Thoughts about the Best Introductory Language

Shlomi Fish <shlomif@shlomifish.org>

Revision History

Revision 1562	2006-08-04	shlomif
Forked the template from a previous work and working on it.		
Revision 1691	2007-04-10	shlomif
Finished writing the document - about to release.		
Revision 1837	2008-04-25	shlomif
Many spelling or phrasing errors corrected, and with some clarifications added.		
Revision 1838	2008-04-25	shlomif
Corrected some problems in the new text. 2nd Revision.		
Revision 2319	2009-02-27	shlomif
Added missing id's to footnotes, so they won't be randomly generated.		
Revision 4855	2011-06-05	shlomif
Convert many ASCII single-quotes and double quotes to Unicode ones.		

Table of Contents

Introduction

The purpose of this essay is to contemplate what is the best introductory programming language to teach for beginning programmers, or for a beginning programmer to learn on his own.

First, I will mention several approaches taken by other people who discussed this issue before, and try to explain why I disagree with them. Then I will propose and explain some relations ("Language A should be learned before Language B") that are good to follow. After that, I will propose my verdict, and discuss some orthogonal alternatives. Finally, I will discuss some different types of teaching and how each should be conducted differently.

As for how I started programming myself, I should note that I learned BASIC at the age of 10 (back in 1987), and then learned C when I was 15 years old (in 1992); I later learned Visual Basic for Applications [http://en.wikipedia.org/wiki/Visual_Basic_for_Applications] and when I was 19 years old I was introduced to Perl and UNIX at my workplace, which was a web site creation shop (back in 1996, when the Internet started to become popular). I have later learned other languages and technologies and still do to a large extent.

One note that is in order is that you shouldn't feel bad about having followed a different ordered in the programming languages you've learned. By all means, you can still learn things on your own otherwise.

The Various (Wrong) Approaches to Introductory Programming Languages

Linda McIver's Thesis Approach

Linda McIver [http://www.csse.monash.edu.au/~lindap/] published along with Damian Conway a paper titled "Seven Deadly Sins of Introductory Programming Language Design" [http://www.csse.monash.edu.au/~lindap/papers/SevenDeadlySins.pdf] that explains the problems they found with most popular introductory programming languages. The article makes a very good read.

Later on, her Ph.D. thesis [http://www.csse.monash.edu.au/~lindap/papers/LindaMcIverThesis.pdf] introduced her idea of a good introductory programming language.

Now, if I had to summarise this language in one word it would be this: sexless. It's incredibly limited, not flexible, and not fun. It has no pointers or references and instead relies on nested structures and arrays. There are two basic data types - a number and a string. The language does not have functions as first-order objects, closures, or objects and classes in the Object Oriented Programming sense. Furthermore, it has very few ways for one to express oneself. As a result implementing many algorithms would be very difficult in it.

When I program, I'm using every tool in my arsenal, and expect the language to be powerful enough to be able to translate my thoughts into code. McIver's language is too limited and limiting, to be effective for programming in, and being planned exclusively for beginners, lacks the richness and interesting idioms that make programmers like or even love their languages.

This is a language that I won't enjoy programming in. And I don't believe a professor who doesn't enjoy programming in a certain language can effectively convey it to his students, while lacking the enthusiasm and love for the tool he chose.

McIver's approach is flawed in the sense that she is trying too hard to save the students from all possible problems they may encounter in trying to understand their introductory language. However, programming is hard to learn, and learning the first language is always difficult. Creating a "flawless" language that lacks any sex-appeal is not going to make it better, but much worse as both the professors and programmers will detest it.

The "Structure and Interpretation of Computer Programs" Approach

"Structure and Interpretation of Computer Programs" [http://mitpress.mit.edu/sicp/] (or SICP for short) is a classic text and course material on programming, taught at MIT and many other universities around the world. SICP uses Scheme (a minimalistic dialect of Lisp) as its exclusive language to cover many important programming and meta-programming concepts.

I have read the book in my third semester of the Technion (without doing the exercises) and later took both of the SICP courses that were given by my department. I learned a lot from the book, and while the courses did not teach me too much new, I did enjoy working on the exercises.

