Table of Contents

Introduction

This book is not about how to write a computer program. We already have lots of books about how to write computer programs.

Instead, this book is about self-education. It will cover the psychology of learning, how to learn a skill quickly, and how to attain mastery as opposed to simple working proficiency. With luck, it will one day provide a comprehensive guide for people who want to learn programming skills outside of formal education.

Who is a Self-Taught Programmer?

Who is a self-taught programmer? For now, we define it as anyone who learns to program computers outside of a formal education environment, *at any point*. So this includes beginners who go online to try their hand at their very first programming language. It also includes programmers who have reached some level of proficiency through independent study and tutoring. It includes all of the masters, because mastery of a still requires extensive additional self-education outside of formal teaching environments.

Recipe for a Useful Explanatory Book

That said, the guiding principle of this book's composition is the **Principle of Least Surprise.** This chapter from The Art of Unix Programming explains the concept beautifully, if you're interested. How do we apply the Principle of Least Surprise to writing a book?

Like this:

1. Use self-explanatory language. This allows your readers to focus on your message without having to stop and look up your words. Example: the introduction above uses the term "self-education" and not "autodidactism" on purpose.

If you must use jargon to describe something, explain the term to the reader. Examples of terms that would require explanation are "garbage-collected," "concurrency," and "Extended Backus-Naur Form."

2. Use self-explanatory chapter titles. We want readers to be able to find what they're looking for from the Table of Contents. A non-humorous, obvious chapter title is far preferable to a cute title with tech puns and inside jokes. For example, a chapter entitled

"Choosing your First Project" is a much better fit for this book than the same chapter entitled "Scoping the Epic Quest."

3. Organize your thoughts. If you are contributing a new chapter, please make sure that the thoughts are well-organized. If they flow well from one to the next, that is excellent. If they at least follow a logical progression, that is sufficient.

For an excellent cheat guide to well-organized writing, you want to look at Section III, 'Elementary Principles of Composition,' in The Elements of Style. The entire book is available for free online as a pdf.

Other guidelines for contributing to this book:

1. Keep it language-agnostic. That does not mean that you cannot express evidence-based statements on the fitness of a language *for a particular purpose.* For example, it is okay to say "the python programming language requires a minimal amount of setup to begin writing programs, so it is useful as a starter language for a first-time programmer." However, keep in mind that *every* language possesses self-taught users, and the majority of this book's advice should be useful to all of them.

2. Keep it IDE, color scheme, library, and framework-agnostic. The same rules apply as above for languages.

3. Use examples wherever possible. Examples make information easier to absorb, so include examples wherever you introduce new information. If you are explaining how to schedule study time, an example daily or weekly schedule may help get the point across.

4. Pictures and stories are great, too. The brain remembers information better when it receives that information in multiple different formats. If you learned something important about self-education from a particular personal experience, that experience might be worth sharing. If you are explaining how to diagram a computer program, a picture might help your explanation.

5. When you introduce new terminology, add it to the glossary as well as explain it in the text. The gitbook editor makes this easy: the rightmost button on the top toolbar is for adding terms to the glossary. Speaking of which...

Getting Started: The Autodidact's Notebook

I have a secret laboratory. I carry it with me everywhere.

You can tell that it has seen years of use. The spine depends on duct tape to hold the contents together. A few sticky notes grace the first page: refactoring to-do lists from years ago. The cover sports two patches of robust color. A couple of Github stickers used to be there, shielding the original cover design from fading until I peeled them off.

When I bought the notebook as a fresh Rails developer, I planned only on using it to keep track of blog post ideas. Instead, it became a critical resource as I taught myself how to develop for Rails, Spring, Android, and iOS, and Django. The notebook also helped me improve my technical writing, build my network, and master the foundations of data science and machine learning. As I have continued to work with the notebook, I have experimented with dozens of techniques, several of which have an outsize effect on my ability to build skills and apply new knowledge.

~~***We can use those techniques as a lens to discuss how we learn, and how you can use an autodidact's notebook to be an effective self-taught programmer.***~~

Benefits of a Notebook:

1. Writing in it forces you to think (thinking fast and slow)
2. Having a record helps you look back at your progress
3. Helps to codify focused practice (Grit)
4. Gives you the freedom to draw pictures and associate (Moonwalking with Einstein)

=====

get better AT getting better at (solving a second order problem)

Choose a curriculum with a deliverable, and only get one degree removed from it

example: ml curriculum to help you make x. study y library to help you use ml to make x. Don't study whatever you need to customize y library. Don't study the math of y. You need something you can track progress and win at without going down a rabbit hole, and you need to learn how to learn JUST ENOUGH for your use case. Why? Because this is how knowledge works in the real world. Rather than be an expert in one thing, you are best off having CONTEXT on a broad set of skills, with the ability to delve into any one of those skills at any time. This is a way to avoid hammer-nail syndrome and end up using the wrong solution for a problem just because you didn't know a better one exists.

That said, you can make a note of things you want to come BACK to. With more context on why you need to know this thing, you will have more motivation to learn it, too

one step further: learning is about being able to produce answers and apply ideas, not just read or memorize them. So take exercises one step further rather than just copying them, to solve problems on your own. Example: implement gradient descent for multiple feature dimensions instead of just one, or come up w new ways to visualize data instead of just graphing it.

