What to learn

It's common to see people advocate for learning skills that they have or using processes that they use. For example, Steve Yegge has a set of blog posts where he recommends reading compiler books and learning about compilers. His reasoning is basically that, if you understand compilers, you'll see compiler problems everywhere and will recognize all of the cases where people are solving a compiler problem without using compiler knowledge. Instead of hacking together some half-baked solution that will never work, you can apply a bit of computer science knowledge to solve the problem in a better way with less effort. That's not untrue, but it's also not a reason to study compilers in particular because you can say that about many different areas of computer science and math. Queuing theory, computer architecture, mathematical optimization, operations research, etc.

One response to that kind of objection is to say that one should study everything. While being an extremely broad generalist can work, it's gotten much harder to "know a bit of everything" and be effective because there's more of everything over time (in terms of both breadth and depth). And even if that weren't the case, I think saying “should” is too strong; whether or not someone enjoys having that kind of breadth is a matter of taste. Another approach that can also work, one that's more to my taste, is to, as Gian Carlo Rota put it, learn a few tricks:

> A long time ago an older and well known number theorist made some disparaging remarks about Paul Erdos' work. You admire contributions to mathematics as much as I do, and I felt annoyed when the older mathematician flatly and definitively stated that all of Erdos' work could be reduced to a few tricks which Erdos repeatedly relied on in his proofs. What the number theorist did not realize is that other mathematicians, even the very best, also rely on a few tricks which they use over and over. Take Hilbert. The second volume of Hilbert's collected papers contains Hilbert's papers in invariant theory. I have made a point of reading some of these papers with care. It is sad to note that some of Hilbert's beautiful results have been completely forgotten. But on reading the proofs of Hilbert's striking and deep theorems in invariant theory, it was surprising to verify that Hilbert's proofs relied on the same few tricks. Even Hilbert had only a few tricks!

If you look at how people succeed in various fields, you'll see that this is a common approach. For example, this analysis of world-class judo players found that most rely on a small handful of throws, concluding[1]

> Judo is a game of specialization. You have to use the skills that work best for you. You have to stick to what works and practice your skills until they become automatic responses.

If you watch an anime or a TV series "about" fighting, people often improve by increasing the number of techniques they know because that's an easy thing to depict but, in real life, getting better at techniques you already know is often more effective than having a portfolio of hundreds of "moves".

Relatedly, Joy Ebertz says:

> One piece of advice I got at some point was to amplify my strengths. All of us have strengths and weaknesses and we spend a lot of time talking about ‘areas of improvement.’ It can be easy to feel like the best way to advance is to eliminate all of those. However, it can require a lot of work and energy to barely move the needle if it’s truly an area we’re weak in. Obviously, you still want to make sure you don’t have any truly bad areas, but assuming you’ve gotten that, instead focus on amplifying your strengths. How can you turn something you’re good at into your superpower?

I've personally found this to be true in a variety of disciplines. While it's really difficult to measure programmer effectiveness in anything resembling an objective manner, this isn't true of some things I've done, like competitive video games (a very long time ago at this point, back before there was "real" money in competitive gaming), the thing that took me from being a pretty decent player to a very good player was abandoning practicing things I wasn't particularly good at and focusing on increasing the edge I had over everybody else at the few things I was unusually good at.

This can work for games and sports because you can get better maneuvering yourself into positions that take advantage of your strengths as well as avoiding situations that expose your weaknesses. I think this is actually more effective at work than it is in sports or gaming since, unlike in competitive endeavors, you don't have an opponent who will try to expose your weaknesses and force you into positions where your strengths are irrelevant. If I study queuing theory instead of compilers, a rival co-worker isn't going to stop me from working on projects where queuing theory knowledge is helpful and leave me facing a field full of projects that require compiler knowledge.

One thing that's worth noting is that skills don't have to be things people would consider fields of study or discrete techniques. For the past three years, the main skill I've been applying and improving is something you might call "looking at data"; the term is in quotes because I don't know of a good term for it. I don't think it's what most people would think of as "statistics", in that I don't often need to do anything as sophisticated as logistic regression, let alone actually sophisticated. Perhaps one could argue that this is something data scientists do, but if I look at what I do vs. what data scientists we hire do as well as what we screen for in data scientist interviews, we don't appear to want to hire data scientists with the skill I've been working on nor do they do what I'm doing (this is a long enough topic that I might turn it into its own post at some point).