However, there are several problems with teaching Scheme as an introductory language. The first is that it is too impractical. Scheme does not have system primitives that more modern languages take for granted like ones for random file and directory I/O, sockets, graphics primitives, Graphical User Interface (GUI), etc. Moreover, the core language is limited and most practical code tends to become very verbose in it. For example, whereas in Perl one would write `$myarray[$i]++` to increment an array element by one, in Scheme it would be: `(vector-set! myarray i (1+ (vector-ref myarray i)))`.

Most of the SICP exercises are about number theory, recursion, and a lot of other relatively abstract stuff, and too few are about real world and exciting tasks: writing games and other demos, working with files, writing scripts and utilities, networking and working with the WWW, etc. In fact, the Scheme standards define too few useful things. Most of the dazzling number of different Scheme implementations [http://community.schemewiki.org/?scheme-faq-standards#implementations] all extend the language in several ways, but all have their own idea of how to do it. Compare it to Perl, Python and friends which have one main C-based implementation, or to C where the standard library is actually quite useful.

I believe an introductory language has to grow with you. When I studied BASIC, I was able to use it for programming games, graphical demonstrations and animations, scripts, and other uses. I continued to use BASIC on DOS and Windows, until I learned the much-superior Perl, which I'm using today.

The "Teach in C" Approach

In his "Back to Basics" essay [http://www.joelonsoftware.com/articles/fog0000000319.html], Joel Spolsky gave a case for teaching C as an introductory language instead of more high level languages. His argument is that programmers will end up writing sub-optimal code because some low-level elements of dealing with strings and arrays are abstracted away in higher-level language.

C and C++ have been popular introductory languages for teaching programming for many years now. While some schools have switched to teaching Java or a different language, C and C++ are still very popular.

However, C has one major deficiency: it's too close to the processor to be useful. In order to perform an operation on two objects, one should allocate them first, perform the operation, and then take care of freeing both objects and the result (to say nothing of edge cases where allocating or freeing may fail.).

All this work to do something that in high level, garbage collected, languages is as simple as `$result = $object1 OP $object2;`. From my experience with Technion students, they are often get so bogged up in the technicalities of working with C instead of getting quick, dirty and useful code running.

A good introductory programming language should allow you to write a lot of useful code quickly, and not slow you down with many low-level constraints. Beginning programmers have a hard enough time learning how to translate their thoughts and intentions into working code, and solving bugs and the last thing they need is to deal with too many idiosyncrasies of the language only because it is too low-level.

Spolsky's argument about the efficiency of some operations is wrong, because programmers who learn such languages won't often notice the difference from such inefficient operations, due to the incredible speed of contemporary computers and the fact that their data sets are generally too small. Moreover, many instructors and exercise checkers won't penalise for the presence of such issues in their homework.

While the efficiency of algorithms and the underlying implementation of language primitives should be stressed at a certain point, the first task of an introductory course is to make sure a programmer can learn to write code, not necessarily the most efficient one. (Not even according to asymptotic complexity). Learning how to write quick and dirty code is a mental leap that is large enough as it is.

The "First Programming Language Should Make Sure You Write Good Code" Fallacy

You many times hear people saying that beginning programmers should be taught using a programming language that restricts them and forces them to write good code. Languages like Pascal, Ada, Java, and many others were designed to try to save programmers from themselves. And indeed many people believe that programmers should start learning from such a language.

What's wrong with this approach? Several things:

- The more strict the language is, then generally the less expressive it is. Programmers like to express themselves and be able to implement algorithms using the entire power of the language. They don't want to declare a lot of type definitions, many constraints, write a lot of syntax, or otherwise be encumbered in the way.

  It may actually make them think programming is loathsome or otherwise a very strict process instead of a very creative process.[haskell]

- Often, the trial and error will be good for them. Plus, even writing some disorganised, but functional code is better than the program taking them much more time to write (and more time to read and understand after writing).

  I don't expect them to become superb programming in a day. Becoming a better programmer is a process, and cannot be taught in a semester or a year of hard work.

[haskell] A comment to the first revision of this article [http://lambda-the-ultimate.org/node/2194#comment-28811] claimed that "Some languages, like Haskell, derive their expressive power exactly because of the restrictions imposed". My reply is that arguably languages like Lisp have the same expressive power, but obviously a more verbose syntax due to the fact they are using S-expressions [http://en.wikipedia.org/wiki/S-expression] and that they are lacking some functions that were added to Haskell, O'Caml and SML. Perl 6 [http://dev.perl.org/perl6/] aims to combine more idioms from Haskell and Lisp than Perl 5 already has, yielding a language that's generally even more succinct.

