# Learning to Hack:

Exploratory computer science through Project Euler

**Philosophical preamble**

A secret shame of many of my friends, and a running joke among us, is the number of computer programming books we used to buy as kids. We were all in awe of computers, of course, enamored by Steve Wozniak building the Apple II with his bare hands, or by the realistic physics engine baked into Quake, or by all these clever websites earning millions for their authors. We wanted in, and we thought we knew how to get there: convince your mom to drive you to Borders to buy Ivor Horton's 1181 page, 4.6 pound *Beginning Visual C++ 6*.

I remember blowing my caddy money on that book and slogging—slowly, painfully—through four chapters. By the fifth I just wasn't getting it: I was re-reading sentences three or four times before I finally quit. And I was no exception: this is what all my friends were doing, though at the time we hadn't the nerve to talk about it.

We were caught up in the self-help cycle for precocious pre-teens. We bought lame books full of promise and didn't do the exercises. We went to the MIT Open Courseware site, read up on Algorithms and Data Structures, but skipped lectures that went over our head. We read tutorials and perused forums, copied code and plugged it in, but didn't create.

We went through years of this stuff and had nothing to show for it. By the time I was a Junior in high school the best I could do was a program that added fractions and printed out square roots; I had been through three languages (C++, Java, Visual Basic) and seven books.

These are the kind of kids who arrive so eagerly at a high school computer science course. They're excited for exercises at their own level, collaboration with their peers, grades and progress reports and extra help—all the things that will finally, once and for all, ensure that they learn how to program.

What a letdown it is! Most would be well-served *not* to take the course, since it so often turns them off the subject. For everyone else it's a painful interlude before something better.

Why?

The AP curriculum, in a couple of important ways, is designed like one of these horrible books we used to clamor for:

1. It's top-down, which means you get a chapter of "concepts" and then a handful of exercises to show you've learnt them. That's a bad approach in any subject, but with computers it's fatal.

2. It uses the wrong language. As a teaching tool, Java is an *insult*. It's ugly, scary, and difficult to use. It gets you about as far away from "thinking naturally" as any modern equivalent.

It turns out that these two problems are related.

High-level programming languages were designed so that the programmer could stop worrying about the computer—when you're trying to write a Sudoku solver the last thing on your mind is garbage collection, or whether all your {'s and )'s are in the right place. But when you're working in Java it feels like you're working *on* Java.

In the same way, when an exercise follows a chapter on Classes then it becomes *about* Classes. When you're being quizzed on "design methodologies" and "best practices of object inheritance" that's what you focus on. You forget that these things do not exist in themselves, that they were invented by a real person who had a real problem.

There's this idea in education that one can "cover" more in a class if one simply distills the important points, packages them in outline form, and throws in some exercises. Pingry is a good school in large part because it doesn't ascribe to this myth. Example: what I remember most from my AP Physics class with Mr. Coe was a week-long project. We were given a simple question—if you push something, when will it tip over?—and let loose, with ample help when we needed it. Pound for pound it was the most productive portion of the whole semester.

The fact is, **ideas stick when you learn them in the service of some problem**. Just as we picked up kinematics and rotational inertia by trying to answer the slip-tip question, one should learn about Classes and Inheritance and Regular Expressions *along the way*, from the bottom up.

$$\sum_{n=0}^{200} \mathbf{PE}_n = \textbf{a fun, fast approach to computer science}$$

I credit most of my programming ability to `projecteuler.net` (PE), a site with about two hundred mathematics-related computer problems. It was PE that finally got me snowballing toward being a reasonably capable programmer.

(If you haven't already, it might be fun to take a brief tour around the site. They have a nice FAQ on the landing page, and you can wander freely through the problems. Beware, though: you may have the urge to solve one of them...)

As you probably know well, you learn not just by doing, but by knowing *how* you're doing. PE's evaluation is simple: you either solved the problem or you didn't (there is, in addition, the "one-minute rule"—you must obtain a solution within one minute of computer time). Once successful, you earn access to a forum where other, more experienced programmers share their approaches. It's the ideal time to pick up new ideas—after you've wrapped your head around the problem well enough to solve it.

