HOW TO LEARN PROGRAMMING:

Navigating the dense jungle of Computing, Software and Technology

This work is Dedicated to Stephen Wolfram

Cover designed by KDP cover designer
Images of computational systems are from the NKS (A New Kind of Science) book courtesy of Stephen Wolfram and Wolfram Research

Jackreece Ejini
jackreeceejini@gmail.com

Printed in the United States of America

First Printing: Jun 2018
KDP

CONTENTS

INTRODUCTION

I am constantly questioned by aspiring programmers; they want to learn programming and want to know what programming language to learn. The question they really need to ask but don't know they need to is: they want to know how to go about their entire programming career. Some of them are usually disappointed when I tell them that if the task they seek to learn is how to create software then knowing one programming language is not usually enough. They have to not just learn multiple programming languages but multiple technologies to be able to create software.

Although with a single general programming language you can solve any computational problem for which you are beset with, in the real world it is usually not as straight forward as that. Many languages are good at somethings better than others, even though they are called "general" programming languages. This is only in the theoretical sense. Another issue is programming style! We eventually get biased to doing programming in a certain way. Some people like the structure of Java, others will opt for the freedom of Python, or the Super power of Wolfram language.

Apart from choosing a programming language there is also the problem of navigating the entire jungle of computing and software technologies. This will depend on kind of computing you want to do: Are you a research scientist? then your technologies and languages will vary. Do you just want to design a web site, or web application? Are you planning on writing some large sophisticated spacecraft control software or do you just want to have the ability to do computational thinking, that is use your computer to solve your personal problems that have computational aspects?

Initially I planned on writing this work as a kind of sequential list, programming language and capabilities. But that will be boring and I am sure with some effort you can pull out such information from the web. What I choose rather is to give it a personal touch, to tell you my own personal story in the jungle of software development. I believe through such personal stories, which you cannot pull out of a Google search alone, you can have a deep perspective of the direction you want to choose.

This book is like the first stages of a rocket, its designed to take you into orbit. Once you're in orbit there is a vast space of knowledge to explore. Without this book, you might be lucky to do some research of yours and stumble on a language learn it and start working. But you might end up like a Jet plane, cruising high altitudes but unable to reach escape velocity and break free from gravity. Even if you have started programming and are maybe 3 years into it, go through this work there will be some nuggets of wisdom here for you.

I have been programming since 2001, although not very long, I have delved into many waters and I have some wisdom to share for the beginning programmer. I have chosen to write this so that if I am questioned on this issue again, rather than start a long narrative taking about an hour or more, I will just refer you to the book.

What is Computation?

Computation is a fancy word we give to any process that follows definite rules. You are already computing things, the fact that you can speak a language means that you are already computing something. In order to speak a language your brain strings together the words in the right syntax. Syntax is a series of rules that makes sure that we are joining things together in the right way. Without the right syntax, we might use the right words but our sentences would not mean anything. Your brain is automatically processing syntax when it understands something that was spoken or read. Spelling is also important in using language, while in our natural languages spelling is not all that important in an informal setting, as we can usually deduce the meaning of words depending on how they sound, in writing a program we have to be very accurate in our spellings.

So, what are the rules that something must follow to become known as a computation? Like we have seen above, syntax in languages is one of those rules that a process must follow to be called a computation, but in general we can choose any rule we want to describe any system we want, not necessarily language, in language, we also have the rules of semantics that something must follow to be meaningful, because sometimes obeying the raw syntax of a language can end up producing nonsense. Following semantic rules allows our language expressions to have meaning and that is another layer of rules natural language has to obey.

Input -> Compute -> Output. Most computation follow this format, we have some input, we process it and we have some output afterwards. You are already using this format unconsciously while using natural language, you have an idea you want to convey, this is the input stage, then your brain turns this idea into words the Compute stage, then you speak or write these words the Output stage. By implicit computation I mean a computation where you don't explicitly apply the rules yourself. When you talk, you are not consciously applying rules, your brain does this for you. Explicit computation is where we apply the rules explicitly like adding numbers on paper or doing it on a computer.

What can we Compute?

In theory, we can compute process that follows definite rules, but in practice there are only a small subset of things we want to actually compute. There is a whole big field of Computational systems out there and a hand full of them have been thoroughly dealt with by the eminent Steven Wolfram in his book A New Kind of Science. If you discover at this point that this is the kind of thing you are interested in, I would advise that after dropping this book you go www.wolframscience.com there you will find all you need to start exploring computational systems.

What are the Atoms of computation?

The physical world is made of Atoms; in the world of computation we also have atoms. The atoms of computation are those discrete pieces of information we deal with in a computational system. In the wolfram language, a computer programming language, we deal with Numbers and Characters mostly. We also have atoms like colors that we can deal with in the language. From these atoms, we can build molecules, macromolecules, cells, tissue systems and organs and organisms of the software world, powerful systems doing computation at many levels, integrating their actions to provide us software systems help us solve problems of varying complexity.

Why should you use a computer?

What we can actually compute in our heads is quite limited. As an example, when we want to multiply numbers, most people can only multiply 2 digit numbers, while with some training some people have gone as far as 4 or 5 digits, it still requires so much effort to reach that level, and going higher is very difficult but with computers these things are a breeze. Your calculator is a specialized computer that has a fixed program that computes only functions that are inbuilt into it. Your laptop computer is a general-purpose computer, meaning that with a programming language you can make the laptop or desktop perform any computation you might want to perform not just addition or multiplication. So essentially we use a computer to compute things that would be very difficult or very tedious to compute by hands and a programming language is a way to communicate with our computer system instructing it on “how” to compute the things we would like to compute.

I put “how” in quotes because the computer in reality doesn’t know anything! The knowledge it possesses on how to compute things must actually be specified by a programmer. You will have to allow this point sink in to really understand what computing is about because most people have the idea that the computer has intrinsic knowledge of things. Just like a spanner used for tightening bolts, the computer is a tool, a highly generalized tool for building other tools.

A good programming language like the Wolfram Language makes the process instructing the computer on how to solve a problem very easy and intuitive so you can focus on the problem you want to solve rather than on the details of the programming language.

* * *

1. CHOOSING A GENERAL PROGRAMMING LANGUAGE

When it comes to solving a computational problem, the choice of programming language will determine how easily you can move from problem conception, analysis and finally coding a solution. In this write up I will focus on general programming languages.

So, what is a general programming language? A general programming language is one in which we can formulate the solution to any computational problem, it contains most of the structures that enables us represent the problem presented as an abstraction. Examples of general programming languages include but not exhaustively: Wolfram language, Python, JavaScript, Java, C#, C++, C, Lisp. From my research, there are about 178 or more programming languages out there. My purpose is not to give an exhaustive analysis of programming languages, but to pick out a few general-purpose ones for our discussion.

There are other languages that are not very general, because they do not allow us to represent all kinds of problems, amongst them are CSS (Cascading Style Sheets) which is used for controlling the display of information on a web page and HTML (Hypertext markup language) which is also used for the same purpose.

A general programming language can be either Functional, Object oriented or Procedural. There are also languages called declarative languages but I will not discuss about them in this writing. I will focus more on the aforementioned three classes.

The classification of languages as either object oriented, functional or procedural depends of the availability of certain structures within such languages.

What are language structures? They are certain constructs that enable us solve a problem computationally, they are usually abstracted from lower level structures found in lower level languages like Assembly language.

The abstraction hierarchy:

The computer is built up in layers from the raw logic of the electronic circuit board, to the direct way of controlling these circuits using machine language, which is merely a collection of instructions issued as sequences of 1s and 0s. The reason for climbing up the abstraction hierarchy was to ease the solution to programming problems. Transforming some computational problem into machine language is a very tedious job, because we have to force ourselves to express our problems within the universe of machine instructions, so people came up with the idea of assembly language which is merely grouping a bunch of repeated machine instructions and representing them with a mnemonic, a character based representation easy for humans to cope with.

So, certain machine instructions that occur together because they accomplish a particular low level machine task will be grouped into a singular operation with operands. An example will be the typical problem of moving data from one place to another in the computer system. This could be from the RAM to the registers or register to register or in any other modality acceptable by the computer model. Instead of issuing a series of machine specific instruction in binary like this nonsensical example below, nonsensical because it doesn't really represent the machine instruction of any computer but just looks like it so you can get a feel for machine language:

```
1010101110
0010101011
1100111000
```

We can simple have a simple assembly instruction like

```
MOV A  <-  B
```

Where A and B are, variables representing memory locations.

It is this grouping and naming that is at the bottom of computer programming and all the way to the top. This is how the abstraction hierarchy of programming is built via grouping and naming we conquer! The core of the software that enables you to write assembly language called an assembler is actually written in machine code.

In most modern computers, the lowest level of abstraction open to the programmer is the assembly level as most of the machine code is already written in the factory where the computer was manufactured. If you need to do anything below assembler, you would then be designing your own computer from more primitive components.

Above assembly language we have our general-purpose programming languages. Even though assembly language simplifies the process of creating software from raw machine language, it is still horribly difficult to program in assembly language. Because the language structures are so primitive you will require a lot of code to specify some very basic processes which are far out removed from the problem you are trying to solve. It is because of the need to simplify software creation, passing most of the burden of program execution to the computer so that the programmer can focus of thinking of the problem that led designers to start building the higher-level languages.

Assembly code can be very efficient because you are dealing with the computer in a very direct way, it's more like short hand for machine code, not much is required in order to transform assembler code to machine code than some sort of table that groups related machine operations to a single assembly statement with operands. In order to design a higher-level language, you will have to design a software using Assembler that enables you write higher level code. This software is known as a compiler.

In compiled languages, we write code in the language of our choice and then a software called a compiler transforms our code into assembly language structures which is further transformed into machine code for the computer to execute on its electronic circuitry.

Apart from compiled languages there are interpreted languages. These interpreted languages enable you to execute independent snippets of code without have to compile anything completely. They possess some kind of read -> evaluate loop process that reads your input statement and executes the statement after it detects a termination of that statement. But within even interpreted languages are structures that do some kind of compilation and transformation to lower level assembler and further down to machine code. No matter how high on the abstraction ladder we climb, if we write code above it ends up as 1s and 0s below which are the electrical signals on the circuit board of our computers.

At the high-level languages, and for now I would ask that we ignore the distinction between compiled and interpreted languages and just focus on the other categorization of object oriented, functional and procedural, we have certain constructs or structures that enable us express a computational problem.

Above assembly, the we start focusing more on the problem we want to solve and less about the way the computer works. How does assembly limit us to think very hardly about how the computer works? Assembly languages vary from one processor manufacturer to the other and even from processor to processor sometimes. These variations come as a result of what is called an instruction set, which is the unique set of instructions that a particular computer manufacturer designs to enable software to be written with their computer processor. It's also similar to the distinctions between high level languages but is more tied to the particular computer you are using. Assembly Programmers expect a processer instruction set to contain instructions for doing certain tasks, but how these instructions are implemented is a decision taken by the processor manufacturer.

Although the processor manufacturers are totally free to include any assembly instructions they deem fit in their instruction set, it must conform to what the compiler designers who design high level languages expect at least at a basic level.
As an aside, you must also note that the assembler can have instructions that are not recognized by the computer manufacturer but added only to the assembly software that enables you write assembler. But in the end, all things must end up aligned with the instruction set. The purpose of this writing is to help you build a solid foundation of understanding about computing, software and technology and is not a PhD dissertation by any means. There are so many details I am ignoring or glossing over for the sake of clarity.

High level structures are different from this low-level instruction sets in the sense that rather than expediting low level operations or providing an interface to access unique hardware structures, they are usually designed to make it possible to express the problem you want to solve computationally.

Structures in High level languages

Every programmer expects the high-level language designers to include certain structures in their languages just like compiler writers expect instruction set designers in processor companies to do the same. We have grown to expect certain high level structures.

1. Assignment: a means to create and store variables
2. Conditionals: Means to make decisions about how the program will flow, these are also called flow control statements. Examples include IF, SWITCH, etc.
3. Looping constructs: These are structures that allow us to repeat operations, examples are FOR, WHILE, DO, etc.
4. Functions (code abstraction): When faced with a bunch of high level code that have become super useful we don't go ahead to create a higher-level programming language. Rather we just wrap that piece of code up in a function! It has been proven by computer language experts that with the functional construct, we can build up code of any complexity, so there is actually no need to build another abstraction above the high-level languages the same way that the low levels have been built. Different languages have their function definition contract specifications and although the basic idea is the same, there are subtle difference across languages.

The difference between functional programming languages and procedural languages. In a functional language, everything you want to do is specified as a function. It is required that you solve a problem by either using an existing function or creating yours. Even constructs like IF are actually functions. A function is like a machine that does certain tasks and either returns an output or is used for its side effects on other operations. A function can contain other functions too ad infinitum theoretically or depending on the practical tasks you want to solve. Most purely functional language doesn't allow side effects and use functions as the highest and lowest possible structure you can use to specify code. A purely functional language is LISP.

A procedural language allows a mix of other structures and functions too. The execution unit is usually a file, while in functional languages the execution unit is actually a function. You can choose to group related functions into a file and use it in other programs, but in a procedural language like C, you just have to group your program into a file and run the compiler on that file. This a kind of loose description between functional and procedural languages, the core idea to take in is that in a functional language everything is a function, and in a procedural language we can have functions and other structures separate but functioning together in one program.

Object oriented languages

Although the highest necessary structure needed to solve computational problems is the function, some languages include the idea of Classes, it is these languages that are usually termed object oriented languages. Although in a language like Python everything is an object which is actually a completely different conception from what Class based object oriented languages specify. In python, the idea of objects is fully generalized to include anything that has an attribute, that is usually accessed using the famous dot notation which enables you to access an object's attribute like: Object.attribute, attribute but can be either a method or data. Python also has mechanisms for defining Classes but its conception of an object is not limited to the Class definition. Another powerful generalization of the idea of objects is found in languages like JavaScript, but the main idea to keep in mind is that an object has attributes which could be functions called methods or data.

Our discussion here about object oriented languages is about the Class based languages like JAVA, C#, C++ etc., in these languages the basic unit of program specification is usually a Class file. Object oriented programming is borne out of the philosophy that everything in the world is an object, something that has attributes like methods to perform actions and also data, and thus to solve a computational problem, we simply have to define a universe of objects and their interaction with other objects. The CLASS statement determines the beginning of the creation of an object. In some languages, the Class itself can also be an object in something called Class-Objects, this is a bit subtler and more detailed study of a specific object oriented language will prove fruitful in clarifying this idea.

A class indicates the most general specification of a "thing", while an instance of the class is the specific thing. A simple example will suffice. We can have the general class of things called shape, but we can have instances of shapes like squares, triangles and circles. When designing an object hierarchy to solve a computational problem we define a class to represent the most general conception of our problem, in this case Shape, then we design sub classes that deal with more specific Shapes, like Circle and triangles.

A sub-Class inherits attributes from its super-Class (parent class) and also defines attributes of its own. In most object-oriented languages, there are mechanisms for controlling what is inherited from the parent Class which are generally called access control Specifiers. There is usually the Public, Protected or Private access control specifiers in most languages, they determine how subclasses can inherit attributes from the parent Class. An access control specifier is usually added in front of an attribute declaration.

Also in some object-oriented languages, you can have Classes that are either abstract or concrete. An abstract class usually specifies attributes that will need to be specified in a concrete form by subclasses that inherit, this is usually called overriding an attribute. You can also have interfaces which look like abstract classes but are just raw specification of a contract that any subclass implementing the interface should obey.

Each Class usually has data associated with it and also methods, what actions the shape can perform. We can have the data associated with the subclass of the triangle shape as the string "A small Triangle", and then the actions that this triangle shape can perform like draw itself or interact with other shapes are called methods, which are actually functions that are attached to the Class. So, we design Classes to solve our problems.

Defining Classes is just the first stage of solving a problem in an object-oriented language, the magic of object oriented programming languages does not happen until we actually try to use the Class by instantiating it into an Object. So, we instantiate a Class creating a class instance or as most programmers will say an object. Most programmers do not say Class object; they talk about a Class and an object of that Class. This is just lingo as Classes can also be objects. So, when a programmer says she wants to create an object, she is usually talking about instantiating the Class. So even though a triangle is also a Subclass of shape we can have different types of triangles all performing the same tasks but maybe with different names like: "A small triangle", "a large triangle", etc.

If we wish to further subdivide the General triangle class into more specific triangles like Scalene triangle, equilateral etc., depending our purpose we can either just use the general triangle class, using different triangle names or just go ahead to sub class the triangle class into more specific classes.

The main workhorse of the Class is the method. The method is a function that is contained within a class with the special feature that the first argument to the method is actually the Class instance itself. This Class instance that is supplied as the first argument is conventionally called Self in languages like Python. Like I said earlier the function is the highest abstraction necessary to solve computational problems, so even when we find it in object oriented languages in the name of methods, it is still where most of the logic of the problem solution is encoded.

There are two ways of composing the solution to a programming problem using Classes: Inheritance and Composition. When using inheritance, we construct a Class hierarchy decomposing a problem into more specific Classes and then importing this fully functional class machines into a kind of Main class, the final place where the solution to the problem is composed by using the individual Class hierarchies that contain the solution to individual parts of the problem.

When using mostly composition we avoid deeply nested Class hierarchies and rather create a flat architecture of Classes that handle some fundamental aspects of the problem and composing solutions by calling classes that can handle the specific problem. In pure composition, you view Classes as bags of functions and not necessarily philosophical in any way. Although object-oriented programming purist will frown at this view and will prefer building a large hierarchy of inheritance to solve a problem, the pragmatists will always favor composition.

In practice, it is difficult to be "pure" when designing solutions to computing problems that is why languages like Wolfram and Python allow for a much more flexible system of design without restricting the programmer to a single programming style. It is hard to design an object-oriented program using purely composition or inheritance. There has to be a mix because both methods are required in expressing the final solution. The most important thing is what method dominates the design, from experience flatter systems that use more composition and less inheritance are easier to understand and maintain.

Software maintenance

You should know from the beginning that the job of writing software doesn't end when the final executable is supplied. Apart from bugs that will be discovered later while the final version of the software is running there will be other issues that come up later on during the lifecycle of a program. These include stuff like: request for new features; request for the modification of certain features like the placement of a button in the user interface; redesigning to accommodate new workflows; optimizing for new technologies, etc. These tasks that come in during the later phases of software development when the final product has been shipped is what is referred to as maintaining software. Seldom is your job as a programmer over on delivery date. This is why choosing the programming language and design structure of the entire product is very important. As we will talk about later in the section on designing software or hacking it out, your methodology will impact the entire lifecycle of the software you are writing.

Why object oriented programming?

Like I have said previously, we can we can solve any computational problem with functions, so why classes? Classes were created to alleviate most of the problems faced by programmers building large programs or working in a large team. The ability to create Classes which are just contracts for the way a certain problem should be handled, managing large code bases becomes much easier with object oriented languages.

So rather than thinking of designing a whole new level of the primary abstraction hierarchy that led to functions (or procedures are they are primitively viewed) to tackle the definite problem of very large software systems, programmers just extended functional or procedural languages into object or class oriented languages, because in the deepest sense of the word and ignoring any philosophical ramblings, a Class is just a bag of functions and data. It is not fundamental to computation.

So, do you really need an object-oriented language to be able to solve computational problems? The answer is NO; the functional abstraction is enough and if there is a good package management system in place then managing large amounts of code is not usually a problem. The procedural programming language C was used to create the enormous Linux operating system kernel containing millions of lines of code written by thousands of programmers, so if this was possible, then what is not possible with functions and simple procedures and a good package management system. Although gigantic pieces of software are written in object oriented programming languages, a beginner should focus first on learning and mastering computation before delving into those monsters.

There is nothing wrong with object oriented programming and you will be faced with code at your organization written in them, the thing is that most of the time when you want to quickly express your thoughts in code then object oriented is not the way to go.

The monster called Python

I am usually into efficiency of action. When faced with a problem I usually find the most expedient way to an optimal solution. When I wanted to venture into programming after being obsessed by computers when I saw them in movies, without actually seeing one physically. I always contemplated what was the best route to conquer such a large field of action. There was no easy access to the internet then in Nigeria, 2001, so I couldn't do extensive research like is possible today. One day I encountered a book on C/C++ can't really remember the title, but I started reading it straight up.

One reason I decided to write this essay and indeed to include the personal story I have started in the previous paragraph and which I plan to continue after this is because I have so many aspiring programmers coming to me beady eyed with the question, what programming language should I learn? This usually starts an hour or so long discourse that leaves me begging for breath. I am always passionate about programming education so I choose to capture my thoughts in this work, so that when someone asks me the question the next time I can refer them to this rather than go on that long arduous conversation again, although I am willing to talk to people about questions they might have after reading this book.

Continuing my story, since I didn't have access to a computer I decided to do the exercises on paper, so I would write the code on paper hoping for the day I would have access to a computer to execute it. It should be noted that when I had access to a computer I had very few syntax errors in my paper and pen code so I was quite accurate. Of course, there will be errors in results, but it's the effort that really mattered.

Randomly I came across a friend's desktop computer and an offline web document titled: HOW TO BE A HACKER, By Eric S. Raymond and my curiosity led me to open it. I found out that the word hacker hand been abused and that people who broke into other people's computers are actually called crackers, while a hacker was a kind of disciplined individual usually found in the field of programming but can be found in every field from electronics to music for whom excellence in their activity was their primary goal. Eric went on to describe what you needed to learn in order to become a computer hacker, remember the definition of hacker I just gave. In summary, his requirements were:

Get access to a Unix or Linux system and then learn the following languages:
Python, Java, C/C++, Perl, lisp. Also, know some html stuff etc. This is not exhaustive and I think a live version of this document should still be available on the web, but this was a general framework for learning programming and eventually becoming a "hacker" in the field of programming if you followed it to the level of mastery. This is where I will stop with Eric's advice because I have followed the path he laid out and expanded it to suit my own motivations.

So, why is Python the first language you should learn, you can remember that the title of this section is them monster called python. The first language you learn in MIT (Massachusetts Institute of Technology) computer science is python. So, if MIT chooses to use python then there must be deep reasons. After patiently learning all the languages in the sequence that Eric gave, I fell in love with Python. This was due to the fact that you could do a "Hello, World" program in one line with a single statement print "Hello, "World". That is just it, a program that runs, no complication, nothing but just a single statement. In my opinion a superior language must say what it needs to say with as little "extra fluff" as possible, and as you could guess JAVA had the most "extra fluff", it took a couple of statements to print hello world. So, after cruising the seas of programming I decided to settle on mastering python, because it became my favorite.

Another power of python was that it enabled you to program in multiple paradigms. Just wrap your stuff in a file, called a module. You could do procedural C like programming, pure functional programming like LISP, object oriented or class based programming like JAVA, C# or C++. It was a lovely language, even though it was still in its early versions, I think I encountered python 2.2 first but it was still an elegant way to solve computational problems easing the thinking process and shifting the programmer away from the computer and more into the core of computation, although with some issues relating to slow execution.

In these early times programmers mainly used Python as a scripting language for automating tasks on the computer, the same way shell scripting languages were used. But as time went on the language was used for more than that as the performance increased.

Many Python functions for doing mission critical stuff like math were implemented in C, the language with which Python is built, because of the speed of C. A wrapper around the C code enabling you to call it from Python was what enabled such a task.

So, Python is a multitalented friendly monster, not necessarily a large scary monster that JAVA is. I spent so much time with Python till my thinking about computational problems was mostly thinking in python. So why do I call Python a monster? Python is like a slimy monster, much like Blue in the cartoon series Foster's home for imaginary friends. It is capable of so much but just like slime if you are not careful you can slip which in this context means you can write monstrously convoluted code that is hard to understand because of the versatile nature of the language. Most Python programmers agree to certain conventions like naming the first argument of a method Self. These conventions make for an agreeable form of communication between programmers so that too much obscurity in code removed.

My detour to LISP

My main field of interest is artificial intelligence, or more specifically Artificial General Intelligence, what is usually called artificial intelligence is actually a very narrow aspect of intelligence dealing more with pattern recognition or other special features of intelligence rather than what encompasses human intelligence as a whole. The more I investigated into this field, the more I saw opinions that LISP was generally the language to do artificial intelligence with. It was through the work of John Koza in the field of genetic programming, programming computers by means of natural selection, that I realized if I had to comprehend what he was doing I would have to study LISP. But after my brief exposure to COMMON LISP, I was not very impressed with the language because although it was fun to use, keeping track of parentheses was not something I enjoyed doing. Don't get me wrong LISP is a great language and people are using it to do great things. I love the functional nature of the language but we all have our preferences.

I was determined that I was going to find a way to do genetic programming in Python. I was going to search the open source world and if I didn't see a framework that worked for me, I would go ahead to build mine.

I don't like classes

Although many large programs are written in the Class-based object oriented style, it is not my thing to think about computational problems in terms of creating Classes. I think mostly in functions and data when conceiving the solution to a programming language, it is neat and elegant. I noticed that when I was solving a problem in my favorite programming language then, Python, I would naturally tend to start thinking in functions. The only time I had to deal in Classes was when I was forced to read someone's code that had already been done with Classes. But there was a time I had a problem I was solving and when I was designing a solution, I found out that it would be neater to present the solution in Class-based object oriented style. Note I always say Class-based because in a language like python everything is an object, even functions! So, I always include the description Class-based so you don't get confused. We are humans and we have our preferences, but that doesn't mean that if we have to do stuff we don't like we would run away from it. I don't like Classes but when a solution would be neater by using Classes I use them. I think it's wrong to force people to think in any specific programming styles, brains are different, many people find it easier to think in Classes I am not among them.

The thing I dislike most about Classes is not the idea of Classes themselves. Classes form a cool organizational framework for functions and data. I usually call it a bag of functions and data, but they are not really fundamental for computation in my understanding and sometimes just a matter of style. They add more complexity to code than is necessary even though encountering them in programming books using them in toy examples will make you feel that you need it to express solutions to computational problems. The most annoying aspect of Class-based object oriented programming is the massive Inheritance hierarchies that are created as a result of using a paradigm that is more of a philosophy than a fundamental necessity. Inheritance is the most touted benefit of Classes, but when you are faced with mountains of code, and deep inheritance hierarchies it's hard to wrap your head around everything especially for someone like me that loves having a full mental internal representation of any project I have to do.

Not being able to wrap your head around code, even though you have done black belt level project management is a sure way to introduce a lot of bugs into a program. Bugs that are not really as a result of a poorly implemented computation in the actual functions but new bugs that are related to Class management. Bugs that can give you horrible weeks and lost nights trying to catch. If a computation implemented in a function is wrong or slow, you know that you need to optimize it or write a better solution, but if you forgot to call a super-class constructor in a sub-class then you are not facing a computational issue but a project organization issue.

Most people argue that by encapsulation, Class based programs hide complexity at lower levels enabling you to build the abstraction hierarchy much higher to enable you tackle the most complicated of problems. This is not really necessary because even when simulating natural systems, you don't need that high of an abstraction to capture the essence of the system, thus the argument for encapsulation holds no water. When I hear arguments about encapsulation hiding the implementations so that the programmer is focused on the overall structure of the project I wonder if properly named functions and code folding editors are not enough to hide the implementation of functions?

In computation, there are two things that are essential: 1. The data structure to capture the root of the problem. 2. The algorithms that manipulate the data structure to produce the solution. A programming language should make it easy to set up computations, a good package management system like I have said earlier is enough to manage projects of any scale.

Later on, in my programming career I learnt the key to mastering complexity of programming projects especially the ones with millions of lines and deep inheritance hierarchies, you must learn to forget the origins of things and take certain things as a given. If you see that a class subclasses another class, don't go chasing down the rabbit hole all the time to see how that class looks like in the documentation, or you will find out that it inherits from another parent and so on till the real essence of the code you are trying to understand gets muddled in your head. If you see something like:

```
Import java.io.PrintSomething
```

Don't go chasing into the io whatever! Just realize that there should be somewhere in your code where you have a PrintSomething() that prints something! That should be enough for a while. You should realize that your purpose in understanding code, although it might require understanding the depths of any inheritance hierarchy you might be working on, is to have a working knowledge of the code you're reading. The question I always ask myself to access my level of understanding of code is, can I edit this code? If the question is yes, then I know that I have gained some understanding.

This work will not go into the details of any specific object oriented programming language, but I am acting like a prophet here! I am telling you of stuff you will begin to consider in you long sojourn in the world of programming. When you start learning Class-based object oriented programming you will not see how inheritance can make code complicated to understanding and how composition will ease things, but 2 to 3 years down the line and much of what I am saying here will become of much relevance. The take home here is that if you are tasked with designing something that is Class based and you have found out that there is no other way to do the project with functions alone, then opt for more composition and less inheritance. In the long run as the project gets more complicated, you will be happy you knew this.

JAVA, my worst enemy

In the beginning of my programming career when I was still studying I loved JAVA, although from the beginning I was not happy with the fact that I needed more than one line of code to say “Hello, World”, it was still a very cool and highly predictable language. You could expect code to come in certain patterns and my favorite of the time were the getters and setters, the methods you use to get attribute values and set attribute values. So, predictable so expected in many programs I came across.

When I didn’t know much about Python and programming in general, because I studied it from reading the tutorial alone, I found out that I didn’t see what I expected software to look like. You expect that at the end of your programming study, you would be able to design a desktop application like Microsoft word! But what I ended up doing was a lot of was command line programs. When I told my friends, I was a programmer they asked to see some programs I had written, and when I showed them some of my python IDLE (interactive development environment) programs, they were not the least impressed. A further display of the command line and invoking some programs with it, still failed to impress. They wanted to see Microsoft word type of stuff or even Corel draw (The two most popular programs in Nigeria at around 2002), but here I was with no icon to click, typing commands. Although they couldn’t tell me I felt their disappointment and I too was dissatisfied with my inability to pull out good GUI.

I wanted to shift to visual basic which was Microsoft’s main programming language then for creating GUIs but Eric Raymond’s advice that Visual Basic teaches bad programming, kept me off. I didn’t want anything to damage my newly forming programming brain, because I know that a bad seed will affect you for the rest of your life.