I believe that Haskell derives its expressiveness not from its strictness, but rather from its abstractions, and that this expressiveness can be duplicated to a large extent in a less strongly typed language. However, my mastery of Haskell is still somewhat superficial, and so I'm not fully qualified to comment on it.

The “It Should Have a Decent IDE” Fallacy

Many education institutions reject many languages as introductory languages because they don’t have a decent integrated development environment [http://en.wikipedia.org/wiki/Integrated_development_environment] (or IDE for short). An IDE as useful and convenient as it is, however, is not an absolute requirement.

Programming does not happen in the IDE - it happens in the mind. Programmers should learn to write code that does something. By using the text editor (of the IDE or a standalone one) and writing text that does something, they can best learn to program for the real world.

There is a myth that programming using a text editor and a command line is too difficult for mortals. This is false because, as late as the 1980’s or 1990’s, almost all personal computers used a command-line interface (often a BASIC interpreter or DOS), and required programming using non-graphical editors, and it was still adequate for most people. (To say nothing of earlier interfaces such as Teleprinters (TTYs) [http://en.wikipedia.org/wiki/Teletypewriter] or punched cards). Plus, it is hard for a programmer to avoid typing code entirely.

Some useful relations

This section will introduce some useful relations (“Language A should be taught before Language B”) to consider in teaching programming, and explain them. By using these relations one can more easily reach a final verdict.

A High Level Language Should Come Before C

C should not be taught as a first programming language from the reasons I have mentioned above. By all means, one should use a more high level languages which supports Managed programming, and other nice high level constructs. Languages like Perl, Python, Ruby and to a lesser extent Java and .NET are much better than C as introductory languages.

Perl/Python/etc. should Come before PHP

Some people believe that PHP is a suitable introductory language. However, PHP has several major problems: lack of good abstraction mechanisms, many inconsistencies, many functions to do the same thing, and many nuances to its use. People who learn PHP right away, tend to write very bad (and sometimes very dangerous) code in it, and are not well-aware of its pitfalls.

PHP is a fine language for the web and for other uses, especially because its implementation makes deployment of some large-scale web applications easier. However, the other languages in the so-called “dynamic”, “agile”, or “scripting” class of languages are not harder to learn, and less problematic. So they should be taught first instead.

Perl/Python/etc. should Come before Shell

Some people believe that the first language a UNIX user should learn is a good shell (such as GNU Bash [http://www.gnu.org/software/bash/] or zsh [http://www.zsh.org/]). However, Shell has some issues. The first is that the mentality of the UNIX Shell is different from the mentality of conventional programming languages, and causes native shell programmers to be less capable of adapting to a different language, as well as writing sub-optimal code in shell.

The second is that in traditional shell, some operations are not as efficient as they should be. While more modern variants have introduced arrays and string-wise dictionaries, they are still an afterthought. For these reasons, shell is not recommended to learn before a dynamic language.

C should Precede Assembly

It is certainly a good idea to learn Assembly language [http://www.onlamp.com/pub/a/onlamp/2004/05/06/writegreatcode.html], preferably of several different processor architectures. However, C should be learnt first.

The reason for that is that people who dive right into Assembly, tend to write sub-optimal code because they don't understand well how this code is executed by the processor and how to compile it. This is while programmers who've learned C are better equipped to understand how Assembly code works, because it is somewhat more convenient yet still very close to Assembly.

A friend of mine reported that in his workplace, where they write Assembly code for various Digital Signal Processors (DSPs) some of the native Assembly programmers order their instructions in ways that are executed inefficiently because of the special processor pipeline. He then told me that C programmers who learn Assembly make better Assembly programmers.

The First Language should be Practical

A good first programming language should be practical and should grow up with you. I can tell from my experiences with the various BASICs, which were the first languages I learnt, that BASIC was fun because it was useful. Using BASIC on the old Intel-based computers, one could write games, graphical demos, text processing and command execution scripts, and even serious applications. While BASIC is in today's standards a very limited language that should no longer be taught as a first language, I still fondly remember it as being a lot of fun. I even continued using BASIC after I learned C and what was then C ++, because it was quicker and more convenient. (I no longer do, because I now feel that Perl is superior to BASIC in every way, and that's what I'm using now.)

