Learning to Program (The Do It Yourself Way)

Mitch Feigenbaum

1. Where to Start

One of the most difficult parts of learning to program is knowing where to begin. There are hundreds of languages, computers, and programs that promise to teach you the magical art of giving instructions to machines. Any of these programs that promise to teach you to program "In just two weeks" or some similar timeline are definitively a scam. With the advent of the internet, you can learn to be an expert computer hacker[1] for free on your own time by yourself. This book will not teach you how to code, nor is it intended to do so. What this book is meant to do is to get you on the write track and give you the tools to succeed.

But why am I qualified? Because I have experienced firsthand the trials and tribulations involved with learning how to program. The summer before my freshman year in high school, I decided that I wanted to learn to code. I made a list of languages I wanted to learn: HTML, CSS, JavaScript, Python, Java, C#, and C. My goal was to learn all those languages in one Summer, with no prior programming experience. I did not even know what differentiated any of the aforementioned languages. I had only heard of them in name. Many would agree that this is too lofty a goal. However I was young, naïve, and confused.

Suffice it to say that I did not reach my goal at the end of the Summer. By the end of the Summer I had learned HTML, a bit of CSS, JavaScript and Python. I was nowhere near proficient, but I knew of the basic if-else, while, for, and function constructs. More importantly, I had fallen in love with the sensation of programming. To me, it was as satisfying as solving a puzzle or putting together a Lego set. Just like those activities, when you create a program, you can use it yourself or give it to your friends. If you get good, you can even end up selling it to people and businesses. For an amateur programmer, there is nothing more rewarding than seeing others using and benefiting from a program that you yourself have made.

On my GitHub[2], I have posted many of the projects which I have made for fun. In the repository labeled 'Personal'[3], which is reserved for miscellaneous personal code that cannot itself warrant its own repository, I have written the following description.

> These are some personal files I have. I will probably upload here periodically. There is no real structure to this. Just stuff I made. Some of it is junk, some of it is brilliant, and some of it is unexplainable.

[1] The word 'hacker' is often used by the media and non-technical sources to mean someone who, often illegally, infiltrates computer systems. This definition may be attributed to a 'cracker' rather than a 'hacker'. The true definition of a hacker is one who is highly skilled at programming a computer and not just knowledgeable in theoretical computer science.

[2] The link to my GitHub, as well as many other resources to help you on your journey to become a hacker, are in the resources section.

[3] https://github.com/mrf-dot/Personal.git

The best way to learn computer science is not in a classroom setting; it is by example. That is why this book is structured in such a way.

If you were to take a computer science course at a university, you would start with the theoretic and eventually make your way into abstracted levels of computing like web design. This is a good method to learn if you only wish to know the theoretical basis behind computer science, but not optimal if your goal is to become a good programmer. The way that I learned to program was first through true fingers to keyboard coding. It was only later when I learned of the theoretical basis behind computer science. Don't get me wrong: theoretical computer science *is* interesting. However; it is in fact true that you may gain more appreciation of theoretical computer science by first learning to do real programming. It brings meaning to the common constructs in computer science like pointers, strings, mathematics, floating point numbers, loops, and conditionals. Learning about theoretical constructs is a big *AHA* moment. It brings a new meaning to the programs you have created. If, contrarily, you learn to program when you have already learned theoretical computer science, it takes all of the fun out of learning how your programs truly interact with the machine they are run and compiled/interpreted on[4].

As I have already mentioned, learning to program is not something that just happens overnight. It takes months, even years to truly learn and master. When you are reading through this book, do not be afraid to read slowly or reread sections. Computer science is a difficult subject contrary to what many code boot camps would have you believe. Included in the sections are interactive exercises that you may do to gain a more in depth understanding of each language. Hopefully this book will help you avoid some of the common pitfalls and errors that befall many aspiring programmers.

Source code in this book will be clearly marked. Source code may be differentiated because it is offset and centered on the page, in a constant width font. Below is some example source code which prints hello world in C.

```
#include <stdio.h>

int
main(int argc, char **argv) {
    puts("Hello, world!");
    return 0;
}
```

Clearly, the code is differentiable from the surrounding prose text.

[4] Later on in this book there will be an in depth explanation on the differences between compiled and interpreted programming languages.

2. Tools

No doubt the first thing a programmer must do when setting out in their career is creating their programming environment. Your programming environment is as much about the type of programs you use as what is in your physical surroundings. In the next part of this section, I will make some suggestions on the setups that are optimal for good programming practices. If these instructions are followed, it will set the programmer up for future success. However, note that these are merely suggestions. These programs are recommended for a reason: they work. If a programmer already has an environment that is comfortable for them to program in, then making a change to the programs suggested may simply not be worth it. It is up to the hacker to decide what tools they will use.