Unlike Matt Might or Steve Yegge, I'm not going to say that you should take a particular approach, but I'll say that working on a few things and not being particularly well rounded has worked for me in multiple disparate fields and it appears to work for a lot of other folks as well.

If you want to take this approach, this still leaves the question of what skills to learn. This is one of the most common questions I get asked and I think my answer is probably not really what people are looking for and not very satisfying since it's both obvious and difficult to put into practice.

For me, two ingredients for figuring out what to spend time learning are having a relative aptitude for something (relative to other things I might do, not relative to other people) and also having a good environment in which to learn. To say that someone should look for those things is so vague that's it's nearly useless, but it's still better than the usual advice, which boils down to "learn what I learned", which results in advice like "Career pro tip: if you want to get good, REALLY good, at designing complex and stateful distributed systems at scale in real-world environments, learn functional programming. It is an almost perfectly identical skillset." or the even more extreme claims from some language communities, like Chuck Moore's claim that Forth is at least 100x as productive as boring languages.

I took generic internet advice early in my career, including language advice (this was when much of this kind of advice was relatively young and it was not yet possible to easily observe that, despite many people taking advice like this, people who took this kind of advice were not particularly effective and people who are particularly effective were not likely to have taken this kind of advice). I learned Haskell, Lisp, Forth, etc. At one point in my career, I was on a two person team that implemented what might still be, a decade later, the highest performance Forth processor in existence (it was a 2GHz IPC-oriented processor) and I programmed it as well (there were good reasons for this to be a stack processor, so Forth seemed like as good a choice as any). Like Yossi Kreinin, I think I can say that I spent more effort than most people have becoming proficient in Forth, and like him, not only did I not find it find it to be a 100x productivity tool, it wasn't clear that it would, in general, even be 1x on productivity. To be fair, a number of other tools did better than 1x on productivity but, overall, I think following internet advice was very low ROI and the things that I learned that were high ROI weren't things people were recommending.

In retrospect, when people said things like "Forth is very productive", what I suspect they really meant was "Forth makes me very productive and I have not considered how well this generalizes to people with different aptitudes or who are operating in different contexts". I find it totally plausible that Forth (or Lisp or Haskell or any other tool or technique) does work very well for some particular people, but I think that people tend to overestimate how much something working for them means that it works for other people, making advice generally useless because it doesn't distinguish between advice that's aptitude or circumstance specific and generalizable advice, which is in stark contrast to fields where people actually discuss the pros and cons of particular techniques[2].

While a coach can give you advice that's tailored to you 1 on 1 or in small groups, that's difficult to do on the internet, which is why the best I can do here is the uselessly vague "pick up skills that are suitable for you". Just for example, two skills that clicked for me are "having an adversarial mindset" and "looking at data". A perhaps less useless piece of advice is that, if you're having a hard time identifying what those might be, you can ask people who know you very well, e.g., my manager and Ben Kuhn independently named coming up with solutions that span many levels of abstraction as a skill of mine that I frequently apply (and I didn't realize I was doing that until they pointed it out).

Another way to find these is to look for things you can't help but do that most other people don't seem to do, which is true for me of both "looking at data" and "having an adversarial mindset". Just for example, on having an adversarial mindset, when a company I was working for was beta testing a new custom bug tracker, I filed some of the first bugs on it and put unusual things into the fields to see if it would break. Some people really didn't understand why anyone would do such a thing and were baffled, disgusted, or horrified, but a few people (including the authors, who I knew wouldn't mind), really got it and were happy to see the system pushed past its limits. Poking at the limits of a system to see where it falls apart doesn't feel like work to me; it's something that I'd have to stop myself from doing if I wanted to not do it, which made spending a decade getting better at testing and verification techniques felt like something hard not to do and not work. Looking deeply into data is one I've spent more than a decade on at this point and it's another one that, to me, emotionally feels almost wrong to not improve at.

That these things are suited to me is basically due to my personality, and not something inherent about human beings. Other people are going to have different things that really feel easy/right for them, which is great, since if everyone was into looking at data and no one was into building things, that would be very problematic (although, IMO, looking at data is, on average, underrated).

The other major ingredient in what I've tried to learn is finding environments that are conducive to learning things that line up with my skills that make sense for me. Although suggesting that other people do the same sounds like advice that's so obvious that it's useless, based on how I've seen people select what team and company to work on, I think that almost nobody does this and, as a result, discussing this may not be completely useless.