On the other hand, Scheme as in SICP is an awful choice for an introductory programming language, because it feels very impractical. Writing quick and dirty code to do a lot of things in Scheme is very verbose, and plus, the core standard lacks many primitives for common POSIX [http://en.wikipedia.org/wiki/POSIX] operations (like random file I/O, directories, sockets, etc.) much less useful APIs. While some Scheme implementations provide extensions to the language, they do so in different incompatible ways.

Different people I talked to, agreed with me that "You cannot do anything with Scheme". Compare it to languages such as C and C++, Perl/Python/Tcl/Ruby/PHP, Java/.NET, etc. that feel very practical, and you'll see why hardly any industrial-strength code is written in Scheme.

Teaching a language just for teaching programming with, is sub-optimal because the students cannot take this language with them and perform real-world tasks with it. They will have less motivation to experiment on their own, and to remember it for long.

Localised Programming Languages should be Avoided

The Wikipedia has an (incomplete) list of non-English based programming languages [http://en.wikipedia.org/wiki/Category:Non-English-based_programming_languages], that were created at some time. What these languages try to do is make sure young children or other people who did not master the English Alphabet and vocabulary well can start learning programming without knowing English first.

I see several problems with this approach. One is that it is important that children will be taught English starting from an early age - as early as possible. This is because English, being the international language, is becoming more and more important for every one to learn. Tender children who are talked to in several languages, will quickly master them, without confusing them. This will save them a lot of frustration later. (By all means if one happens to know other languages, he should talk to his children using them too, but that is beside the main point.)

Knowledge of English is more important than knowing how to program. So it is a good idea that when teaching programming to teach English first as a necessary pre-requisite.

The other problem I see is that such localised programming languages feel unnatural and wrong. English has the richest technical vocabulary of any other language, and some terms in English are impossible to translate to other languages. And yet another is that such languages tend to be very ad-hoc and incomplete. Finally, code that is written in them cannot be understood by programmers who don't know this language.

So, to sum up, instead of starting with a localised programming language, teach your students some basic English first. It might take longer, but will save more time and frustration later on. Plus, programming is a great way to expand one's mastery of English, especially today when the Internet is so prevalent.[globalisation]

Java Should be Taught After Perl

Joel Spolsky wrote an essay titled "The Perils of JavaSchools" [http://www.joelonsoftware.com/articles/ThePerilsofJavaSchools.html] where he argued that teaching Java in Computer Science curricula is inferior to teaching C and Scheme, which was what he learned. The article is wrong on many points, but it highlights some of the problems with Java.

Java is too verbose. Some people may argue that this can be solved by using a proper IDE, but as Paul Graham explains [http://www.paulgraham.com/popular.html], verbose code also has the "the cost of reading it, and the cost of the space it takes up on your screen.".

Moreover, Java code tend to be very monotonous. Almost all Java code looks the same, and feels boring.

Steve Yegge's very funny article "Execution in the Kingdom of Nouns" [http://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html] illustrates another problem with Java. Everything has to be a noun, with no verbs or even the many keywords which Perl 5 is infamous for but which Perl programmers love. And instead of having some Perl 5-like operators for converting between data structures, you have a hideously long casting lines.

Java was supposed to be kept simple, and many important concepts like closures, multiple-inheritance, defining methods at runtime (a la Smalltalk), runtime code evaluation (the Lisp-derived "eval" operator, which is now common in most dynamic languages), operator overloading, and many other elements had been kept out of it. As such it turned out to be very unusable. Java 1.5/5.0 introduced many drastic enhancements, but not enough proper abstractions. As a result, Java is now bloated, but talented programmers still normally find writing code in Perl, Python and friends more natural.

Paul Graham's essay Java's Cover [http://www.paulgraham.com/javacover.html], which he wrote to explain why he decided not to learn Java is very instructive. I read Graham's article, some time after it has been written and felt it reflected my feelings about the language. Back when Java started to become hyped , I had ended up learning Java to see what the hype was about and to write some browser applets. While having felt that I have truly understood what the essence of references in Perl 5 was, only after learning Java, I still felt that Java was too over-rated.

