Teach Yourself Programming in Ten Years

Peter Norvig

Why is everyone in such a rush?

Walk into any bookstore, and you'll see how to *Teach Yourself Java in 24 Hours* alongside endless variations offering to teach C, SQL, Ruby, Algorithms, and so on in a few days or hours. The Amazon advanced search for [title: teach, yourself, hours, since: 2000 and found 512 such books. Of the top ten, nine are programming books (the other is about bookkeeping). Similar results come from replacing "teach yourself" with "learn" or "hours" with "days."

The conclusion is that either people are in a big rush to learn about programming, or that programming is somehow fabulously easier to learn than anything else. Felleisen *et al.* give a nod to this trend in their book *How to Design Programs*, when they say "Bad programming is easy. *Idiots* can learn it in *21 days*, even if they are *dummies*." The Abtruse Goose comic also had their take.

Let's analyze what a title like *Teach Yourself C++ in 24 Hours* could mean:

- **Teach Yourself:** In 24 hours you won't have time to write several significant programs, and learn from your successes and failures with them. You won't have time to work with an experienced programmer and understand what it is like to live in a C++ environment. In short, you won't have time to learn much. So the book can only be talking about a superficial familiarity, not a deep understanding. As Alexander Pope said, a little learning is a dangerous thing.

- **C++:** In 24 hours you might be able to learn some of the syntax of C++ (if you already know another language), but you couldn't learn much about how to use the language. In short, if you were, say, a Basic programmer, you could learn to write programs in the style of Basic using C++ syntax, but you couldn't learn what C++ is actually good (and bad) for. So what's the point? Alan Perlis once said: "A language that doesn't affect the way you think about programming, is not worth knowing". One possible point is that you have to learn a tiny bit of C++ (or more likely, something like JavaScript or Processing) because you need to interface with an existing tool to accomplish a specific task. But then you're not learning how to program; you're learning to accomplish that task.

- **in 24 Hours:** Unfortunately, this is not enough, as the next section shows.

Teach Yourself Programming in Ten Years

Translations

Thanks to the following authors, translations of this page are available in:

Arabic
(Mohamed A. Yahya)
العربية

Bulgarian
(Boyko Bantchev)

Chinese
(Xiaogang Guo)

Croatian
(Tvrtko Bedekovic)

Esperanto
(Federico Gobbo)

French
(Etienne Beauchesne)

German
(Stefan Ram)

Researchers (Bloom (1985), Bryan & Harter (1899), Hayes (1989), Simmon & Chase (1973)) have shown it takes about ten years to develop expertise in any of a wide variety of areas, including chess playing, music composition, telegraph operation, painting, piano playing, swimming, tennis, and research in neuropsychology and topology. The key is *deliberative* practice: not just doing it again and again, but challenging yourself with a task that is just beyond your current ability, trying it, analyzing your performance while and after doing it, and correcting any mistakes. Then repeat. And repeat again. There appear to be no real shortcuts: even Mozart, who was a musical prodigy at age 4, took 13 more years before he began to produce world-class music. In another genre, the Beatles seemed to burst onto the scene with a string of #1 hits and an appearance on the Ed Sullivan show in 1964. But they had been playing small clubs in Liverpool and Hamburg since 1957, and while they had mass appeal early on, their first great critical success, *Sgt. Peppers*, was released in 1967.

