Adam13531's advice for how to learn programming

Monday, April 4, 2016 1:46 PM

Setting expectations

- At first, you're going to learn about concepts like variables, 'if' statements, and outputting text, and they may seem uninteresting. I think that most people who try to learn coding expect to be changing high-level attributes quickly, e.g. "let's add a fiery sword into the game!" When you end up learning about variables instead, it may not be obvious how this eventually gets you to the point where you can add features to your game/program.
- You probably won't be making very interesting programs for a little while (at least a month or two); they'll more likely be small, console-based programs around math functions or simple guessing games.
- Programming has a lot to do with being self-sufficient
    - Googling error messages
    - Gaining general knowledge
        - What are syntax errors?
        - What's an interpreter?
        - Etc.
- Programming isn't necessarily for everyone, it can be too frustrating, boring, etc. However, I don't think intelligence/skill/talent plays too much into this not-for-everyone mentality; if you keep at it, you will eventually learn it!
- It doesn't really matter which programming language you learn first.
    - You will likely learn many languages if you're going to keep programming
    - Technology changes rapidly enough where the popularity of languages changes over time (e.g. you may be forced to stop using your favorite language (*cough* FLASH))
    - If you want someone to pick a language for you, just go with Python to start with
- Finishing hobby projects is rare. A lot of people wonder what the best technology or language is, and I think the answer is almost always "whatever will keep you motivated".
- Only focus on two things when you start out:
    - Making basic progress ("am I learning?")
    - Having fun

How to learn programming

- In general, it's no different from learning anything else:
    - Studying
    - Practice
    - Talking to other people (teachers, coders, etc.)
- More specifically, there are lots of resources out there (books, sites, tutorials). I've intentionally removed specific resources from this document for two reasons:
    - I haven't consumed many of them myself, so I can't vouch for them.
    - As mentioned earlier, learning self-sufficiency is important! Do a quick search for "awesome Python/JavaScript/Haskell resources".
- When it comes to practicing, try to come up with a project that is slightly beyond your current skill level.
    - If the scope of the project is too large, then it can quickly lead to being demotivated, so pick something that you think is feasible.
    - Reinventing the wheel is *amazing* for learning because it lets you focus on a smaller scope. For example, if you wanted to learn game programming by making your own game, you now have two tasks: designing a game *and* learning how to develop that game. If you instead recreate an existing game like Tetris or Pong, then you can focus solely on the code. The

only caveat here is relatively obvious: don't claim other people's work as your own.
    - At first, coming up with project ideas is just a matter of combining the last two bullet points: pick a project you like, then tackle a small portion of it. E.g. rather than trying to make all of Pokémon, maybe you can just make NPCs walk around.

Things to learn ASAP

- How to search effectively for the solution to your problems
    - StackOverflow is a good place to start for almost ANY basic programming question (e.g. "how do I add to an array in JavaScript?" or "what does 'syntax error' mean?")
- Learning more interactively
    - Take notes
    - Test out programs for yourself
- Debugging - the basics of this are placing breakpoints and inspecting variables. Most modern languages have an easy way to debug them (e.g. Visual Studio for C++, PyCharm for Python, browsers for JavaScript).
- How to break down your problem into the smallest possible repro (http://sscce.org/)
    - Describe the problem
    - Describe the expected outcome
    - Provide only as much code as is needed to reproduce the problem

Bugs

- You haven't fixed a bug until you've fully understood it. The reason why this is so important is because when you come across a bug, you immediately make a bunch of assumptions. Maybe the bug is that an image isn't showing on-screen, and your initial hunch is "the image file doesn't exist". You look in the folder that should contain the image, and you see it there. Then you think, "maybe one of my libraries got updated and that's having trouble rendering the image". You investigate that and make some changes, and in the process, the bug is solved, but you don't know why. If you stop there, it's possible that the bug still exists but that you don't realize it. In this particular example, let's say that the bug was caused by trying to access the file on a Tuesday for some reason. You're clearly going to encounter this bug again, but you'll assume you dealt with it. That's why it's so important to understand the root cause of a bug.
- Whenever you find [and eventually fix] a bug, you should ask yourself these questions:
    - Where else might this bug exist in my code?
    - How will I prevent myself from hitting this bug in the future?
    - What techniques/knowledge did I learn that could help me find any bug faster in the future? For example, maybe you learned that logging is helpful or that you tend to make typos in filenames.