With not much more than a pencil and paper, a Python interpreter, and access to Google, one can go from not knowing how to declare variables to understanding classic sorting algorithms, data structures, and program design (i.e., the stuff of the AP curriculum), proceeding one fun puzzle at a time.

Problem 1, for instance, reads:

> If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The

sum of these multiples is 23. Find the sum of all the multiples of 3 or 5 below 1000.

I have watched a complete beginner (no programming experience) work up a solution to this problem in just over an hour. And that includes time spent downloading and installing Python, reading a brief tutorial, and figuring out how to use the modulus operator (which checks for divisibility). He got hooked enough to spend most of his free time that week working on the next ten or so problems.

It's not long before one is solving problems like 96:

Devise an algorithm for solving Su Doku puzzles.

By problem 2 a student will have already had a thorough discussion of recursion vs. iteration, and an idea of how to implement both. By problem 4 they will have seen the Python "String Methods" page, and by problem 11 they'll be reading and writing files and thinking about how to structure data. Along the way they'll learn about regular expressions (perhaps with a diversion into finite automata, one of those subjects that's not as complicated as it sounds), memoization, and exception handling.

That's what I mean by bottom-up learning—adding to your toolbox as you go; see the **Appendix** for a quick example.

The pressure to write cleaner code, and to think more carefully about program flow, will come from the forum. As will detailed explanations of the math behind the best algorithm and exposure to other programming languages.

All the while one is solving difficult problems, debugging, and discovering the many breakpoints (syntax errors, sloppy indexing, choosing the wrong data structure) of even the smallest programs.

This is not to mention all of the great mathematics to which one will be exposed. How many high-schoolers know about Ramanujan's generating function for partitions? Or what the Collatz conjecture is about?

Along with prepared materials[1] about various relevant concepts (say, the idea of black-box abstraction, Big O notation, and illustrations of tree recursion; also, a guide like *Dive Into Python* for quickly getting a handle on all of language's powerful tools) and bull sessions among students about their approaches, I think that a course whose homework is solving Project Euler problems would get you pretty far in about the fastest way possible. `euler` himself, the site's creator-curator, chose his problems carefully.

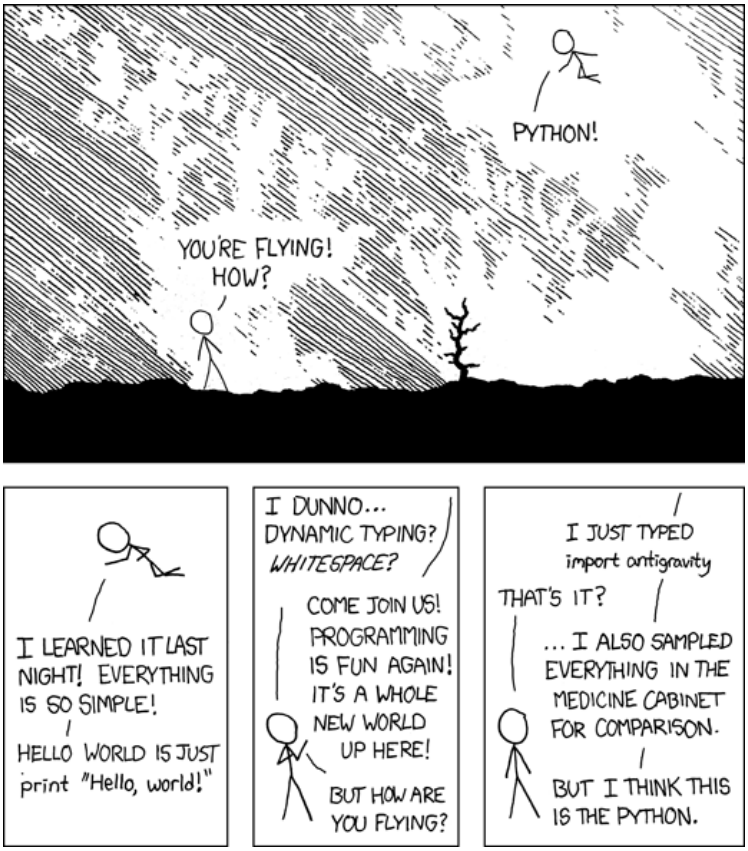
[1] I have created a wiki with ideas, approaches, resources, etc., for most of the problems I intend to use (obviously all 200 would be insane). This is the "lesson plan," so to speak, although it's a wiki for a reason: I don't expect anything to go according to plan!

