsoft Excel spreadsheet package and in [3] we described in detail how the use of the 'mail merge' facility in word processing packages provides an excellent forum for introducing many features of programming languages.

EXAMPLES

Below we describe specific features of software available to our students and how we have used them to explain programming concepts. Since 1984 all entering freshmen at Drexel have been required to have individual access to a microcomputer. To facilitate this Drexel provides Apple Macintoshes and an accompanying bundle of software to freshmen for purchase at a very attractive price. Virtually all freshmen take advantage of the bundle so that Drexel's undergraduates essentially all own compatible hardware and software. The hardware models and software platforms have varied as the state of microcomputing has matured, but the software we use has always been available in the mainstream and represents in each case a high standard of its type (whether word processing, spreadsheet, database or other).

By focusing on these concepts as each piece of software is introduced, and by illustrating how the concepts recur in each piece of software (even if the syntax may change slightly) the student is able to compare and contrast the underlying structures and how they are manifested in different contexts.

Macros

Macro utilities allow the user to store a sequence of keystrokes and/or mouse clicks which can be 'played back' by invoking a key sequence assigned by the user. Prior to System 7 a utility called MacroMaker was distributed freely with the Macintosh. AppleScript, a more robust scripting language developed by Apple, is now available with enhanced versions of the system. The following concepts can be illustrated through macro utilities:

- Creation and use of a stored program (parameterless);
- The sensitivity of program performance to data (the positions of file and folder icons used by a macro are important; the macro is defined for use only within a specific application);
- Programs which build on subprograms (macros can call other macros in the same domain of definition);
- Scoping (different macros defined for use with different applications can have the same names);
- Data files (sets of macros are stored in files).

Programming with a word processor

Students receive MacWrite Pro for word processing. The following concepts are illustrated through MacWrite:

- Input/output;
- Data files for storage/retrieval;

- Pass by reference, pass by value (the user may insert the date/time in a document, and specify whether it is fixed or updated continuously).

Additionally MacWrite has a 'form letter' (or 'mail merge') feature which allows the user to create a document containing substitution fields the values of which are filled from a separate data document. Through this feature, the following concepts are illustrated [3]:

- The use of variables and identifiers. The importance of order in specifying input.
- Data files. Output files. The creation of output through specifications written in a language understood by the computer.
- IF/THEN/ELSE statements. Nested and compound IF statements.
- Syntax vs. semantics.
- The concept of garbage in/garbage out, and debugging. (Nearly everyone has received in the mail at some time a form letter whose fields were mismatched or inappropriate)

Programming with a spreadsheet:
Students receive Microsoft Excel which can be used to illustrate the following concepts [4]:

- Defining variable names;
- Data types;
- The use of variables in assignment statements;
- The use of intrinsic functions and parameters;
- Formatting, and characters as numbers;
- One and two dimensional arrays (we find this medium to be especially useful for arrays, due to the immediate visual feedback);
- Records (Several fields associated together);
- Relative versus absolute addressing;
- IF/THEN/ELSE statements; nested and compound IF statements.

Further Excel comes with a rich macro language which can be used to illustrate all features of a programming language, especially:

- Programs;
- Use of procedures and functions, with and without parameters;
- Defining one's own procedures and functions;
- Iteration, count controlled and conditional;
- Interactive input;
- Modular programs.

The release of Excel V5.0 includes Visual Basic as another macro language. Ironically our approach of teaching programming with nontraditional languages now involves a traditional language! This affirms the notion that programmable features are becoming an increasingly indispensable part of today's software products.

Programming with a computer algebra system:
Students receive the student edition of Maple which can be used to provide further reinforcement of the following concepts in a setting useful for other courses (particularly calculus where symbolic algebra systems are seeing increasing use):

- The use of intrinsic functions and parameters;
- Defining one's own procedures and functions;
- IF/THEN/ELSE statements (For instance, to define step functions whose characteristics are provided as function arguments);
- Iteration (For instance, to study the limit of a sequence).

The preceding examples are illustrations of how these concepts have been used with particular pieces of software in use at our institution—comparable features exist in other software commonly used elsewhere.

CONCLUSIONS

While computer literacy is important in allowing users to understand and cope with computers, computer programming skills allow the user to control the computer. These skills can be learned through a concept based approach using software rather than a traditional language. Thus it may not be necessary for students and practitioners in many fields to know how to program in a general purpose language, yet the underlying concepts are more important and useful than ever and should be learned by all students. These concepts may be taught in more intuitive, powerful and user friendly environments as exist in popular software such as word processors, spreadsheets and database managers, and in common devices such as VCRs, CD players, programmable thermostats and microwave ovens.

We have been teaching introductory courses using this approach at our institution. We have found that students are more readily able to accept this approach to programming than traditional approaches using a general purpose language. Students taking this course leave it with not only a sense of computer literacy and familiarity with selected software tools, but deeper understandings of the essential components of computer software, how to identify and use advanced features, and a readiness for computer programming in other more traditional forms. We have used this approach for majors outside computer science and for computer science majors who we deemed at risk of not passing the traditional introductory computer programming course because of lack of previous programming experience.

We believe such an approach is feasible for a wide variety of students because it relies less on mathematical skills than traditional approaches. In addition the immediate creation of nontrivial results motivates students to persevere, particularly when the examples can be linked to activities they find important to them in their studies. Finally we remark that this concept based approach using software should be readily accessible to secondary school students as well as undergraduates. We feel that it can be used as a successful introduction to computing which will ready these students for further programming in whatever form they choose.

ACKNOWLEDGEMENT

Partial support for this work was provided by the National Science Foundation's Division of Undergraduate Education through grants DUE-#9254013 and DUE-#9354589 to the authors.

REFERENCES

1. Popyack, J.L. and Herrmann, N. (1991) *Using Software to Teach Computer Programming Concepts.* MacAdemia'91, University of Pennsylvania, Philadelphia, PA.

2. Herrmann, N. and Popyack, J.L. (1992) *A Software-Based Approach to Scientific and Statistical Computing for Science, Social Science, and Engineering Freshmen.* Computers Across the Curriculum: A Conference on Technology in the Freshman Year, City University of New York, New York, NY.

3. Popyack J. L. and Herrmann, N. (1993) *Mail Merge as a First Programming Language.* SIGSCE Bulletin: The Papers of the Twenty-Fourth SIGSCE Technical Symposium on Computer Science Education, **25** (1) pp. 136-140.

4. Herrmann, N and Popyack, J.L. (1994) *An Integrated, Software-based Approach to Teaching Introductory Computer Programming.* SIGSCE Bulletin: The Papers of the Twenty-Fifth SIGCSE Technical Symposium on Computer Science Education, **26** (1) pp. 92-96.

5. Herrmann, N and Popyack, J.L. (1995) *Creating an Authentic Learning Experience in Introductory Programming Courses.* SIGSCE Bulletin: The Papers of the Twenty-Sixth SIGCSE Technical Symposium on Computer Science Education, **27** (1).