---

globalisation A few people who read this article claimed I was advocating globalisation. However, consider what Eric Raymond writes in "How to Become a Hacker" [http://catb.org/~esr/faqs/hacker-howto.html]:

> 4. If you don't have functional English, learn it.
>
> As an American and native English-speaker myself, I have previously been reluctant to suggest this, lest it be taken as a sort of cultural imperialism. But several native speakers of other languages have urged me to point out that English is the working language of the hacker culture and the Internet, and that you will need to know it to function in the hacker community.

One should note that the proliferation of English today is not the first time that there happened to be a Lingua franca [http://en.wikipedia.org/wiki/Lingua_franca] in the world or a limited part of it. I also feel that having one spoken language that everyone of importance is familiar with (although possibly not so well) is better than not having any good common way of communication, and thus was shown to be inevitable times and again in history.

Perhaps I'm getting too carried away in criticising Java. My point is that, as Joel Spolsky indicated in his "JavaSchools" essay, teaching Java as the first language, makes many of the people who have learned it airheads, who cannot think outside the limited constraints that it imposes on the programmer. Teaching an expressive and rich dynamic language [http://en.wikipedia.org/wiki/Dynamic_programming_language] such as Perl or Ruby instead, will not exhibit this problem, regardless of what Joel says, as these languages constantly require a programmer to think outside the box, and introduce the programmer to many different (often built-in) patterns and paradigms.

My Verdict

According to these constraints one can conclude that one should start learning how to program from a high-level, dynamic and practical language such as Perl, Python or Ruby.

Eric Raymond recommends this in his excellent "How to become a Hacker" document [http://catb.org/~esr/faqs/hacker-howto.html]. He suggests one should start with XHTML, which while not being a programming language but rather a formatting language will still introduce many programming idioms and disciplines as well as prove useful later on.

After XHTML, Raymond recommends one to learn Python. However, I'm not sure whether Perl 5 or Ruby will not be as suitable as Python, or more. Unfortunately, I cannot reach a conclusion here, but rather give some of my thoughts on each three languages.

(If I need to teach programming, I'll start with Perl because I know it very well, and like it a lot. However, programmers who are well versed in Python or Ruby, may wish to teach them instead.)

Perl, Python or Ruby

Perl

The core Perl language is huge. That may be a good or a bad thing for teaching programming in. The Perl language can be usable by learning only a small subset of the language. However, as budding Perl programmers learn more they tend to diverge in the what they know, and use different subsets, which makes understanding code of peers with different background (much less experts) more problematic. This problem is naturally not limited to Perl 5, and given good, searchable documentation can be made less substantial, but is still a pedagogical hurdle.

Perl is very expressive. I believe programmers will appreciate its "There is more than one way to do it" philosophy. A correspondent once told me he'd prefer to teach beginners Perl instead of C, similarly to the fact that he'd prefer to teach English over Esperanto, because beginners would prefer a language that allows them to express themselves. [Esperanto]

Historically, Perl had a lack of good online documentation for beginners, and other problems with the treatment of newcomers [http://www.shlomifish.org/philosophy/perl-newcomers/], but this has improved lately.

Perl has a rich (and so far unmatched) collection of re-usable modules that provide functionality called CPAN - the Comprehensive Perl Archive Network [http://en.wikipedia.org/wiki/CPAN]. Uploads to CPAN are not moderated (on purpose) and therefore it is sometimes hard to find a suitable CPAN module

[Esperanto] Several people contacted me saying I have misrepresented Esperanto here. I should note that I'm quoting someone else, and I admit that I don't know Esperanto well enough to be sure if it indeed suffers from many problems attributed to artificial languages.

The point is not to dismiss Esperanto, but rather to say that many people appreciate expressibility, and some of them also appreciate irregularity (or even inconsistency) in their spoken or programming languages, as it makes life more interesting.

out of the many bad or unsuitable ones (if there actually is one available). [rethinking-cpan] They may prove useful in teaching programming in Perl.

Perl has a rich and active culture surrounding it, including many diversions as obfuscated code, golf challenges [http://perlgolf.sourceforge.net/], riddles, many specialised mailing lists, Local Perl Mongers groups, and conferences.

Python

Python has a small core language and it tries to be elegant. It has an excellent online documentation, and many introductory books for it are available online. The online Python community has too much elitism, and tends to deprecate Perl a lot, for some reason. I am not blaming anyone in particular, but this tendency is present to some extent by some of the greatest names in the Python world, and by some Pythoneers I personally know.

