Each computer science department has its own origin story. In some schools it was created out of the math department, in other schools it emerged from the electrical engineering department. It is easy to recognize their foundation when you examine their curricula. Math folks study methods while engineering folks use computation to solve specific problems. The math influenced departments are more theoretical, stressing the abstract over the concrete. The engineering folks grade you on whether it works, not so much on its methods. I have been beaten by both sides. Luckily, I learned programming mostly on my own, by getting paid to write code.

My mother taught me a version of FORTRAN in the mid 60s. I used it again, for my one course in computer programming, at the University of Wisconsin. Punched cards into a window, stacks of printout back. That sort of hassle did not make it fun. When I bought my first computer in 1978 it booted directly to a BASIC prompt. You wrote code and ran it. When you turned the machine off your work was gone. There was a lot of typing involved. Then came the cassette tape, the storage system from HELL. Next came assembly: Z80, 8080, 6800, 68000, eventually 8086 and ATmega32; then I learned Pascal, to use the Macintosh API. C came around 1979 so I was using Pascal and C at the same time. I learned C, then how to use it to program the Mac, in the early 80s.

I was learning from books, magazines, and from monthly meetings of the computer group. The bad thing was there was no single BASIC. Each company had its own variations. While I was learning how to write code, I also learned how to translate code. Once I got good at that, I got out the FORTRAN books and translated that code into BASIC. It was all BASIC until I got my hands on Forth. That language really taught me how to code. It is the source for my "make sure your function fits on one page" notion. This was enforced by the language, because it was written in pages. A few pages of Forth was enough to make a 3D app on my TRS80. Plus, it was blazingly fast. I had to add wait states when I wrote a fly-through of my scene. The first time I tried it I thought nothing had happened. Forth is a threaded interpretive language. You write a program out of words which you have crafted. Each word is made out of other words, or the base language. The final program is started by typing one word at the prompt. A high level, interpreted language which ran at assembly language speeds. Eek! I was hooked. However, the market place forced me to change. First Pascal, using a clunky, multi-disk system on the TRS80. Then I got the very first Mac in 1984. With that Mac I learned how to write windowing code, and C, at the same time.

Machine language is what a computer uses. All languages are either compiled to create the executable machine language, or are interpreted to create the machine language. I have written exactly ONE app in machine language. It is a real pain when variables, numbers, addresses, and operators are all in 1s and 0s. Once you have written an app in machine language you NEVER want to do it again. Doing it makes you love how easy assembly language is in comparison :) Assembly language is where variables, numbers, addresses, and operators are easily recognizable. It is kind of fun, and it is very close to the machine. But, it does take time and patience to create a medium size assembly language app.

COBOL was invented by Grace Hopper to take the next step. She took the subroutines she had been saving in a notebook, and started crafting a compiler. That compiler took the high level language code

and translated it into each specific computer's machine code. High level languages let you transfer an app from one computer to another with a different architecture, to compile and run it. Assembly and machine language won't allow that to happen, since they are so closely associated with the hardware.

Next came interpreted languages like BASIC, Python, or Perl (plus many, many others). While using BASIC, for instance, you are actually running a program. The program is called BASIC, and it interprets strings of text into machine code with a few internal steps. Each time a line of code is run it needs to go through the interpretation process, which slows the app down. Fast, modern processors mask this, but the interpretation limits the size of the problem you can work on.

A compiled language only needs to do this "interpretation" once to create an entire program in machine language (the .exe file). That is why compiled languages are more compact, and much faster (20 to 80x is normal). With a faster computer, and a compiled language, I can solve much larger problems than if I were using an interpreted language.

I know C is more difficult to learn. It's advantage is it is much closer to the machine and much faster. There is also a vast library of code written by professionals, and academics, which is readily available for your use. Working on a Windows box blocks you from a lot of the benefits of this. That is why I advocate you set up MinGW on your computer. It puts the Unix shell into your system, while being light and compact. Now, when I use the command prompt under Windows, I can either use DOS or Unix commands.

Unix tools are designed to be strung together, for instance:

```
sudo find / -name "*mp3"   \
| grep "Music"   \
| awk -F/ '{print $(NF-1), "\t", $NF}'   \
> musicFiles.txt
```

There are four commands in that one long line. The '\' character is just a line extension signifier. This command string is entered as one long string without the \s if you're typing. If read from a script, use the \ character to provide whitespace. Now, as to what this does, I'll go line by line. sudo find ... tells the system to look for all the mp3 files I have on my entire network. The ouput gets piped (| Unix tool) to grep, which is a regular expression tool. That takes the list from the find command, and looks for those files with Music in their directory path. The output gets piped into awk, which parses out the last, and next to last 'columns'. This gets sent directly to the musicFiles.txt file. This command string is very fast because it does not write output to the display. By typing this command string I found all of the mp3s in the Music directory subtree, grabbed the last two levels of the directory structure, and saved them to disk. All that work gave me the list needed to create my mp3 music database.