Although Python had the native TKinter GUI kit, it was not impressive to me and the documentation was scant and I couldn’t find any good web document at the time teaching it. I forayed into wxPython and it was good but couldn’t get access to open source programs out there written with it so I could learn from. And at the time I couldn’t think of any good desktop application to write because my interest where more like research and discovery, mostly in information retrieval, artificial intelligence etc. and the command line and the interactive development environment was enough for me.

What kept me with JAVA was that there was this neat well-built GUI library, and glancing through some JAVA book I had at the time I could see some lovely GUI application. My goal was to study JAVA well and write some application in it, probably educational so that I could prove to friends and folks that I was now a programmer and people should stop questioning me about why I spent so much time on the computer. But my real interests were more of a research kind of thing and that I knew I couldn't do efficiently because thinking in JAVA is slow, thinking in python is fast.

Another thing that put me off with JAVA is that array stuff. In python if you want to put some items in a container, it could be a list of names or numbers, you just put it in a list. In JAVA, back then you would have to declare an array which looked clumsy to me. Another annoying thing was that I hated static typing, we don't do that in Python, we just say name = "Jackreece" rather than something like String name = "Jackreece"; I'm more into flexibility, efficiency and expediency I want to get on the computer and spend more time thinking of my problem rather than trying to beg the computer to represent my computational thoughts properly. A good programming language should be close to natural language and yet very efficient in execution. Programming is about solving computational problems, the computer should be able to figure out if something is a string, and integer or a floating-point number. The more we allow the computer to figure out routine stuff automatically the more time we spend thinking about the problems we have to solve.

* * *

2. WEB PROGRAMMING

After my foray into JAVA, eventually writing some toy application in the new JAVA swing GUI framework, I was able to impress my folks and it was time to focus more on what I wanted to do. I realized that in order to create some useful service, the world of Desktop applications was not where I wanted to be. I saw something on the horizon, web applications, and because of the success of Google, I knew that the future would require programmers to build applications that presented their interface in a web browser. So, despite being initially fascinated with Desktop GUI I decided it was time to prepare for the future by learning web technologies. The central question in my mind was, how does Google work! If I could learn the stack of technologies that enabled something like Google to work, then I could do some kind of web thing of my own.

Earlier I was exposed to HTML, a friend of mine had a very primitive laptop with a TFT screen and poor battery capacity. He didn't know what to do with it because it didn't even have a CDROM drive so it couldn't be used to watch movies. I saw it as an opportunity to learn HTML, but to convince him to leave it for me at least for a while I had to do something remarkable! And the most impressive thing you could do with HTML to attract the admiration of non-programmers apart from changing the background color of a browser window was <marquee>some text</marquee>. To add more spin to it, you could change font size, still change the background color and do other tricks. After these demonstrations, my friend was convinced that it was better the computer stayed with me.

For weeks, I furiously learnt html by getting websites on a 128mb memory card from the cyber café, and studying the code and trying to replicate what I saw. But my tenure with this computer was short lived as he came back for it because the real owner wanted it so I surrendered it, with my knowledge still intact!

Most of my study done this period was in the cyber cafes and because it was quite expensive what I actually did was offer to help people type their mails, because they were obviously slow in typing and it was tedious for them so I would type quickly, I had learnt touch typing on an old word processor, a full machine dedicated to word processing with a 7-inch screen that I had access to. My "clients" were impressed and usually left their remaining access time for me. This was what financed my day long stay in the cyber cafes studying computer science, web programming and anything with the word computer attached to it. Sometimes I spent nights in the café because there was a flat rate for browsing throughout the night, I was sleeping a mere 3 hours a day.

Apart from HTML, I picked up JavaScript and downloaded Perl, that text processing language that did the role of PHP back then. Perl was also on the list of languages I got from Eric's website so it was only natural that I learnt it. So, I did a lot of local web page stuff and then had a project brought by a friend. The project was to design a small website to showcase some tourist sites in my state, cross river state in Nigeria. Although I didn't know all that was needed, I believed I would solve whatever problems I faced along the way and by the end of the project I would be a web designer. So, after some research I found out that I needed a WYSIWYG (what you see is what you get) tool, this tool would simplify the process of generating the needed html for the website and then I would need JavaScript for controlling the interface, and some CGI gateway stuff for form processing and all.

The tool I had access to was Microsoft FrontPage, I heard of Dreamweaver but couldn't get access to one so I settled for FrontPage, I think at the time it came bundled in Office. Then we used the Corel Suite for the graphics and with these and some hard tinkering we had a website fully functioning on the local computer! My job was just to deliver this and it was my partner's task to find out how to host it etc. so I was done with that project and learnt a lot along the way.

Now to my basic question! How does google work? I lead my life by answering such grand question, for instance my current question is: how does human intelligence work? This is not some HowStuffWorks.com kind of questioning but a kind of questioning that guides me on a search for information, knowledge, wisdom and how I will manage my time. My life is currently governed by such grand questions and I do everything in my capacity till I am able to answer them satisfactorily. This answers can usually come out in some kind of blogpost or book but the google question had to be answered in both words and code.

I read Larry Page and Sergei Brin, 1998 PhD thesis paper and even though I didn't understand all of it, I took home some points! Google was built on the LINUX operating system, Red hat to be specific. Many parts were written in C, and of course the web stuff was done in HTML, JavaScript etc. although I didn't have access to the full architecture, I had to fill in the blanks. The most important question is how to you write an application that does some backend work and delivers results every time it is queried. This is a bit different from the static website idea, this was a dynamic website what we usually call web applications.

Backend and Frontend

A web application usually consists of a backend and a frontend connected together to deliver some service to the user. The web page involved the front-end coding stuff, while the backend constituted code that managed the entire system, plus code that implemented the algorithm.

In summary, you had the base of the system, the LINUX operating system, a bunch of C code for implementing the algorithm, databases and other system specific stuff like filesystem management etc. and you know that to run an application like google you needed more than one computer working together so you had a bunch of computers that had to be connected to function as one big computer to deliver such a big service. In those days, there were no cloud computers and you had to build your infrastructure yourself. Although the frontend stuff is resident on the google servers, they are what format the results on your screen and provide interactions.

Analogies with Desktop applications

In Desktop applications, you have a similar structure as to what you have in web programming. There is the "Business code" running in the background and there is the interface code providing you interaction. Although this separation is hidden from the user, this is how the code for Desktop GUIs are designed. In Web application programming, while the backend runs on the service provider's computer, the GUI is delivered via a web browser on your own computer. It's like the backend is your power company and the front end is your light bulb or your television.
As I went deeper into the study of web applications, I discovered that most of the complication of designing a dynamic website (web application) could be eased out by using a web application framework. I have mostly focused on Google but another famous web application existing at the time was Yahoo mail and of course Hotmail. I went ahead to find a web application framework that I could learn, it had to be something written in Python. I found ZOPE application framework. When I discovered

it then it was in Zope 2. I proceeded to learn it. Nowadays you have Django and Ruby on Rails, and also the PHP with MySQL system for doing web applications.

Moving data around

In the early days, you would pass data around as a text file or for certain applications as a binary file. As time went on the XML (Extensible Markup Language) emerged as an ideal way to pass data between applications. So you usually had to mark data up with XML and pass it on to its destination. At its destination an XML parser would then extract the information for the destination application. Nowadays JSON has overtaken XML as the default format for passing data between applications.

✻ ✻ ✻

3. GAME PROGRAMMING

The most popular activity people perform with their mobile devices these days is playing games! People spend more time on games than on making calls or even using popular social media sites. As a result, you can imagine that the demand for new games are almost insatiable.

Many people are getting into programming for the sole purpose of creating games and I think it will be a good thing to give them an idea of the scope and scale of game programming.

The number one platform for games these days is the mobile phone mostly Android and IOS. On android, most applications are written in Java and more recently google announced that Kotlin will be the official language for android so if you plan on doing games on Android you should arm yourself with these languages. With Xamarin you can write cross platform games in C# and as I have mentioned in some other area of this work it's also a language you would have to learn if you want to write apps once and run them on windows, android and IOS. Of course, we know that IOS also has its native languages which are Swift and ObjectiveC, if you plan on developing games for that platform alone then you should use the native languages for higher efficiency.

Apart from mobile we know that we also have PC games and Console games. By PC games I refer to games that are written for the windows operating system. There are also games for Mac and Linux and of course most PC games are ported to these platforms. There are console games like Sony PlayStation and Nintendo Wii, etc. on which many games are developed for.

A console is specialized computer for running specific applications, in this case games. Your desktop computer (PC) is a general computer for running all kinds of applications, including games. A console is optimized to run games alone. The need for consoles arose in the early days of gaming, although there was some commercial benefit to being able to sell specialized computers to people in the form of game consoles, the main reason consoles arose was for the sake of performance. In order to pack more specialized hardware optimized for game play like better graphics processor, more memory, etc., computer engineers decided to design game consoles.

In the early days of game production, a single programmer would be tasked with doing everything required to bring the game from conception to completion but as games became more complex and computers improved more people became involved in the game production pipeline. Below is a list of the kinds of people that work on a game project and a brief summary of some of the task they will be involved in.

1. Graphics designer: These individuals work on creating graphics assets that will be required in the game. They are involved in the creation of characters and landscapes or whatever graphics the game might require. These people do not write the programs that make the game possible, they are mostly artists and work with software tools like Blender, Photoshop, Gimp, Maya, etc.
2. Graphics programmer: These individuals are usually the people that specify how the graphics transformations will occur during the game play. They usually have an intimate understanding of the underlying graphics processor and are charged with creating software routines for controlling it. They are also versed in math because the graphics processor merely does matrix operations, so a strong knowledge of linear algebra and other disciplines of mathematics are required and also a strong computer science background.
3. Physics programmer: Most games contain natural environments and thus the physics programmers is tasked with writing code that simulate stuff like gravity, viscosity of water and even fire, etc. Some games contain unnatural environments that possess laws that differ from that which we find in nature. It is also the task of the physics programmer to make sure that these laws work in a coherent manner throughout the game play.
4. AI programmer: Many games contain a good deal of artificial intelligence; the enemies you are trying to attack in first person shooter games are usually controlled by some AI that is usually done by the AI programmer of the project. Most modern games even have some form of learning where rather than hard coded intelligence, characters in the game can also learn from your actions and their environment and thus adjust their behavior to optimize their actions in ways that were not hard coded by the programmers.
5. Game engine programmer: In the early days, when writing your game, you will have to write your own game engine for yourself. A game engine is a kind

of middleware software that enables you to focus on a high level description of the game using scripting languages rather than managing the mechanics of how the game is executed on the computer processor. We can remember that programming is always about building abstractions. If we have some large scale goal we have to attack, rather than trying to build it out very close to the machine, we build a tower of abstraction above the machine so that at the highest possible level we are able to do a lot of basic stuff with simple high level commands. Game engines are a layer of abstraction that enable the game programmer focus on a higher level view of the gameplay rather than managing hardware.

6. Network programmer: most games these days are also multiplayer games that sometimes have the different players located over a network such as a LAN or even over the internet. The role of the network programmer is to solve low level networking problems like latency issues and how to route packets around a network to ensure smooth gameplay among all players engaged in a multiplayer system. This is one of the most difficult task in the entire game development process and thus needs specialized skill and attention.
7. UI programmer: This person is tasked with creating the UI that enables you navigate the game. Anything that has to do with stuff like selecting characters and seeing how much ammo you have during game play or other task that require viewing or interacting with the software which is not actually directly related to controlling the actions of game assets is usually the work of the UI coder. Most UI are in 2D but sometimes it's done in 3D.

In the very early days most games were written in Assembler! And as you can imagine that was a horrendous task all by itself. Most modern console games are written in C/C++ because of the need for efficiency in running a graphics intensive application like a game. Most 2D mobile games like I have said earlier are written in Java/C# or even JavaScript.

Scripting languages like Python and Lua are also used in gameplay. These scripting languages are used to perform high level tasks like creating the character storyline etc. They are not used for the performance sensitive aspects of the game.

Unity is a game engine that is used for creating games using C# and other scripting languages. If you are new to game programming and want to start with a simple easy to use environment I would advise that you learn Unity. It is a box of tools that will ease your workflow. You must note that Unity cannot be used to create assets like characters and environments, these must be created using other tools like Blender, etc. It must also be noted that Blender has its own game engine so if you are more inclined towards open source like I am you should look into the entire Blender stack.

* * *

4. WHY LINUX?

Before I even found out that Google used Linux, it was clear from Eric's page that I had to learn Linux and he gave an example that learning to program with windows was like dancing with boots, I hope I got that statement right but it sounded something like that. But so far I had not found the need to learn Linux. Most of what I did required only Windows as a platform, I worked mostly in python and barely needed to play around in my operating system. But as the realities of creating my own web application at a scale such as Google dawned on me, and with the absence of cloud computing in those days, I started spending more time with Linux.

The founders of google had little money and couldn't definitely use Microsoft Windows servers for their young product because of the prohibitively high cost of their licenses and the endless limitations within the OS plus the fact that its closed source and thus has limited customizability. They had to use something that was Opensource, something they could have access to the source code so they could perform endless optimizations with it. Although the base product was Linux Redhat, they must have customized it completely to something that was much more suitable for their product. This would have been unthinkable to do with Windows server software because you don't have access to the source code and you are limited by the license on how many processors you can run it on etc. so you get my point.

Linux is basically a kernel and a shell. In modern Linux distributions or distros as they are popularly called, there is usually some form of a desktop environment, but the power of the Linux system is having powerful access to the shell or what is called the command line in windows. So you issue commands to the kernel via the shell and it returns results. This is what you basically do with a computer operating system. The shell can also be scripted, meaning you can write small or large programs that do a lot of shell activity for you automatically. With this scripting ability and full access to the kernel, you can turn your computer to any beast of your imagination. If this powerful scripting environment is not enough for you, you can take things further by extending the Linux operating system in any way you deem fit using system programming. The

Linux operating system is fully programmable and there is an API that enables you access the system internals programmatically via a language like C. With the shell scripting languages, you can automate some tasks but of course - but with C and the Linux API you can do very powerful things.

If this power is still not enough, you can look into the freely available Linux source code and edit it, recompile it to perform any magic that you deem fit. So in summary, the LINUX operating system gives you full access to your entire hardware and enables you build computer systems in a way that suits your need.

So what is the place of Windows?

It might appear that I have been Windows bashing all along but that is not the case. I remember trying out Redhat Linux version 6.0 and the primitive desktop environment almost drove me mad. My first encounter with a computer was with Windows 98, and around that time there were systems that were still running the Windows 95. There was a computer school in my area that offered me unlimited access to their systems if I could help them teach their students MSDOS. Although they had no internet access it was a good place to sharpen my knowledge of computers. In reality I have always been surrounded by windows, although I will soon have a dedicated Linux computer soon, I currently have 2 laptops running windows as their primary operating system. I use a virtualization software to run some Linux distros but so far I only go there when I have to do some Neural network with Google Tensorflow.

Windows brought computing to the masses and we cannot underestimate the difficulty of that task, we have to appreciate them for that. Windows server software is used by large corporations, the ones with suit and tie "IT" masters - you know the kind of company that has some huge software application they purchased from another big company like them that do all those big things. To these companies' money is not an issue and purchasing a $3000 license will not give them a headache. For the young startup trying to put together a product with nothing more than some small personal funding, learning Linux Kung Fu is inevitable.

Azure and Windows 10

With the advent of Windows 10 Microsoft is a changed company in my opinion. Although I have been a Microsoft hater in the past, I love Windows 10 and have since repented from my ways. I am even considering studying C# seriously, Microsoft's answer to the Java programming language. My love for Microsoft was activated by my first use of Windows 10, of course it was a free upgrade which to me was the best thing the company had ever done. Azure which is Microsoft's cloud platform is also a good thing for the company and for developers, you can now develop applications on any scale and launch them on the cloud without the need to build your own infrastructure. Back then in the old days, where girls and guys still built their infrastructure by themselves, I usually joked around that I would one day run a company from my laptop. With the cloud infrastructure like Microsoft Azure, Amazon AWS, Wolfram Cloud, Alibaba, etc. You can get applications up and running on another person's well managed computer aka the cloud. This is another level of abstraction that makes the task of writing software easier. It enables the programmer focus more on the problem they want to solve and less on the infrastructure stuff like power management, cooling, disk failure management etc. With the cloud you just pay for these stuff and go ahead to develop whatever it is you are developing.

C# is also a very neat language and with a powerful IDE like Microsoft Visual Studio, you have a very powerful environment for developing all kinds of application. Also it is my opinion that Microsoft makes one of the best user interfaces, and with C# and Visual Studio you can create very neat and powerful Desktop applications. Also Microsoft is making great effort towards integrating python and this is good news because python lovers can work on Visual Studio and with some work launch python applications on the Azure cloud.

Apple

Mac OS is a great product. I cannot say much about it because I have not owned a Mac before. I have experienced it by using other people's computers. The interface is lovely and I think it is the best environment to do any kind of visual artistic work in. I know a lot of coders code on Mac, but I have never imagined myself coding on Mac. If Mac and IOS is your destination, then learning Objective C and Swift are the languages you need to learn. I don't see myself venturing into the world of Apple soon, but as a fashion statement, I would eventually spoil myself with some Mac products, maybe an IPhone and a good Mac Book. But I don't see myself doing any serious programming on that platform.

UNIX vs. LINUX

Unix is the ancestor of Linux, the whole history of Unix is available online, what I am providing here is a perspective. From the commands alone it's not always clear what the difference is between Unix and Linux except in some details. If you know Unix, you can use Linux and vice versa. The reason why you hear many people talk about

Linux these days is that Linux has had a very friendly license right from the start, while Unix flavors sometime come with some licensing issues. Both operating systems are based on similar philosophies. A simple mindset to use when thinking about these operating systems is that Unix is old and Linux is new. That is all, you can work healthily in any of them as long as you know one. The thing is more open source effort has been put into Linux and that is why it is so happily recommended as the programmers' operating system. Mac OS from Apple is actually based on a solid Unix kernel, which is a plus for it. Indeed, the only thing that excites me about MacOS apart from the fancy well designed User Interface, is the fact that there is BASH shell available. The BASH (Bourne Again Shell) is my favorite shell environment on Linux so you could guess that I would fall in love anywhere I see it.

✻ ✻ ✻

5. THAT DRAGON CALLED C

C is ridiculously powerful! Well that is an understatement, let me say it again in all caps: C IS RIDICULOUSLY POWERFUL!! With great power comes great responsibility. Improperly handle C and you will end up shooting yourself in the foot! When working in C you are as close to the computer as necessary to do anything using the direct capacities of the system. Such power can be very dangerous and you have to do everything from memory allocation yourself. A typical Python programmer or worse, a JavaScript programmer who has no experience in C has no idea what memory allocation is! These things come for free the higher up the abstraction ladder you go. When you are in the pits, standing next to the fire spitting dragon called C, you have to be careful about everything you do. Another way to get burnt with C is that mechanism called pointers! Pointer mismanagement has ended up creating more distressed programmers than any other individual bug in the world.

In C, you can even have inline assembly code as nitrous to boost your speed, if the speed of the already fast C language is not good enough for you or if you have intentions of doing some other stuff.

So what is C good for? First of all, it's very fast! So if you're doing some calculations, especially with numbers and you want it sped up, it's a good idea to implement it in C. Since C is very memory economical, we find C used in embedded systems, like your door alarm, microwave etc. These kinds of systems are constrained to have very tiny memories so you need a very tight language like C to work efficiently in these systems. Some girls will just go ahead and do everything in assembler. If you one of those kinds, you're blessed but if you're like the rest of us, then settling for C will be the wiser choice.
Most operating systems kernels like the Linux OS is written in C with sprinklings of assembler here and there. C is also the language that most other high level languages

are written in. Python is written in C, Java in C etc. When I say written in C I don’t mean that everything in it was written in C. Most of the performance sensitive aspects or what we will describe as the core of the systems was written in C, while most parts could be written in the language you are designing or some other assistive language. Python has some parts that are written in Python - this is because what is called the parser of the python language was written in C so therefore once you have a parser you have a language, which you can now use to write other aspects of the language. This is actually more complicated than this, but if language/compiler design is what ticks you then you should immerse yourself more in this topic.

There are Python packages like SciPY and NumPY where you will see that “performance sensitive” portions are written in C. This is usually how C is used - if you’re the type of programmer that just wants to use libraries in python to develop a web application then you might never encounter or need C in your life. But if system programming calls unto you, then mastering C is very important.

✻ ✻ ✻

6. A BULLDOZER CALLED C++

As if the difficulty of C was not enough, someone called Bjarne Stroustrup, a Danish Computer scientist went ahead to add Classes to C giving birth to C++. Why do I call C++ a bulldozer? It's because, if you have the skill with it you can bulldoze any problem, but just like a bulldozer, you don't just go to regular driving school to learn how to use such a powerful machine, you must undergo serious training. After you have been tempered by the dragon fires of C, which will build your discipline and knowledge of how the computer works for the sake of the computer, you are ready to take on the big bulldozer C++. The thing is, why not use python? If you want to solve a computational problem fast then you should think Python and above, the above I will come to later. But if you are interested in solving a problem and you want to control how the computer solves that problem, without much assistance then you can opt to use languages like C/C++. In many cases these is not necessary except you are tasked with doing stuff like writing device drivers, middleware or some other stuff whose purpose is more of controlling the computer and its resources rather than solving a human computational problem. The keynote in your mind should be "performance sensitive" and if you have not tried out the solution to the problem in a higher level language than C/C++ you shouldn't hastily declare the problem as performance sensitive. First of all, solve it in a higher language. Most of the times there are packages that are already written in this higher languages that solve your problem more efficiently using the low level ones when needed. If you find out after adequate performance tests that your solution still needs a faster implementation that cannot be obtained from a better computer that is not much more expensive, or faster algorithms in the language you used then you can go to C/C++. These languages are black belt stuff and should not be the weapon of first resort.

If accidentally you were trained in C/C++ as your first language, just like me and that was due to the fact that it was the only programming book available, then you might be lucky but still heed advice and go for a higher language. Even Java is better, except you really know what you're doing.

As a beginning programmer except you know clearly that you will be designing systems that use these languages, C/C++, learn them then stay off! Do your work in some easier language for the time being, you will end up producing more bugs than solutions in your programs if you do not know what you're doing.

✻ ✻ ✻

7. COMPUTATIONAL MATHEMATICS

These days there are a lot of coding camps promising to teach coding in a few weeks. But I always remember a statement by Peter Norvig once the director of search at google, it takes 10 years to learn programming! This might seem discouraging as the great Peter even disregards a Computer Science graduate trained for 4 years as not yet a programmer. As discouraging as it may sound, it is actually true. The subject computer science should in reality be changed to computer arts because at the practice of it computer programming is more of an art and less of a science.

You learn to code in weeks, meaning that you will know the syntax of a language, but that is just a minor but important step, there are mountains upon mountains of convention which you will have to learn, which is why programming is more of an art.

The great master of computer algorithms Donald Knuth made everything clear when he named his great algorithms bible: The art of computer programming. Truly programming is an art but what is art? In the usual understanding when you think of art you think of lovely pictures, and all forms of expressions that express beauty, but this is not the sense that the word art is used here. Although there is an aspect of beauty in code, especially beautifully expressed code. The word art here is used to express skill rather than the results. In fine art, the painter is faced with canvas, paint and brushes; in music the piano or whatever music instruments; in dancing your body. These tools are to serve as means of expression of inspiration. In the art world the inspiration is towards beauty and pleasantness while in the world of programming we usually have a problem we want to solve and go ahead to express with our tools of a programming language and a computer, mostly it's just about the programming language.

Programming is an art form akin to writing and less like images, sounds, etc. When you write you are doing art, and the purpose of that writing is to solve some problem. We write to express the solutions to our own problems, which might also be the solutions to the problem of the reader. In programming we write code to solve our problems, with a computer as the means of expression and with other programmers as our audience. Although the output of programs is usually consumed by those we call users, they are usually not participants in the full art form of programming. Programming is a unique art with multiple audience obtaining different benefits. Other programmers reading your code are users as well as the non-programmers who will fire up the application you have created.

Sometime just like a piece of writing like a poem that we don't intend to share, programming can be a very personal thing. But seldom do we write code for the sake of just appreciating the beauty of code. We use code as a sophisticated language to communicate our ideas about the solutions to computational problems.

In the early days of computing, when it was all about making computing machinery better, the word computer science was coined to represent the activity of people who were working on the computer. Their methods were similar to that which we find in traditional science which involves hypothesis, experimentation, theory and conclusions. Just like the biologist studies biological organisms, the computer scientist studies computers. It is imperative that the computer scientist studies programming which is indeed the major way to communicate with a computer but much of her activity is usually related to the way the computer works itself rather than how user applications are written. This is why Peter Norvig says it takes 10 years to learn programming, not that it takes that long to learn the syntax but that to start creating reasonable programming art forms it might take 10 years of practice and mastery. He was talking about mastery here not creating an ecommerce site or some basic desktop application. So a computer science graduate has tasted computer programming and can write some programs, but he has not yet swum the waters long enough to call himself a master.

Another reason is that computer science doesn't just require a knowledge of programming. There is the hardware, algorithm analysis and the topic of this section computational mathematics to master. So programming is not her primary activity for 4 years, but I think dedicating 4 years into just programming languages will make you proficient in programming but not the entire field of computer science which is vast.
When I first got access to the Google founders PhD thesis, I was expecting to encounter some code that describes how the then famous page rank worked, but to my utter disappointment I encountered mathematics. Due to the fact that I did not have a solid

math background I was dismayed. I had previously thought that an algorithm to run on a computer was going to be in code, but here it was written in some funny looking notation, which later I found out to be the summation sign and some other expressions. At that point I decided to study math, but what kind of math I needed to know to be able to understand such expressions I did not know, so I began to research.

Back then for some weird reason I couldn't find a calculus book for free online for a while and I couldn't afford one at the time so I temporarily stopped my search. A while later a friend gave me a calculus book which I studied but I still didn't find the mysterious summation sign. After sometime I actually realized that the math I was seeking was called Discrete math and it was the primary mathematics for computer science, and indeed programming. If I had known something like MIT OCW at the time I would have been saved of much stress and just seen the Math for Computer course being offered there.

The importance of computational mathematics is that algorithms are not always written as pseudocode. They are sometimes written as mathematical expressions using mathematical symbols like Summation and Product. Some books on algorithm analysis are actually written in mathematical notation. So while a coding school can teach you coding and many people will argue about coding being the only thing you actually need to start a career in tech and programming, I will say that if you ever want to grow in tech and programming then you must be able to read math expressions which you will find scattered all over the computer science literature. There is no way out, you must learn discrete mathematics which I call computational mathematics.

Computational mathematics is not just about summation and products; it is a bunch of mathematical techniques which enable us to tackle computational problems without pseudocode. Why not just use pseudocode? This is because computational mathematics is a more general representation and hence more elegant and direct in expressing solutions to problems. It is so important that the master of algorithms Donald Knuth in collaboration with some others produced a book titled Concrete Math that gives us the skill to tackle concrete mathematical problems without a computer. Although after expressing the solution in some mathematical form, coding it on a computer to automate the execution is quite straightforward. This book concrete mathematics is actually an expanded form of his initial sojourn into the topic of computational math which is the mathematical preliminaries section of his book: art of computer programming. The beauty of that book apart from the mathematical preliminaries is the fact that algorithms are not written in pseudocode but in some kind assembly code called MIX and later MMIX for the 64-bit version of the architecture. It is actually an instruction set for a hypothetical computer created for that book. But there are now emulators for the architectures. He wanted to make his book as broad as possible and avoid limiting it to any pseudocode structures that might be hot today and fall out of favor tomorrow. For a man with such experience, you must understand how important it is that he didn't do such a fundamental book in pseudocode.

With an understanding of computational mathematics, you will be able to understand and write algorithms using mathematical notation and you will be able to do algorithm analysis with Big O notation.

A very useful computational mathematics course I would recommend is MIT's free 6.042 course, mathematics for computer science. On the MIT OCW website you will find it, videos and a book all for free. It will give you wonderful insight into the importance of mathematics in computer science.

* * *

8. COMPUTATIONAL SYSTEMS

At the base of all programming activities is the core concept of computation, so what is computation? We have actually provided some definition for computation in the beginning of this work but this section will act as a refresher introducing another angle to enhance your understanding at a most fundamental level. The new angle is: Computation is the transformation of information.

The simplest computation we can think of is addition. When we pick out two numbers to add, these numbers are units of information. We want to transform them by applying an operation in this case addition. This is computation. A computer is merely a device that automates away the process computation, which is transforming data units. In a simple transformation like addition of 2 and 3, we transform two information units to another information unit which in this case is 5. We can represent this transformation symbolically with an expression like 2 + 3 = 5.

The computer is a device that knows only 1s and 0s, at the core of the operation of a computer is electronic circuitry that know only the two states, the presence of a voltage and the absence, both symbolically represented by 1 and 0 respectively.

This is all we ever deal with in computation, we build a big stack of abstraction over these basic forms of operation. The logic gates in your computer operate by manipulating voltage levels, that's all they do. In reality, there is no 1 and 0, these are just symbolic representations that are suitable for us humans to operate with. The computer knows only current and no current.

The field of computational systems is not really about building user-centric software applications, but rather it is interested in exploring the basic properties of computation itself. In the study of computational systems, we build abstract models idealizing a system of computation so we can run experiments to discover properties of these systems.

So why study computational system? What good can come out of it? As a programmer your main interest is usually creating software to solve computational tasks so talking about abstract computational systems in this kind of work might look out of place. But in my experience, I think every programmer should gain some exposure to abstract computation, even if they do not intend to work in this field. What good can it be then?