Malcolm Gladwell has popularized the idea, although he concentrates on 10,000 hours, not 10 years. Henri Cartier-Bresson (1908-2004) had another metric: "Your first 10,000 photographs are your worst." (He didn't anticipate that with digital cameras, some people can reach that mark in a week.) True expertise may take a lifetime: Samuel Johnson (1709-1784) said "Excellence in any department can be attained only by the labor of a lifetime; it is not to be purchased at a lesser price." And Chaucer (1340-1400) complained "the lyf so short, the craft so long to lerne." Hippocrates (c. 400BC) is known for the excerpt "ars longa, vita brevis", which is part of the longer quotation "Ars longa, vita brevis, occasio praeceps, experimentum periculosum, iudicium difficile", which in English renders as "Life is short, [the] craft long, opportunity fleeting, experiment treacherous, judgment difficult." Of course, no single number can be the final answer: it doesn't seem reasonable to assume that all skills (e.g., programming, chess playing, checkers playing, and music playing) could all require exactly the same amount of time to master, nor that all people will take exactly the same amount of time. As Prof. K. Anders Ericsson puts it, "In most domains it's remarkable how much time even the most talented individuals need in order to reach the highest levels of performance. The 10,000 hour number just gives you a sense that we're talking years of 10 to 20 hours a week which those who some people would argue are the most innately talented individuals still need to get to the highest level."

So You Want to be a Programmer

Here's my recipe for programming success:

- Get **interested** in programming, and do some because it is fun. Make sure that it keeps being enough fun so that you will be willing to put in your ten years/10,000 hours.

- **Program**. The best kind of learning is learning by doing. To put it more technically, "the maximal level of performance for individuals in a given domain is not attained automatically as a function of extended experience, but the level of performance can be increased even by highly experienced individuals as a result of deliberate

Hebrew
(Eric McCain)

עברית

Hindi
(Vikash Tiwari)

Hungarian
(Marton Mestyan)

Indonesian
(Tridjito Santoso)

Italian
(Fabio Z. Tessitore)

Japanese
(yomoyomo)

Korean (John Hwang)

Persian
(Mehdi Asgari)

Polish
(Kuba Nowak)

efforts to improve." (p. 366) and "the most effective learning requires a well-defined task with an appropriate difficulty level for the particular individual, informative feedback, and opportunities for repetition and corrections of errors." (p. 20-21) The book *Cognition in Practice: Mind, Mathematics, and Culture in Everyday Life* is an interesting reference for this viewpoint.

- **Talk with** other programmers; read other programs. This is more important than any book or training course.

- If you want, put in four years at a **college** (or more at a graduate school). This will give you access to some jobs that require credentials, and it will give you a deeper understanding of the field, but if you don't enjoy school, you can (with some dedication) get similar experience on your own or on the job. In any case, book learning alone won't be enough. "Computer science education cannot make anybody an expert programmer any more than studying brushes and pigment can make somebody an expert painter" says Eric Raymond, author of *The New Hacker's Dictionary*. One of the best programmers I ever hired had only a High School degree; he's produced a lot of great software, has his own news group, and made enough in stock options to buy his own nightclub.

- Work on **projects with** other programmers. Be the best programmer on some projects; be the worst on some others. When you're the best, you get to test your abilities to lead a project, and to inspire others with your vision. When you're the worst, you learn what the masters do, and you learn what they don't like to do (because they make you do it for them).

- Work on **projects *after*** other programmers. Understand a program written by someone else. See what it takes to understand and fix it when the original programmers are not around. Think about how to design your programs to make it easier for those who will maintain them after you.

- Learn at least a half dozen **programming languages**. Include one language that emphasizes class abstractions (like Java or C++), one that emphasizes functional abstraction (like Lisp or ML or Haskell), one that supports syntactic abstraction (like Lisp), one that supports declarative specifications (like Prolog or C++ templates), and one that emphasizes parallelism (like Clojure or Go).

- Remember that there is a "**computer**" in "computer science". Know how long it takes your computer to execute an instruction, fetch a word from memory (with and without a cache miss), read consecutive words from disk, and seek to a new location on disk. (Answers here.)

- Get involved in a language **standardization** effort. It could be the ANSI C++ committee, or it could be deciding if your local coding style will have 2 or 4 space indentation levels. Either way, you learn about what other people like in a language, how deeply they feel so, and perhaps even a little about why they feel so.