An example of not doing this which typifies what I usually see is a case I just happened to find out about because I chatted with a manager about why their team had lost their new full-time intern conversion employee. I asked them about it since it was unusual for that manager to lose anyone since they're very good at retaining people and have low turnover on their teams. It turned out that their intern had wanted to work on infra, but had joined this manager's product team

because they didn't know that they could ask to be on a team that matched their preferences. After the manager found out, the manager wanted the intern to be happy and facilitated a transfer to an infra team. In this case, this was a double whammy since the new hire doubly didn't consider working in an environment conducive for learning the skills they wanted. They made no attempt to work in the area they were interested in and then they joined a company that has a dysfunctional infra org that generally has poor design and operational practices, making the company a relatively difficult place to learn about infra on top of not even trying to land on an infra team. While that's an unusually bad example, in the median case that I've seen, people don't make decisions that result in particularly good outcomes with respect to learning even though good opportunities to learn are one of the top things people say that they want.

For example, Steve Yegge has noted:

> The most frequently-asked question from college candidates is: "what kind of training and/or mentoring do you offer?" ... One UW interviewee just told me about Ford Motor Company's mentoring program, which Ford had apparently used as part of the sales pitch they do for interviewees. [I've elided the details, as they weren't really relevant. -stevey 3/1/2006] The student had absorbed it all in amazing detail. That doesn't really surprise me, because it's one of the things candidates care about most.

For myself, I was lucky that my first job, Centaur, was a great place to develop having an adversarial mindset with respect to testing and verification. When I compare what the verification team there accomplished, it's comparable to peer projects at other companies that employed much larger teams to do very similar things with similar or worse effectiveness, implying that the team was highly productive, which made that a really good place to learn.

Moreover, I don't think I could've learned as quickly on my own or by trying to follow advice from books or the internet. I think that people who are really good at something have too many bits of information in their head about how to do it for that information to really be compressible into a book, let alone a blog post. In sports, good coaches are able to convey that kind of information over time, but I don't know of anything similar for programming, so I think the best thing available for learning rate is to find an environment that's full of experts[3].

For "looking at data", while I got a lot better at it from working on that skill in environments where people weren't really taking data seriously, the rate of improvement during the past few years, where I'm in an environment where I can toss ideas back and forth with people who are very good at understanding the limitations of what data can tell you as well as good at informing data analysis with deep domain knowledge, has been much higher. I'd say that I improved more at this in each individual year at my current job than I did in the decade prior to my current job.

One thing to perhaps note is that the environment, how you spend your day-to-day, is inherently local. My current employer is probably the least data driven of the three large tech companies I've worked for, but my vicinity is a great place to get better at looking at data because I spend a relatively large fraction of my time working with people who are great with data, like Rebecca Isaacs, and a relatively small fraction of the time working with people who don't take data seriously.

This post has discussed some strategies with an eye towards why they can be valuable, but I have to admit that my motivation for learning from experts wasn't to create value. It's more that I find learning to be fun and there are some areas where I'm motivated enough to apply the skills regardless of the environment, and learning from experts is such a great opportunity to have fun that it's hard to resist. Doing this for a couple of decades has turned out to be useful, but that's not something I knew would happen for quite a while (and I had no idea that this would effectively transfer to a new industry until I changed from hardware to software).

A lot of career advice I see is oriented towards career or success or growth. That kind of advice often tells people to have a long-term goal or strategy in mind. It will often have some argument that's along the lines of "a random walk will only move you sqrt(n) in some direction whereas a directed walk will move you n in some direction". I don't think that's wrong, but I think that, for many people, that advice implicitly underestimates the difficulty of finding an area that's suited to you[4], which I've basically done by trial and error.

Appendix: parts of the problem this post doesn't discuss in detail

One major topic not discussed is how to balance what "level" of skill to work on, which could be something high level, like "looking at data", to something lower level, like "Bayesian multilevel models", to something even lower level, like "typing speed". That's a large enough topic that it deserves its own post that I'd expect to be longer than this one but, for now, here's a comment from Gary Bernhardt about something related that I believe also applies to this topic.

Another major topic that's not discussed here is picking skills that are relatively likely to be applicable. It's a little too naive to just say that someone should think about learning skills they have an aptitude for without thinking about applicability.