1. Questions about computability: That is what is computable can only be answered properly by a study of abstract computational systems.
2. General features of computation: What can we hope to expect from any kind of program? Practical programs like your word processor and browser software are incredibly complicated systems which we cannot hope analyze in an exhaustive manner, so we build abstract computational systems with a moderate set of features to enable us exhaustively analyze computation to aid us in understanding our practical programs. Usually the software that does realistic things for us in the real world is usually highly unpredictable in its operation. Even if we design software so carefully there will always unpredictability in many possible usage scenarios and of course the notorious issue of computer bugs which no matter how we try to eliminate them still find a way to seep into our code. This is because we cannot explore all possible paths of software execution and thus it will be very hard to say absolutely anything about the programs we write beyond a certain level of confidence. So, to study the deeper behavior of programs we setup simple arbitrary computational systems of whose behavior we can study and analyze completely with sufficient effort.
3. Computational raw materials: Abstract computational systems like cellular automata can be a source of computational raw materials, these are features of these systems that can most of the times be directly harnessed by a programmer to solve a problem for which explicit coding would not work. A definite example of this is randomness. It is very hard to write programs that generate robust randomness but a simple program like the Rule 30 cellular automaton just gives it to us for free. There are even computational systems that can serve as cryptosystems for doing cryptography which have been proven to be sometimes more powerful than human engineered cryptosystems. By exploring computational systems like cellular automata, we can actually mine solutions out of what is called the computational universe by just using exhaustive systematic search. Previously the study of abstract computational systems was black belt programmer stuff but with a

book like Stephen Wolfram's A New Kind of Science even kids can now start exploring the computational universe. We can find all kinds of things in the computational universe beyond structures that can be solutions to programming problems, we can also find art! If you are into computer based art, then you can step your designs up a bit by exploring computational systems where many 2D cellular automata provide high quality designs for free. Even musicians can get sound patterns that could be used as inspiration for creating what Stephen Wolframs calls A New Kind of Music currently being championed by the Wolfram Tones project.

4. Master computation itself not just the computer: The computer is just a tool for physically expressing computation, but computation itself can be substrate free. Many programming books will not reveal this truth, they will either ignore it because the authors are not aware of it or you may find it in very advanced books on computing, the ones with the titles that do not inspire you to open it. The truth is that the computer is just a tool for expressing computation and therefore the computer has properties that are independent from the properties of computation itself. Abstract computation enables you to use the computer and a programming language, which are generalized tools of expression to study computation itself. Deep understanding of computation, itself can give you an edge as a programmer, it will enable you think in a powerful way about the software you want to design and how best to go about it. It's like weight lifting, you lift dumbbells and barbells which doesn't seem like a reasonable task but the strength you gain transfers to other fields of endeavor in your life that require physical strength and endurance.

Computational complexity

There are several ways of understanding complexity in computing, but the word usually connotes how much resources are needed to perform a certain type of computation. These resources are usually either space resources: that is how much memory is needed to do a computation. Or time resources: how long does it take for the program produce the desired output. When studying computational complexity in this sense, the most favored measure of complexity is actually the time element. We want to know how much time a program will take given infinite memory, but of course we know that memory is not infinite. The time measure has been found to be very convenient in the practical evaluation of program performance.

Another way of looking at complexity championed by Stephen Wolfram is observing the evolution of computational systems and judging how much structure they contain.

With this measure, we can say that a highly complex system is one rich in localized structures, and a less complex system that is a simple system contains less structure and can be highly repetitive.

Another alternative way of judging complexity which I find a lot of fun is to measure the compressibility of the output of a system. If the output is highly compressible then It is a simple system, and if it is difficult to compress the output then the system producing it must be doing something complicated.

There are notions of complexity like Kolmogorov complexity and I recommend that you do some research on it if you find this topic interesting.

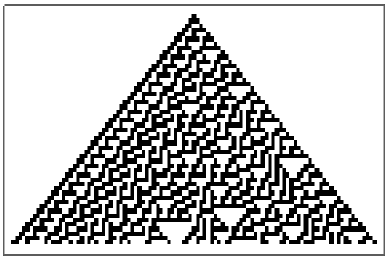
Varieties of computational systems

There are many computational systems out there but without having to go to the wild you can simply look at the ones found in A New Kind of Science which is a thorough and concise treatment of computation and computational systems. The field of computational systems is vast but A New Kind of Science is a powerful entry point that will ease your journey. Below are some of the computational systems treated in the book.

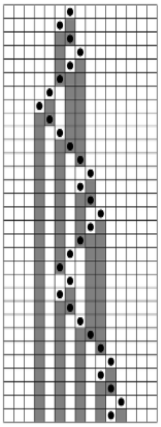
1. Cellular Automata evolution

```
In[1]:= ArrayPlot[CellularAutomaton[30, {{1}, 0}, 50]]
```

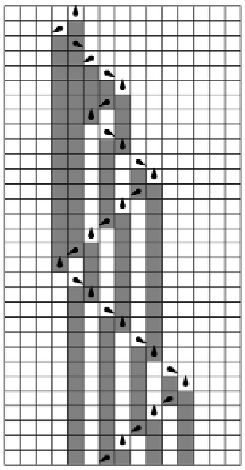
Out[1]=



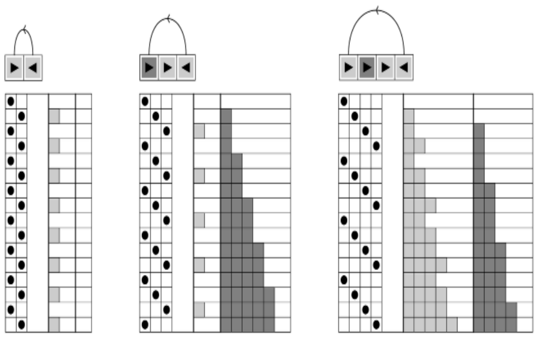
2. Mobile Automata evolution



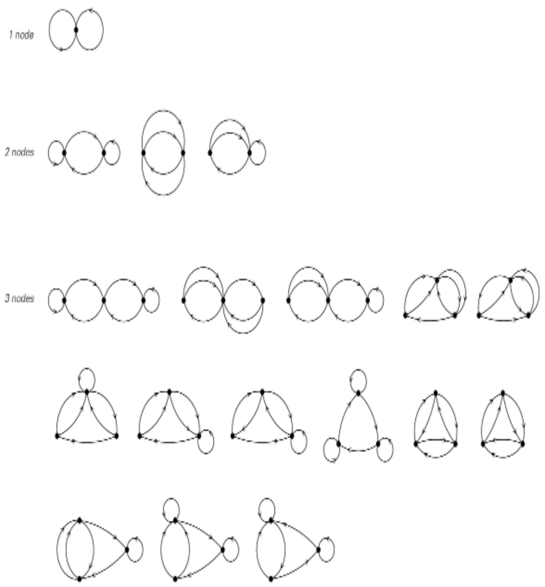
3. Turing machine evolution



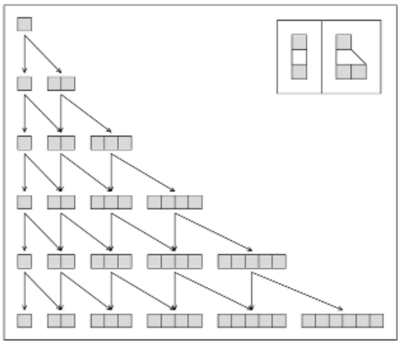
4. Register machine evolution



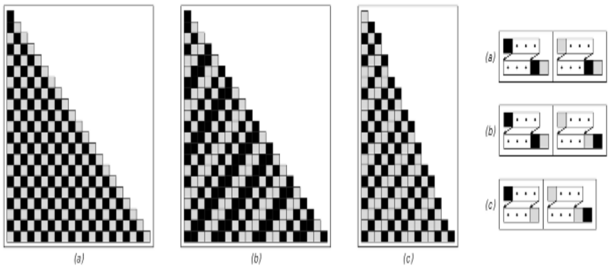
5. Network System evolution



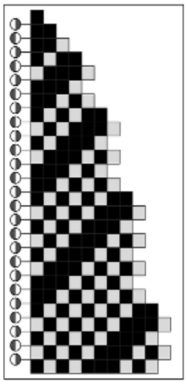
6. Multiway System evolution



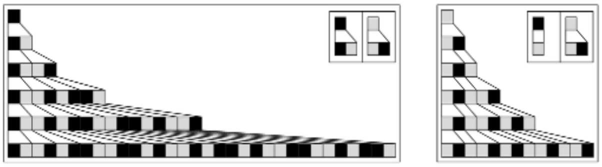
7. Tag System evolution



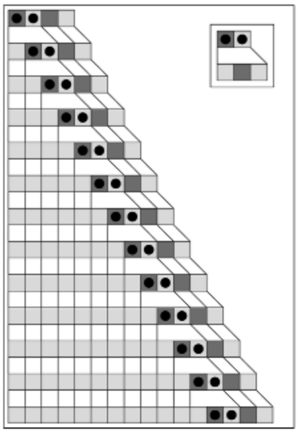
8. Cyclic Tag System evolution



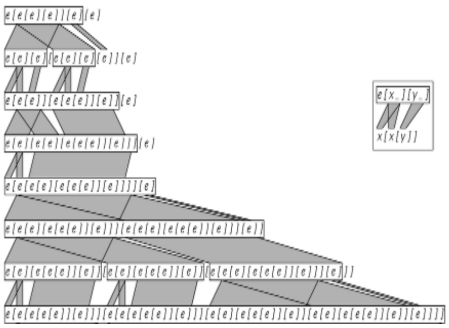
9. Substitution System evolution



10. Sequential Substitution System evolution



11. Symbolic System evolution



The world of computational systems is very broad and requires some effort to be familiar with, except you're a computer science grad student, you will not be exposed to much of these kinds of systems in your undergraduate course. This is why Stephen Wolfram's NKS is a very good place to start for the casual programmer who just wants to dip in without spending so much time and effort researching this field. Even if you want to go deep into computational systems, I would advise that you start with NKS book because it's very deep.

Of all the computational systems outlined in the book, my favorite is Cellular Automata, this is because the output from these class of programs are much richer in structures and the threshold for complexity, that is how far you have to go before you find a complex CA is very low. Also, Cellular Automata has had a wide range of application from Algorithms research to Image processing, etc. the list goes on and on. With some research into the field of complex systems you will discover many more applications.

In the NKS book there is this simple program called rule 30 Cellular automata which happens to be a very good random number generator that was even used at one time as the random number generator in the Mathematica system. Rule 30 has undergone some of the most extensive test of randomness to prove its nature as a true random number generator.

Apart from setting up the rule 30, of which there are 256 elementary cellular automata, there is nothing much involved in "engineering" the rule to make it a random number generator. It just happens that almost by accident this is a very good and robust random number generator. As a simple program encoding the Rule evolves, the output it produces is quite random, and we can simply sample this output by maybe taking a single column of evolution. This kind of discovery of programs that do something useful is described as mining the computational universe. There are rules that are useful cryptography and there is even a rule that has been proven to be a universal computer. By universal computer I mean that it can compute whatever any other computer can compute. Although it took a while and a lot of effort to prove this fact it was eventually done. So imagine designing a computer based on the rule 110 cellular automata! That will be some interesting line of research.

Apart from this very specific uses of computational systems, the study of computational systems especially by the beginning programmer will introduce you to concepts that only Grad students contemplate. Approaching the field from the wild is very difficult without a lot of experience, but Stephen Wolfram simplifies it very much. The NKS book is divided in two parts, the main part and the notes section, which is longer than the main section. The main part deals with the core of the matter, while the note section deals with the vast details.

I have read the book 3 times and still not penetrated its depths totally so don't think that because it's a simplification of the entire field of computational systems it must be really simple, the book contains 1280 pages of some of the most succinct writing you will encounter.

Although the main issue discussed in the book is fundamental science, there are portions that I recommend for programmers who don't want to delve into science. Below I will list out sections and speak briefly of them so you get a general idea of why I think you should look into these sections even though you might not want to delve into the entire book. The entire book is available online at www.wolframscience.com

Chapter 1: The foundations of a New Kind of Science

You don't want to delve into the study of computational systems without first seeing what it's all about. This chapter will help you achieve that. It first outlines the basic ideas and the relation of NKS to other areas. Then it talks about some past initiatives and ends with the personal story of the science contained in the book.

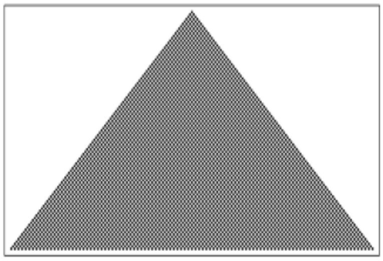
Chapter 2: How do simple programs behave?

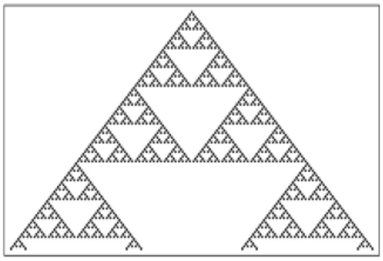
This chapter is also very essential for building your foundation for understanding simple programs. I highly recommend since it is crucial to understanding the rest of the book. It begins with a powerful overview of how simple programs (computational systems) behave. This is the most important section for you the programmer but there is nothing wrong with reading the last section titled: a need for a new intuition.

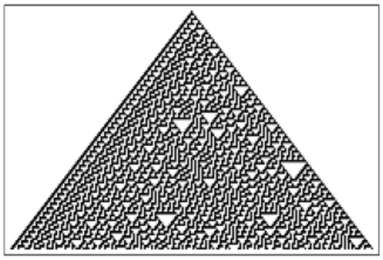
Chapter 3: The world of simple programs

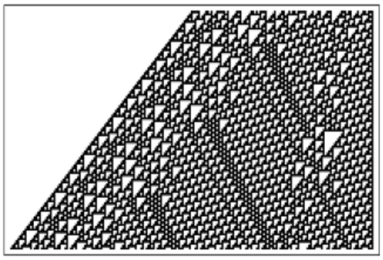
This is the last foundational chapter after which the books delves into more specific topics. Reading it will be very valuable for you. The first section: The search for general features is the most important for a programmer because it shows the four classes of behavior that can ever be possible:

These four classes are so fundamental to not only things in computer programming but in other fields of technical application. No matter what a system is doing be it a computer program or a human mind or the cell of a paramecium, it can be classified as functioning like one of the above system. You can have either:


repetition (rule 250)


nesting (rule 90)


randomness (rule 30)


localized structures (rule 110)

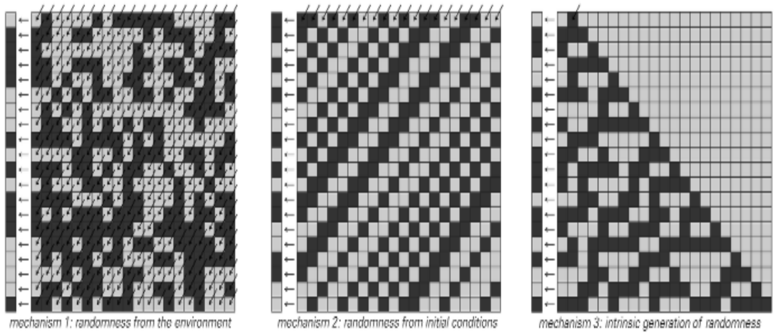
i. Repetition (rule 250): Systems showing highly regular or repeated behavior, doing the same thing over and over again.
ii. Nesting (rule 90): Systems where the whole is contained in the parts, which is nesting or self-similarity.
iii. Randomness (rule 30): Like random noise or the randomness of a random number generator for a computer.
iv. Localized structure (rule 110): Systems that appear random for most of the time but show definite signs of localized structure. As an interesting point rule 110 is a universal computer.

The rest of the sections in this chapter talk of other kinds of computational systems apart from cellular automata. If at this point your juices have been stimulated, then you should go ahead and study the other kinds of systems that are enumerated here.

Chapter 7: Mechanisms in Programs and Nature

After going through the first three chapters linearly we skip and move to chapter 7, not because there aren't any nuggets of wisdom for the computer programming students in the preceding chapters but because I am trying to whet your appetite for studying computational systems as much as possible so I am picking out chapters that are directly and immediately applicable to you. After tasting the first few chapters many readers would have chosen to just read the whole thing but for those who still need some convincing chapter 7 is where you should be now.

The most important topic in this chapter for you is the issue of randomness. How is randomness important? In the world of algorithms, we have a subset of algorithms which are called randomized algorithms. As opposed to deterministic algorithms, that have their entire activities figured out in a rigid way from the beginning, randomized algorithms use randomness in making certain decisions in the execution of the Algorithm. In this chapter you will learn everything about randomness most importantly the three mechanisms of randomness:


mechanism 1: randomness from the environment
mechanism 2: randomness from initial conditions
mechanism 3: intrinsic generation of randomness

Among the three mechanisms of randomness, randomness from the environment; randomness from initial conditions and intrinsic generation of randomness Stephen argues that the most robust method of getting randomness is through Intrinsic generation of randomness. How is this important? This shows that the quality of the sources of randomness that we want to use is as important as what we are doing. There is a detailed explanation on the weakness of mechanisms 1 and 2, and how intrinsic generation of randomness which we get from simple programs like cellular automata are much more superior. Section 2 of this chapter gives a thorough treatment of the Three mechanisms of Randomness while sections 3 and 4 go into describing randomness from the environment and Chaos theory and Randomness from initial conditions. This book is just a highly compressed but very comprehensive presentation of massive research into computing, rather than going on your own difficult path of research, it will be better to start with this book even if you are old to the field.

Section 5 deals with the Intrinsic generation of randomness which is the most important section for you to study and master.

Chapter 8: Implications for everyday systems

When programming you are usually creating models, the abstraction we have been talking about is basically the process of model creation. But what is a model? A model is basically an idealization of a system that ignores certain details. The abstraction hierarchy that computers are built on involve creation one model of an underlying system after another.
The first section of this chapter: Issues of modelling, goes in-depth about the fundamental issues of modelling, which will be very useful for you as you would eventually have to invent models when trying to solve computational problems.

The sections after this go into models in nature, you might have to write a program that models some kind of natural phenomena be it fluid flow for the oil industry or material science. Apart from the fact that NKS methods are superior to methods based on traditional math in many cases, and this section deals with models created with NKS, you will learn how to build robust models that capture the most important features of various systems by reading this sections.

For those interested in building financial software reading section 8 will ground you in a fundamental technique for modeling those systems.

Chapter 10: Processes of perception and analysis

This is one of the most important chapters for computer programmers especially those that want to go into Data science, Big data, Data analysis, Artificial intelligence, etc. Stephen deals with data at the deepest level of understanding possible.

After a brief introduction in section one and some a good description of what perception and analysis. Stephen revisits the issue of Defining the notion of randomness again in section 3. A clear definition of Complexity and Data Compression is discussed in detail in sections 5 and 6. These sections, 5 and 6 basically answers the question, what can be compressed?

Section 7 goes into great details about Visual perception, this section will be useful for those endeavoring to get into the field of computer vision. Section 8 for those who want to go into audio analysis as their domain of programming, while section 9 goes in depth into statistical analysis, of course the NKS way is emphasized all the time.

Section 10 goes into cryptography and cryptanalysis, indispensable if your domain of computing involves computer security. A deep analysis of some NKS systems capable of providing good cryptography is also discussed.

Those with a mathematical bent will love section 11 and those who are thinking of artificial intelligence should really read section 12 on Human thinking for some deep insight into the issue. Concerned about alien life and extraterrestrial intelligence? section 13 does justice to these issues.

Chapter 11: The Notion of Computation

This is a chapter that goes deep into computation. After reading this chapter and the accompanying notes section, you will come out with an understanding of programming and computer science that will change your destiny in technology permanently and for the better. You will gain knowledge and insight that only people with advanced degrees in computer science have. I totally recommend this chapter to every programmer, newbie and advanced alike.

It is important that you carefully read every section in this chapter as it will ground your knowledge of computing solidly and every subsequent programming task you do will be done with depth and clarity as to what is really going on computationally. This chapter deals with computation as a pure thing, away from any complication that physically implemented systems suffer.

Important concepts you will learn from this section include the concept of universality and how systems emulate each other. Very interesting chapter indeed. Pay good attention to section 3 and section 4, as they are the most powerful sections in the very powerful chapter.

Chapter 12: The principle of computational equivalence

Of course you wouldn’t want to miss out on the overarching conclusion drawn from all the ideas in this book. This is a very important chapter that will strongly affect your technological future in a positive way should you agree with the ideas contained in it.

Sections 1 to 4 go ahead to clearly state the principle of computational equivalence in terms of the Basic framework; the outline of the principle; the content of the principle and the validity of the principle. Programmers should take not of section 5 that goes deeper into explaining the phenomenon of complexity. A very important section introducing an idea that has only previously been hinted at is Computational Irreducibility.

Computational irreducibility is important because many of us especially those trained in the traditional sciences have gotten used to the idea that the future of any system out there can be predicted absolutely with the right mathematical equations of course. When we encounter systems for which a specific point in their evolution cannot be predicted we usually assume that in time more sophisticated mathematics will do that for us, but computational irreducibility says that there are systems for which no formula will ever be produced that can predict a future state in its evolution. This implies that we just have to run certain programs and wait for them to reach a state we seek to predict rather than try to unsuccessfully predict their future state by deriving formulas. This means that there might be no effective way to summarize the behavior of a system and capture this behavior in a formula of some sort.

The rule 30 cellular automaton is an example of such a system, enormous effort has been put into finding a formula to make meaningful predictions of its future state to no avail. It's not really a matter of waiting for more sophisticated math, it's something that cannot be done because the principle of computational irreducibility says so. If it could be done, then computational irreducibility will not be a principle and traditional mathematics equations would soon be able to predict the future state of any system. This is a section that will alter you thinking about all kinds of systems, even those that you will encounter in the daily course of your career as a programmer.

Another important part of this chapter for computer programmers is section 8: Undecidability and Intractability. Undecidability involves questions like what will be the final outcome of a computation after perhaps an infinite number of steps.

Another must read section for programmers is Section 11: Implications for Technology. If you're a "how do I use this" kind of person then you will find this section the most interesting, although more will be gained from going to the note section of this particular section. In the notes related to this section found at http://www.wolframscience.com/nks/p841--implications-for-technology/ you will see the topics outlined below. Pick any to delve into details of the topic that might interest you.

» Self-assembly

» Nanotechnology

» Covering [implications for] technology

» Applications of randomness

» [Implications for] chemistry

» Special-purpose hardware [for cellular automata]

» Applications to design

» Ornamental art

» Sources of randomness

» Generating textures

» Randomized algorithms

That's all for the main part of the book, if you have been blown away by what you have seen in the main part of the book then the note section will blow you away even more. The note section treats all the topics in the book and more in excruciating details including runnable code that you can download and run to see some of the programs you have been reading about in action. I strongly recommend that you read the note section associated with the chapter and areas that have interested you during your reading. There is a wealth of knowledge there that you will not find anywhere in a prepackaged form as neat as this. Launching your own research program into computational systems without using this book as a background will be a horrendous pursuit. The book itself should be a foundation for further research because the book doesn't claim to contain all the information about computational systems. Should you get really interest in the science of complexity, then you should check out the Complex systems research journal at http://www.complex-systems.com/

All the code in the notes section of the book is available for download. Now in another area of this work you remember that I mentioned that to be a programmer after reading several books on the language you are interested in, which I recommend that you learn the Wolfram language first, the next step is to find some code to read and tinker with. The Wolfram demonstrations page has about 11,000 programs with source code available for you to read and play with, covering every thinkable domain. Now I will recommend another path. This is the path of the master. If you read this section on computational systems to this point, then it indicates that you don't just want to operate at the surface of programming and you want to penetrate its depths. The path I offer you now is the path of the master, where you will work with programs dealing with the most fundamental aspects of computing. The code you will be working with cover powerful programming styles that you might never come up with on your own until you have been programming for about ten years and have written tons of code and read tons more.

The code in the notes section of the NKS book, are the most general pieces of code I have ever come across because they are dealing with the most general issues of computing, doing the most basic things computers do which is transform data. If you want to shorten your path to programming mastery, I recommend that you download the code in this section, and not just read it, analyze it completely! The code deals with general computational issues so apart from what you have learnt in this book there is no additional domain expertise required to be able to understand the content of this code. The code is self-contained and all you need apart from a Mathematica/Wolfram language system is this NKS book and the Elementary Introduction to the Wolfram language book both available for free online.

I would also recommend that if you choose this path of mastery, then you try to read the entire book too and not just the sections I handpicked. It's not much of an effort for an aspiring master as it is just 1280 pages, there is so much knowledge and wisdom contained in this book about programming, technology and even science that the effort of reading 1280 pages is entirely worth it.

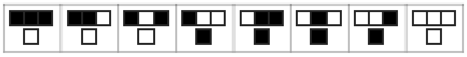
Reading the entire book, will enable you experience all the chapters and the note sections associated with them so that you will be able to deal properly with the accompanying code. You will come out from this endeavor a better programmer for life, ready to conquer any domain, system or code that you will be presented with.

I have read this book 3 times and haven't still drank completely of the wisdom contained in it, start your journey now and become a better programmer for the rest of your career.

How large is the computational universe? There are only 256 elementary cellular automatons in 1 dimension but this is not even by any means the totality of what is possible. The computational universe is infinite in size! Why do we call it elementary CA (cellular automata)? this is because of the nature of the Rules. If you read the book as I recommend you will see why they are called Elementary CAs in much details but for sake of brevity, I will give you some rule of thumb. ECAs are those 2 color, black and white usually, range 1 rules. But nothing limits us from having any number of colors or any range size. In a range 1 system 3 cells interact to produce one cell in the next step of evolution. We can have range 2 systems where we have 5 cells interacting to produce 1 cell in the next step and they can have 3 colors or more.

```
In[1]:= RulePlot[CellularAutomaton[30]]
```

Out[1]=



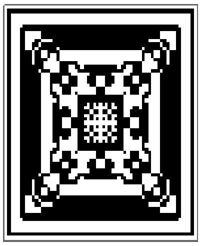
Rule 30 a 2-color range 1 rule.

When we enumerate the total possibility of rules in a 2-color range 1 system we find out that there are about 256, for more colors and larger ranges you will have larger numbers extending to the trillions, quadrillions, and all the way to infinity! You can actually view the evolution of any rule if you have a Wolfram language system in place or if you don't want to do any programming and just want some visual representation of the evolution, a simple wolfram alpha search will just give you a result you can search for a rule like: CA rule 4525248585554412244889977452211144 5885 in Wolfram Alpha and you should get a result! That is how large the computational universe is.

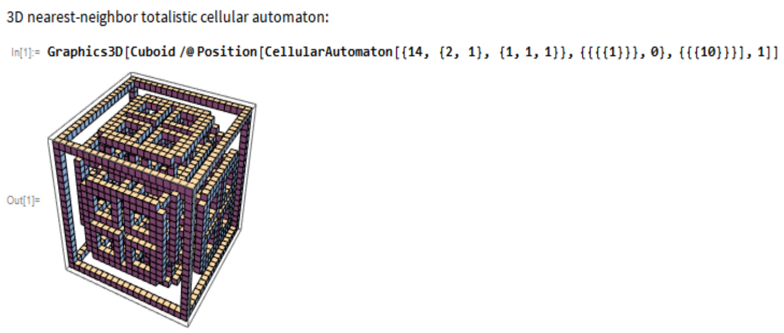
To go further there are 2D cellular automatons, 3D cellular automatons or any number of dimensions of cellular automatons. Although you might not be able to easily visualize cellular automatons beyond 3D nothing stops you from exploring higher dimensional CAs that is why I said that the computational universe is infinite.

```
In[1]:= ArrayPlot[CellularAutomaton[{14, {2, 1}, {1, 1}}, {{{1}}, 0}, {{{30}}}]]
```

Out[1]=



2D totalistic cellular automaton



3D nearest-neighbor totalistic cellular automaton:

```
In[1]:= Graphics3D[Cuboid /@ Position[CellularAutomaton[{14, {2, 1}, {1, 1, 1}}, {{{{1}}}, 0}, {{{10}}}], 1]]
```

3D cellular automaton

* * *

9. THE POWER OF ABSTRACTION

Our job as programmers is to create abstractions. An abstraction is a way to represent lower level stuff by higher level stuff by grouping a bunch of lower level stuff and representing them with at least 1 high level stuff. The first abstraction in programming is actually the link between the computer hardware and the first level of software, machine instruction. Then we build other layers of abstraction above this base level, till we are at the highest possible level of abstraction in one dimension. That is the high level languages. The function is the highest necessary level of abstraction, people have built classes that combine functions and data, called methods and fields respectively as another level of abstraction.

The other dimension of abstraction is the actual creation of software. A difficult problem I encountered in my early days was that after mastering the syntax, I went online, got some open source python code and when I opened it and I got confused. The syntax didn't change, but there were other things that you do not learn in programming books. A few examples are: Idioms and design patterns. It was so shocking to me because all the code I encountered conformed to the syntax specification, but understanding them was very difficult. Apart from the fact that some functions were referenced from other parts of the program, which was another source of difficulty because I had not yet mastered the art of understanding complexity, or learning to know what to ignore when trying to understand large and complicated systems. The complications here were different. What I didn't know at the time was that I was facing another level of abstraction, one I had not yet been forewarned about. Because I was using Python at the time, it was easy to pull out some segment of code that I wanted to understand and write a mini program with it that allowed me to poke at it from different angles. And because python was interactive this was not very difficult to do.

It would be quite clumsy doing this most necessary task in a compiled language because after writing these mini programs you would have to compile and run, or with C compile, link and run. With the IDLE of python, I would just whip up some tiny piece of the code I was trying to understand and poke at it from many angles till I gained comprehension. Even this method has its limits, when some code you are trying to poke at has linked complexities. When encountered with this, you just have to wait until you read most of the code, and hope your memory holds and even then, probing the standard library would also be important. If after reading the code you still don't understand all the parts, you must ask yourself if that part is really essential in your understanding of the entire code, that is if understanding it is required to modify the code or improve the code. If the answer to the above question is false, then you must forget about that code fragment and take it on faith that it works and maybe should be left on its own for now till your knowledge improves.