Portuguese
(Augusto Radtke)

Romanian
(Ştefan Lazăr)

Russian
(Konstantin Ptitsyn)

Serbian
(Lazar Kovacevic)

Spanish
(Carlos Rueda)

Slovak
(Jan Waclawek)

Turkish
(Çağıl Uluşahin)

Ukranian
(Oleksii Molchanovskyi)

- Have the good sense to **get off** the language standardization effort as quickly as possible.

With all that in mind, its questionable how far you can get just by book learning. Before my first child was born, I read all the *How To* books, and still felt like a clueless novice. 30 Months later, when my second child was due, did I go back to the books for a refresher? No. Instead, I relied on my personal experience, which turned out to be far more useful and reassuring to me than the thousands of pages written by experts.

Fred Brooks, in his essay *No Silver Bullet* identified a three-part plan for finding great software designers:

1. Systematically identify top designers as early as possible.

2. Assign a career mentor to be responsible for the development of the prospect and carefully keep a career file.

3. Provide opportunities for growing designers to interact and stimulate each other.

This assumes that some people already have the qualities necessary for being a great designer; the job is to properly coax them along. Alan Perlis put it more succinctly: "Everyone can be taught to sculpt: Michelangelo would have had to be taught how not to. So it is with the great programmers". Perlis is saying that the greats have some internal quality that transcends their training. But where does the quality come from? Is it innate? Or do they develop it through diligence? As Auguste Gusteau (the fictional chef in *Ratatouille*) puts it, "anyone can cook, but only the fearless can be great." I think of it more as willingness to devote a large portion of one's life to deliberative practice. But maybe *fearless* is a way to summarize that. Or, as Gusteau's critic, Anton Ego, says: "Not everyone can become a great artist, but a great artist can come from anywhere."

So go ahead and buy that Java/Ruby/Javascript/PHP book; you'll probably get some use out of it. But you won't change your life, or your real overall expertise as a programmer in 24 hours or 21 days. How about working hard to continually improve over 24 months? Well, now you're starting to get somewhere...

References

Bloom, Benjamin (ed.) *Developing Talent in Young People*, Ballantine, 1985.

Brooks, Fred, *No Silver Bullets*, IEEE Computer, vol. 20, no. 4, 1987, p. 10-19.

Bryan, W.L. & Harter, N. "Studies on the telegraphic language: The acquisition of a hierarchy of habits. *Psychology Review*, 1899, 8, 345-375

Hayes, John R., *Complete Problem Solver* Lawrence Erlbaum, 1989.

Chase, William G. & Simon, Herbert A. "Perception in Chess" *Cognitive Psychology*, 1973, 4, 55-81.

Lave, Jean, *Cognition in Practice: Mind, Mathematics, and Culture in Everyday Life*, Cambridge University Press, 1988.

Answers

Approximate timing for various operations on a typical PC:

execute typical instruction	1/1,000,000,000 sec = 1 nanosec
fetch from L1 cache memory	0.5 nanosec
branch misprediction	5 nanosec
fetch from L2 cache memory	7 nanosec
Mutex lock/unlock	25 nanosec
fetch from main memory	100 nanosec
send 2K bytes over 1Gbps network	20,000 nanosec
read 1MB sequentially from memory	250,000 nanosec
fetch from new disk location (seek)	8,000,000 nanosec
read 1MB sequentially from disk	20,000,000 nanosec
send packet US to Europe and back	150 milliseconds = 150,000,000 nanosec

---

Appendix: Language Choice

Several people have asked what programming language they should learn first. There is no one answer, but consider these points:

- *Use your friends*. When asked "what operating system should I use, Windows, Unix, or Mac?", my answer is usually: "use whatever your friends use." The advantage you get from learning from your friends will offset any intrinsic difference between OS, or between programming languages. Also consider your future friends: the community of programmers that you will be a part of if you continue. Does your chosen language have a large growing community or a small dying one? Are there books, web sites, and online forums to get answers from? Do you like the people in those forums?
- *Keep it simple*. Programming languages such as C++ and Java are designed for professional development by large teams of experienced programmers who are concerned about the run-time efficiency of their code. As a result, these languages have complicated parts designed for these circumstances. You're

concerned with learning to program. You don't need that complication. You want a language that was designed to be easy to learn and remember by a single new programmer.
- *Play.* Which way would you rather learn to play the piano: the normal, interactive way, in which you hear each note as soon as you hit a key, or "batch" mode, in which you only hear the notes after you finish a whole song? Clearly, interactive mode makes learning easier for the piano, and also for programming. Insist on a language with an interactive mode and use it.