But while it's pretty easy to pick out skills where it's very difficult to either have an impact on the world or make a decent amount of money or achieve whatever goal you might want to achieve, like "basketball" or "boxing", it's harder to pick between plausible skills, like computer architecture vs. PL.

But I think semi-reasonable sounding skills are likely enough to be high return if they're a good fit for someone that trial and error among semi-reasonable sounding skills is fine, although it probably helps to be able to try things out quickly

Appendix: related posts

- Ben Kuhn on, in some sense, what it's like to really learn something
- Holden Karnofsky on having an aptitude-first approach to careers instead of a career-path-first approach, which is sort of analogous to thinking about cross cutting skills like "looking at data" or "having an adversarial mindset" and not just thinking about skills like "compilers" or "queuing theory"
- Peter Drucker on how to understand one's strengths and weaknesses and do work that compatible with ones own inclinations
- Alexy Guzey on the effectiveness of advice
- Edward Kmett with another perspective on how to think about learning
- Patrick Collison on how to maximize useful learning and find what you'll enjoy

Thanks to Ben Kuhn, Alexey Guzey, Marek Majkowski, Nick Bergson-Shilcock, @bekindtopeople2, Aaron Levin, Milosz Danczak, Anja Boskovic, John Doty, Justin Blank, Mark Hansen, "wl", and Jamie Brandon for comments/corrections/discussion.

---

1. This is an old analysis. If you were to do one today, you'd see a different mix of throws, but it's still the case that you see specialists having a lot of success, e.g., Riner with osoto gari [return]
2. To be fair to blanket, context free, advice, to learn a particular topic, functional programming really clicked for me and I could imagine that, if that style of thinking wasn't already natural for me (as a result of coming from a hardware background), the advice that one should learn functional programming because it will change how you think about problems might've been useful for me, but on the other hand, that means that the advice could've just as easily been to learn hardware engineering. [return]

3. I don't have a large enough sample nor have I polled enough people to have high confidence that this works as a general algorithm but, for finding groups of world-class experts, what's worked for me is finding excellent managers. The two teams I worked on with the highest density of world-class experts have been teams under really great management. I have a higher bar for excellent management than most people and, from having talked to many people about this, almost no one I've talked to has worked for or even knows a manager as good as one I would consider to be excellent (and, general, both the person I'm talking to agrees with me on this, indicating that it's not the case that they have a manager who's excellent in dimensions I don't care about and vice versa); from discussions about this, I would guess that a manager I think of as excellent is at least 99.9%-ile. How to find such a manager is a long discussion that I might turn into another post.

   Anyway, despite having a pretty small sample on this, I think the mechanism for this is plausible, in that the excellent managers I know have very high retention as well as a huge queue of people who want to work for them, making it relatively easy for them to hire and retain people with world-class expertise since the rest of the landscape is so bleak.

   A more typical strategy, one that I don't think generally works and also didn't work great for me when I tried it is to work on the most interesting sounding and/or hardest problems around. While I did work with some really great people while trying to work on interesting / hard problems, including one of the best engineers I've ever worked with, I don't think that worked nearly as well as looking for good management w.r.t. working with people I really want to learn from. I believe the general problem with this algorithm is the same problem with going to work in video games because video games are cool and/or interesting. The fact that so many people want to work on exciting sounding problems leads to dysfunctional environments that can persist indefinitely.

   In one case, I was on a team that had 100% turnover in nine months and it would've been six if it hadn't taken so long for one person to find a team to transfer to. In the median case, my cohort (people who joined around when I joined, ish) had about 50% YoY turnover and I think that people had pretty good reasons for leaving. Not only is this kind of turnover a sign that the environment is often a pretty unhappy one, these kinds of environments often differentially cause people who I'd want to work with and/or learn from to leave. For example, on the team I was on where the TL didn't believe in using version control, automated testing, or pipelined designs, I worked with Ikhwan Lee, who was great. Of course, Ikhwan left pretty quickly while the TL stayed and is still there six years later.

   [return]

4. Something I've seen many times among my acquaintances is that people will pick a direction before they have any idea whether or not it's suitable for them. Often, after quite some time (more than a decade in some cases), they'll realize that they're actually deeply unhappy with the direction they've gone, sometimes because it doesn't match their temperament, and sometimes because it's something they're actually bad at. In any case, wandering around randomly and finding yourself sqrt(n) down a path you're happy with doesn't seem so bad compared to having made it n down a path you're unhappy with. [return]