If you really can't let go then online forums like StackOverflow.com may be able to help, or you go full on research. But if you have a deadline to meet then leave that segment of code if it can be left alone.

Programming is not just a difficult task because syntax is difficult. A good language will usually have minimal syntax, the difficulty with programming lies not just in developing efficient algorithms to solve problems, that is hard, but it lies in understanding and creating code beyond the function or class level of abstraction.

The reason why it takes a while to master programming is because you will have to develop wisdom, which takes an irreducible amount of time to get. To reach this wisdom you have to go beyond knowing the syntax, you have to work not just on your own programming projects, but read other peoples code and even work on other peoples' projects with a wide variety of programmers. There are a lot of smart people writing good code out there, you can really get good by studying the works of these masters to become a master yourself. You can go far by picking a project, and find a solution to it using only the materials you found in your programming language API library but your code will not have that touch of mastery. Even though you must have mastered algorithms using computational mathematics, it will still be long way before you start writing elegant code that someone can describe as beautiful.

Black boxes

A program is usually a black box to the user. You cannot see the code that was used to design your word processor but you can fire it and use it as a black box. The black box I am about to describe is not one you usually cannot read. Although most black boxes are really black because you cannot see their content. This box is more of understanding and not seeing.

In programming, there are some pieces of code usually wrapped in a function that are the result of some long line of research. They are difficult to understand, highly optimized to remove any solely pedagogical items. We usually find it difficult to understand these black boxes not because we don't have access to the code, we can see the code staring at us, but it is at such a high level of abstraction that if we are not familiar with the domain that is being described we will find it almost impossible to understand. It is not usually a sign of mental weakness that we cannot understand these pieces of code, but it is because the code is usually abstracting the core structures of a domain we are not familiar with. Sometimes even without heavy optimization, the code is still not approachable.

So, what do we do when faced with this situation. If the entire project is in your field and if you find out that you cannot understand most of the code that is being written, then you have to go ahead and study your field more. You might need to go back and learn the basic algorithms of your field, etc. but if what you are faced with is just a small segment of your project, like if you are reading a python package like SciPy, and maybe you are part of the open source project and you want to make a contribution. If you see a function like fastMathOnGPU(), this is just a hypothetical function not a real function in the SciPy package, and you are basically interested in contributing to another part of the project that uses the function but doesn't need to modify the code then just let it be. This is part of the art of mastering complexity, you must learn what to let go of at the moment and focus on viewing certain things as black boxes. Use the code in that part of the program you are responsible for and ignore the mechanics. This is what object oriented programming sought to achieve by explicating the idea of encapsulation to hide the implementation.

There is this thing that is available on every modern IDE it's called the collapse button, use it to collapse the long and complicated code so that you are just left with the function name fastMathOnGPU() and none of its code staring at your face, go ahead and work on the part with which are familiar with.

* * *

10. THE UNIVERSE OF OPEN SOURCE

In the early days of programming it was usually a sole programmer probably in her garage coding away at some problem after reading some manual on the language they want to use and after tinkering with their machine sufficiently to get it do a Hello World program. Earlier still was the days of punch cards where programmers would punch their programs on those cards and carry it to a computer operator in shoe boxes and wait a day for their results. This computer operator was usually a bearded guy with thick glasses wearing a white lab coat in a glass room that housed the computing machinery.

In these times, the skill level needed to be a programmer was very high. Although most of the programs were short compared to our multimillion line modern megaliths, soon to reach multitrillion lines when programs start writing programs without human supervision. Google has a codebase of about 2 billion lines of code! To be a programmer in the early days you had to be an exceptional human being, with good aptitude for rational reasoning and sometimes a lot of math.

These days the landscape is different. Although you can learn a programming language and go straight ahead to solve some problem, except the problem is very unique to you alone and I doubt most problems you want to solve are, then rather than start hitting out code on your keyboard it is better to go Opensource first.

Opensource software like that you can get on sourceforge.net or on other sites, is one that comes with the source code for which the executable of the program was created. The most famous Opensource software in the world is the Linux operating system. The Opensource community is a loose network of developers' resident all over the world who come together to develop software and give it out mostly for free, but more importantly with the source code available, so that other developers can study and modify and choose to share back their modifications or hold it back.

Opensource is the greatest thing to happen to the software industry. Usually a software company has a close-knit group of paid developers that develop their products for them. The quality of these developers usually depends on how well the hiring managers did when they hired them. And sometimes no matter how perfect a close-knit team of developers are, software is so complicated that there are bound to be bugs in it no matter how carefully constructed it was. In the Opensource community you have all kinds of people collaborating on projects from highly skilled to newbie coder. There is usually a piece of software for managing what finally goes into production and this is usually controlled by the core team, and most of the time the originator of the project, but anybody can contribute. The reason why Opensource is so robust compared to software produced by a closely-knit group of developers at a typical software company is because apart from the diversity of the skill, the highly non-political environment where quality of the code and not devotion to any particular coder is the key metric is much favorable for producing high quality software products.

In most large companies, no matter how perfectly you structure the work environment, there will always be the human factor of politicking and struggling for supremacy where the quality of the product is sometimes jeopardized to fulfill someone's vanity. Also, the feeling that you are just working for money may prevent you from putting in your best. In most companies' people are so scared of losing their jobs they will accept some non-optimal short term fix rather than a long-term strategy of producing robust code that might take some time and resources.

Another problem is where companies make non-technical managers supervise highly technical programming projects. It is assumed that the "management" skills are all encompassing and can make every project no matter its nature a success. This is more of a myth propagated by the managers themselves to retain a job rather than a reality. Without sufficient knowledge of computing, software and technology as a whole you can't just "manage" a highly technical project like a software project or even manage the kind of people that participate in such an endeavor. It is always a good idea if a programmer who has been in the trenches of software engineering with some education in management is called upon to manage a software project not the typical manager.

Software for business is usually created to satisfy some business requirements that can require that the programmers be supervised in some way to make sure that they are meeting these specifications. In this case the need for the purely business trained manager is important, but they should interface with some highly technical manager and not interact directly with the programmers involved in the project.

In the typical large software company, the clamor for power, position and job security can jeopardize the delicate art of producing robust high quality software. In contrast to this scenario is the Opensource community where coders come together to give their best in projects they are passionate about rather than projects they are forced into because they work for a particular company. Code is criticized without the fear of hurting the feelings of a superior, the code is more important than bruising anyone's ego.

Programmers are usually highly intelligent people and like to be free! To attract the best coders available software companies, have to find a way to structure their workflow around this need for freedom. Although a software company has its goals and if someone applies to work there she must have decided that this is the place she wants to be, programmers should be given some liberty in choosing their projects and should also have some time to do their own personal projects without judgement. The industrial man/laborer stereotype doesn't really work with programmers. You cannot muscle programmers to "perform"! Create the right environment and programmers will do at their best, provided you did your job hiring the good coders. Many forward-thinking companies are inculcating Opensource methodologies in their workflows, aspiring programmers should seek these companies if they want to be more productive in their careers.

Linux operating system is the most robust piece of software created by humanity, not because the guy Linux Torvalds, who manages the project is a god but because thousands of highly skill developers people you can never box into one company came together to create it thus sharing their expertise and experience in programming. No single company no matter how large and powerful can ever own such a group of developers, although they are all free to benefit from the produce of the project.

So, how does the newbie coder use the open source community? This is the modern way to learn programming.

1. Get a language you want to learn.
2. Get a book or 3 to learn and familiarize with the syntax
3. Do the exercises if you're into that kind of thing, I personally don't like exercises but do really cool ones.
4. After the above you can either choose to start a small personal project or skip to 5.

5. Go online if its Wolfram language, go to the wolfram demonstrations page or if its python, then go to the python package index, or you can go to sourceforge.net or any other place you know, scroll through till you find a project that interest you, download it and start reading the code.
6. Write mini programs to understand aspect of the code that look alien to you.
7. Contribute to the project, don't be ashamed you can never know what finally gets pushed into the final product.
8. If you really need to use the project for your own thing, you can fork the project and start making any modifications you want to it, since you will now be the sole administrator of the project.
9. You can build some other software that use the project or other projects as a base.

This is a modern path to learning programming. Learning programming only through doing exercises contained in programming books or you own personal project will not make you a world class programmer except you are some kind of super genius. This was the way it was done before the internet, but it is no longer a very efficient way to learn. Sometimes your project has been done and you can get an Opensource version of it and modify it to suite your taste or use it as is. If that project doesn't exist as a whole intact, there are parts that have already been written by someone else. It is usually better to just get your hands on those parts and work on integrating them into a solution. Don't try to reinvent the wheel except you are just looking to perform mental acrobatics, which is good and healthy for your brain. If you want to create a solution to a problem in software in the shortest necessary period of time, then you cannot ignore Opensource.

Many Opensource solutions from standard sources have been properly vetted for quality. Especially when the developers are large in number, there is a high probability that due to the free nature of the system, highly skilled developers must have ascended the ladder. So you can be guaranteed that most of the time the code available is of very high quality. If you find code you suspect might be low quality, you can simply profile it using a profiler to find the bottlenecks and improve it yourself.

How many languages should you learn?

This will depend on what exactly you want to focus on. Although a general programming language can be made to do anything, like I have said previously, even though it is possible theoretically practically this might not be feasible. I have already outlined some of the specific strengths of some languages and will go into others later. The main question we are trying to answer here is how many languages should you learn. It is very rare to see a full software project done in only one language. Usually a software project is done in a bunch of languages, with different languages doing specific things. For instance, if you are inventing some new algorithm, you would probably want to do it in Wolfram Language. If you want to do a GUI then C# or Java would be your point of call although there are GUI libraries in many other languages.

As a beginning programmer, you should learn several languages so that you have a solid appreciation for the entire software enterprise and also make your brain flexible when thinking about solutions to computational problems. The thing to keep in mind is that you will seldom need to know one language if you want to become a real software engineer. You will have to learn a stack of tools and languages even if you don't need all. Familiarity with different languages will aid you in understanding any kind of programming methodology.

Sometimes you learn a language not because you will be required to write code in it, but because you will be required to read it later on when faced with some project that includes it. Reading code is the major reason you should be conversant in many languages because many software projects come loaded with different languages doing what they do best.

Your first programming language

This is the most critical decision you can ever make in your entire programming career because it is going to shape the way you think about solving computational problems for a very long time, indeed the rest of your life. Although all general programming language can solve the same kind of computational problems, they are all designed differently. Naturally I prefer interpreted languages because of the ease with which you can start whipping out solutions, without writing too much code. Apart from the syntax, there are these other things I talked about previously like the conventions of the language. Two languages can have arrays and they can require different syntax to specify them, but the conventions surrounding them in how they are used in real world programs can vary greatly.

As you work primarily in one language, especially if it is your first language, you will inevitably begin to think in that language. Meaning when you have a problem to solve, your mind will automatically start formulating a solution in this language.

If you learn computational mathematics before you encountered programming, you might begin to think of an algorithm mathematically, but once you get used to directly coding your solutions on a computer, your brain will begin to think in programming automatically and skip this mathematical stage. But your mathematical capabilities will enable you think efficiently about the solutions to problems in certain situations.

Now thinking in a particular language can both be a blessing and a curse. In my programming experience, although I first encountered C/C++, I never really spent so much time with them. After taking a tour of many programming languages, I ended up settling with python for a long time and this affected my thinking at first in a positive way. I thought about the solutions to problems in python and python is a good language, but when I encountered Wolfram language I found out that thinking in python had limited my "freedom" of expression. Although I will go into some detail about the Wolfram language later on, for now it will suffice to say that Wolfram language opened me up to computational freedom.

I found out that even as simple and easy as python was, Wolfram was simpler, easier and more elegant. I never conceived it possible that a programming language could be simpler and more expressive than python but Wolfram changed everything. And the funny thing is that the language has been existing for thirty years and has been the core language used in the design of an enormous stack of computation called Mathematica which has been used everywhere from designing Jet engines, to analyzing financial data in Hedge funds.

In order to change my way of thinking from the already good python, I had to work exclusively in Wolfram language for a long time and eventually my thinking changed. This happened because naturally I have a very flexible mind, but for many people it would have taken much longer to accomplish. So, choosing your first language is very critical.

Yes, I would say that I am affiliated with Wolfram Research but this statement is not biased. I got attracted to the company because of the elegance of the language not because of any other factor. And if my affiliation should end I would still say that Wolfram language is the simplest most flexible language I have encountered.

As a beginning programmer, I would advise that you should learn these languages in this sequence no matter what you are going to end up doing in your career because they form the base of many programming paradigms! You can add any other languages to this sequence depending on your domain of expertise, but I would say that from my experience each of these languages will bend your brain in a certain way, which will aid you greatly in your career, no matter what you will end up doing. Please follow the sequence if you can, I have thought about this greatly, I have searched deep and investigated and this is what I came up with:

1. Wolfram Language
2. Python
3. JavaScript
4. C
5. C++ (optional)
6. C# | Java

The list above is the result of great contemplation and labor spanning a period of 17 years, there is no bias in this list. Learn all the languages in this list in sequence. When you have reached C, then C++ is optional, but to enlarge your brain you should spend some time with it even if you will never use it in your life because you might encounter some project that has some C++ and you might want to know what is going on in there. You might never write code in C++ except you are required to. As for C# or Java you can learn both or just one. The thing is that it's not just about the languages but more about getting a general philosophy about programming that will shorten your path to mastery.

If you are an absolute beginner who has never touched code before, then you are lucky because you have the opportunity of standing on the shoulders of giants. You will understand deep things about solving computational problems that only adept programmers know in a much shorter span of time.

After learning this sequence then you can choose any of them to specialize on depending on your needs, or you can add any new languages. If you note this list might look ancient but it isn't, this is 2018, and I am aware of the developments in programming like JavaScript frameworks, Ruby, Go, Swift, and even the Rising star Kotlin. But this list will teach your brain fundamental programming and give you such a solid foundation in Software engineering that you will pick any of these later mentioned languages and master them in no time and use them as you see fit. Just like the depth of the foundation determines how tall a building you can build; the depth of your programming foundation will determine how high you will soar in software engineering.

* * *

11. WHAT IS COMPUTATIONAL THINKING?

Of course, whenever we are faced with a computational problem, a problem we seek to solve with a computer, we have to think computationally in order to solve it. The word computational thinking is a new buzz word and although the concept isn't new, people only started using the term recently. There have been attempts by many people to define it and there are even courses and books on the subject. But the point some of this people miss is that computational thinking, although it can be used as generic term for any software activity, the real definition should apply to the computational activities of the non-programmer who doesn't intend to create software but just wants to answer some computational questions using programming as a tool.

There has been mathematical thinking for many years, and many people who are skilled in this disciple can decipher the solution to complex problems using mathematics as a tool. This could be Calculus or the much recent Discrete math or Concrete math as Donald Knuth et al. choose to call it.

Computational thinking is no different from mathematical thinking in that both of them are methodologies beyond just thinking in natural language that we use to solve problems. In computational thinking, rather than math we use programming tools, like programming languages to solve our problems, even though we might not write full blown software.
Most people are teaching computational thinking as mathematical thinking and this is NOT the direction this new discipline should go. There is nothing wrong with mathematical thinking it is a very powerful way to solve problems. The problem is

that many people are not good at math and selling mathematical thinking disguised as computational thinking will turn many people away, people that could have benefited from it if they were shown a better path.

The mathematical thinking that is disguised as computational thinking takes traditional mathematical analysis and converts the equations and systems into computer code. This is good for the student versed in math but not the lay man.

Computational thinking should be as automated as possible; the user should not know much of math beyond arithmetic. The user should be able to combine built in tools like functions and classes to solve their problems without needing to do math or develop generic algorithms. The user should be able to work meta-algorithmically, meaning combining pre-built algorithms to solve their problems.

In summary, computational thinking requires using prebuilt computational tools like built-in functions to solve problems facing a non-programmer. The only extra requirement is that the user should be versed in the programming language that they intend to use and there could be no mathematical background beyond basic arithmetic. You can do computational thinking in any programming language but it is more practical to do in an interpreted language and in an interactive environment like a Wolfram Notebook or Python IDLE.

An interactive environment makes it possible for someone to get output from even a single line of code. It is this constant feedback kind of programming that makes computational thinking for the lay person possible. Although you can do in the python environment, it will be very difficult because despite the fact that the python standard library contains a lot of built in functions, most of those functions are programmer centric, and can only be used properly by a programmer. Even if you understand python language, you might not be able to use most of the python functions for basic computational thinking because they were not designed to be used directly for such purposes but were designed to be used as building blocks for programs that could be used directly for computational thinking.

Without bias the Wolfram language is the best language for doing computational thinking. This is because it contains built-in functions that we could use at once to do some immensely complicated task like applying a Gaussian blur to a picture. There is a single function that could identify any image and there are also functions for doing things like finding out the language that a particular text is written in, to many other tasks that require vast and highly complicated mathematics to perform available via a single function.

You don't need to have any skills in image processing to be able to start doing Image processing in Wolfram language, there are built-in functions for doing almost anything computable. Don't take my word for it visit www.wolfram.com and take a tour of the language documentation to see what is possible.

If you are faced with some data you want to gain insight into, you don't need a PhD in statistics to start doing some data science, with built in functions and some programming with nothing beyond basic arithmetic you can start doing some basic data science. That's what I call computational thinking using built in functions to accomplish real world task with a computer.

There is a limit to how much you can solve by just thinking about it. To go beyond natural language, we have to use to kind of structured form of problem analysis and find solutions. The first rung of the ladder after natural language is mathematical thinking. Beyond this we have the much general tool of programming, and with language like the Wolfram language and with much skill python we can usually solve problems directly using the computer to automate the process of finding a solution. We can do things on a computer that would require some very complicated mathematics and intense calculational effort to manually achieve. The reason computers were invented in the first place was to automate away boring mathematics that was previously done manually.

Generally, the emerging discipline of computational thinking should involve educating people without a mathematical bent on how to use programming and built in capabilities of programming languages to solve problems in the real world. These people should not be required to write full blown software to solve their problems. A good language, with lots of built in capabilities, should be used and based on need the user can build more complicated solutions based on combining already built-in capabilities and using programming alone without much mathematical skills. Those with good mathematical skills can take care of themselves.

✻ ✻ ✻

12. THE CRAZY WORLD OF MOBILE DEVELOPMENT

What I give you in this work are seeds. Every field I touch I give you very valuable seeds. Seeds of wisdom that will guide you on your journey in the Wild jungle and sometimes very dry desert of computer programming. I do not discuss the hyper latest fads in technology. Many things have come and gone in the World of technology and programming specifically. The hot thing in the 90s and early 2000s was warez, small applications that did many things and were highly popular, some of them were useful but most were viruses. Everybody was peddling desktop applications or as they were called warez. Nowadays its web applications or more specifically mobile applications.

If you are venturing into the world of programming now, it might seem that the only kind of programming you can do is developing web applications, but programming is much broader than that. And having a solid overview of the fundamentals will enable you to be able to move into the future when a new wave of technology arrives. So, you can learn mobile/web development and of course land a good job, but like the desktop application programmer of the early 90s, when a new wave of technology comes, like the internet and html came into prominence in the 90s, you will be wiped out of the food chain. This might not be sudden and there might still be need for your services but the money will shift hands.

Whatever you do for a web browser will need only minor adaptation to work on mobile devices like phones and tablets. This is because browsers run on both desktops and mobile systems and designing applications with certain tools can make it possible to run it on both the desktop and mobile, of course it will have to run in a browser. The major setup for doing this kind of programming is the JavaScript, HTML, CSS stack.

Of these 3 technologies, the most famous is JavaScript which has eaten the mobile world. It is very hard to think of mobile development without JavaScript. Almost everything you encounter while on the web has some JavaScript running somewhere.

I included JavaScript in the list of languages you have to learn because it is a fundamental paradigm that shows many basic features of programming. If you ever want to design anything that will be viewed and interacted with by a user on the web you should know how to code JavaScript. Apart from that, the programming mechanisms available in JavaScript will contribute to this mental structure that will enable you tackle many other web related languages and technologies without difficulty.

Except you were some guy doing assembler for some processor manufacturer or your Job requires deep backend stuff with C, you will see JavaScript someday. In the earlier days, the desktop application programmer didn't need to have any knowledge of the web, the web too was in its infancy with HTML the only technology to use to specify web pages. But with JavaScript being used as a local scripting language on many desktop applications and the availability of server side JavaScript solutions like node.js, it should be very difficult to ignore JavaScript these days.

What makes JavaScript powerful on the web is its ability to turn a dumb interface into an active one. When you write HTML, and use CSS to style your web page, by way of appearance you will be presented with something beautiful, that is if you did your job well. But this beautiful thing you have created will be no more active than the pages of a book you hold in your hand. To give it that spark of interactivity, that automatically folding menu, actions when you click button and so forth you need JavaScript.

JavaScript is not only useful for the User Interface; it is a general programming language and can do any computation you want. Most often when you have collected raw data from a user in a web form, you might want to do some extra processing on it before submitting it to a web server for further processing. This is where JavaScript comes in. JavaScript is sandboxed so the chance of writing code that damages your computer is limited but possible with effort. You can also use it to monitor user browsing behavior and collect data to improve your product and services but this is also a double-edged sword and can be used for unworthy purposes.

JavaScript also has a great future because devices like virtual/augmented reality use it to do most of their UI stuff. It is very obvious that VR/AR is here to stay so this is another incentive for learning the language.

So, with JavaScript/HTML/CSS and some good coffee by your side, you can create cross platform applications that run on both your web browser, Desktop, and mobile devices. This is one way to create applications especially mobile applications since this section is about that. In order to create truly native mobile applications, you will need some more tools.

As a new programmer, I will advise that you learn core JavaScript language itself before you delve into the wilder world of JavaScript frameworks. So, what are these frameworks? They are basically JavaScript libraries that ease the burden of developing with JavaScript. Just like the standard library of a typical programming language like python or Java contains prewritten code that enables you get up and running with doing some very basic programming stuff, JavaScript libraries also share the same role. The most popular frameworks are JQuery, AngularJS, and ReactJS. There are many others but this is not a book on JavaScript programming or frameworks.

After learning the basic JavaScript, I advise that you pick the 3 frameworks that I mentioned above, study them all and then decide on which one you will choose to stick with then master it.

There are 3 major mobile platforms in the world: Android, IOS and Windows. There are other mobile devices like Ubuntu and Firefox, but I will not talk about them because I'm not very knowledgeable in them, my apologies to the creators.

Each major mobile platform has their core language which is used to create apps in them. Although They have browsers and can run more generic JavaScript/HTML/CSS code. Let's go through the major Platforms one after the other.

Android

To create native android apps, you will need to know Java and some XML. Java is used to write the code while the XML is used for specifying the interface. There are two types of android apps: Fully local apps that are self-contained and don't need internet access and the other type are apps that have local aspects but depend on some kind of cloud service to do most of their computation, thus requiring internet access.
Although it's hard to put a hard line separating both types of apps, because even though you are not doing any remote computation on some cloud service, if you enable

ads on your apps, then the ads come from Google's servers on the internet. It should also be noted that Google owns android. A fully local app would be one that makes no connection to the internet at all.

An internet enabled app will have to connect to some kind of API on an online service to provision whatever it needs from that service.

What is an API? This is an acronym for Applications Programmer Interface. When you are using functions from the standard library of your programming language, you are using a function or class exposed as an API to you the programmer. You don't have access to the function/class body, but you can call the function through the function signature that is exposed to you. This is what APIs are all about. APIs are not limited to your local standard library; with the internet, you can have access to an API via remote execution.

A company like Google might have a huge cloud based application like Google maps, you cannot download and install Google maps even if the executable is available to you because as an individual you might not possess the computational capacity to run it locally. This doesn't stop you from being able to compute with google maps. Google exposes some functionality of Google maps via its API remotely. If you are tasked with writing an app that enables a user view the map above her current location, you could simply follow the modalities of accessing the Google maps API. This will include some kind of registration and then you can see the function call signature required. Locally in your code all you need to do is to specify this function with the appropriate arguments and when you build the app and call the function by clicking a button or making some other kind of request, the function accesses the internet and remotely calls the google maps API function. The function returns some result, mostly in the JSON (JavaScript Object Notation) format or XML and this data is returned to your client.

In the course of programming the app, you must have anticipated how the data will be returned and process it in the programming language, which in this case is Java. After processing the data, you must now format it in a user-friendly manner for display to the user.

This is basic procedure for developing non-local apps that require some internet service to do computation locally. So fully local apps are very rare, even a game that runs locally might need to maintain some online knowledge of user game play statistics and user accounts etc. This is a light load but there are heavy load apps like the google maps example.

Another cool example of an app that uses the internet for most of its processing is the Wolfram Cloud app. This app enables you to develop Wolfram language code on mobile. It has local elements that run on your local device but most of the heavy lifting is done online. The entire Wolfram system is very large and can't possibly fit on your phone. So, you don't download it locally when you download the app but you download only a very thin aspect of it, the client-side with mostly user-interface and local file management stuff. When you write your code, and run it in the app, it is sent to the Wolfram cloud servers, where it is run and the result is returned to you as an output cell. This is so fast and transparent that you won't even know that all that processing is not done locally.

So, what do you need to start writing Android app? There are numerous android tutorials out there. The best I think should be that on UDACITY because it is Google affiliated. Basically, you will need to learn how to program in Java, this will require reading some books on Java. Android has its own app framework which you will learn from an Android tutorial but all is based on a foundation of Java. Some basic knowledge of XML will be useful especially if you have never seen or heard of XML before. Android development is usually done with Android Studio, which is the standard development environment.

Another powerful approach to developing for Android operating system is using the new Kotlin language. As someone who is not really in love with Java I am welcoming Kotlin as a new direction to proceed in. I would advise though that you learn Java first because after learning how to build basic android apps from a good tutorial you will have to go online and check out some Opensource android apps to learn and master your skills. There are more Java based android apps but I know this will change in the future. If you are more adventurous, you can just dive into Kotlin. Go online and do the needed research to see what is necessary to develop with Kotlin.

So, what is Android basically? Just like you have Windows operating system for your desktop or Linux, android is an operating system. It is open source so you can get the entire source code and if you know what you are doing you can fork out your own version and name it whatever. Many people have done this. At its base, it's a Linux kernel and then the experts at Google built layers above it that enables you create your own apps. But know that when you are using android, you are working on a very solid Linux base.

IOS

When I write, I only put out knowledge that I know and have fully internalized. This is because I don't want to pass on information I am not sure about. I write about things I can defend so that if called upon to speak about what I have written I will not bring out anything that is not part of me. I am not a fan of copy-pasting information I have not digested. I say all these because I will not talk much about IOS. If you want to know more do your research. This work is a torchlight guiding you in the jungle, it is not the jungle itself. The opinion I give you here are to guide you in your general programming career.

To program in IOS you will need to know how to develop in Apple's Swift programming language. Yes, the programming language is called Swift. To work in this language, you will need to download and install Apple XCode. This is the development environment for IOS apps just like Android Studio is for Android. This is all I can say about IOS because I have not delved into it deeply. In a future edition of this work I will do greater justice to the topic because recently I am getting interested in IOS.

Windows10

Like I have said previously Windows 10 is the repentance of Microsoft. In the past I, have been a Microsoft hater but with the release of Windows that hatred has been reversed. Windows 10 promises to deliver a UWP (Universal Windows Platform), which will enable you develop applications that can work across desktop and mobile. This is similar to JavaScript's capacity but this time around you do not need a browser. You develop deeply native apps using Microsoft's major language C#. Apart from UWP Microsoft also offers Unity which is an engine for developing 3D applications, the most notable of which are games.

I have not developed for windows but I am considering it very seriously. C# is among the fundamental languages I listed and a great way to utilize the knowledge you have gained after learning C# is to develop a cross platform app for Windows10.

Another major benefit is that Microsoft has the Azure cloud, this means that developing cloud applications should be a breeze as the integration between C# and the cloud should be tight. The major tool for developing all kinds of things on Microsoft's Windows operating system is Visual Studio. There is a free community edition out there and its well featured.

Xamarin: A cross platform tool

With Xamarin, a cross platform tool that is tightly integrated with Visual Studio, you can start developing applications in C# that run everywhere. By everywhere I mean on Android, IOS and Windows10. I have not gone in deep into Xamarin but it’s on my horizon of study. I think it will be better to write once and run everywhere, maybe applying platform specific optimizations as you see fit.

The field of mobile applications is vast; this is not an exhaustive treatment of the subject but after reading this section you have at least a foundation from which you can explore deeper.

* * *

13. MASTERY OF PROGRAMMING

What is mastery? This is the journey to perfection. We can never be absolute masters of anything as there will always be more to learn about any topic we are engaged in. but we can develop ourselves to the level where our work will be of the highest quality possible.

In the field of programming becoming a master requires knowing more than programming. You should have mastered advanced algorithms, computational math and lots of other fields that might seem unrelated to programming but will influence your entire approach.

To master something, you need to narrow yourself to one thing and in our case this would be one language. But you cannot master the entire field of programming by learning and working in only one programming language. You will need to be exposed to different approaches to programming to develop a deep understanding of programming that is beyond the limits of any one language.

Learn the fundamental programming languages I listed at the earlier portion of this work, you can add more languages to that and you don't need to go very deep. Surfing the landscape of programming languages will give you exposure to different programming styles and widen your scope and intellect. After this your next step will be to pick one programming language that you want to master. This could be one related to the job you are doing or maybe you work in a language you don't really like but you have one which you love and want to master.

Your job might not require that you design a whole program yourself, you might have to work with many other coders and fit into the workflow. You don't have much freedom to control everything and you have to adapt to the workflow of the group and the approved style. You might not even be the project head. In another scenario you might have to work on other peoples' code in a very large code base that you can't change much and you might not be permitted to access the entire code base. If this is the situation you are faced with and you don't like the language you are working on, then you must go on the path of mastery with what you love.