People who know Perl very well, can learn Python with fewer mental blocks than the other way around. This is in due to the fact Perl is richer, and supports more paradigms. A Perl programmer told me he was able to start working on a Python program right after starting to edit it using his editor, and it worked, after some research.

Python's philosophy is "There's one good way to do it.". It doesn't mean that there aren't other ways, but there is one commonly acceptable way to write most code. Whether this is a good thing or not for an introductory language is debatable.

If PHP is the new Visual Basic, and Java is the new COBOL, then Python is the new Pascal. (Although, all these languages are better than their previous ones). In a way teaching Python as a first language, like teaching Pascal, makes a programmer used to limited paradigms and one strict way of doing things. (like teaching Esperanto instead of English). As a result, trying to learn other diverse languages is becoming more difficult.

If you've learned Python as your mother language, you should take the mental leap and learn Perl, which is the Tower of Babel of languages, and also has many DWIMmeries ("Do-What-I-Mean"'s) and other expressiveness. (Of course, a Perl programmer should also learn Python due to its elegance, and the fact it is extensively used and useful.)

Ruby

Before I discuss Ruby a word of warning: I don't know it very well. So far all the limited tasks I tried to accomplish using it worked well after some trial and error, but I still did not take the time to thoroughly study it.

Ruby was written after its creator was unhappy to some extent with both Perl (possibly 4 at the time) and Python, and so he created a language that tried to combine the best elements of Smalltalk, Perl and Python. Ruby aims to be elegant and consistent, yet still very expressive and shares Perl's "There's more than one way to do it" philosophy.

As of version 1.x, Ruby does not support multi-threaded programming, has poor support for Unicode, and is much slower than Perl or Python. Some of these problems will be addressed in Ruby 2.x.

The worst problem with Ruby, however, is the lack of good documentation. Ruby has one old edition of the "Programming Ruby" book [http://www.rubycentral.com/book/] available online, and that's it. Furthermore, this book is intended for absolute beginners and will be too slow paced for people with extensive experience in similar languages.

[rethinking-cpan] As of April, 2008, there is an effort under-way to revamp the CPAN experience [http://perlbuzz.com/2008/04/rethinking-the-interface-to-cpan.html]. The author of these lines is heavily involved with it, so he may be a bit biased. Plus, the effort is still in its infancy.

All the other books from the Pragmatic Programmer series are not available online (including the new editions of the "Programming Ruby" book). What many people end up doing is downloading them from "warez" sites or from Peer-to-Peer networks, but I wouldn't encourage professors to tell their students to do that.

I recall trying to find out how to tag methods in Ruby, in a similar way to Perl's method or variable attributes. Google was no help and no one on Freenode on #ruby-lang told me and I asked several times, and people tried to research it. Eventually, someone I knew on #perl was able to give me the answer. He then claimed that many of the slightly more unconventional, but useful, tricks in Ruby were completely undocumented.

As such, one may still encounter problems teaching Ruby as an introductory language. If these problems are remedied by the Ruby community, with some amount of work and effort, then this may be better.

Final Verdict

All things considered, I'd say that Perl is the best choice now, as Python is too strict and unexpressive, and Ruby is documented in an extremely inadequate way. Again, any of the three languages would be a fine choice, and all of them should be learned by any programmer who is worth his weight in salt.

Note that other than the main players in the dynamic language arena, there is the new crop of such languages: Lua [http://www.lua.org/], Io [http://www.iolanguage.com/], The D Programming Language [http://www.digitalmars.com/d/], and others. These languages may be more suitable in some respects, but on the other hand, may not yet have the brain-share, comprehensiveness (especially as far as APIs are concerned), usability, richness or "sex-appeal" [consistency].

Some Types of Teaching

There are several different types of teaching programming to laymen. This section aims to cover the most important ones and what needs to be considered when they are done.

The first type I'll discuss is a self-teaching enthusiast who is trying to teach himself programming, perhaps with some help from his friends or people he is interacting with on the Internet. Such an enthusiast usually has a lot of motivation to learn, but on the other hand, will probably not put up with a material that bores him or seems trivial.

The second type is a programmer who tries to teach a child or a teenager programming. Such youngsters are often mostly motivated by things that seem fun to them: games, demos, drawing pretty pictures programmatically, etc. They will have little nerve for a tedious programming language such as C, in which every task takes a boatload of code.