Currently, I have six computers on my network. Two of them are on almost all of the time: the server and this box, which is where I do my typing on a Steelseries keyboard (very fine). The server is a headless Ubuntu Linux install named Hermes. Headless means it has no GUI at all; everything is command-line which I can administer directly at that computer, or indirectly, via a terminal from my windows box. Being headless means it is much smaller, with fewer pieces. This makes it easier to

learn all the parts. The GUI code takes up a lot of space, both in the directory structure complexity, and the amount of processor cycles it requires.

I use Hermes as my file server, movie server, project repository, and music server. I can pull eight movies simultaneously without stressing the network, or the server. I built the box out of some very cheap parts. When I found what I could do with it I realized I did not need a high end box to do the job. No video card because that is built into the motherboard. I can use my main box more effectively now that I have the server. ALL of the data I create or download is stored on it, not this box. ALL of the programs I write go on it too. I actually store very little on the main box. C drive is solid state, but it holds ONLY the OS and NOTHING else. D drive hold my applications and config files. Everything else is on the server.

I can perform backups silently, and often because the server is running Linux. When I rebuild my old server I intend to mirror one box onto the other so I will always have a back up. Instead of mirroring to a neighboring drive, I mirror to a neighboring box on the network. I also store all of my windows backups on the server. Setting up a CRON job to do the back ups is easy. Once written and tested, it runs in the background, with no problems. My main box is quite uncluttered, and fast, by using this architecture. All of the hard drives have a balanced amount of space with excess capability for the future.

Using the Ubuntu server gives you a MySQL server too. MySQL is used internally by Samba, which is used to map Windows drives onto the Unix file system. You access MySQL by calling it with putty (very capable terminal program) from any Windows box on the network. On a Linux box you would open a terminal window and ssh (secure socket connection) to your servers. Once you call up the database you want, you send it MySQL commands to post and retrieve the data. With a few Unix commands like: grep, awk, tr, find, ls, and the piping mechanism of Unix, you can create new tables for your database with a few scripted lines.

MySQL is a relational database which can be created from multiple, independent tables. That means you can create each table, one at a time, often days or weeks apart. You don't need to be so exacting when you create your database schema. I have been finding tables on the 'net and then massage them a little into tables for the database. Lots of good practice in how to use the tools of Unix, and I create the tables quickly from the latest data. If you set up a cron job to gather financial data, you could fill a database without any need of supervision. Set up the script and let the computer do the work. You can even set it to email you when some event occurs.

Installing MinGW lets you use GCC, the Gnu Compiler Collection. gcc is an up to date compiler for C, C++, Ada, and FORTRAN, plus a number of other languages which I cannot remember off the top of my head. It also has a number of 'back ends'. With a compiler the front end is the language you chose, but the back end is which processor are you compiling for. While I normally compile for the X86 family, gcc also lets me work with my microcontrollers. I use Atmel chips these days but have used Motorola and TI chips. <soapbox>I was using Atmel chips years before Arduinos came along and cannot figure out why they changed all the nomenclature. I avoid the Arduino stuff because it is hard to read the tech sheet and make heads or tails out of what I am supposed to do with an Arduino

term. To me a shield is used to shield EM fields. They use the term to mean daughterboard. </soapbox>

Microcontrollers used to be programmed exclusively in the assembly language of the chip family. But now that gcc supports so many of them, you can use C, or any of the other languages of the gcc family, to write code for a microcontroller. Normally there are only a couple lines of pseudo-assembly language in them. I guess that is where the Arduino stuff comes in and covers up. I would rather work closer to the machine and KNOW what is going on. I like things to be as simple as they can be and still work. Makes debugging much less painful. Layer after layer of abstraction is great for letting folks get started but not for day to day work. All those layers throw out a lot of control. No, I am not advocating assembly language for micros, a higher level language works wonders. Plus, you can use a tested library of the assembly language routines, instead of writing your own.

Using gcc to compile and link a windowing app in C is not as difficult as you may think. It can get tedious, with a lot of menus, but that is a choice. Writing a windowing app for Microsoft is mostly a matter of taking a few template files, and adding in other chunks of code from libraries on the net. Then you do write a few lines of C-like code.

Yes, they are actually in C, that is what Windows is based on, but they don't look like C. The original Mac was based on Pascal. Cool language but a little too academic, not enough short cuts for professionals, but it did teach me my formatting style. The biggest advantage of learning how to write windowing code is the impression you give folks with your coding skills. While it is not that much different than writing command-line apps ,it impresses even the academics.

Writing windowing code for either the Mac or Windows OS, is a matter of learning, and using a set of APIs. You link in the windows.dll file to make commands available. The various windowing commands are built into the .dll file. gcc just compiles them and links in all the right commands.