When you pick the language you love, then the real path of mastery begins. You will have to read lots of code and write lots of code for some time. You must practice, practice and practice even more. This is the only way you can approach the pinnacle of mastery. When you feel you are at the top of your game there is only one way to test your prowess, participate in the open source community. Push some code out there and you will know your level, people will call you a master when they have seen your work. Another way would be solving a great computational problem.

Although you don't always need external validation to test your mastery, sometimes feedback is important. When you have reached a certain level of mastery in programming you will know it yourself, no one needs to confirm this for you.

* * *

14. DATABASE PROGRAMMING

Except you are writing some throwaway program that doesn't really need to store data, there is no way you will write a program without having to store some kind of data.

The first point of call when storing data that your programs generate is the regular file, that is usually provided by your file system. Usually during programming, you generate a list that you have to store, you could just write it into a plain .txt file and that is it. When you need it you could simply read it out again and continue performing computation on it.

Indeed, you really don't need to add an extension to the file you want to store, you could just name it anything that agrees with the file naming convention on your operating system without an extension and read it back anytime.

As the data you work with becomes complicated, storing data without preserving some structure on your file system can make for difficult reading when you need it. So we start having to think of the structure with which we want to store a file. Basic formats for storing structured data is usually XML, but JSON has emerged as a better alternative recently because of the ease with which you can parse the data when you need it later on or communicate that data with another program or programmer that may need it.

With languages like the Wolfram Language, you can read files in many formats by default but if you are performing a computation and don't need to try to force your data into any particular format, you could simply store the resulting expressions like that in a file. You could store entire programs like that too and read it and still execute it without any issues.

There are also mechanisms in programming languages that write binary data to files. In all these mechanisms you are using your file system as a database. This is ok when you are not dealing with large amounts of data and when you don't need any kind of structured querying of the data, that is extracting well defined parts of the data from the file without reading the entire file into memory. Although some languages allow for stuff like memory mapping when dealing with large files, that is rather than bring the entire file into memory, you simply map portions of the file from the persistent storage like the hard disk into memory and deal with that. Managing memory mapping is difficult in code but much needed sometimes. If you find out that the amount of data you are storing is getting larger and larger, then reading from the file system is not really efficient and maybe tedious even if you're a master programmer who understands how the computer ticks and can design huge stacks of programs that deal with multi-terabyte files without difficulty. You really don't need to do this because programmers have come up with the solution to this, database systems.

Although your simple file system theoretically is a database, in order to solve complicated data storage and query requirements, we need a stack of software usually called a database management system. When people say database, they are actually referring to database management systems.

Database management systems usually have some kind of query language that we can use to talk to the database, so we use these systems to store data and retrieve data just like you could do with your file system manually. The power of these systems is that they allow you programmatic access to data storage, meaning that you could make a database connection in your program, issue a query and get results and process those results in your program without playing around directly with your file systems.

As the amount of data you deal with scales up, the database management will do all the low level work to make sure that your data is safe from corruption. A lot of effort has gone into the designing of these systems so you can have a great degree of trust that they will do their work.

We have various types of databases: Relational databases, Object oriented databases, Graph databases, etc. The root of their difference lies fundamentally in the way they structure their data although there are lots of other peculiarities to each of these kinds of databases. The most popular database is the relational type and the most popular relational database product is MySQL. There are other relational database products like Oracle, PostgreSQL, etc. MySQL is an open source product so you can be sure that the robustness that goes into the development of Opensource products lives with it.

Relational database management systems are queried using a language called SQL, meaning Structured Query Language, thus the SQL at the end of MySQL. This language has become an industry standard for querying relational databases and is one you would have to learn if you want to use a tool like MySQL to manage your data. It is a simple language and although it is not a general programming language it is a very important one to learn. Whether you choose to specialize on web-programming, mobile programming, etc., you will deal with data especially from form input, which you will need to store in a structured way. It is not advisable to this on your file system as it is not robust enough and was not designed for such usage.

There are other database systems which I think you should look into as your needs for your program might not be very relational-database-like in nature and might need to use other database types. As an example if you're working with data that has a graph structure then trying to use graph databases will be a good idea.

For developing web applications there is this database called MongoDB. Although I have only had a cursory glance at it I think it will be worth learning if web applications development is something that you are going to do. Also I once heard that Facebook uses something called RocksDB maybe checking that out too would be a great idea.

I am talking to you like a programmer, there are so many technologies for solving one problem that you cannot possibly master all by yourself as a single individual, you have a life to live and other commitments, but the way we proceed as programmers is hinting at things we think will be cool to other programmers so they can try and give us feedback. I don't want to Wikipedia bomb you by copying and pasting information, that is just adding unnecessary duplication to the information available in the world, something I passionately dislike.

In summary databases are an indispensable part when we write programs of sufficient size and scale, it is essential that you learn SQL at least or any other database paradigm you think will be important to your data storage solutions.

* * *

15. DATA SCIENCE

This is a buzzword in the programming community containing all activities relating to the art of trying to extract meaning from data. Data Science as it is practiced is an outcrop of statistics that was usually done with pen and paper. Data science is a bunch of programming techniques that is applied to data to discover trends, extract meaning or gain insight. The amount of data involved is not really important, what is important is the kind of conclusions that can be obtained from that data.

This is the best I can do to define this often confusing looking field, the thing to hold in mind is applying statistics and other modern techniques to extract meaning from data. So, what do you need to know to be a data scientist. Apart from a good programming background, you will need to know something about probability, statistics, inference and some computational mathematics.

The most popular language for doing Data science is the R programming language. Although many programmers do it in Python, of course you can do it in a general programming language. To me the best language or system to do Data science in is the Wolfram programming language, this is because R is more for statisticians who want to use a programming language to automate their skills, and with python which is also good the major problem you will face is integration of the tools you will want to use. I will get to this much later when I discuss Anaconda, the python mega package.

The reason I choose the Wolfram language for data science activities is that it has the lowest learning curve ever, meaning you don’t need to be a super mathematician or statistician to start doing some very powerful data science with the Wolfram language. The standard library also solves the big problem with Python which is integration of all the tools, in the Wolfram language you are dealing with functions and symbols that have been crafted to work together very seamlessly so you don’t have to convert between data conventions from the various tools you have to work with in other languages.

This strong integration is responsible for the ease and power of the Wolfram language for data science, another thing you will have free access to when doing data science with the Wolfram language is the most powerful visualization capacity ever. This is the ultimate data science super power. In order to discover patterns, trends and relationship in data, we cannot do without our eyes. Data is usually a jumble of numbers, text, sound or images, all represented as numbers on a computer, and since our human senses cannot make sense of these data items the way they are, especially when they are in large numbers we resort to visualization of some sort to make sense of the data.

No matter how powerful our mathematical tools are they will only operate on data to create more data and although in simple situations, we can simple calculate out a single figure that represents a quality of our data, with large amount of data we cannot get single figures that make much sense so we resort to some visual representation so that our most powerful tools for perception and analysis, our humble eyes can just pick out the pattern of interest.

In the section on computational systems, I refer you to a chapter in Stephen Wolfram's NKS book, where issues of perception and analysis are thoroughly dealt with. That chapter will be very valuable to you as a data scientist because fundamental issues concerning the nature of the kinds of data that we obtain in the real world will be pierced through and we will see what kind of data can be reduced to formulaic condensation and what kind cannot.

Statistics always prides itself with being able to model any kind of data using its formulas but in NKS you will find out that there are systems out there that produce data which cannot be summarized by any formula. This is not usually an issue of the lack of sophistication of the mathematics that we use but because of a phenomenon called computational irreducibility. Read the chapter on computational systems which is like a mini review of Stephen's book, but to delve into the issues I highlight will require some deep reading of the book, an activity that will be of great profit to you as a data scientist.

In my knowledge Wolfram language/Mathematica has the most powerful visualization system in any other package on earth as of now in the public domain. If you are new to data science or already established in the field and using other tools, I would advise that you take the Wolfram system for a ride. It will be worth your while.

* * *

16. BIG DATA

I have usually posed this question to many people, what is big data? Some people answer with the response: data that is really big! when I ask at what number of gigabytes or terabytes can we call data big? Some say at the terabyte level and above. I have not usually been comfortable with the word big data and like always, when I encounter some flashy looking tech name I suspect a marketing ploy!

In the tech industry and especially in computing you have to be able to distinguish between new technology and marketing hype. Just like the word "cloud computing", this word actually represents everything you are able to do with someone else computer while connected via the internet. Back in the days running software remotely was called "remote execution". So when I heard the word: cloud computing resounding in cyberspace, I was hesitant to get curious about it because I suspected it wasn't something fundamental. But whatever it is called we know it is useful to do things in the "cloud" rather than build your own infrastructure if you want to optimize cost.

There are so many technologies that I have not included in this guide, this is because I am either not familiar with them because there're emerging or they look like one of those hypes that we have around for a while and soon they are replaced by other stuff. The C programming language has been around since the 70s and is the stuff behind your operating systems and even used to design your so called high level languages, this is what I truly call a fundamental technology.

I am not saying you shouldn't learn hype technologies, because you might be required to use them in your job. I am only saying that you should learn the fundamentals before the hype. PHP is an example of a bad programming language alongside Visual basic, but PHP was used to build the hyper successful Facebook social media platform, although Facebook has cleaned up PHP with their own home brewed language called Hack, if you are required to learn PHP because you are tasked with maintaining some code written in it or you have to write some new stuff in it, that is after you have pleaded with your manager that it will be best to do it with another language, then learn PHP. But for your own good please don't make PHP your first programming language. If you find the path I have laid out tedious and you just want to do web stuff alone, then your first language in that case should be JavaScript. Not angular, react or any other library package, just raw JavaScript please. After your mind has been programmed to think in full JavaScript then learning the idiosyncrasies of different packages can be done.

When I ask marketing types when cloud computing was "invented" and by whom? Most have the guts of saying that as recent as 2015! But the truth is that Yahoo mail, an early web mail service used in 2001, is actually a cloud application. You managed your mail locally with your web browser while the rest of the service ran on a remote computer. Although "cloud computing" has evolved in recent times and more capabilities have been added it is not really a new thing.

Big data is that kind of technology that looks like a hype but is not really a hype. A search on the word big data will usually bring up some kind of web results that will go into the details of what it is and the tools that are used. But in this work, I am here to give you the inside scoop of things not to provide a manual. So the person I think I'm talking to, who is the one that really needs my advice is the new programmer, contemplating the idea of getting into programming and wondering what she should do. This work is wisdom on your path, in essence compressing your entire research into this field. Of course I know that the masters are watching, please your advice is welcomed: jackreeceejini at gmail dot com.

When you hear the word BIG DATA, what should come to your mind is data that is coming like a hurricane or torrential rainfall! Data that you cannot sit down, think about in a static form and build a model. Data that is coming in huge amounts and you have to understand this data on the fly! This is what Big data means to me. The kind of stuff that can produce big data are numerous but what you might already be familiar with are sensors! Yes, your sensors that monitor stuff from humidity to body temperature etc. are usually producing the kind of data can be handled with the tools that constitute the paradigm of Big data. Any form of data that is constant and steady and you have to make decisions about or understand without having to store them first, that is make them static, is big data, especially if it is coming in torrentially.

A service like Google receives millions of queries per second! That is big data. If you have a 1000 sensors monitoring maybe weather in different locations, recording data at millisecond intervals or less and sending you all that data constantly and you don't have the storage to store terabytes of data per hour and still want to make sense of that data, then that is big data!

The field of big data is a collection of tools that enable you work on these kinds of data sources. The programming paradigm that is mostly applied is called Map/Reduce programming. Research on this if it looks like the field you are likely to get into. One famous tool in the field is called Hadoop. I'm not into big data myself but merely included it in this work to help give you a starting point if you find out that what you want to do requires handling data this way and didn't know previously that your activities go under the umbrella of the word BIG DATA. Wolfram data drop simplifies handling big data, look into it.

* * *

17. INTERNET OF THINGS (IOT)

Another buzz word in the world these days and I thought I should mention it in this work for the new programmer because it is going to be with us for some time. Although personally I don’t think it’s the next big thing as many people predict but I know that sooner or later we will have billions of new kinds of devices that will need to communicate via the internet, things that are not personal computing devices, this is my definition of IOT.

If your refrigerator is connected to the internet, that is IOT. If your home temperature regulator has to communicate via the internet, that is IOT. Now imagine a world filled with those devices, communicating either with themselves or with their manufacturer or with you the user. Programming these devices is what defines the entire field of IOT programming.

IOT became possible because of the arrival of SoC (System On Chip) which are full featured computer systems on a single chip! The most famous of this is the RaspberryPI computer, that matchbox sized computer that can connect to a network and has a reasonable memory. This is another paradigm of programming and I hope I have given you linkers to explore this field. The programming languages you have learnt will enable you to work in this field whether its Big data or IOT or some other fancy new field that I have not mentioned because I either don’t know about it or I don’t think it’s important.

We use programming to control these IOT devices, no different from the way we use programming to control our desktop computers. The field of IOT and Big data are close related, but big data is just concerned with processing the data, while IOT programming is concerned with controlling the devices themselves. So you have some device with a bunch of sensors built by some manufacturer. The manufacturer provides a library in some programming language that enables you to make system calls to the device. Usually you will connect this device to a computer using a USB and use some development environment to access the device, the development environment enables you to programmatically access the device and thus gain control over it, making it do your wishes within the limits of its predesigned abilities. The libraries provide the capacity of the system, but your ingenuity is what will create some useful application out of a bunch of circuits. Programming is usually about such ingenuity.

Although these fields can come with their own programming libraries knowing the fundamentals of programming will help build the foundation that you require to penetrate these or any other fields, at least until quantum computers change what the fundamentals mean to us. So learn the fundamentals and you will be able to conquer any field or paradigm.

✻ ✻ ✻

18. CLOUD COMPUTING

Like I said earlier, when you are using a remote computer you are actually doing cloud computing. But you know marketing will always put a spin around technologies so that they can package it properly to appeal to top executives and the general public. In reality the bunch of technologies that constitute what we refer to as the cloud has been around longer. When the web mail systems starting propping up in the early 90s, this was already cloud computing.

As for modern cloud computing, there have been some extra bells and whistle added to cater to all kinds of business problems. I must say that as a programmer you should become deeply familiar with the cloud. In those days' programmers were tasked with building their own infrastructure and maintaining it themselves to run whatever service they had come up with and intended to share with the world. Although this is still a very useful skill to learn because as computer power increases it will still be useful to build some local system to run some service, and wait for it to scale before buying cloud resources, but you must also note that cloud resources are cheap. But nothing stops you from building your own infrastructure to do some kind of deep geek stuff that you can't run on the cloud maybe for security reasons.

As a programmer you might have some idea that you have whipped up code for and want to run it as soon as possible without building any infrastructure beyond your laptop, then you need to purchase some cloud service.

We have Amazon web service (AWS), Google cloud, Microsoft Azure and the Wolfram Cloud, etc. These are some of the most popular cloud providers and they don't vary much in price and quality of service.
Cloud providers offer certain classes of services. They are SAAS (Software as a Service), PAAS (Platform as a Service), IAAS (Infrastructure as a Service). With this three types

of service we are faced with another class of abstraction in computing. If you remember how the computer is built from layers of abstraction, from the hardware to assembler and eventually the high level language level, with the cloud we have 3 basic layers of abstraction abstracting the underlying computer resources, that is someone else's computers.

A cloud computing system is usually a pool of resources that you can have access to via a network connection, mostly remotely via the internet using tools like a web browser to access these resources. These services are hosted on what we call warehouse scale computers, a large pool of computing resources networked together to function as one and deliver services to remote users. These entire pool of computers can be accessed at three levels of abstraction and this is where the three classes of cloud services come from.

At the service level of IAAS you are usually giving access to barebones computing resources, disk (can include direct access to disk), memory and even network access. These services are usually packaged under virtual machines or containers and the program usually has to have a lot of knowledge of the underlying system to utilize it efficiently. This is analogous to the C programming language level of computing access in a regular computer.

Up next is PAAS. At this level you may not have full access to the underlying machine but you are presented with a development environment to do some programming. This is the most popular level at which programmers work. It is very cost effective to operate at this level because you only pay for what you use in terms of program execution or service utilization. At IAAS level, if you do not know what you are doing, you might run into some serious cost issues because things might run out of hand due to poor design strategies and also due to a weak understanding of the underlying platform offered by the cloud provider. If you want to work at the IAAS service level, then you should go for containers rather than virtual machines as a newbie, you run less chance of messing things up this way. But for the beginning programmer always start from the PAAS level, as this is what you need to start writing and running things on the cloud.

The Wolfram Cloud: If you want to develop for the Wolfram language then it will be a good thing to start at the Wolfram Cloud, you can even develop all your code in a web browser even if you don't have access to any Wolfram product locally installed. If you do have a Wolfram product installed, then it will be best to use that as you can take advantage of your local computing resources as well as your cloud resources. A good work flow with the Wolfram desktop system is to develop locally and offer your service on the Wolfram cloud.

At the highest level of the cloud abstraction is the SAAS level. The most popular service at this level is an email service like Gmail that you might already know. Many non-programmers do not know that when they are using webmail they are already using the cloud – and think that the cloud is only a recent invention, well thanks to clever marketing. As a programmer your point of concern should be that you might eventually want to offer some of your applications that you develop as a service running on the cloud don't take this name special, anytime you hear of the "cloud", just imagine using someone's computer over an internet connection.

You might have to develop on the cloud and then run your apps on that same cloud, or you might develop locally, test your service locally and then deploy a live version on the cloud. The most important thing to think of is that the cloud offers robustness, you might be too early in programming to build your own infrastructure. But if you have the resources to do this, then go ahead and build yours.

✻ ✻ ✻

19. WEB DESIGNER/WEB DEVELOPER

Many new programmers or the general public at large often confuse the terms web designer and web developer. The most important thing to know as a programmer is that a web designer is usually concerned with issues of presentation like graphics, placement of elements on a page, choosing appropriate colors and designing the general look and feel of a web page/site or web application. The tools of a web designer include graphic tools like Photoshop or InkScape, WYSIWYG tools like Dreamweaver for generating page layouts and the associated HTML, CSS authoring tools for controlling styling, etc.

The Web developer on the other hand is concerned mostly with writing code in some programming language that adds functionality to the web application, and also code that runs at the backend. While the web designer is more of a digital artist that uses digital tools to produce the interface elements, the web developer is more of a programmer that works in languages like JavaScript, PHP, etc. Most websites have to connect to some database to obtain or store data. While the web designer controls the elements of display of the data, if it has to be displayed, the web developer is the one responsible for writing those queries.

So what really matters are the level of abstraction they work in. While the web designer is concerned with the look and feel of a web site/application, the topmost layer of the abstraction, the web developer is concerned with the functionality of the whole system i.e. working at many more layers of abstraction of the base hardware. The web designer is working at the highest level of abstraction possible in the entire system, the level of the user interface, while the web developer works mostly on the lower level, the backend stuff. But the effects of the web developer actions are felt by the user.

✻ ✻ ✻

20. CODER/ PROGRAMMER/ DEVELOPER/ SOFTWARE ENGINEER

This is another categorization used by people in defining three classes of people that work in the software industry. Although it is not realistic because everybody writing code is a programmer sometimes it's a kind of industry lingo used to differentiate people based on their skill level. There is no standard definition for these classes of people who work in the software industry and everywhere you go you will find differing definitions. I have been in this industry for a while so I think I have the right to define what these classes of people do.

Coder:

This is more like someone who has learnt some basic markup, or JavaScript, etc. To me this is the lowest skill level, the type you get after doing a course that sounds like: Learn JavaScript or Java in 24 hours. This is the entry level kind of individual with very little knowledge of the entire field of programming. When you get your hands on your first programming book and start entering the examples in some kind of editor and running the code, you have become a coder. You might not be able to solve many real world problems beyond the toy problems that you have been exposed to in your books. There is nothing to be ashamed of in being called a coder, every programmer has passed through this stage. The problem is if you stay a coder, computer programming is so exciting that you don't want to remain at this level, except you find out that it is not the thing for you. But anyway even if you have been briefly exposed to programming at the level of being a coder, the experience will last you a long time.

Programmer:

This is someone who has learnt one language to a very high level. This person is the kind of person that reads programming books ascribed to as Intermediate. He has gone beyond syntax, solved some difficult exercises in some programming books and can now start applying his knowledge to find the solutions to real world task. This kind of person is not really much interested in developing software for other people to use so they are not usually concerned with user interface issues. They want to use programming as a tool to solve some problem that was computational and made them get into programming language. This people usually just learn one general programming language, their contributions to the open source community is usually in the form of libraries for other programmers to use. Most programmers would run code on the command line, they are usually interested in solving their problems, which sometimes might be the problems of others. They might choose to share their code of just have thousands of pages of code lying on their computers without good filenames.

Developer:

This is a programmer that decides to develop software that will be used by programmers and non-programmers. This kind of person usually has software as a product in his mind. She works in many languages, some general and some highly specialized, knitting smaller applications written in these languages into some larger highly featured software product. They might have some knowledge of computational math and algorithms, but usually very narrow as applies to the kind of work they do. The could be self-taught or computer science grads who are taking up a job as a developer for some company or their own. This is usually the kind of people you think about when thinking of people who work in the software industry. They read books tagged Intermediate and can even go Advanced. Most developers work on varying types of projects, some of the time they are web developers, desktop developers or Mobile developers. They usually have a large collection of tools they have mastered which enables them to their work efficiently. The key note is that a developer knows more than one programming language and develops stuff that is targeted at an audience usually consisting of the general non-programming public.

Software engineer:

There is only a subtle difference between developers and software engineers. Many people actually interchange these designations one with the other, but since we are talking unofficial lingo here, we could go further to show the subtle difference.

A software engineer is someone that has mastered the entire field of computing. They might be highly educated in a formal sense with a PhD, or without a formal computer science education but highly skill through many years of software development and mastering many fields of computing. While a developer usually works with tools that are created by others and might not be able to create robust tools, a software engineer can design a computer language, write compilers, work comfortably with assembler write and maintain large complicated bodies of software. She could also be highly skilled in math; seldom will you find a good software engineer without good math skills. They think algorithmically and are usually the kind of people that write the "standard" or fastest algorithms available. You don't need to possess genius level skills to call yourself a software engineer. A nice way to look at each of these classes from coder, to programmer, developer and software engineer is to see it as a categorization of interest. You can call yourself a software engineer or developer if you feel that you are on that path. Like I said earlier the path of mastery is endless, so on the spectrum of software engineers, you have aspirants, beginners, intermediate and advanced software engineers.

So this is how you should think of these classes, rather than think that there is a hard fixed level of skill that you must reach to be called either of them, think in terms of what your computing curriculum looks like. If it contains skills that are possessed by software engineers, then you are an aspiring software engineer. When you start hitting your study milestones, your confidence as a Software engineer will start increasing.
So for those who want to get into the software industry, set your sights to where you want to reach and then you can know what designation fits you.

21. DESIGNING SOFTWARE VS. HACKING IT OUT

When writing software there are two approaches commonly used, designing or hacking it out. Developing software from some kind of formal specification is usually what is referred to as designing software, while hacking it out is where we have some subjective knowledge of what we want to build and start hacking out the beast one functional piece at a time.

In a scenario where we are working to extend an existing system, if we seek to use the path of formal specifications, we write out in some detail a specification of how we want to extend the system, be it adding new features of performing a major rewrite. In an industry setting where you are working on code that must be carefully crafted to fit in with the company's goals and criteria and of course without much novelty, then following a formally laid out specification is the best path to take. In practice this specification will be handed to you from someone in the management side of the business who understands business requirements and needs you to implement solutions in code.

In another scenario we are not trying to extend some existing piece of software but are given a high level goal of what we are to design, like the request: write a custom web browser that does this and that. We write some kind of specification which cannot be in too much detail because we are writing something from the scratch. Major requirements to adhere to might be pointed out but not to great detail. In this scenario we use a mix of hacking out solutions and following a formal specification to guide our activities. In a simple project whose results can be well anticipated because it is not something new but a customization of something existing like the custom browser we talked about then much novelty is not required and the project will be biased towards following the specification. In some situations, there might be parts of the software that require extensive hacking in an unplanned manner. If a solution to the problem, exist in the open source community then it will be better to extend or compress that solution to fit in with your needs rather than reinvent the wheel.

If it is a novel problem, then guided by the product requirements we go ahead writing one primitive functionality after the other, stitching them together to produce higher level components until we solve the business problems we were faced.

The keynote for designing software is working from a formal software specification. What is a software specification? This is a document usually written by the people requesting the software that contains what the software is supposed to do. It could be a very high level specification containing only the highest level features that are needed, giving the programmer total freedom on how to produce these features, or it could be a low level specification, giving details like the function names, class names, security policies and recommendations on how to implement them.

In the low level specification scenario, the programmer is not really at liberty to choose how he wants to do things and must follow the specification to the word. There are other details that could be included in a software specification documents which we don't need to go into at the moment. Our goal is to talk about the two broad methodologies commonly used for writing software.

Software specifications are mostly used in formal business environment like in big companies, etc. most of the time some programmers choose to design specification for themselves to guide their development effort, this is a good practice and can lead to quicker development times. Specifications are usually only good if they don't change too often, because sometimes after going through a long process of writing code to satisfy some specification, then some modification to the specification will require that you destroy some code you have already painstakingly written and perform all kinds of adjustments on your project.

Specifications are also good in solving some well-established problem, for instance if you are designing an ecommerce site, then you are not inventing much and a good well written specification will guide you and make sure you are not missing any feature or business requirement. A specification is like the outline of a book. It is usually easier to write a book when there is some kind of outline to guide you, all you need to do is write on the topics you have already listed.

Hacking out code is a method suitable for finding the solution to novel problems for which the structure of the solution cannot be easily pre-anticipated. This is like the computational thinking we talked about earlier where you are faced with a novel computational problem and you are trying to code out a solution. You cannot write a detailed specification of how the solution should look like because you are probing new grounds, so you start fluidly whipping out code allowing your creativity to flow unhindered so that you can access the entire solution space in your mind. You iteratively produce solution after solution, quickly testing them out and discarding them if they turn out to be bad of inefficient solutions. This is the path of research and discovery.

Sometimes hacking out a solution is good when you are trying to prototype a new product. You don't need to go through the process of writing out some huge official document first. You have an idea, some programming skills and you want to design something, you just start with the first function that comes to mind. Then to another function, then at a certain stage you decide you need to make a certain group of functions a package that you can import into other parts of the program, or you find that some functions operate on the same data and are highly related so you build a class out of it, etc. These are the activities that encompass hacking out code. It's like getting a cutlass and hacking out a path in the jungle. Your GPS is showing your current location, your compass is pointing in a particular direction and you just go out facing the desired direction and start hacking the bushes. The only guide in your head is the general direction you are going to not any particular well-trodden path.

Analogously, a software specification is a detailed survey of the jungle, usually well explored by some other people who have left documents of their activity. You know that the goal has been reached before but you want to go there yourself. So you use as much knowledge as possible to design your method for reaching that same goal.

When hacking out a piece of software we can usually come up with some kind of informal outline to guide our activities if we need to, maybe to avoid us derailing along other paths. Or we can just allow the activity to flow, derailing to other paths we may find interesting. Although research into the solution of a problem can be systematized and this systematization can prove valuable in increasing efficiency, too much systematization of research, that is adding too much structure to our research activities can actually limit serendipity. The take home is that hacking out software is good when you are researching some new field or trying to gain more insight into an already existing solution or in general just prototyping new software.

* * *

22. EFFICIENT PROGRAMMING

When you initially venture out on the path of programming, you will have to at least learn the language syntax from some books, and then like I said previously the next thing for you would be getting your hands on some code and start reading and writing small programs, which initially could be fragments of the larger program you seek to gain a better understanding of. There is another addition to this path I want to add here, not because I forgot but because it now necessary to reveal further. By the time you become cross the river from being a newbie programmer to being able to read books for the intermediate level, you will now be exposed to a whole new flavor of programming books.

Reading code and writing code will help a great deal, but there are intermediate books that do not teach programming language syntax. These books assume you have a certain understanding of the language syntax of a particular programming language and go ahead to teach skills and techniques that you might encounter in the real world, or that have specific domain applications. It is not the highly domain specific books I am concerned with here, but the more general ones that have skills that could aid you on your path towards mastery.

You might encounter books titled: High performance {insert a programming language here, e.g. Wolfram, C, Python, etc.} programming; Memory management in {programming language}; Writing {programming language} libraries, etc. These kinds of books go into specific areas of programming that are general and not domain specific. If you have your hands on good code you might encounter good programming practices that encompass the topic matter of these books, but you might not be conscious of it. You could be reading the Linux kernel and come across powerful memory management routines and absorb them and use the techniques in your own programs but reading a good book about memory management will make you conscious of what you have experienced and point out new dimensions of understanding that may have been closed to you.

Sometimes you might not have encountered any advanced techniques in your programming but are anticipating writing software that you need to have advanced highly efficient solutions to certain problems. In that case then chasing down books with such titles might give you some heads up, but this heads up knowledge must be complemented with some real world code. Any good book on these items will point you in the direction of some powerful open source code repositories that contain good code in addition to the toy examples in the book. If you are lucky, then the author of the most powerful open/closed source implementation of the kind of library you are looking for is actually the author of the book, then you get a unique opportunity to learn from the master himself.

Beyond the intermediate level books there are the ones with the designation: Advanced. You need not be scared of these kinds of books if you already are able to read the Intermediate level books. Most of the time the only reason for such designation is because the author assumes that you have not only studied the language syntax, which is level one stuff, but that you have actually worked with some real world code and are at least aware of certain things working programmers are aware of. The reason authors write advanced books is because they want to skip over a lot of basic stuff and just concentrate on some subject matter, usually at great depth and length.