A different type of pedagogy altogether is introducing programming to students in university. Such students are older, have more mathematical background, and will find other things aside from games enjoyable. On the other hand, they tend to have less willingness to experiment on their own, or to play with the computer. They expect to learn programming so they can either go on with their degree, or use it to learn the rest of their degree.

When people teach programming in the so-called K-12 school (i.e. pre-college or university), then such students will have less mathematical background than their college counterparts, and may find learning

[consistency] Some people assume that the more consistent a language is the better. However, just as most people prefer expressive and inconsistent natural human languages like English, many of them would prefer their programming language to have some inconsistencies, Do-what-I-mean-erries, gotchas, etc. In Perl 5's case it is well known that these make the language more expressive and succinct [http://www.paulgraham.com/power.html] in the hands of a competent programmer.

programming (as they find learning most everything) a burden. On the other hand, they tend to be brighter and more curious.

The final type of teaching is in training courses. It is known that such people often have to be spoon-fed the material. Plus, they may not be as bright as those who were accepted into high-class universities or colleges.

How does this influence the choice of the introductory language? It probably doesn't. However, it influences the way the language should be taught and which parts of it should be taught first.

Conclusion

I talked with a few people on the IRC about it and some of them told me something along the lines of "What makes you think that you know better than all the universities and colleges (and other schools) that are now teaching Java?". Well, this is the majority must be right fallacy:

1. Everybody thinks that the Earth is flat (or the Sun revolves around it) so it must be true

2. Everybody thinks that drugs should be illegal [http://www.shlomifish.org/philosophy/politics/drug-legalisation/] so it must be true.

Etc. I can think of many other cases where a common consensus, even among experts turned out to be false. But I'll still explain a bit.

Universities have tended to teach the "hottest" language on the market. They used to teach Assembler. They used to teach COBOL (an awful language by all means, and one which proved to be a dead-end in language design). They taught Fortran and PL/I. They taught Pascal. They taught C and C++. And now they teach Java. I believe none of these languages were suitable as an introductory programming language, but they were taught because they were used in the industry.

During the course of IT education, several languages need to be studied - at least one dynamic language such as Perl, Python or Ruby ; C; an assembly language; Lisp (Scheme, Common Lisp or perhaps now Arc [http://www.paulgraham.com/arc.html]); Haskell, O'Caml or SML; and probably some specialised languages when they are appropriate. But the first language need not be what is the most hyped language in the industry, or even what most the rest of the studies will be conducted in.

From my impression of the Technion [http://www.shlomifish.org/philosophy/computers/education/opinion-on-the-technion/], the institute as a whole believes that students can effectively write all their code in C. In some courses, the choice of C++ and Java are given, but these languages are not effectively taught. Most students, during their studies, had not been exposed to such advanced paradigms as regular expressions, dynamic-typing, Perl 5-like nested data structures, run time evaluation, closures and dynamic functions, and others that are considered common knowledge among developers of dynamic languages, and any software development enthusiast who is worth his weight in salt.

So my opinion still remains: Perl, Python or Ruby are the best languages for introducing non-programmers to programming, while Perl is the best, and Python is probably still the worst of the three. However, note that any decent programming training will introduce his developers to more than one language, and a prospective programmer should not worry if he started out with a language that I consider sub-optimal. With good ambition and motivation and with the right attitude ("I know that I do not know"), one can become a better and better programmer regardless of his initial background.

Other Good Food for Thought about Teaching

This section will bring other good for thought about teaching.

“Live as if you were to die tomorrow. Learn as if you were to live forever.”

This is a quote attributed to Gandhi [http://en.wikipedia.org/wiki/Mahatma_Gandhi]. The “Learn like you were going to live forever” part is not widely understood by many workers. Many programmers believe that their knowledge of a few programming languages is enough, and that it is not necessary that they learn completely different ones.

It is well known that learning a new and different programming language will make you a better programmer also in the original languages you know. Programmers who don’t learn new programming languages eventually stagnate. They are bounded by their limited knowledge, and cannot think outside their box. They deserve the stagnation they receive due to this bad attitude, and mental laziness.

If you want to grow as a programmer, make sure you keep studying new languages and technologies. Not only they may turn out to be useful, but they’ll also make you think in completely different ways.