Below are listed required items needed to program. There is no need to get fancy. Use whatever is comfortable and inspiring.

Chair

This needs to be a happy compromise between comfort and professionalism. Keep in mind that you will be spending hours in hacking sessions, so make sure that the chair you use will be comfortable to sit in for long periods of time. For those who are health-inclined, a standing/walking desk might be right. Many people who have standings desks are very content and even claim that it improves their brain stimulation. Linux and Git creator Linus Torvalds has a setup in which he uses his computer while walking on a treadmill at low speed[5].

Computer

The type of computer a hacker needs varies based on the what they intend to do with it. I am not an expert on building PCs, so I am not qualified to give recommendations on specific parts to be used to build a computer. For a simple setup it might be optimal to get a laptop instead, especially if the prospective programmer is on a budget or must get a laptop anyways for school or work. Particularly if one wishes to use Unix-like operating systems such as Linux or *BSD, the Lenovo Thinkpad (particularly old models such as the X220) are prized. This is due to the lack of proprietary firmware such as EFI and secure/fast boot which impede operating systems other than Windows™. The specifications do not matter very much, especially if light editors such as vim are used. There is no need to spring for twenty gigabytes of RAM. If your computer can play Minecraft with a steady frame rate, it is more than capable to write and compile programs on.

Editor

The editor is where you will be spending most of your time as a programmer. The method in which a program is typically written is edit→compile→debug. Most time spent in the editor is not used on strictly writing code; rather, the time is spent looking for errors, consolidating existing code, and planning out the program. Thus, a good editor must not only have functionality to type characters into a file but also to jump between different parts of the file and make repetitive edits such as deleting entire lines or changing all instances of a particular variable name. In my experience the best editor for programming is VI, as it has native functionality to quickly perform repetitive actions through muscle memory. Many new programmers start out using an integrated development environment (IDE). IDEs may be useful for large projects, but can be confusing or misleading to those

[5] https://www.youtube.com/watch?v=SOXeXauRAm0

new to writing code. It is best to start out using an editor such as Vim, and, if the functionality of an IDE is truly needed, switching to an IDE later. Many IDEs such as Microsoft® Visual Studio™ have extensions that allow use of VI keybindings. Even if a programmer is not per se using VI or a VI clone, knowing VI keybindings is a useful skill that pays dividends. As a side note, it also looks really cool for the uninitiated.

Headphones

When writing code for long stretches of time, it can be helpful to have music to listen to. Listening to music can help pass the time and make the programming process less frustrating. Headphones usage is a personal preference. Some people simply cannot focus if there is music playing in their ear. Others derive productivity benefits if they are listening to music. A good pair of over the ear headphones can make programming a lot more bearable. Note that over the ear headphones are best, as on the ear headphones or earbuds will become uncomfortable after a short period of time. Over the ear headphones are also typically better at keeping distracting outside noise out.

Keyboard

The keyboard is the primary way that a programmer interacts with a computer. It is highly recommended to get a keyboard with high key travel. Mechanical keyboards are the best, but gel keyboards may also do. Keyboards with short key travel, like those typically found on laptops, will cause repetitive strain injury. As an additional note, it is a de facto requirement for programming to be able to touch type. The higher your words per minute (WPM), the better. A rate of about forty WPM is the minimum requirement to effectively be able to write programs.

Monitor

The monitor setup is intended to reduce eyestrain and give the programmer a view of all their code. A three monitor setup is optimal, as typically you will have three programs open at the same time. On one monitor the editor is open, where code is written, deleted, and modified. On the second monitor a terminal window is open, where the program is compiled and run. On the third monitor is documentation or a web browser for reference.

Mouse

The mouse is not extremely important when programming. Some editors and most IDEs rely on the mouse for a number of actions. If a keyboard shortcut based editor such as VI or EMACS is used, the mouse will almost never be used. When programming the mouse will be mostly used to switch between editing and compile/debugging windows. If you are using a laptop, having an external wireless mouse can be more ergonomic.

Notepad

A pen and paper should be in reach when writing code. Writing information like variable names and procedure descriptions on paper can be quicker than having to scroll through code.

Operating System

A Unix-like operating system is ideal for programming. They have native access to the latest GNU tools for programming are by default equipped with command line editors and programs. Programming on the command line in an operating system such as Windows™ requires the installation of a host of software and a bit of tweaking. If one must program on Windows™, there is a bootstrap guide which will simulate a Unix environment to the

farthest extent without requiring a switch to Unix or administrator privileges[6].