So reading intermediate and advanced books in addition to reading code from good sources and writing code fragments and even full blown projects is a great way to start gradually developing the sense needed to detect efficient code from slow code. Although with a good profiler you can discover bottleneck in programs and even attempt to fix it, as a beginning programmer, except you are extremely talented you might not be able to get an efficient fix for such programs. Writing efficient code is a skill that develops as you keep on iteratively improving your programming skills through the paths I suggested earlier. You have to expose yourself to a lot of code and books and then you will be able to write efficient stuff with time and practice.

When you're in your newbie days, reading books dealing basically with syntax and filled with lots of toy examples and exercises, you are not really expected to write really efficient programs. But sometimes new programmers are hit with shock when they first experience real world code. This is code mostly written to solve some problems in the most efficient manner possible. Although readability is required for even the most advanced code, because other programmers may have to work on it too. It is certainly not written to be as descriptive as possible for the newbie.

Such encounters with advanced code usually discourages some new programmers from continuing in the programming field. Imagine a web service like Instagram, the large online photo sharing app. Now let us think of some kind of process they might be using internally, even though we don't have access to their code base and we are not going to think about actual programming code.

Imagine that you were a programmer tasked with writing an efficient program for finding photos that a particular user would be interested in and displaying it in the search page of the app. You will have to take into consideration what the user likes in general, usually gotten from the user activity and also the nature of the kind of photos the user posts and even the nature of the user's comments and what they comment on. This might look simple if you are doing it for a hundred users, your brain will definitely come up with some kind of naïve method for finding the solution and come up with some naïve code which appears to work really well on the laptop, but then extend it to a thousand users and then we notice a slight slowdown.

If you are not using a profiling tool to measure your performance, your natural perception might not even record a slowdown in performance. As you scale into a million users, then your naïve implementation will show definite slowdowns, and this is running locally on your computer without any extra overhead of network connections. Now imagine that Instagram has over a billion users! Now you know that you need an efficient solution to this problem.

This is where efficient code, not just code, becomes important. Solutions that work on a small scale can fail at larger scales, sometimes looking like they were not solutions at all in the first place. This is why learning just coding is not the way to become a programmer. And modern coding schools, without the goal of growing the programmer to be eventually able to write efficient code are just teaching programming syntax.

Whenever I talk to aspiring programmers, who are usually motivated by the success of the big software companies, I tell them that you have to learn computational math; algorithms and then programming. They are usually saddened by this news. They expected I will just point out a programming language they should learn and that will be all, but I point out some unsavory extras.

The reason I show the full path is because I am not just interested in them being "coders", I am thinking of their long term future in the programming industry. If you want to write a Facebook, then you must go beyond coding and into the real art of software engineering. It takes some time and effort but it is possible, if you put in the needed time. There are more difficult things than programming and people have excelled in it. Except you just want to experience programming, then you can just pick a language like the Wolfram language and catch some fun. But if you display any great ambition then you have to go through the rigors of some full computer science experience, either formally or through self-study.

In the Instagram example above, the solution will actually require that you have been grounded in algorithm analysis. People have worked on more abstract representations of the problem and come up with intelligent algorithms that solve the problem in the shortest amount of time. You can dig the open source world for libraries that solve that kind of stuff. High quality libraries are usually written in the most advanced form possible, although readable with good documentation sometimes, the code is still advanced for a newbie without a lot of experience in programming.

If you don't find the solution in plain code, then you will have to dig for some research paper where the algorithm must exist. In order to understand this research paper, you need to be familiar with the lingo of the algorithms world, which some study of algorithms will ground you in. if you're are lucky you might find your solution in pseudocode, which you can easily produce code in some specific programming language. But you might also find your code written in mathematical notation, and without knowledge of computational math, you will not be able to understand this math, talk less of implementing it in some programming language. Like I said in an earlier example the page rank algorithm in the 1998 Google paper was actually in mathematical notation. I hope I have been able to convince you that to write efficient code you need knowledge of computational math, algorithms and programming.

You don't need to be a master mathematician or algorithms specialist to be able to utilize the enormous body research out there. If you are scared of math, you need not be. Just make some attempt at studying a good computational math book like: Concrete mathematics by Donald Knuth et al. or even enrolling for the free MIT OCW course, Mathematics for Computer Science (6.042). You might not be able to write solid expressions in mathematical notation, but with some time you will understand things like Summation, Products, some probability, asymptotics and stuff like Hypergeometric functions which come up in all kinds of computer programming problems. Just get into it and learn the lingo, there are people who are really good at this stuff and have spent sufficient time mastering it. Most of their work is out there online for free or for a small fee by subscribing into some journal. Your purpose should be getting into the field long enough to understand some research paper on algorithms.

Apart from Concrete math and the OCW course you can find numerous sources online to learn this stuff. For free or for a small fee. Computational math is usually called Discrete math so when you see it you don't get confused.

Efficient programming is more of an art than a science, although hard research usually produces the most advanced algorithms for doing all kinds of things, you learn it by using the apprentice mindset. It's not something you can learn of a book, but by reading code of masters and emulating them. You can research and implement the already preexisting solution to some problem you are facing, but when faced with some new problem for which there is no existing solution or you are not satisfied with any existing solutions, then you enter the scary realm of pure research! This is where having followed the path of the apprentice will yield its fruit, armed with your knowledge and experience, you will delve into this realm with all the confidence knowing that you can always do better.

* * *

23. DESIGN PATTERNS

These are the conventional ways of accomplishing certain tasks in programming that have become standard. It is an aspect of efficient programming but I choose to explain it separately in another section, because it is much more narrow and applies mostly to programming. This is actually a misnomer because every field has its design patterns. Whenever a field matures design patterns seem to appear. My focus here is on design patterns in programming.

Design patterns look sometimes like programming styles but it's really not a stylistic issue we are trying to deal with, it is a matter of writing recognizable code that works optimally. Certain problems we encounter in programming have been solved in a particular way and it is that particular way that most people have come to accept, because it makes for efficient code, readability and reusability.

A design pattern can either be programming language specific or abstract in nature. A programming language specific design pattern is one that works only in a certain programming language. This can be a convention for applying cleanup operations after opening a file, or wrapping stuff up in try…finally clause to free up resources even though a particular operation fails, the finally section, saying at all cost, no matter what happens do this or that.

The language definition and syntax provide the raw bricks, but the way they are used usually develops conventions that end up as standard design patterns every programmer is expected to use, but it is not enforced. It is a way people have come to do things, and I know the non-conformist might object. If you don't do things this way nobody will enforce it on you but your code will be distasteful to others, except by some rare luck you have people adopt your way of doing things and make it a convention also.
Programming is a difficult activity and no one wants the extra overhead of trying to understand a new way to do certain fundamental task for which there is already some conventional powerful solution and methodology existing.

Abstract design patterns are not language specific but can be implemented in many languages. These are conventions for doing certain task that are more general to programming and not just stuff that a particular language requires. For example, there are ways of doing parallel programming, which requires that some resource be locked by a particular process or thread which is operating on it. Or there are certain conventions which govern how a process should access network resources to reduce overhead. These are not algorithms per se but powerful and efficient conventions that people have grown used to. And when writing code that needs to do such general tasks people expect your code to contain code written in some pattern that they can readily digest.

One can stretch the field of design patterns to even include naming conventions for classes and functions, but for this discussion lets limit it to the methodologies of performing certain programming tasks. Although following a good naming convention can make your code tastier to a wider class of programmers. There is the famous Tabs vs. Spaces war in the programming world, where one group of programmers insist that code should Tabbed, that is using the Tab key on the keyboard to do indentation while the other group swear by spaces. This is simply a matter of which kind of indentation makes code readable or more pleasant to one group or the other. As simple as the issue sounds it is one that is filled with emotion with each group trying to rival the other in pushing forward their agenda. It’s actually one of those fun things that make programmers interesting people.

* * *

24. DEBUGGING

As a program grows, the possibility of bugs increase. Debugging is a skill/art that must be learnt and eventually mastered by all programmers. Although your hello world code will be seen to contain no bug, it is rare to write a bug free one thousand liners.

There are different kinds of bugs, a bug is an error in a program. The first kind of bug you will encounter is syntax errors. This is usually caught by the compiler or interpreter you are using, with valuable information as to where the bug or bugs occurred. By reading the error message we can go back to the source code and find them for correction. Most programmers don't consider syntax errors as bugs, but for the beginning programmer you will experience a lot of this kinds of bugs.

The next level of bugs are the semantic bugs, now these are what we call real bugs because your program compiles but doesn't produce the expected output. This is the first and I think the greatest source of discomfort in the entire field of programming. Sometimes debugging a program is much harder than writing one.

When faced with a semantic error there are two pathways we can follow: 1. If it is a small program then we can eye ball it, inserting as many print commands as are necessary to give us an idea of the intermediate state of variables. 2. If it's a large program? use a debugger. What is a small program? The answer will depend on the programming language used. In a programming language like Java, 1000 lines of code is still a small program, while in a compact programming language like Wolfram 100 lines is already a large program.

Also what is a large or small program will depend on your experience as a programmer. If you think what you have is a small program, then debugging for semantic errors can by means of eye balling. From the error, you simply scan a region of the program where you think the error must have originated from and try to fix it. This might not be feasible for large multi-file projects or even large single file projects, and this is where a software system called a debugger comes in.

Beyond semantic errors are a higher level of errors which I call usability dysfunction, or feature error. A semantic error is when a program gives output that you do not expect and because it is not what you designed into the program. This is difficult enough to spot and sometimes correct but usability dysfunctions are even more difficult to spot. This subtle kind of errors exist in all programs and are usually not encountered during normal program execution. These errors arise in unusual usage scenarios that the programmer did not pre-contemplate.

A program is written and runs perfectly, passes all tests that the programmer conceives, yet when it gets into the real world somebody discovers an exploit! These are the subtlest of bugs and sometimes bypass the most carefully crafted tests. These kinds of bugs have to do with the randomness of the world. With good and careful design and a lot of testing we can produce very powerful and robust programs. But somehow, these programs get into the world and some user finds an exploit. These exploits can be found even in programs that don't have their source code exposed. People are clever and can perform weird "attacks" at your program that reveal weaknesses.

Contrary to your expectations, open source software has fewer bugs than closed source software. Because the code is open, all kinds of people can break it in all sorts of ways, doing test beyond what any closed team can come up with even with the most intelligent people in a closed team. The people who put open source software through its paces usually come up with ways of fixing the holes in the product, making future releases more robust to increasingly greater attacks.

So when thinking of constructing some new product, it is good to use tools and libraries that are open source even if your final product has to be closed source for commercial reasons. It reduces the chance of your software having this third kind of bugs and increases the robustness of your product.

The world of debugging is as wild as the entire world of programming itself. There are entire books on debugging which treat debugging either as an abstract stuff or go in-depth into how to do debugging in a particular programming language or using a certain set of tools.

In summary you will spend more time debugging, than actually writing new programs and I advise that you put in some effort into mastering this skill, no less that what you would put into the actually practice of programming. It will save you many sleepless nights ahead.

✻ ✻ ✻

25.TESTING

When you write a function write a test for that function. A test is basically a short program using the function/classes that you have created, making sure that certain assertions about what the functions do are true. When hacking away at some research you might not need to write tests because you mind is in creative zone. As a programmer you will write a lot of throwaway code, code that is written once and never used. Most people don't burden themselves with having to write tests for this kind of code but some do.

When you are writing any piece of code that you will use at a later date or will be used by other programmers, be sure to include tests. This is the only way to assure both yourself and others that your code does what it does. Testing programs is usually called unit testing in programming parlance.

There are libraries and frameworks for doing testing. Most of these libraries are specific to the programming language you want to use. Most programming languages have some built in frameworks for doing some kind of tests using statement like Assert, etc. explore what you have in yours and if you find this insufficient it will be wise to explore the libraries out there. There are also good books on testing out there and it will do you great good to avail yourself of some. Testing like most other specialized skills in programming is an art. There are people who have taken this art to a high level and have written books and articles on the topic. Explore them to start your own journey towards mastery.

* * *

26. DOCUMENTATION

Apart from code, we have documentation of the code. The documentation expresses in natural language what the code is meant to do. This assist you to gain an understanding of the code without having first of all read the code. In the section on complexity I will go into some detail about how documentation can help you in navigating complex programming projects. Documentation writing is a skill that can be honed through reading good documentation. There are also books on how to write good documentation out there that will aid you in this skill.

At the project level, the read me file usually contains the entire goal of the project, if you have the source code. This is usually short or long depending on how much explanation the project maintainers want to do. In every other folder containing files or other project folders, there could be other read me files going into some explanation of the folder and all the elements it contains.

At the file level, we can have a major documentation at the beginning of the file. This is usually some long text bounded by the commenting mechanism of the language you are using. This long documentation goes in-depth into the purpose of the file and what it offers.

Beside some global structures like constants and variable declarations, etc. we could have some single line documentation or comments about what the constant or variable provides, etc.

After class definition there is usually some documentation of the purpose of the class and after method definitions, we have some more about the method. In the code body inside a method or function, there is usually some inline documentation.

It is important to document your code because other people will gain insight of what you are about to code and even if they don’t understand your implementation, at least they will have an idea of what the code is meant to do. Documenting your code is not just good because other people are going to read it, it is also good for you the coder. It is common in programming to write code that you don’t understand later or after month. If your code was well documented in natural language, then you can gain some insight as to what the code is doing even if the implementation might look hairy now.

Documentations also carry one special type of comment called: “to do”, it is here that programmers put their future implementation goals. So when the programmer or someone else charged with maintaining the code come across this to dos, it will enable them know the future direction of the code they are working on and whether to implement them.

✻ ✻ ✻

27. CONQUERING LARGE CODE BASES

When you do your regular coding, you are usually either writing a few thousand liners of your own conjuring or working on some other few thousand liners. But there are other monsters out there, the one million and above liners. How do you deal with such monsters? Initially when I started out in programming I thought that one must memorize every piece of code one is confronted with.

This is an outcrop of the kind of formal education we have, which is basically a memory testing competition. Those who succeed very well in academics usually have gigantic memories for memorizing the "facts". So when I got into programming, it thought I have to memorize code like I have been thought to memorize words and formulas in the world of formal education. It is totally impractical to memorize a million lines of code, except you are Rain man (in the movie titled with that name).

I once asked an advanced programmer, what do I need to know to be able to understand a million lines of code? And my question was dismissed with: you will never need to work on such a large code base in your life. Such a bad answer, as he doesn't know the heights I want reach in my career. Even if you work in some large company, you will be presented with a large code base, but seldom are you responsible for all of it. There will be people working on all other parts of that program and you might be personally responsible for a few tens of thousands. So there is an element of truth in what that guy said, but what if you are a startup person tasked with designing some service where the code will run into millions.

If you are just a small part of the team, then there will be no need for you to master the entire code base, you will still be working on a small part. But what if you are the owner of the project? Then you will have to master your own responsibilities and at least have an idea of what other people are doing.

Once you have imported some million line or more project into your development environment like visual studio, you should have some complexity management understanding under your belt.

Complexity management is a huge skill and needs an entire book to explain in detail, but there is the fundamental notion that can be stated in a single sentence. Complexity management is the art of learning to ignore some details for now. As a programmer seeking to understand a large code base, you must learn to ignore certain things at the moment while focusing on other things. This is an underestimated capability. In the early days of my programming journey I found it very hard to understand multi file projects, because I did not have this ability.

Our academic training makes us try to memorize too much at once, and makes us focus on details that might be unnecessary at the moment. You must learn to look at the biggest possible picture and then drill down to all the specific details as need be, it's kind of a top down approach to understanding things.

Understanding the details to the point of even pulling the atoms apart on your computer is important, but at later stage. If you try to understand one tiny part and then proceed to one tiny detail you will not have an overarching mental model of the entire program, and except you are some savant, you will tend to forget most of the details you have encountered.

Depth first/ Breadth first:

Curiosity usually proceeds in a depth first manner, drilling and drilling even deeper into the core of things. When trying to understand massive and complicated structures, that method will only get you lost in the details. The way to proceed should be breadth first, staring from the outside of things and progressively drilling down layers and layers deep.

Unravelling containment hierarchy:

A programming project is usually a nested containment of files and folders. Its structure can be explored like a graph in a good IDE. The leaves of this graph is usually the file level. To be aware of the entire project, one will have to explore the folder structure, this is the first level of complexity. In a well-designed project, the folder structure will actually contain a lot of information about the structure of the entire program.

Navigate the entire folder structure till you get to the leaf, which is the file. Do not open the files. All you want to see for now is how the entire structure of files and folders is.

Hard memorization vs Familiarization:

From academics you might be used to hard memorization using tools like mnemonics or other memorization systems you know. If you have a good memory, then good for you! You can memorize the entire folder and file structure of a program by heart. This will give you a good grasp of how the project is structured. If you don't have a perfect memory like me, then you will need to familiarize. Your super memory might eventually fail you when we add extra layers of complexity to our understanding, especially class hierarchy. If you want to understand a large project do not try to hard-memorize it. Rather try to be familiar by going over certain things over and over again, until you are comfortable with them, even though you might not have a perfect memory of them.

If you try to use a memory system to memorize an entire code base, you will spend 200% more time mastering the code base. The first 100% will be spent encoding things into your system, the next 100% will be spent using your system. Simply familiarizing yourself with the code base will be time well spent.

So your first step in exploring a code base is going through the file structure till you are comfortable with where things are. During this process you will have a sense of where the root of the program is. In a single file program, the root of the program is usually called the main function found in many compiled languages. This is the function or class that pulls together all the resources specified in other files together to make the program run.

In a GUI the logic of the main loop of the program is usually placed in a central location on this single file, class or function. Except the project is a library, designed to be used by other programmers in other projects, many projects usually contain this file.

Luckily the main file is usually at the highest level of the tree so it is easily accessible. But you shouldn't rush at it. You should explore all other folders in the project to see what they contain, familiarizing yourself with the file/folder names and their containment structure that is, what is in what. After getting used to the project from that high point of view, then you should seek the main file, this is the file that ties everything together.

The main file of a project need not call everything in the project directly, but it does usually reference stuff that reference other stuff and so on. When you open the main file, I will advise that you find a place in your IDEs menu where you can see something like “Fold all”. A good IDE will allow you to fold code, that is where you have function or class definitions you can fold the code so that only the function and class names are showing. There are non-foldable items like global single line definitions but use the Fold all functionality to fold all possible foldable.

During the folding process some IDEs usually leave just the documentation and the class/method and function names available. This is the next level of understanding you have to gain. The path is to drill from the outermost files to the innermost.

When you have folded the code in the IDE, explore the file tree in a kind of breadth first method by going through the files in the upper most layers first without drilling too deep into the first major folders, this is just one method. In another method we use some kind of depth first method, where starting from the outmost project layers, after going through the first files, that are at that level, usually in no enclosing folder, we pick one folder, explore it properly to its depths and then back out to the upper most layer to explore other folders from the topmost layer.

This is a highly structured kind of searching, if you are not a fan of too much structure then you don’t need to follow this hard breadth first of depth first thing, constantly asking if yourself if you are doing the right thing etc. the only structure you need to know is that at this stage you must fold all the code in all the files you are going to explore. This is of course after you must have familiarized yourself with the file and folder structure of the entire project, that is folder/filenames and some knowledge about their nesting.

When you open a file to explore and must have folded it, all you do now is read the documentation in the file and the class/method or function names and any other names that do not fold. Make sure you say fold all. This is all you will be needing in terms of “code reading for now”, code folding is the most important thing you should do. Explore the entire project in the best way you see fit, according to what might catch your eye if you don’t want to follow a structured search.

When you have gone through a portion or all of the project in this manner, if you still intend to explore the entire code base then you can go through it all. But if you find out after your initial explorations that you only need to explore just a fraction of the code, then you proceed to the next level.

In the next level you actually start looking at some implementation, there is no hard structure to follow so I will not be telling you which file you should read or how. By now you have gotten sufficiently familiar with the entire project and you know which direction you want to go, and at what level of detail you want to master the code.

You may find out that you want to read the entire code – if this is the path you choose, then go ahead and unfold the files and start reading the implementations. If you find out that only a certain aspect of the code interest you then go to those sections, unfold and start reading the implementation.

You could choose to unfold all at once, or unfold only one function or class after the other. Whichever method you use will depend on your needs. Some people prefer being faced with the entire complexity at once while some prefer it in stages, so whichever kind of programmer you are will determine if you unfold all at once or in steps.

When you are going through the implementation, the level of understanding you want to gain will also determine how much detail you want to go with each piece of code. If just you are just merely looking at the code to see how certain things were implemented, then your read-through will be faster. If on the other hand detailed understanding is what you want, then you will have to spend time understanding the code in greater detail.

You will find that you easily understand some code, but some will pose so much difficulty you might choose to write a mini program to capture the essence of certain things you want to understand. This mini program may be easy to write if there are little or no dependencies with other pieces of code or external libraries, but it might be difficult if there are so many dependencies with other code or external libraries. If it's just code in a different file of the same project, then it might make sense to open the other file/s to find out the roots of the currently investigated piece of code. Sometimes you might have a large network of dependencies that you have to understand before understanding the code you are working with. This is part of the difficulty of the activity of reading code.

So it might not be easy to write a mini program, or even write one at all if the dependencies are too much, especially when they are tied to external libraries for which you don't have the source code. Mini programs are easily implemented when your goal is to gain deeper understanding of algorithms expressed in code, that is when you want to analyze some algorithm completely tearing out tiny fragments of code and writing programs to experiment with them till you gain deep understanding.

When writing a mini program is not possible and you have gone through the chain of dependencies in the file you are working on, ending up in the programming language documentation and even exploring the interfaces of the external libraries, if you still do not gain an understanding, then I advise you skip it and move unto some other piece of code. Spend only as much time as is necessary in a code fragment and when you realize that you are spending too much time forget about it and move on. If later you still find yourself feeling that you should go to that code again then do so.

When reading the code remember that you are not expected to memorize it. There is no examination here on re-writing code you are reading, what you are trying to gain is an understanding of how the code does what it does. It's like reading a book, it's not always about the words, although sometimes there might be some captivating sentence you wish to explore further, it's about the general story and the overall theme not the words. Same applies to code reading. In my earlier days I was terrified with code reading because I thought I had to memorize to prove my understanding but as I have advanced I realize that all the understanding that is expected of you sometimes is to be able to edit the code, maybe to change certain things, or to add extra features. Sometimes you might be faced with debugging someone's code, but I don't think in your early days of programming this kind of task would come your way. But if it does and if it's a performance thing, then run a profiler don't just eyeball the code.

If it's a deeper kind of bug, maybe a poorly implemented algorithm, then you have to do some research and implement a better version. But it is rare to encounter another programmer's semantic error in production code, except maybe a friend asks you to help debug their code.
Basically as a programmer you should welcome challenges at any level of your career, it doesn't matter if you just finished your computer science degree or maybe just completed the reading of your first programming book. If you are presented with a challenge in a field of interest don't run away saying: that is too advanced for me. Doing research to solve this kinds of problems can lead you to areas in the world of technology that you may not have thought of approaching on your own.

If you are presented with some complicated PhD grade algorithm that doesn't seem to work well and you are called up to fix it, muscle up and start working at it. Do some research on the problem, there is this thing called the internet where you can get information on almost anything. Thank your challenger for giving you this challenge because it is usually a better path to gaining knowledge than reading a ton of disconnected books, although the path of books is not a bad thing as some random piece of knowledge gained somewhere can always come up handy at a later time.

In this way out of raw ingenuity, you can come up with a solution that no one thought of and gain some fame. Maybe this new problem just needs some fresh eyes and nothing more. This is a kind of a project directed development away from the traditional method I have outlined here, this kind of project directed development might not give you a very wide view of the programming world but it will give you a sense of confidence for tackling other problems.

* * *

28. INTELLIGENCE OF A PROGRAMMER

Most of the time the people drawn towards programming activities are extremely intelligent people on the other end of the distribution of intelligence. There is no doubt that programming can be a challenging activity but this doesn’t mean that people at all levels of intelligence cannot participate in it. Solving problems is more about motivation than solely about intelligence. Although possessing great intelligence can make it easier to navigate the solution space, it is not impossible for someone with average intelligence to also navigate that space, even with some difficulty and find the solution to some problems.

Extremely intelligent people are also difficult to work with especially in a team scenario, sometimes they are impossible to work with because the person trying to coordinate them in a team might be of average intelligence and they might look down at this individual as not being worthy of instructing them in anything. Extreme intelligence is also a rare thing and although it is much glorified by society, it is not really a prerequisite for success in any endeavor. Most of the time, the average person does better than the savant at tasks that require delivering a service to people – which most programming is all about.

Extremely intelligent types are needed in the world of software because they help us tackle the most difficult problems. They should not be antagonized but understood for who they are and what they offer us. But it should be clear that the entire field of programming is not dominated by them and those hiring managers who arrange interviews like a classic memory testing championship, hoping to find those types of people will be mostly disappointed because they are not very much in the population and they might not even want to work with anyone because they can usually get by on their own.

Your intelligence doesn’t really determine your success as a programmer, what is more important is your ability to focus and persist on some endeavor till you attain your goal. If you possess high intelligence, then fine for you but it might not be that much helpful if you lack any persistence and give up when a problem gets hairy.

Most of programming these days is done in teams and rarely will you be alone, so the ability to suppress your ego will determine the success of the project. If you feel you are above everybody then working in a team will be difficult and the conflicts that will come up will end up destroying or delaying delivery dates.

The essential factor in team work is fluidity of expertise, every member should be able to identify their core strength and if two people possess the same strengths and interest then even within the team they should work together in a sub team.

So much has been written about team work in programming but what I present here is a kind of street perspective on these issues. Most team methodologies are too structured and real world programmers are the kind of people that don’t like too much structure in what they do. Structure works well for corporate types, and there are many programmers in suit and tie. But the typical bearded hacker is hard to put in a structure.

Everybody wants to work alone but when some huge project is to be conquered, then you need others. The key to working in a team is conquering your own ego. You must maintain your individuality without trying to force your views on others. If you are moderate in your views, defending them only when they are being grossly overridden then the team will succeed.

Each team member has to be good at what they do, not to extreme levels but to a high level of confidence. When a team is formed there should be an intellectual leader and a project leader. The intellectual leader should be the most experienced programmer of all. While the project leader should be someone with good people management skill.

Sometimes the smartest person in the team should be elected as the intellectual leader, but most of the times these kinds of people do not like to be put in any kind of leadership position. In this case they should be left to solve the hardest technical problems while others build the accessory parts.
Conflicts are sure to arise in any team endeavor; the key is ego reduction. With a reduced ego any conflict can be solved. If the group unanimously identifies an individual that is jeopardizing their efforts, the have the freedom to opt that person for elimination. No one should be eliminated because they are not super intelligent.

Elimination should be a matter of behavior most of the time and not just capability, except the individual is grossly underperforming. Someone who is not very smart can still be important in group endeavor, there is always some light task that can be given to this individual as far as they are motivated, hardworking and focused.

The best way to access if someone is a good programmer is how far they are willing to go to solve some computational problem, not the person's academic qualification. If someone has a PhD from MIT, that might guarantee that the person knows certain things and can execute them speedily. If the person is lacking in character, then intelligence alone will not help the person contribute the most to the company.

Hiring managers always want the PhD types thinking they are some kind of panacea that solves all their problems, but except the company is a young growing company then these PhD types are not really needed.

When a company is starting up, they might need some very intelligent people to solve some fundamental technical problems the company will face, the Gilfoy types in the TV series: Silicon Valley. This is the time for the PhDs, no one needs too much character to be able to solve some deep algorithmic problem. They could walk in naked in the morning and curse at will and that would not reduce their intelligence. The early days of the company needs very intelligent people who can be made to do large difficult tasks. The only person needing any kind of character or soft skills is the founder.

Poor team coordination may not result in the catastrophic failure of a startup because in the early times, a company with maybe 5 people hacking out some massive project might not be working as a real team at all. Each person just gets assigned some enormous task and they work on it, integrating it loosely with other parts created by other people.

For programmers intending to start their own companies I strongly recommend the HBO series, Silicon Valley. That series captures what a real group trying to build a company go through.

As a company matures and more people come in, the need for character grows proportionally to the size of the company. At this phase, probably the product has stabilized and some money is coming in, the hiring efforts should shift towards good people. This should be immediately the company releases a product and is getting some money from sales or investments.

The reason for focusing on getting good people at this stage is because the need for sacrifices and long term employees is crucial at this stage of the company. The company is still in a massive stage of growth, and has not reached its peak so you need good people who love the company and are willing to make certain sacrifices for it to grow.

It is not as if the era of the PhD types is over but they should be a smaller fraction of the company. They are great assets for the company and should be kept to solve the most challenging problems. Every company should make it a lifetime effort of accumulating PhDs, allowing them to do their personal projects while calling upon them in times of need, when some serious technical challenge comes up. But companies need a lot of good hearted people who might not possess PhDs for the daily challenges the company will face delivering service.

Sometimes I wonder why there are jobless but job seeking math and computer science PhDs? It's ridiculous that companies will do anything to buy up a small company that sells some ridiculous product than shop for as much PhDs and Professors as they can afford. These people are at the pinnacle of their fields and can solve many problems that are ridiculously difficult, they are great assets to a company more than buildings and cash in the bank or even stock price.

Every other employee is also a greater asset than any other physical asset a company may have. Most CEOs fail to see that a company is just its people, products and services come secondary. The most important aspect of building a company is in the kind of people you hire, they will determine the quality of products and services.

There is a traditional saying in my tribe, of course you know I am African, that goes like this with some alterations for easy of interpretation in English: A human being is the only true wealth.

Many companies fail at understanding this and treat human beings as cogs in a wheel to be picked and dropped or not really cared for at any stage. No great company grew without having good people to support it. It is only after the company is established by good people, that bad companies start undermining the value of their employees, thinking that their true wealth lies in their non-human assets. Not many big evil companies succeed for long in this philosophy as they are soon overtaken by younger agile companies who put employee value as number one.

* * *

29. SHOULD YOU DO A STARTUP?

Everyone seems to be doing a startup these days and if you feel that this is the direction your career should go rather than work for some company then you should do it. Doing a startup is one of the most challenging human activities anyone can engage it. It’s like giving birth to an entity, the company, that never existed before.