Setting Goals and Expectations

Do you already have a goal in mind? Or are you looking for more guidance on how to set goals that will advance your skills?

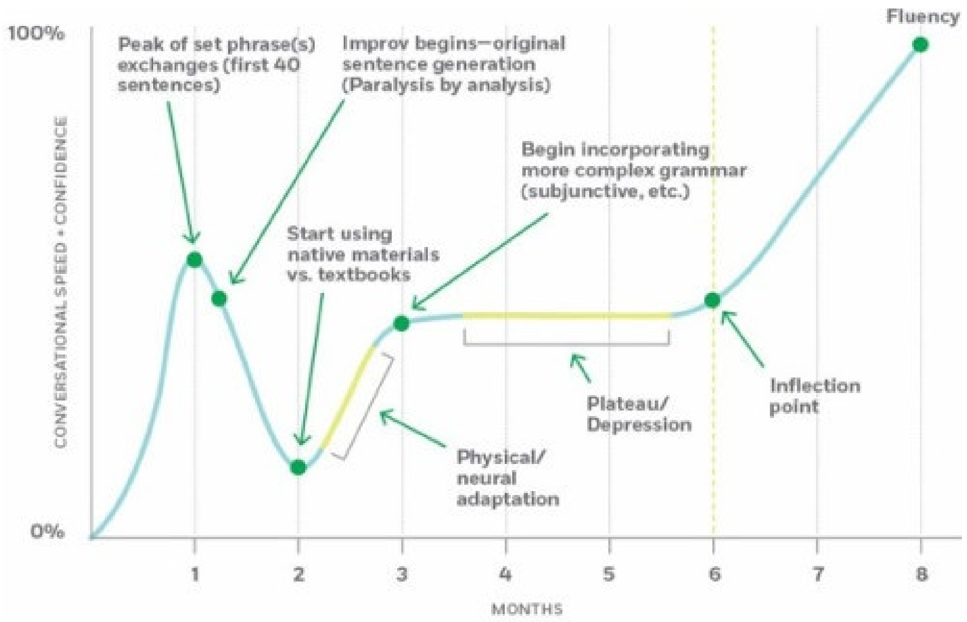
The idea of goals comes with some pitfalls for a self-taught programmer.

The first is the idea of "doneness"—when we accomplish a goal, we consider ourselves done with something. In programming, here is always more to learn, and the technology is always changing. So a goal like "learn iOS" might be too broad, since it could be difficult to establish a finish line.

Instead, it can be helpful to establish a specific project in which to learn something new about programming. For example, you may have a specific iOS app in mind, and you would like to complete that app. A project offers gives you an established finish line and a way to move forward incrementally until you arrive.

It is helpful to anticipate the way that your motivation will wax and wane during the learning process.

This chart represents how you may feel over the course of learning a new programming concept:



That dip at about two months important to notice. When you learn something new, you start with a high level of confidence after a few small successes. Those small successes lead to new things: the chance to try out your new knowledge in the real world, or in the context of a project. Outside of its limited initial context, your new knowledge and skills look much smaller. That experience can be jarring, but those obstacles are an important part of the learning process.

Those obstacles—the struggles, the dips in confidence—will be easier to overcome if you see them coming. You will not always feel excited to study, and you will not always feel like you're progressing. Luckily, the trick is *not* to stay motivated. It is to go ahead and put in practice time even when you are not feeling motivated.

You can use your notebook to set and track goals.

1. Write down the goal and break it into smaller milestones
2. Put a date range on the immediate next milestone, but not the others down the road (when learning is involved, time estimates tend to be way off, and we want to avoid that messing with your confidence)
3. Note changes in the goals or milestones as you go
4. Record which milestone you're working on at the start of each work session in your notebook.
5. ALSO devise a symbol to put in the notebook to share how you're feeling about working toward your milestone that day. Are you motivated? Are you feeling lethargic, angry, or anxious? Do you feel behind? Later, you can note how these days affect your productivity. You might find that you *don't* learn less on those days than on your motivated days. Even if you do learn less on those days, you can go back after a period of time, count them up, and see how much of your learning happened on days like that. Are you 10% ahead of where you would have been if you quit on days you didn't feel like it? 25% ahead? 50% ahead? Furthermore, you can look at trends in these days. Everyone has days like this. I'm serious, everyone. But if you start to have fewer of them, look at why. What has you so motivated? Maybe you can use that in setting future goals. If you start to have more days like this, what's wrong? Are you feeling stuck? Are you not enjoying your project? These trends can clue us in to opportunities to change our system in ways that work better for us.

Sticking to a Study Schedule

Smallest forward step - every day

The Materials You Will Need

A project

problem ordering: https://mkremins.github.io/blog/doors-headaches-intellectual-need/

Choosing A Learning Project

y

Learning from Others

ex. 1 reading, summarizing, and responding to books (once per 100 pages - don't want to get too granular)

ex. 2 reading and annotating other people's code

Programming with Others

Ex. 1) taking notes during pairing to come back to things, what you learned, etc