2.1. Vim: The Programmer's Editor

Vim is an editor based on keybindings rather than mouse based context menus. The mouse is never used in vim. Rather, different modes are entered to modify text. In my opinion, Vim is the best editor for programming. It allows a level of modification at such high speed that it is sure to give improvements over simple editors like notepad, and its free and open source. If you wish to use VI, I recommend using Neovim[7] as this version tends to be the most up to date and features many features out of the box that must be specified with a build of Vim. It also has some features enabled by default like syntax highlighting and language servers that must specifically be enabled with Vim. Using a VI based editor is a learning curve, but those who put in the effort unequivocally do not regret it. If you wish to use Neovim, install it using the instructions for your particular operating system[8]. Once Neovim is installed, you may learn how to use it with the following shell[9] command.

```
nvim -c Tutor
```

If you wish you may use my configuration[10], which contains shortcuts for the C, Java, and HTML programming languages.

2.2. UNIX: The Programmer's Operating System

In computer circles oft heard are praises of UNIX and its offspring Linux, BSD, and MacOS. The UNIX environment may seem strange to newcomers. One principal of the UNIX environment which seems particularly odd is that everything is a file. Programs are a file, directories (folders) are files, and even hard drives are files. This idiom however is in the interest of simplicity. By stating that every object is a file, it allows for a programmer to specify the creation of a file without giving it superfluous information. Information such as type, permissions, and creation are stored in structures called inodes, which link to the physical file on the drive. Programs on UNIX are simple and general by design, and can yield specific input through mechanisms such as piping and conditional execution. This book will not go in depth on the specifics of the UNIX operating system or the shell, but the AT&T Documentary "The UNIX Operating System"[11] is a great reference for starting out in UNIX.

---

[6] https://github.com/mrf-dot/linuxize-school-pc

[7] https://neovim.io

[8] My aforementioned Linuxize PC guide for Windows™ installs Neovim and my configuration automatically.

[9] The 'shell' is the language used when you open up a command prompt.

[10] https://raw.githubusercontent.com/mrf-dot/deb-bootstrap/main/nvim/init.vim

[11] https://www.youtube.com/watch?v=tc4ROCJYbm0

3. Logic

Logic is the basis of all computer programming languages. By learning some simple rules of logic, a programmer can consolidate their code and better check it for errors. There are three basic logical operators that are used often in programming. In the next section "Operators", these logical operators will be reviewed.

3.1. Operators

$\neg$ The *not* operator negates the statement. It checks if the compliment (opposite) of a logical statement is true.

$\vee$ The *or* operator checks if one or both of two statements is true. Only if both statements are false will it return false.

$\wedge$ The *and* operator checks if both of two statements are true. If one or both statements are false, it will return false.

3.2. Truth table

A truth table asserts the result of a logical operation in multiple circumstances. In computer science, the Boolean values true and false may be represented by the numbers 1 (true) and 0 (false). Below is a truth table of basic operations using the $\neg$, $\vee$, and $\wedge$ operators.

Statement	Value
$\neg 0$	1
$\neg 1$	0
$0 \vee 0$	0
$1 \vee 0$	1
$1 \vee 1$	1
$0 \wedge 0$	0
$1 \wedge 0$	0
$1 \wedge 1$	1

3.3. De Morgan's laws

De Morgan's laws are logical proofs that stipulate

1 The compliment of not A or B is equivalent to not A and not B and

2 The compliment of not A and B is equivalent to not A or not B. The following proofs show the validity of De Morgan's laws.

Proof 1

let $x \in \neg(A \cap B)$

$x \notin A \cap B$

$x \in \neg A \wedge x \in \neg B$

$\therefore \neg(A \cap B) = \neg A \cup \neg B$

Proof 2

let $x \in \neg(A \cup B)$

$x \notin A \cup B$

$x \in \neg A \vee x \in \neg B$

$\therefore \neg(A \cup B) = \neg A \cap \neg B$

4. Programming Constructs

4.1. If-Else

4.2. Ternaries

4.3. Loops

4.3.1. While

4.3.2. For

4.3.3. Goto

4.4. Primitives

4.5. Objects

4.6. Methods

4.6.1. Parameters

4.6.2. Body

4.6.3. Return

4.6.4. Invoking

4.7. Arrays

4.8. Pointers

5. Beginnings: HTML, CSS, and JavaScript

6. Python: The First Frontier

7. Picking up: Moving to Statically Typed Languages

8. Java and C#: Object Oriented Programming

9. C: The Final Frontier

10. Novelties

10.1. Brainfuck and Turing Machines

10.2. G/TROFF

10.3. Meta Computer Science

11. Resources

12. Exercise Keys

12.1. HTML

12.2. Python

12.3. C#

12.4. Java

12.5. C

12.6. Roff

Table of Contents