Three Levels of Learning

Rabbi Hanina used to say “I learned a lot from my teachers, and from my friends more than my teachers, and from my pupils the most.” I believe this means that there are in fact three levels of learning:

1. **Level 1 - Learning** - this is a passive learning of the material, where one inputs the material.
2. **Level 2 - Experiencing** - in this level you work with the material you learned, and try to implement what you’ve learned and integrate it. This requires more understanding, because you have to work with the material.
3. **Level 3 - Teaching** - in this level you teach the material to someone else. This requires the most understanding because you need to organise it properly and convey it to someone else.

Perhaps there’s a fourth level - **Science** in which the knowledge is expanded. However, this implies that to truly understand the material, one needs to experiment with it (preferably in production) and better yet teach it to someone else.

The old adage “He who can - does. He who cannot - teaches.” which was said by George Bernard Shaw [http://en.wikiquote.org/wiki/George_Bernard_Shaw] is amusing, but simply not true, as I’ve demonstrated here. Being a great teacher is much more difficult than being a great doer, and is much more enlightening. [those_who_can]

Learn as Many Languages as Possible

Learning one computer language is not enough. Knowledge of only one computer language or a few cripples the mind and causes the brain to run in circles. Different programming languages introduce different insights: various easier ways to do certain things, different restrictions , different syntax, different APIs, different ways of doing things, different high-level mechanisms (or lack of them). All of this gives different understandings of how to program in any language.

Many people believe that their limited knowledge is adequate. Java programmers are especially notorious for being opposed to the ideas of them having to learn different languages. The Pragmatic Programmer book [http://www.amazon.com/exec/obidos/ASIN/020161622X/ref=nosim/shlomifishhom-20] says a

[those_who_can] What is true, in my opinion. is that “Those who can - do. Those who can’t - complain.” However, often people who can and do, still complain. I recall this quote being attributed to Linus Torvalds [http://en.wikipedia.org/wiki/Linus_Torvalds], but it predates him [http://shlomif.livejournal.com/39215.html].

programmer should learn a new computer language at least every year, and I tend to agree with it. I compiled a tentative list of the technologies I found the most enlightening [http://www.shlomifish.org/philosophy/philosophy/advice-for-the-young/#technologies_to_learn], and I recommend programmers to learn at least all of them.

Learning How to Read Code and Enhance Existing Code

At present, universities and other spend most time teaching programmers how to write code. However, most of what programmers have to do for work or for pro-bono work (like open source projects) is to read code, and to enhance existing code.

Joel Spolsky ("Joel on Software") [http://www.joelonsoftware.com/] gave the following "cardinal rule of programming" in his famous "Things you must never do, part I" essay [http://www.joelonsoftware.com/articles/fog0000000069.html]:

> It's harder to read code than to write it.

My friends and I later discussed this topic in the Hackers-IL mailing list [http://tech.groups.yahoo.com/group/hackers-il/message/3576]. Even if code is given for reading in university, it is usually extremely well-written, highly organised, highly legible, code, rather than the real code that programmers are likely to encounter in the wild.

It's a shame most of the code students write as part of their curriculum is only for themselves, and ends up being of little value to the world at large. Even if some code ends up as an open source project, it is usually too incomplete and lacks essential functionality or correctness to be of any use in the real world.

As Joel points out in the article, most programmers end up saying that the code they are working on is horrible and that they wish to completely rewrite it if they have the chance, instead of refactoring [http://www.refactoring.com/] it to make it better.

Furthermore, since reading code is harder than writing it, then it makes sense that programmers who are good at reading (or refactoring code or enhancing it) are much better programmers, than programmers who are only good at writing new code. I wish I had a dollar for every time I heard of someone trying to rewrite an existing functional and relatively bug-free codebase from scratch, just because this codebase was deemed of too little quality, and that afterwards this rewrite ended up at nothing. These cases practically dwarf the number of successful rewrites I recall.

To sum up, it will be a good idea to teach first-time programmers how to read real-world code, or the code written by their co-students, and how to enhance it by extending it, and cleaning it up.

Thanks

Thanks to Pete_I on Freenode, Omer Zak [http://www.zak.co.il/], chromatic [http://www.wgz.org/chromatic/], Jonathan Scott Duff, Sagiv Barhoom and others for reviewing early drafts of this essay and giving some editorial assistance.