Given these criteria, my recommendations for a first programming language would be **Python** or **Scheme**. Another choice is Javascript, not because it is perfectly well-designed for beginners, but because there are so many online tutorials for it, such as Khan Academy's tutorial. But your circumstances may vary, and there are other good choices. If your age is a single-digit, you might prefer Alice or Squeak or Blockly (older learners might also enjoy these). The important thing is that you choose and get started.

Appendix: Books and Other Resources

Several people have asked what books and web pages they should learn from. I repeat that "book learning alone won't be enough" but I can recommend the following:

- **Scheme:** Structure and Interpretation of Computer Programs (Abelson & Sussman) is probably the best introduction to computer science, and it does teach programming as a way of understanding the computer science. You can see online videos of lectures on this book, as well as the complete text online. The book is challenging and will weed out some people who perhaps could be successful with another approach.
- **Scheme:** How to Design Programs (Felleisen *et al.*) is one of the best books on how to actually design programs in an elegant and functional way.
- **Python:** Python Programming: An Intro to CS (Zelle) is a good introduction using Python.
- **Python:** Several online tutorials are available at Python.org.
- **Oz:** Concepts, Techniques, and Models of Computer Programming (Van Roy & Haridi) is seen by some as the modern-day successor to Abelson & Sussman. It is a tour through the big ideas of programming, covering a wider range than Abelson & Sussman while being perhaps easier to read and follow. It uses a language, Oz, that is not widely known but serves as a basis for learning other languages. <

Notes

T. Capey points out that the Complete Problem Solver page on Amazon now has the "Teach Yourself Bengali in 21 days" and "Teach Yourself Grammar and Style" books under the "Customers who shopped for this item also shopped for these items" section. I guess that a large portion of the people who look at that book are coming from this page. Thanks to Ross Cohen for help with Hippocrates.

---

163 Comments norvig.com Jonathan Lam

Recommend 108 Tweet Share Sort by Best

Join the discussion…

Eric-Wubbo Lameijer • 7 years ago

Thank you for this post! I was trying to help someone learning to program (though she does not plan to take only 24 hours for doing so...)
Two remarks content-wise though:

1) learning by doing is not necessarily the best or even the fastest way to learn, even though it definitely has its place. If I would teach anyone programming, I'd strongly recommend to study and try to understand good code; this would be the technique of 'worked examples'. So far, lots of 'worked examples' and a bit of problem solving seems to be better than one example followed by lots of problem-solving. (see for example http://www.jstor.org/discov...

2) The 10-year rule (or 10.000-hour rule) very much depends on the field; it used to be about 6 years for painters, but can be over 25 years for musicians. (Sternberg, Handbook of Creativity, around page 230) It tends to depend mostly on the competition in the field - the author of 'Moonwalking with Einstein' became an US champion after about 1 year of training!. Assuming that a majority of programmers work hard at deliberate practice (which I doubt), it may indeed take 10 or 15 years to become a top programmer. But to be functionally literate and 'worth hiring' may not require that many years. Besides, one could definitely ask who is a 'top programmer': someone who can write code that is superbly simple and clear? Someone who can solve a complicated problem with a snazzy new polynomial algorithm? Someone who can keep track of a project involving millions of lines of code? Someone who knows hundreds of language functions and libraries? Someone who actually understands what a non-computer-scientist customer is talking about and can transform it into a solution? Not all programmers will be equally good in all of those things, being the