**Python: the software equivalent of a big, friendly dog**

Implicit in this discussion has been the choice of Python as a programming language. Here's the pitch from their website:

> Python is a dynamic object-oriented programming language that can be used for many kinds of software development. It offers strong support for integration with other languages and tools, comes with extensive standard libraries, and can be learned in a few days. Many Python programmers report substantial productivity gains and feel the language encourages the development of higher quality, more maintainable code.

That's all true, although this strip from the popular webcomic XKCD might do a better sales job:




Where I'm working this summer (the complex systems group at the University of Michigan, as part of the Google "summer of code") we are actually in the process of porting everything we can out of Java into Python. Recently, MIT switched its legendary first-year course (The Structure and Interpretation of Computer Programs) from LISP to Python.

Python is a real achievement. It's the kind of thing that makes one wonder why high schools continue to use Java; by the time the students graduate from university all the interesting job ads will have "Python or Ruby" in place of "Java or C++."

**Can James teach?**

I am not naïve enough to think that this course will somehow teach itself, and I recognize that the dubious factor here—can James teach?—is also the most important. I suppose for now you'll simply have to take that on faith, or at best, on the word of some Pingry teachers who know me.

**Appendix: an example of "bottom-up learning" using Project Euler**

There are two approaches to teaching someone about hash tables. You could introduce the idea in a lecture, stress that they have $O(1)$ lookup time, show how they're constructed, etc., and then give a few obvious "implementations" as exercises.

Or you might be working on PE problem 14, where you're told to apply the following algorithm to every $n < 10^6$: (1) if $n$ is even, go to $\frac{n}{2}$; (2) if it's odd, jump up to $3n + 1$; (3) stop when you hit 1. So for $n = 12$ you generate the chain $12 \to 6 \to 3 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$. The question asks you to find the starting number less than a million with the longest such chain.[2]

The trouble is, most of these things end up being about 100 steps long, which means you would need about $10^8$ computations to go through every single one—no way your home computer will finish *that* in a minute!

What you'll notice, if you work through a few examples, is that you tend to do a lot of redundant work. The chain for 13, for example, gets to 10 in four steps: $13 \to 40 \to 20 \to 10 \cdots$ But you already know from working with 12 that once you get to 10, you're only six steps away from the finish ($10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$). There's no reason you should have to do all that work again.

It would be really useful if you could build some kind of memory as you go along, a way to associate `length_of_chain(10)` with 6, for example, or more generally, a mapping between numbers and the corresponding "steps till 1" value. That way you could just look up values rather than doing all this chain-computing, and probably save some time.[3]

You're in luck: just as a dictionary maps words to definitions, a Python `dict` can, with one line of code, map your 10s to your 6s. That's what it's there for.

This, in my experience, is how you learn things for good: you bang your head against a problem ("This thing is going to take forever..."), think of a clever idea by working through a few examples ("Why should I compute `length_of_chain(10)` one million times?"), and jump to the natural conclusion: "There must be some way to keep track of this stuff..."

You're 90% of the way to "hash tables," and when you finally find them you're going to treat them with respect, revere them, love them—who knows, maybe even *study* them—because they helped you knock another problem off the list.

[2]Incidentally, figuring out whether *every* number has a chain that ends in the $1 \to 4 \to 2 \to 1$ cycle is one of the great unsolved mathematics problems, known as the "Collatz conjecture." It has been confirmed, using programs not much more complicated than the ones we'd write, for $n < 10^{18}$.

[3]This, incidentally, is called "memoization" or "caching," but who cares about the *names* of things?