Yes, there is some actual C code involved, but most of that is pretty simple. The hard work is being done by the commands you look up, and string together. I have been streamlining a few templates to make it simpler for me to write code. Using a makefile with gcc is much simpler, and faster, than using the full blown Visual Studio. <soapbox> Since it is a Microsoft product it gets updated about every 15 minutes. You are ALWAYS out of date unless you subscribe. I find that an annoying waste of my time. So I stick with the makefile/gcc route and find I work faster, with less stress. By using the win32 library, with the makefile/gcc toolchain, I can write decent looking, well performing Windows apps with much less stress all while being totally free. Same with Ubuntu and MySQL - FREE. The gcc/microcontroller system - FREE. Well, the microcontrollers cost money but that's hardware. All of the software needed to program them is free. I mean free as in beer and free as in unencumbered. </soapbox>

By the way, the HTML option is kind of the opposite of the MySQL option. Where MySQL can find and use a non-local database, HTML code can use a CGI script to post to a non-local database. Keep your DB in the cloud, or on your network, and access it anywhere. A CGI script is called a common gateway interface. It performs the authentication to log you in, then you can send MySQL commands

wrapped in HTML code. But a CGI script means your ISP lets you use CGI scripting. They may charge you a little for it.

Hopefully, this gives you some idea of what I can teach you. I hope you can gain some knowledge by doing this work on your own computers. As in any programming, there are many ways to write things correctly. Choosing a language, or an architecture, should be based on the problem, not on what you know or don't know. Since you have learned Python you actually know some C. Perl, Python, Ruby, etc. are based on the C syntax so it is easy to work in all of them. I find it is best to have the tools at home (and free) where I can use them to make all the mistakes necessary to learn them.

While Mac OS was written in Pascal & Windows in C, Unix was not written in C. But, within a few months of the first C compilers being developed, the Unix OS was rewritten in C. That is why it is the language to use under Unix. When writing code on a Mac, or under Windows, you are removed a few levels from the OS, as well as from the machine. C under Unix/Linux allows you to get very close to the machine. I favor control, over ease of use, unless I'm in a hurry. First, get the problem solved. Then do it well. Solve the problem with scripting tools or languages. Then, while folks are finding new uses for your code, you rewrite it in a faster, more compact language. Throw away code is a good way to beat out an idea. Then you write the pseudo code into your notes for later.

The first link is the main reason I am using Ubuntu. All of the installation commands listed on that site worked. That is not often the case which makes life quite frustrating. The second link is for when you want to go farther. https://www.howtoforge.com/setting-up-a-linux-file-server-using-samba

https://help.ubuntu.com/lts/serverguide/serverguide.pdf

A five year old box would be just fine for this. Once you get it networked to your normal computers then you can have a large increase in storage space. The neat thing is that accessing and using those files does not effect the speed of the computer you are using. As long as your network does not get bogged down, everything runs faster than normal. It is an architecture thing. The network data does not come in along the same bus as regular data from memory. Balancing the loads on the busses maximizes throughput. Using the GPU for graphics code, and for parallel apps, can offer astounding results. Oh, forgot, by using Linux as a server it has fewer parts, and a more sparse directory tree. This really helps you learn the OS if you're interested.

Think of algorithms as recipes. But include everything. Do you have all the ingredients? Do you have all the tools and utensils? Do you need a table, a stove, a bowl? Perform a thought exercise. Choose a cookie recipe you like, or pie or cake. Then figure out all the steps, and sub-steps, necessary to make them. There are many correct answers. You can make your cookies from scratch or from a mix. But remember, if you truly want scratch made cookies you first have to create the universe. Think just a little smaller and you'll learn more rapidly

Imagine algorithms as functions. $f(x) = x^2$ has input and output. But this one is more interesting: $f(x) = x^2 - 2x + 1$ More interesting because it can be factored. $f(x) = (x+1)(x-1)$ The reason I bring this up is because computer functions can also be factored into smaller parts. Those parts, once generalized, can be reused either in the same program or in future programs. While I was teaching C to undergrads I

stressed this. The best students took my advice and found later assignments much easier to write, from tools they had built themselves for previous exercises. You should build your own toolbox for each language you use. Ultimately, you can save your algorithms as pseudo code in your notes and rewrite them in any language you like.

Good software is built through craftsmanship and artistry. You need the artistry to find the most elegant way to solve the problem, while you need the craftsmanship to write the most bug free code. If you enjoyed the cookie thought exercise you might like writing code. But, like art, code is a verb; you have to do it, not just study it, to learn more. Instead of writing word by word, you will start thinking in phrases or idioms, then your code will write itself. I think of writing code not as an end in itself, but a tool to explore whatever problem space I am interested in at that moment. The computer becomes an extension of me, processing far more data than I can ever imagine, while doing exactly what I want it to do (or more accurately: what I told it to do).