Whether you could fail in this endeavor or succeed, if your instincts tell you that this the direction you should take then do that, it will teach you almost everything you need to learn in life. It will test every single faculty of your being for a prolonged period of time. You can’t even say you have succeeded in your startup until 10 years after the date of your incorporation, that is if you don’t fail within the first 5 years.

Startups are the Olympics of the tech world. If you are capable of being punched uncountable times by life and you find yourself getting up each time you are sucked down, then startups are for you. If you cringe and get bleeding ulcers anytime you fail at anything, and find it hard to even get up from your fall, then don’t do a startup, get some safe job in a good company and dedicate your efforts under their umbrella. If you really love the company then you can still have the challenging aspects of the startup world by engaging in the most challenging problems the company is faced with, if you are lucky enough to get assigned to one. Your name might not appear on Bloomberg but you will get the fulfillment of solving some challenge and of course your colleagues will cheer at you, and maybe some bonus might come your way.

For the startup guys out there, I will share some survival tips in this section. This wisdom has come from observing the world of companies in great detail for some time. I also consider myself as being very intuitive and can sometimes pierce to the depths of things and gain knowledge with very little input from my senses. I have not done a startup myself, most coaches don't play on their teams, but I am highly confident of the tips I propose here.

When people think startup, they think of a group of people with skills determined to solve some problem, encapsulate it in some product and offer it to get something in return. It might be immediate in terms of direct sale of the product or a subsidiary service later on which will generate revenue for the company running atop some free service.

But starting up is actually a mindset. Many founders usually knew right from the earlier periods of their lives that they will eventually have to start a company someday and structured their entire life to gather all they will need to get to actually starting a startup. So the startup is not just the legal entity with a company name, it is actually the product of a lifelong pursuit of one or more of the founders.

To startup you need to have the problem solving mindset. There has to be some problem that has been burning in your mind for a long time and you now seek to solve it and encapsulate the solution as a product or service for the world.

There are also accidental startups, where you are tinkering away at some problem without thinking of the need to start a company and you solve it and share it with the world and are overwhelmed with demand that you need to incorporate a company fast enough to make sure you meet demands.

There are startups these days, where people just come together and say: "let's do a startup", just for the sake of it without any deep motivation from any of the founders and just because they hope to cash out soon. These startups for the sake of starting-up is what is eroding the tech scene with so much junk that good money is not going to the real people that need it.

In my opinion, a single founder or at most 2 or 3 founders are all that is required to start solving some problem. Not every solution can be successful as a product for a startup. Sometimes the product you are developing is better off developed within some company, so maybe you should apply for a job there first before trying to build a company around a minor feature of some product.

A startup is usually for only those that want to solve some fundamental problem with a fair reach. And maybe better than a predecessor's solution or maybe never solved before. These are startup worthy ideas; you should not just start an empty idealess startup because you want fame or money.

If you have concluded that you need to do a software based startup then you have to understand that added to the programming requirements I listed at a section of this work, you have to know way more than that. This is mostly for startups in Software and maybe a little bit into hardware.

Gaining a certain level of mastery in the particular programming language you are going to be building your project in and then gaining fair knowledge of some other programming languages you will encounter when working with open source code bases and libraries is a requirement. Apart from programming itself, you might need to learn how many other software tools work, these all enhance your work and make you reach delivery date faster.

If you are alone then the brunt of the work will be on you. If you have a cofounder and if it's a technical person then you guys can share the work appropriately. If your cofounder is not a technical person and maybe a business person then you will have to do most of the early work of building the project yourself. If you have money to hire people, then do that to ease your burden. Maybe you find out that front end web development isn't your thing, then you have to hire some front end guy, either in house or you outsource. But no one can do the core of the product design for you. Some non-technical people start software companies and hire everybody they need from the get go. Maybe you have a software idea and don't have the time to learn programming yourself or not inclined to do so, then you will need to hire a core team made up of people who have certain specialties.

You cannot run away from doing a lot of study because you must know how to communicate your idea to these people. You need not tell them how to implement it but you can have some illustrations and specification to guide their development. At least you know what you want when you have it. Apart from having a fair understanding of the world of software, you will need some people management wizardry to be able to accomplish such a feat as building a startup.

For the software person apart from programming and knowledge of many tools to aid your activity, you will need lots of extracurricular activities to be able to build a successful company. By extracurricular I mean not directly related to programming or software. You will have to know about business, finance, some accounting, money management, people management, copyright law etc. if you are building a hardware then add some hardware knowledge to this mix. Even some knowledge of psychology will help you in not only understanding yourself and your team, but also the people you seek to deliver your products to.

You can always hire people to do these stuff for you but in the beginning if you are very lean on money then you have to do most of these things yourself. And even if you hire people you will still need to understand what they do so you can oversee them properly. This is the only extra information I have for people who hope to start some software startup later on in their career. There are many books that go into details of these topics and I recommend that you read them.

Remember that as a startup founder and a programmer you have to be fit and healthy so you don't run into health problems, as the kinds of task you might be faced with might generate a lot of stress. Life is not just about the things you want to accomplish but about taking care of yourself and the people around you. To do a startup successfully without incurring too much damage to yourself and your relationships you have to be a balanced individual. You have to be able to work hard and play hard when the time is appropriate. If you are severely unbalanced psychologically you might run into all kinds of issues that would prevent you from fully enjoying your success when it finally comes.

Put in your heart into what you are doing. You might be the next big thing or if your startup doesn't hit its expected targets, you might have your core technologies and yourself acquired by a bigger established company. Most companies love people that have ventured in some kind of startups and consider them more matured employees, because they have also tasted the bitter waters of running a company. They won't hesitate to hire you after your startup fails, most of the time. Also most of your core team will find it easier to get employed if they have worked in your startup.

The death of your startup is not the end of your life, start another company up till you get it if your strength is still intact or seek employment somewhere.

* * *

30.NETWORK PROGRAMMING

When starting out in programming you usually write software that is meant to work mostly on your local computer or what we call localhost. Later on you might need to write software that communicates over some kind of network connection. If you are writing a web application or a mobile app that needs some kind of web resources like calling an API from an external service, the you are already doing network programming.

In your programming code you will have to start some web connection, different programming languages have their modalities for doing this: make a web request; create variables for holding the result of your request and close your connection. This is the most basic kind of network programming you can do, connecting to the web to retrieve some information programmatically.

Before you became a programmer, the only way you thought you could connect to web resources was probably through a browser alone, but now you know that you can programmatically access web resources and use the returned data in any way you deem fit for your programs.

To really understand network programming, you have to get some books on the topic and also get familiar with the TCP/IP stack. This is a bunch of protocols which makes it possible for agents on a network such as the internet to communicate. Knowledge of networking especially as it applies to internetworking will be very important if you will have to write applications that communicate over the internet. This is very essential for those who want to get into IOT as your devices will mostly have to communicate over some kind of network. For close proximity stuff some kind of local area network will be the most optimal solution, either wired or wireless. But in order to control some device from any distance beyond the limits of a LAN (Local Area

Network) via Ethernet or WAN (Wireless Area Network) via WIFI, then you will resort to communicating over the internet.

A good general programming language will have a library for doing network programming. There should be functions of opening connections, sending and receiving packets of data etc.

Study networking then learn some network programming from some book and start experimenting with writing simple programs that make some kind of network request.

* * *

31. DISTRIBUTED COMPUTING

Most of your algorithms will run sufficiently on one computer but as things scale up and the need for greater amounts of computing resources increase you will have to write programs that not only run on one computer but on multiple computers connected with some kind of networking solution.

These computers can be in a local network or distributed across the globe as in grid computing. Distributed computing can seem like parallel computing but as we will see later there are subtle differences. Distributed computing is usually done over computers that might be heterogeneous, as in they might be using different operating systems, or network protocols. The keyword here is that they are loosely knit.

As computing requirements scale up when implementing some algorithm, buying a bigger and more powerful computer might not be an effective solution to the problem in the long term especially when you want to manage cost and increase robustness, the ability to survive node failure. Rather than relying on one big powerful computer, it is better you get smaller less powerful ones and connect them over some network connect so that they can share the job of running the algorithm.

Without distributed computing, there will be no modern behemoths like Google and even the Amazon cloud. In the early days of Google, they actually got small cheap computers and connected them in a distributed fashion in order to scale up their computing power. Even at the present they still use lots of low cost servers to build their warehouse scale computers, that offer their cloud product and other services to users. As a startup person if you are not using the cloud to deploy your product and are thinking of building your own infrastructure, in most cases you are thinking of this kind of distributed computing system.

There are two ways we can design a distributed computing system, we can design it such that the entire network of computers appears as one computer to the program, then we do not need to modify our algorithms to run in a distributed fashion. We can just directly run our programs that were meant to be run on one computer then some layers of management software will make sure that the program is abstracted from the underlying network of computers, essentially everything looks like one computer. This management software is usually known as middleware software.

In this kind of distributed architecture, the one where our programs are ignorant of the underlying network, we can continually add new resources to the network and the management layer will just include it. To our programs we will witness a boost in computation power available. Grid computing networks are built like this and are used by some large scientific institutions to outsource computation to clients worldwide. If you have ever participated in one of this projects, you are asked to download some software to your computer which will use your idle computer time to provide computing resources for the organization. So the client software you end up installing wakes up your computer when you are not using it, connects it to the global grid and computations can be distributed to you. There is some kind of management software that coordinates everything.

The second way of designing a distributed architecture is one in which we are totally in control of the execution of the program on the network. The kind of management software you use in this kind of system, exposes the underlying network to your programs. You cannot just use software that was written to run on one computer to run on this kind of Distributed computer. Your algorithms will have to be modified to run in a distributed fashion. This brings the entire field of distributed algorithms to sight, algorithms that are designed with the view of running on more than one computer. Apart from coordinating the execution of the core algorithm on more than one computer you must also coordinate network resources etc. This a very complex field of endeavor and there are books and courses dedicated to it out there. If you realize that your programs will have to run on more than one computer, then you should research these two main paradigms of distributed computing. As a newbie you might have to use the highly managed one, and when your skills are better you can start gradually venturing in the wild world of distributed algorithms.

* * *

32. PARALLEL COMPUTING

Parallel computing seems like distributed computing and officially a parallel computer is known as a distributed computing architecture. This work is not about the “official” report about computing. This is about street knowledge of programming things not the official type given by suit and tie wearing management types who don’t code. This work is a programmer’s view of all things programming.

Whenever you are trying to do two things at the same time on a computer you are doing parallel computing. The first level of parallelism a programmer will be exposed to is thread level parallelism. In the beginning of time there were mono-processor computers, these processors could do only one thing at a time. In order to make the computers operate with the “illusion” of being capable of doing more than one thing at a time, we created thread level parallelism.

A thread is a single unit of execution of a computer program. When multitasking operating systems came into being, what they did was essentially divide the multiple things you wanted to do with the single processor computer into threads, and then made sure that the operating system switched execution between these threads so fast that you were not able to realize that only one process was being run at any one time. This is how the “illusion” of multitasking came to being.

When you write a program that has to run multiple threads of execution, for example in GUI programming you might want the user interface to be still responsive while other processes run in the background, you divide the program into threads, one thread for the GUI process and one for the background execution. The underlying operating system will manage the switching from GUI to background process so fast that you will not notice that at every single cycle of processer execution only one thing can be done at a time. Modern processors these days can even directly manage thread level parallelism or TLP as it is called.

Beyond the mono-processor days, we moved to multicore systems, first it was just two cores but nowadays some commercial units contain as much as 32 cores or more. With multicore systems where a single processor can have more than one core, the operating system designers had to optimize their code to transparently perform computations one more than one processor cores. Another side paradigm was the multiprocessor systems, where we had more than one processor on one motherboard. In all these cases it was the job of the operating system designer to make sure that your programs ran transparently on this kind of tightly knit parallel system. You were not normally exposed to the processors and this is a good thing except you really need to work directly on the cores or processors.

When programming on these multicore or multiprocessor systems, the major form of parallelism is still the thread level parallelism you are exposed to by the operating system. Your programming language usually has libraries for managing threads. This is usually called multi-threaded programming. There are facilities for starting a thread, obtaining a lock on some device or region of memory, releasing the lock and doing all kinds of things. Your programming language documentation will usually provide a specification of its thread management interface.

Up a level away from your individual computer is the parallel computer. This is usually a homogeneous network of tightly knit computers designed to appear as single computer. This is similar to the highly managed distributed architecture, the difference here is that most parallel computers often have highly specialized processors for running mostly specialized applications. These are what we call supercomputers.

Purpose designed parallel computers are mostly designed for one task alone, mostly to run just one application at an enormous scale. They usually have a different kind of operating system and different programming languages or highly specialized libraries in a regular programming language.

There are no hard and fast lines dividing parallel computers from distributed computers, sometimes it's just mostly tech lingo. But the ideas to have in mind is that when faced with a heterogeneous loosely connected set of computers coordinating with each other to run a program or bunch of programs, then you have a simple Distributed computer. If the network of computers is highly homogenous, tightly connected and designed to run highly specialized kind of programs, then you have a parallel computer.

Most parallel computers are designed from a specialized kind of processor called a vector processor, which of course requires different programming techniques. When parallel computers get really big we usually call them super computers. This is mostly a convention, because most of the super computers of the past like the cray supercomputers and mostly equivalent in processor power to what your mobile phone is packing.

Do not be afraid when called upon to work on a program for a supercomputer, as far as you have thoroughly learnt computer programming, you will be presented with the documentation of the supercomputer and even if it has a special language, once you are armed with solid foundation in computer programming, it’s simply a matter of reading it up to know its syntax and conventions and then you can start programming it. Learn the fundamentals and every other tool no matter how mysterious will become easy for you to comprehend.

✻ ✻ ✻

33. GPU PROGRAMMING

GPU is an acronym for Graphics Processing Unit. It is a specialized kind of processor for doing one thing, processing graphics for display on your screen. Most of the time you are not in control of your GPU, the CPU dispatches what needs to be done on the GPU to it. Everything that is being drawn on your screen is usually done by the GPU. In the earlier days' processors had some inbuilt hardware to process graphics. This is because graphics processing requires a lot of floating point operations so processor designers usually had some specialized region on the processor for doing these floating point operations.

As the graphics requirement increased and people started running 3D games the need for a specialized device separate from the CPU became pertinent and some companies seized on this need to provide specialized graphics processors for running these games more efficiently. Although the CPU still has some graphics processing capability and can be adapted to do some low quality graphics work when these GPUs are not available.

GPUs are almost like full featured computers with their own memory and processing cores, but they are specialized computers. As the field of Neural Networks grew, the need to perform fast matrix by matrix multiplication, the primary task of a GPU also increased. This is the core algorithm behind the training of neural networks.

AI researchers started running their deep learning models on GPUs, which is tons faster than running on CPUs. This gave birth to an entire field of programming, GPU programming. The major GPU manufacturer in the world is NVIDIA corporation, their GPUs provides an entire API that enables a programmer to access the device from the computer it is installed in.

As a programmer, when you hear that something exposes some API, that indicates programmability. It means that with an appropriate development environment, you can access this device programmatically, making a request and getting response. The request is usually via some function or class and there is usually some documentation on how to access the device.

Your programming language should have access to the CUDA libraries to be able to do GPU programming on NVIDIA devices. This is an entire field, and if you are doing deep learning stuff you will have to know some CUDA. Most AI architecture abstracts away the GPU, making it easy for a programmer without extensive knowledge of CUDA to be able to run her deep learning or other AI model on the GPU, most of the time it's as simple as an instruction to use GPU. You will only need extensive knowledge of GPU programming if you were rigging up your own system manually to do some specialized stuff.

Another use for GPUs is in the field of crypto mining, where you have to solve some cryptographic hash very quickly. Miners in search of Bitcoin gold set up large rigs of these devices to speed up their work. But nowadays people are using ASICs (Application specific integrated circuits) to do most of the mining stuff.

✻ ✻ ✻

34. ARTIFICIAL INTELLIGENCE

Ever since we had computers, people have always dreamed of the day that computers will be able to perform more than man on all cognitive tasks. AI has a rich and elaborate history and it will be a good thing to learn about this history by doing some research of your own. What I have here for you is a very fast summary of my own perspective on the development of AI.

In the beginning the primary language for doing AI was the LISP programming language, a language invented for the sole purpose of doing AI, but nowadays things are a little different but there are still people out there maintaining expert systems built with LISP.

We have what is known as GOFAI (Good Old Fashioned AI) and the modern neural network paradigm that everybody seems to be delving into. In the GOFAI kind of AI, we try to develop algorithms for building Intelligent agents, problem solving by searching, knowledge systems, reasoning and planning, other kinds of learning beyond deep learning, communicating, perceiving, and acting, etc. Why do we call this AI? mostly it's still just algorithms but the problems are more of the kind a human will face not necessarily computer-centric algorithms. So programming a computer to solve the same human-intelligence related problem is usually termed Artificial intelligence.

At first when neural networks were invented they couldn't do much and a lot of researchers thought they were useless, but when a method known as back propagation created by Geoffrey Hinton and others became functional aided by improvements in computing capacity, the dumb neural networks started producing results.
Ever since neural networks came to the lime light again it became the dominating method for doing AI due to massive success on image recognition tasks, many people

think it's the only way to do AI but this is actually a false perception because there are many more parts to the general phenomena of intelligence than what deep neural networks are capable of. Before the current Deep learning paradigm, which is an application of back propagation, there were other AI paradigms. The best book if you want to learn GOFAI is: AI a modern approach by Stuart Russell and Peter Norvig. This will ground you in some fundamental AI paradigms and also introduce you to some neural networks. I seriously recommend that you read this book as much as you can because the techniques it contains can teach you very powerful methods to use in solving different kinds of problems you will encounter in your general programming career. It will give you an edge that you cannot obtain from any other place, believe me.

Deep learning:

This paradigm has taken the world by storm such that if you are new to programming you might think that this is the only way to do AI but as you have seen earlier GOFAI techniques have always been with us for some time. If you just want to jump into Deep learning, then you will have to read about it. It's a large field on its own with lots of details to it. Gaining some general knowledge of neural networks before you delve into deep learning and eventually writing programs using it is a good way to ground yourself in some knowledge. This always prevents you from getting stuck later on when some problem arises in what you are trying to do. Your fundamental knowledge of programming languages will come to bear here too because deep learning is a technical paradigm and not a programming language.

For the beginner I recommend doing deep learning within the Wolfram language, this is because the mechanisms are straightforward and much simplified. The Wolfram language is compact too and doesn't require much coding to start doing meaningful AI stuff. There is also a large library of prebuilt networks taken from the research community with lots of neurons available for straightforward utility in your own projects.

The effort required to do neural networks in other languages is quite much because this paradigm is in a research phase currently so there are lots of nonstandard ways to do things. If you want to venture into other languages, the next one I would recommend will be python. You have tools like TensorFlow and Keras which simplify the process of building and training neural networks. You should become familiar with the documentation of these platforms because they introduce a lot of nonstandard conventions which you must get familiar with to use them successfully.

There numerous other platforms for python out there and indeed you can do deep learning in any programming language. Python is a good language alongside Wolfram language for a beginner to start in.

If the kind of AI you are involved is generative of information not solely predictive, then you will need to learn about Recurrent Neural Networks. All these paradigms are actually algorithms. You should keep this at the back of your mind. Apart from the data that you are going to work with, all these "networks" are actually specified in code, so having a good foundation in programming other things will easily transfer to these specialized fields.

There are numerous other AI paradigms out there depending on the field you want to conquer. For those who want to venture in robotics then learning about Reinforcement learning will be of much good. I specify the primary areas of application for these paradigms.

1. Deep learning for pattern recognition and prediction.
2. Recurrent neural networks for generative tasks
3. Reinforcement learning for intelligent agents, robotics.

These are just primary applications, if you dig into the literature you will see people adapting these paradigms to do all kinds of novel tasks that stretch the boundaries of their major applicative fields. But as a beginner go with the basics, when you become an expert you can now start performing deeper kinds of experiments towards you own kinds of special applications if you have any.

Deep learning and other machine learning techniques fall into 3 main categories of what is known as learning algorithms. They are:

Supervised learning:

Where we provide training dataset to train the model. The training datasets for supervised learning usually come in the form of a mapping between the exampleDatum and the class it falls into. As an example if we have a handwritten image of a number in the exampleDatum side of the mapping, the class it falls into will be on the right side of the mapping as the number itself as it is typed out on a computer. The exampleDatum is a data element that we want to train the system to be able to recognize automatically in a classification problem, while the class is the correct classification of the exampleDatum. As another example on the exampleDatum side we can have a picture of a cat and on the class side of the training example we have the text: cat. After putting the supervised learning system to work on the example data, we can then test it on an image not contained in the training dataset to see if it will recognize it. Deep learning is a supervised learning system, it is what is behind image recognition systems like Wolfram ImageIdentify and Google images. As we train a supervised system with input data, it generalizes the data by extracting features that are general to examples of a particular class. It might be of note that supervised learning systems usually require a lot of data, the simplest training data out there is the MNIST digit classification data. It already has 50,000 training examples. There are training datasets out there especially those for real world images that are in the billions. Your first dataset is most likely to be MNIST because it is the default dataset used in many introductory AI courses. Apart from neural networks, there are Support Vector Machines which are a supervised paradigm that I will advise you to check out.

Unsupervised learning:

In this method we do not assign a fixed class to a piece of training data. We just put the data through the system and expect the system to group things together based on some criteria. This can be used to discover relationships between data elements that we cannot pick out on our own or that don't fall into a well-defined class but are intrinsic amongst the data element.

Semi-supervised learning: This is usually a mix of both supervised and unsupervised techniques. This is deep magic and call only be used successfully by the expert at machine learning.

There is a major reason for studying GOFAI techniques as compared to these other popular AI methods. For example, in deep learning, your task is getting enough data and preparing it for training, validation and testing. Getting a deep learning environment setup like the Wolfram system or Tensorflow, using standard techniques encoded as library functions to do learning. After the learning and you got yourself a trained model, then you start performing inference, asking questions of that data.

For the basic digit classification task, MNIST, that you are introduced to in almost every work talking about neural networks, after training a model you take an image of a hand written digit and ask the network what it is. This later stage where you ask questions of the model you have created is known as inference.

In all these procedures you will be doing some programming. In the Wolfram language the amount of programming you will be doing is very little because the procedures have been automated to a very high degree allowing you to just focus on what you want to accomplish which is training a network on some data and asking questions on it. Programming is kept on the minimum.

In Wolfram language, even if you want to go beyond basic digit classification into much standard image recognition tasks using convolutional neural nets, you don't have to go building your own net. You could just import a standard net like the LeNet network and start performing inference on it without having access to millions of pieces of data to train the network to. So you don't have to struggle to build your own net models as a beginner. You could just use the built in models created by experts in the field.

In a language like python some books can go about showing you some simple implementation of the core algorithms that make a neural network work, but they are slow for them to be descriptive enough to the beginning programmer. If you want to see the real source code behind stuff like TensorFlow, then you can get it because it is open source but you must be an expert programmer to make sense of much of it because it will not be written in a highly descriptive manner and the most description you can get will be in the documentation. It is written for the purpose of efficiency with all the algorithms implemented in the most efficient manner possible. And still yet references to other libraries written in languages like C will make it still difficult for the beginner to wrap their head about.

AI is just a bunch of specialized computer algorithms. Deep learning is just one, and trying to understanding a real world deep learning platform is difficult for the beginner but not impossible to understand. If you want to get adequate exposure to the Algorithms that we call AI, then you should get your hands on some book treating GOFAI techniques. To my knowledge, the best is Stuart Russell and Peter Norvig's book: AI a modern approach. This book also has open source code on the techniques explored in the book which will prove very valuable to you as a beginning programmer who wants some exposure to advanced algorithms.

If you remember from the section on computational systems, I mentioned the NKS code as a good source code that will aid you in gaining a deeper understand of what programming is about by exploring code that deals with the core of computation itself. Reading code on GOFAI techniques like the one in AI a modern approach will enable you to experience the advanced algorithms which is what we call AI.

* * *

35. ROBOTICS

Any specialized field of computing that utilizes software can be understood using a very basic method of thinking. If the system is about controlling some hardware, then you the programmer should know that your software will be about creating an abstraction of the underlying machine and then controlling the abstraction which in turn will put electrical signals where they need to be to control the actual device you are trying to utilize.

You will be working on several classes of software in your entire career depending on your inclinations or tasks.

1. Software that controls specialized devices.
2. Software that provides environment for other software to run: operating systems, virtualization and containerization software, etc.
3. Software for coordinating a network of devices.
4. Software that solves human cognitive problems, Artificial Intelligence.
5. Software that solves general algorithmic problems, like sorting, searching, etc.
6. Software provide solutions to human work and life problems, like word processors, browsers, spreadsheets.
7. Software for managing data, like database software.
8. Software for analyzing all sorts of things like data science stuff.
9. Simulation software for creating artificial environments like Games, etc.

There are many other specialized fields where software is applicable, extensive deep research will uncover some. An exhaustive treatment of the kinds of software that you will have to write is not the intention of the above subsection. What I want to create in your minds is a generalized picture of what software is and what you should think of when you encounter a new field of technology that applies software.
To create software for anything, you are actually abstracting the system, creating handles for control. Sometimes these handles for control are already created by other people and offered in the form of code libraries. You now call on these libraries and

use their returned results, if they provide any, to perform extra computations. Sometimes some library function does not provide any meaningful return value and is called merely for side effects. We craft software to solve problems by abstracting away the problems into a model, which we interact with. This computation can have a physical effect like controlling a robotic arm, moving a game character on the screen, or allowing you to type in data and format documents.

In the field of robotic our main concern is controlling a physical object. Primarily we might want it to move, either its entire self or some aspect of itself. There is usually some kind of computing platform that allows us run software that will control the hardware of the robot. Depending on the layer of abstraction you want to work with you can build everything from scratch or just use layers already provided by people working in this field of robotics you want to explore.

It’s always good to find out what is available first before you go ahead to build everything yourself. I talked about this earlier when I said before you write some code from scratch, it’s better to investigate if it has already been done by someone else in the open source community. The software that is out there might be better than what you can come up with as a beginner except you just want to solve the problem as a form of mental gymnastics. It’s better to modify the existing code and use it than going ahead to write from the scratch.

Same with any other field of technology, there are already other tools available. Some might not be open source and might require that your purchase some license. If you have the means, do purchase as much specialized tools as you can afford, for the field you intend to work in and some generalized tools too. This will shorten the time and effort required to build something. You can customize and modify the tools, most tools allow this. And if the majority of your tools are Opensource, then lucky you! You can perform endless customizations, but make sure you know what you are doing so you don’t break your code.

This work is about programming wisdom. I don’t go deep into the specific fields I mention; I give you fundamental information that will shorten your path to acquiring the skills you need to conquer your chosen field as a new programmer. There are lots of books out there on each of the topics I mention here. What I’m doing here basically is teaching how to go about learning these things, that is learning how to learn.
Even if they know exactly what they want to do, many new programmers ask me how they should go about it. I usually start talking for a long time about many of the things I have written in this book and every time I do that, the talk varies a lot because depending on time constraints and other factors I don’t go into as much detail or

breadth than I think I should have gone. That is why I am putting everything I know in writing so that I could just refer interested persons to this work rather than start that long hour plus conversation with the countless people that confront me on these issues.

To learn robotics, you will need a solid programming background, this is the foundation for all technology that requires software at some level. Then you will need to lay your hands on some good text book on the subject of robotics itself. This book might include some treatment of planning algorithms, etc. some are math heavy so background in continuous math or computational math will be needed. Then you will have to get your hands on some kit like Arduino.

Arduino is like a one stop shop for people interested in learning about robotics and even Internet of Things. It's a complete package with hardware and software tools for getting your hands on some practical device for starting your robotics journey.

* * *

36. ALGORITHMS

Apart from talking about the Wolfram language and Computational mathematics, my favorite topic when discussing with new programmers is always about the importance of knowing how to think algorithmically.

What is an algorithm anyway? I have been using the word algorithms loosely all through this work without defining it properly. An algorithm is a set of rules for achieving some goal. When confronted with a problem we want to solve, our brain goes ahead to think of a way to solve the problem in a stepwise manner.

An algorithm is like a recipe for creating a dish. We have specific instructions on how to combine the ingredients to make the dish. The recipe is very specific at times. It might include the amount of each ingredient, when to include it, that is after what and how long before we include another one or if we are to mix a set of ingredients together to achieve our intended effect. There are non-specific aspects like "a pinch of salt", or a cup of spinach, this is usually the most confusing aspect for me when I am trying to cook from a recipe. What is a "pinch of salt"? what is that standard "cup". And what is a "touch of lemon".

Algorithms for computational problems are very specific in that you are usually specifying them in a highly systematic language. In computing we are faced with certain tasks, for example how to sort a list of numbers into ascending or descending order. If you are told to do a sorting task on paper, and you have been given a list of numbers written with a pen and told to provide a sorted list of those numbers. You will start creating a new list by looking at each element in the given list and find the one with the least or most magnitude depending on if you are doing ascending or descending order respectively.

Now imagine you are tasked with expressing this "algorithm" which you used to manually sort out the list of items on paper. Can you describe what you were doing? If you can describe what you were doing as you took out each element from the list you were given and wrote it in the new list, then you are now specifying an algorithm for doing sorting.

If you were careful in observing your own actions and list them in a list of steps, another person could just use your algorithm when faced with the same task of sorting items on paper. The person would not have to think of what they need to do, just follow your steps. In this case the person in question will be acting like a computer executing your steps.

Now if you were to specify the task to a computer you would have to modify the specification a little bit to suit the structures available on the computer. The way you specify it depends what level of abstraction you are going to be working on. If you are working in assembly code the specification will be different than if you were specifying it in C, python or even Java.

But when planning out an algorithm, most algorithm designers do not write in an exact programming language. They usually use pseudocode so that any programmer working in a specific language can implement it.

Pseudocode is like real code, but usually written to be consumed by another human being and not a computer. There is also flowcharting, but I don’t know many people who use this. If you will prefer a graphical approach to specifying your algorithms, then flowcharting will be excellent.

When you have some pseudocode you can go ahead and implement the algorithm in some programming language of your choice for execution. There is another advanced way of specifying some algorithms, where certain aspects of the algorithm will be given in mathematical notation. Knowledge of computational math will enable understand these bits and even aid you in specifying your algorithms in this form.

So what are the bits we need to know in specifying pseudocode? A simple pseudocode for specifying addition of two numbers could go like this:

```
01. Create three variables: A, B, C
02. Get the first number and store it in A
03. Get the second number and store it in B
04. Add the first to second and store the result in C
05. Return the result
```

This is just one way of specifying it and you can do it many other forms depending on your style preferences. The key is to be as clear as possible so that other programmers can easily interpret your pseudocode.

Other things you can see in pseudocode are: Do … While …, If … Then …, For … Do …, While … Do… etc. if you have learnt programming then translating this statements in pseudocode to real code will not be too difficult.

Efficient algorithms:

In the sorting example I gave above, a human being may have to scan the entire list over and over again while picking out individual elements to list in a particular order. This is an inefficient process. When dealing with a small list we might not notice any overhead from going over and over the entire list searching for the next element to include in our list, but when the list becomes large enough the process breaks down. There is an entire field of algorithms research that is concerned with problem of producing the most efficient kinds of algorithms for all kinds of computational problems, of which sorting is just one of the numerous kinds available.

When we code our algorithms in the most natural way we think about them, like in the sorting example: searching through the list over and over, we create what is traditionally called naïve algorithms. A naïve algorithm might make it easy to explain the basic process of solving a computational problem to someone, but when it comes to implementing a fast solution we have to think of efficiency both in computer resources and time of execution.

The computer resources we seek to economize when writing algorithms are the: memory usage and number of processor cycles needed to accomplish the task. In general, we want to bring memory resource utilization and processor cycles down to a minimum. The number of processor cycles is also related to the amount of time spend executing the code. We want to keep this running time as low as possible. The most optimal algorithm for solving a problem will usually have the fastest execution done in the shortest amount of time with scant usage of memory, an ideal that is always very difficult to achieve. This is what the entire field of algorithms research is about, producing optimal algorithms.

There are standard algorithms already included in libraries and available in your programming language to do things like sorting. It is always better to use what is available rather than writing yours, except you're an expert in these things. So much directed effort has gone into researching and producing these standard algorithms. If this is not your field of specialization, use what is available. If you are not satisfied with what you have, remember there is a maxim in the field of algorithms: we can always do better. Sometimes getting better algorithms that are faster than what is available is a herculean task available to only the brave.

Don't be too sure that you have produced the fastest algorithm possible just because it passed some specially tailored problem of yours or because you used it on a small dataset. Your algorithm that successfully sorts a list of one hundred thousand digits might fail when it encounters a dataset of a million digits. If you feel that your calling is in the field of algorithms, then do endeavor to get as deep as possible in the field. Good mathematical skill is a must if you want to go deep into algorithms.

The computer knows nothing intrinsically, it's a machine that allows you specify algorithms for anything computational. Even addition and multiplication have to be specified for the computer. Algorithms for addition and multiplication are even expressed in hardware on the CPU so they are very fast. The computer is this mountain of interconnected algorithms that enable us accomplish our tasks.

Algorithm analysis:

When you specify some algorithm for performing some task on a computer, we usually want to know how fast they are. This is an entire field of endeavor known as algorithm analysis. We want to know how an algorithm performs as you increase the size of the input, usually denoted by small letter n. Algorithm analysis is sometimes known as Big O analysis because it uses the Big O notation to express running times of algorithms. When you hear professional programmers talk about the speed of an algorithm, the usually do it in Big O notation. Learning about Big O notation is really important.

Most people who go to "coding" academies are not usually exposed to these things like algorithms analysis, and in my understanding they are missing out on some of the core ideas of computer science and programming. You can learn the syntax of a language and even go ahead to write a program to solve some problems, but I bet you that without knowledge of algorithm analysis you will end up writing inefficient programs most of the times.
When you have understood algorithm analysis you will know when your algorithm is blowing up memory resources or spending a lot of time on a problem. The entire field of Big O analysis is large but in regular programmer lingo, you can hear someone describing an algorithm as being n^2 (n squared) in its input size. Or simply an n^2 algorithm, formally expressed in Big O notation as O(n^2). Another popular running time is a Log[n] algorithm. What this means is that the graph for the growth rate of the algorithm's running time as a function of the input size n is growing at an n^2.

So therefore for an n^2 algorithm as you keep increasing the input size the running time of the algorithm grows like a graph of n^2 or is quadratic or polynomial in n.

So we use these algebraic expressions to represent the running time of algorithms, because that is what we are more interested in. it's quite harder to know how much memory an algorithm will consume apart from the size of the input. The operating system and your programming language usually handles the details of this, but there are techniques you learn as you advance in programming that will enable you trim memory requirements for your program. Most of the time we don't know the environment where our programs will run in after we ship them away. We might know that it will run on some specific operating system but the hardware configuration will be unknown to us. We can usually include system requirements for the products we ship, but there is no guarantee that our requirements will be met. That is why we have to be cautious with memory consumption. The higher the level of the language, the lesser your concern about issues such as memory management because there are automated facilities provided by your programming language for what is called garbage collection.

When you create a variable in your program, memory is allocated for it as long as the variable exists. When we are done with this variable, that means when there is no longer a reference to the underlying memory, we are usually stuck with some useless piece of memory that we no longer have access to because we have destroyed the reference to the variables through our programming.

This process of destroying references to memory locations is a normal part of programming and should not be looked at as some kind of error on our side, except a variable is destroyed prematurely before it is used. There are programming scenarios where this is possible.

When we have inaccessible regions of memory created by our programs, we need some method of freeing up these pieces of memory so that we can reuse them in further code. In languages like C you would have to deallocate these piece of memory manually, but in higher level languages like Java, Python, Wolfram, etc. there is a program called a garbage collector. This software constantly monitors the memory and reclaims memory regions for which there is no reference to.

A well implemented algorithm usually has a good memory utilization strategy. There are other specific methods that programmers use to avoid memory excesses. In your computer there is something called the memory hierarchy: Registers -> Cache -> RAM -> Persistent Storage. The registers and cache are on your CPU, but the RAM and

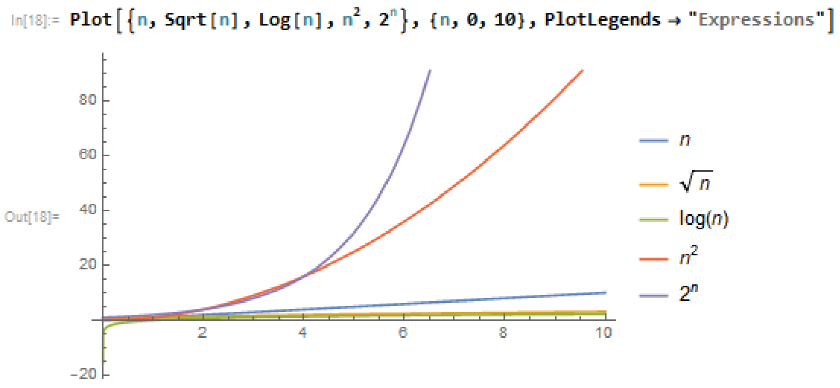
Persistent storage like Hard Disks (HDD or SSD) are usually away from the CPU, connected by a bus. In regular programming activities, the operating system manages stuff like the Registers and the Cache, while you as the programmer have access to the RAM and persistent storage.

There are mechanisms for creating register variables in languages like C, but that is black belt stuff and you shouldn't be toying with that. When you are writing a data intensive program, and assuming you are not having to connect to a database, then you will have to read of your data from a file in your hard disk and bring it into ram for computation.

If you know the limitations of the computer you are using like how much RAM is available to your program etc. you can design your strategy around your individual computer, bringing in as much data to your ram as you can withstand without causing your operating system to page (taking out stuff from RAM and writing it to hard disk when RAM is filled). Paging generally slows down a system and it's not something you want. So you should read as much data as is reasonable for the system you are using, you want to make sure there is some RAM available for other processes.

For a program that you are shipping off to be used by another person, you should be more cautions and do your research properly on the most common hardware configurations out there. Do not expect that your system requirements statement that you provide for your software package will be obeyed. Be cautious with memory allocation, but know that more memory request to the persistent storage will incur some overhead because the hard disk has to be read and then your data transferred into main memory (RAM).

So before our brief detour into the hairy world of memory management we were talking about time management, that is controlling how much time our programs spend on their task. We talked about what is usually referred to as algorithmic complexity, including ideas about Big O notation etc. One method I use in explaining the most basic running times for most algorithms to programming students is by showing them a graph. The graph below was concocted to show you the nature of some basic running times. It does not represent any particular program but just gives you an idea of how the growth rates look like when governed by some algebraic expression.



```
In[18]:= Plot[{n, Sqrt[n], Log[n], n^2, 2^n}, {n, 0, 10}, PlotLegends → "Expressions"]
```

Some basic algorithm run times plotted with Wolfram Mathematica

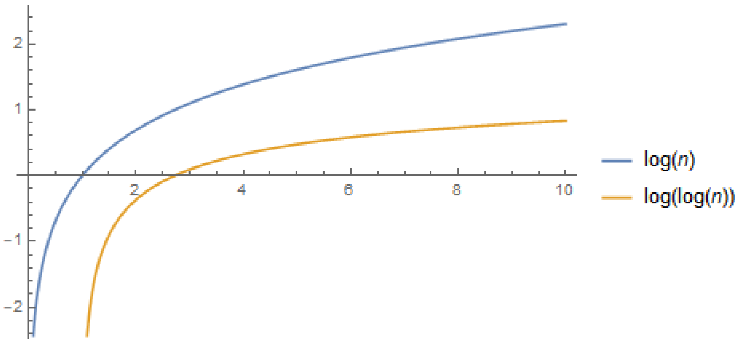
In the graph above the input size n is on the horizontal x-axis while the running time is on the vertical y-axis. We can see that as the input size for an n^2 algorithm increases the time spent on execution increases very rapidly. Note that since there are no negative inputs to a program, i.e. there is either zero or more inputs to any program, the graph is mostly positive.

When we say that an algorithm is Log[n] in its input size n, it means that if we repeatedly take numerical values of the running times of the algorithm as we gradually increase the input size, the graph of these running times when plotted will look like the graph of the expression Log[n].

From the graph above we can see that 2^n is the fastest growth rate while Log[n] is the slowest. Algorithms whose runtimes grow like 2^n are known as exponential algorithms. This is usually something distasteful because it means that as the input size n increase the runtime increases exponentially. In some algorithms we cannot run away from exponential time. But when algorithm designers see something exponential, they try to find ways to optimize it to reduce the runtime or make it polynomial in n as they usually say.

In Big O notation we say that the algorithm is O(Log[n]) or O(n), etc. depending on the results of our systematic analysis. There are other run times you will encounter as you go through the available literature on algorithms. You will encounter things like n Log[n], Log[Log[n]], etc. Do not be bewildered by such expression, by understanding the fundamental runtimes you can easily deduce what these expressions mean. A nice method I encourage is using Wolfram Mathematica to plot any running time you encounter to have an idea of how the graph of the growth rate behaves with respect to the standard runtimes I have shown you above. This will reveal information of how much faster or slower some fancy expression is compared to what we already know and from there we can gain insight into its performance. As an example if we want to know which is faster, Log[n] or Log[Log[n]], we simply plot it.

```
In[19]:= Plot[{Log[n], Log[Log[n]]}, {n, 0, 10}, PlotLegends → "Expressions"]
```

Out[19]=


From the plot above, it is clear that a Log[Log[n]] algorithm is much faster than a Log[n] algorithm.

If you have engineered a function of your own takes a list of numbers and don't know much math to do standard algorithm analysis, you can keep running the function with increasing input sizes, and just collect the data obtained after some runs and plot it. You should be increasing the input size, i.e. the length of the list logarithmically (i.e. 10,100,1000,10000,100000) so that we reach the upper limits of the program on time.

When you plot this data and compare it to some of the standard growth rates you have seen, you will have an idea of how fast the algorithm is without the standard algorithm analysis methods.

Deterministic vs Non-deterministic algorithms

As you progress you will see some algorithms with the tag deterministic and others non-deterministic. A street level way of distinguishing the two is this: A deterministic algorithm is one that has a definite execution path. Most everyday computer science problems you will encounter like problems of sorting and for instance multiplication, etc. are the kinds of algorithms that have a deterministic nature. You can predict what these algorithms will do every time they execute.

For non-deterministic algorithms, you have algorithms that have to make some kind of indefinite decision every time they run. When we are designing algorithms to solve decision problems, we usually end up with non-deterministic algorithms that use randomization is some form. Due to the fact that we include randomization in these algorithms, we cannot for certain know what decisions they will make during program execution to arrive at their results.

So how do you learn algorithms? There are many pathways you can take to achieve your goal but this is one I will recommend for you. First of all, I will strongly recommend MIT OCW courses on computer science. They are free and of the highest quality. Imagine being tutored by some of the best Professors and PhDs from such a prestigious institution like MIT. There are all sorts of courses these days from Coursera, Udemy, Udacity, etc. I actually did one on Coursera by Tim Roughgarden of Stanford university and it was very good. The thing with most MOOCs (Massively Open Online Courses) is that in an effort to make the courses more attractive to people they shorten the length of the lectures. In a field like algorithms this shortening results in too much summarization without the instructor going into sufficient detail. MIT OCW courses are actually full class sessions. The instructor is teaching a real MIT class on a blackboard! It's just recorded directly for you to watch.

In my opinion the full lecture structure of MIT OCW courses makes it a better approach for teaching computer science than the summarized short video form on many other MOOC platforms.

Go to the website ocw.mit.edu and check out the courses below.

1. Introduction to Algorithms 6.006
2. Design and Analysis of Algorithms 6.046J

There are many textbooks out there on algorithms but one stands out. It's the book written by the Master of Computer science, Donald Knuth himself. The title is: The Art of Computer programming. The book is in multiple volumes. Currently I think volumes 1,2,3 have been completed. Studying this book will show you the depths of algorithm design. Unlike most other books, he doesn't do it in pseudocode or any high level programming language. He actually designed a hypothetical computer where you write the code in an Assembler like programming language.

Writing algorithms at such a low level, so close to the machine enables you to implement the most efficient possible forms of the algorithms treated in the book. Many algorithms that are deemed fundamental are covered in the book. The early part of the book introduces you to the mathematical preliminaries that are required for gaining a deeper understanding of the algorithms but it is quite short and succinct so Donald in collaboration with others decided to produce a more descriptive version of the mathematics he uses in his algorithm work in another book titled: Concrete mathematics. These books are treasures to any student of computer science and programming.

* * *

37. HEURISTICS

This is a pattern of solving problems using rules of knowledge that we have obtained from experience rather than some fixed solid algorithmic solution. We use heuristics when we are faced with dynamic problems for which we cannot capture a total solution at once.

Heuristics programming involves some kind of commonsense problem solving system, where we cannot anticipate all our input but we have some kind of rule system for dealing with certain cases. Even if we cannot classify the full input we can expect it to contain certain structures or behaviors that we know about and can thus identify.

This technique of doing things emerged out of classical artificial intelligence research. When confronted with a scenario where a program cannot perfectly predict the future, it is usually wise for the programmer to use a heuristics approach to designing a solution. There are some basic rules that are known about the environment and the agent navigating such an input space is able to judge things based on its database of heuristic knowledge. This database can be expanded as more patterns and behaviors in the input space are detected and stored.

A major application of heuristic based programming apart from regular agent based artificial intelligence is in the field of computer security. Most of the time we cannot classify all possible threats a system can encounter in the course of its existence. Although there are certain known threats that can be easily identified, it is very hard to identify threats that are new of which we know nothing about.

Antivirus, antimalware, firewalls and other computer security software usually possess of library of known threats. When scanning a system for viruses and other malware or when screening incoming traffic for malicious packets in firewall systems, the threats are being matched with what is known in the database of threats and if there is a match there is usually some kind of protective measure that is activated.

This is the ideal situation but in reality the world is being swamped by new threats every day and it is hard to keep updating the database on a constant basis with this new threats in time enough to catch them. So in order to deal with new threats that might not be contained in the database we build up some heuristic rule to identify in general what a virus, worm, Trojan, etc. looks like. This rule is then stored and used to process incoming traffic.

Security threats are not always arbitrary in their behavior, they act in a certain way: trying to access protected resources, obtain information that is beyond their access level, trying to overwrite a file, etc. The database of threats usually contains the names of the specific threats and their file signatures etc. but a heuristic database contains patterns of behavior and structures to expect in a typical malware. So when a file or packet is scanned that is matched with the database of known threats without yielding any match we usually pass it through some processing using our known heuristics to see if its behavior matches the behavior of our known threat. If there is a match, then some kind of quarantining is done to prevent the threat from activating its actions.

If on further checkup we find that we have a concealed threat then we do appropriate actions like deleting it, etc. if we found out that our heuristic process was overzealous then we restore the file or process back to privileged access of the system.

Heuristic programming is not limited to catching security issues in systems, it can also be applied to any system where we cannot fully anticipate our input but are required to deal intelligently with whatever input we are presented with. You can find it financial modelling, robotic control, IOT devices that have to interact with the wild, etc. even in daily life most of the time when you are dealing with issues you are not using some kind of fixed algorithm or scanning some database of known things, you are mentally applying heuristics to tackle problems of a novel nature.

So how do we learn heuristics programming? The internet is laden with books that go into great detail about this paradigm. I am your light bearer pointing you in directions you might not have known of so now you are aware of the field of heuristics go ahead and research to find out more.

* * *

38.DATA STRUCTURES

The two fundamental aspects of computer science are Algorithms and Data structures. When we write an algorithm to solve the problem, we are assuming that there is some underlying data structure to hold the data we wish to deal with. In your programming language you have data types like: Integers, Floats, Double, Shorts, Bytes, Chars. These are kind of atomic data types because they can be used to represent some quantity. There are also compound data types like Strings, lists, tuples, arrays and dictionaries. In many languages the String data type, which is a combination of characters is usually a primitive data type.

Data structures are abstract structures, implemented by lower level data types that store data in a certain way and include procedures for accessing the data. So whenever we have a systematic arrangement of primitive data types, more frequently the container data types, and some procedures for accessing the data in a certain way, we have a data structure.

Data structures are important because they give us a structured way to store and retrieve data for our programs which results in improved efficiency of the task we want to perform. One of the simplest data structure ever is the stack. Just like a stack of books we take things out from the top of the stack and put things back on stack using LIFO (last in first out). A stack is primitively stored using a list or an array. It is just a linear collection of things. What makes it special is that the word “stack” denotes an abstraction in the way we want to deal with the list. In this sense the order of access is the most important thing which in this case is LIFO. The underlying data type can be anything that allows us to access the last element placed in it as the first.

Another basic and very useful data structure is the Queue or Pipe as some people will like to call it. It’s a FIFO (first in first out). In this sense it operates like a pipe, where the first thing you put in is pulled out first from the other side of the pipe/queue just like you have with a queue in real life.

While a stack has one end closed a queue has both sides open. In programmer sense we are all dealing with a simple underlying data type like a list, it’s just the way we access the underlying datatype that defines what a data structure is. Some data structures might require certain arrangements of underlying data types to implement them, and no matter how complicated the abstract definition of a data structure is there will always be some arrangement of underlying data types and procedures to implement it.

Data structures are a fundamental aspect of programming. As you write programs of increasing complexity and scale you will find out that treating data improperly will affect the efficiency of your programs. It is important to learn some of the fundamental data structures found in a standard text book rather than hacking out your own. The nature of the problem you are trying to solve will also determine the data structure you will use. There are many textbooks out there and I don’t really have a favorite so I cannot recommend any. You should look out certain topics like Binary search trees (BSTs), B-trees, AVL trees, etc.

✻ ✻ ✻

39. WHY YOU SHOULD KNOW COMPUTER ARCHITECTURE

If you have not been exposed to a standard computer science curriculum, you might have learnt programming without any knowledge of the underlying computer architecture. Most coding schools just teach you programming in a high level language and with what they impart on you, you can go ahead and start writing programs. All the layers of abstraction you deal with in a computer work quite independently of others and you can work at any without bothering so much about what goes on above you or below.

This is the path of the regular programmer, you can go ahead and solve many computational problems with your basic skillset. If you add algorithms, data structures and some math you will take your skills to a very high level. But for the master programmer, the individual that wants to have the entire machine abstraction hierarchy in view while designing software, you will need to know about the underlying architecture of your machine.

Although in another section of this work I said in other to understand complex systems, you must learn to forget some aspects of the system while focusing on others, this doesn't mean that you will not eventually build a full mental image of the system, it only means that you will concentrate on one part of it at a time while learning the system.

Learning about computer architecture is just one part of the entire complex structure of the computer system, and taking some time out to learn it will actually transfer into your thinking when you are designing or trying to understand programs. You don't need to actually build a physical computer yourself, but knowing how basic the underlying hardware is can give you an all-encompassing view of practical computing that will improve your understanding of the tasks you are trying to perform.

When working in a high level language, there are layers of abstraction that shield you away from the underlying machine. This is the greatest power of computing, the ability to create simple interfaces to much more complicated underlying structures. The truth of the matter is that even though you can theoretically build anything at the high level language, the underlying hardware is limited in certain ways and this can influence your programming a lot.

A common situation is branching, which is one of the most expensive operations for the computer hardware. When you learn programming, you are shown conditional branching statements like IF and SWITCH. There are certain programs you are implementing where you will notice that you can replace some heavily nested IF block with a simple SWITCH statement. This is because on most processors its faster to execute SWITCH than deeply nested IF.

There are other things you will get to know about when you study the highly constrained environment of the computer hardware. In casual programming these things are usually not an issue, but when you want to write extremely efficient programs for systems with limited resources you must start considering these issues.

If you plan on building computer systems like your own infrastructure, then knowing about computer architecture is a must. For that purpose, the best book in the field are the books written by Hennessey and Patterson. Just type those names in your favorite search in engine and you will see their computer architecture books turn up in the result page. You can simply add "computer architecture" to the names to make your search more specific.

It is also fun to know something about the way your computer works, it's an enlightening experience and will make you appreciate the greatest tool ever created by mankind the digital computer.

* * *

40. COMPILER TECHNOLOGY

If you are not working at the assembly level of computing, you must sometime write code that uses some kind of compilation. Even the so called interpreted languages have some kind of compiler technology deep inside the stack of code.

A compiler is a software that takes your code in text and transforms it into machine instructions that your computer can execute. Without the compiler you would have to work in assembler and all that beautiful pieces of code your write would not exists, rather you would be writing directly in the language of the machine which is not an easy task.

Different compilers generate different code depending on what their creators think the most optimal machine code for your high level code is. There are also optimizing compilers that go through multiple cycles of compilation making the generated code much more efficient each time.

When I recommend that you explore a topic like computer architecture and now compiler technology, I am not insisting that you be an expert on these topics. Reading one architecture book like Hennessey and Patterson will ground you in computer hardware even if you don't intend on building your own computer. Having an idea of what is going on these fields will aid your general knowledge about computer programming.

Some people want to know as little as possible but that is not the kind of person I am and that is not what I advise people who come to me seeking to learn programming. More knowledge is better than little knowledge when it comes to computers. You don't need to be overwhelmed by all what you have to learn, to complete this curriculum might take many years to accomplish but it is better to know what is ahead and brace yourself for it.

When you finish going through this work, you will know how to start your individual journey. Don’t be discouraged if you find it hard to learn in the beginning, programming doesn’t work in a way your brain functions naturally. Your brain will have to do a lot of adaptation to get into the structure of computational thinking. It’s like driving, after learning with so much effort it becomes automatic. When you really learn a programming language and associated technologies for a while, you will start naturally thinking computationally. You don’t have to force memorization of all the concepts and ideas you will encounter. If you find that something is very useful to you, then keep familiarizing yourself with it until without effort you will know it.

If you plan on designing programming languages, then learning about compiler technology is a must. It is a wide field on its own with many new concepts to learn and master.

✻ ✻ ✻

41. HOW DO YOU TEACH KIDS PROGRAMMING?

Recently I was tasked with teaching a bunch of kids with ages between 5 – 8 how to program. These were kids that could read, write and do basic arithmetic but I couldn't figure out how to present the abstract concepts of programming to them for easy comprehension. I wanted to teach the kids a language that they can keep using for the rest of their lives, if they choose to become computer programmers.

Long ago before I encountered the Wolfram language I had considered python the best programming language because of its ease, I had even considered recommending it as a programming language that even kids could learn. But on deeper inspection and after some experience in trying to teach people programming I decided that it is possible in python but would be difficult.

The thing is you can teach a 12-year-old python programming very easily, at least you could start with showing the kids how to do arithmetic in python and then proceed to showing them how they could write programs to automatically solve certain things for which they already have formulas for. As an example, they could learn how to write programs that finds the hypotenuse of a right-angled triangle using Pythagoras theorem or they could solve high school physics problem using programming.

This is a nice way to teach those kinds of high school kids programming. Their capacity for very abstract reasoning is not in general fully developed so teaching them how to apply programming to tasks they are already familiar with from school would be the best approach.

For kids that are way younger than that, it will be difficult because they have not had much exposure to formulas so showing them how to use programming to solve formulas would not be possible. Another thing is that with a language like python, you could still teach basic arithmetic and just make the kids memorize the language features abstractly just the way alphabets are taught but that too would not be very useful. It can expose the children to programming ideas like what an IF statement is, etc. but these kids are way more capable if only the proper programming language and environment were provided for them.

They could do more than learning about the names of language features like FOR, DO, WHILE, IF, etc. they could actually do programming! It will be unwise to make kids of this age range read a standard book in programming so you would have to create an environment where they could interact with prewritten code.

Wolfram language is the default choice to teach kids in – this is because the language is powerful enough to be used in many large-scale software systems as well as ridiculously easy to learn plus there is already an environment called the Wolfram programming lab, where anyone with little or no experience in programming can just start interacting with code, without having to first read a large book on programming.

Without the Wolfram programming lab, it would have been a hideously difficult task trying to teach these kids programming the traditional way, that is through regular programming books. Videos can also aid, but there is nothing like direct code experience when it comes to teaching someone programming. The hands-on experience gained from running and modifying code cannot be surmounted by any other means no matter how cleverly crafted.

With children of ages 10 and above, using the Wolfram programming lab will be a breeze. The system is so well designed that the kids can just use the system on their own and by the end of their tour of the system they would have enough knowledge of programming to understand any piece of code written in the Wolfram language and can start writing their own. But for younger kids, some kind of instructor is needed to guide them on a learning path. Starting in the next paragraph is a learning path I invented on the fly as I was tasked with taking some kids less than 10 years old through the Wolfram programming lab.

First of all, they must be able to run code! This is the first requirement on the learning path I crafted. In the Wolfram notebook interactive computing environment, all that is needed to run code is to take the mouse pointer to the cell in which the code is contained and use the Shift key + Enter.

One of the beauties of teaching these very young kids programming is that you take nothing for granted. You have to show them keys and how to even do key combinations. You have to show them how to use the mouse pointer to locate a notebook cell. Some kids might already have some experience with using a computer that has a keyboard and touchpad – but the kids I was teaching had only used a tablet computer.

The laptop I used to teach them was also a touch screen but I discouraged them from using the touch screen to locate a cell. This is because in the real world they might be faced with computers without a touch screen and it will mystify them when they try to touch the screen with no response. I was sowing fundamental knowledge in these kids so I was careful to make sure that their foundation is solid and their future in programming hitch free.

The Wolfram programming lab environment also provides a button for running code, but I also discouraged the kids from using it because in the regular Wolfram notebook environment they would not have access to that button – so I emphasized the use of the keyboard for code execution and then the mouse to locate the cell.

We spent several lessons focusing on the simple act of executing code that was already contained in the Wolfram programming lab. It was fun for them because the program output was very visual, like images, words and stuff like shapes etc. When their code execution skills were stable enough, we proceeded to learning to identify function names. So we would execute code and then count all the functions that were involved. At a later stage I would tell them to pronounce the names of the functions involved.

It was easy to identify the functions because we were using built-in functions and all built-in functions in the Wolfram language follow a naming convention where the first letters are all capital. Sometimes they made mistakes in identifying which names were functions and which were symbols, a fact I had concealed because I wanted to give them information in bits. I didn't want to introduce the idea of symbols yet but when we encountered a symbol and someone pointed it out as a function, I added further details to clarify the difference between the Function name and a Symbol name. I showed them that a function usually has square brackets (FunctionName[ ]) after its name. With this detail they stopped making the mistake.
When learning mistakes are as important as the information one is trying to learn. People have to see examples and counterexamples to enable them properly identify true cases. A definition should not be too detailed at first – some general definition should be given at first then later on more details filled in. If the individual is learning

in a guided environment, then they should be allowed to make a mistake before shown the correction. The fact that someone had made a mistake in identifying something creates a stronger emotional condition so that the correction is absorbed more permanently.

The learning environment must also be judgement free, punishment as a way to improve accuracy is unnecessary. Rather the student should be corrected with understanding. Even if the mistake is repeated, correction should be repeated. Brains are different and some brains need more repetition than others. As far as there is willingness to learn the student should be encouraged and the teacher must be of infinite patience. Sometimes some homework to help the student overcome their weakness in the safety and freedom of their own personal time will be of much good, if the learning environment is too hectic and much attention is not being focused on the single student due to time constraints.

The next step for us was learning what function arguments are and learning how to count them! Teaching these kids was a very fulfilling endeavor to me because I was able to break down things we normally take for granted into the most basic forms possible. I just told the kids that anything between an open: [, and closing:], square brackets was an argument to a Function, the stuff that started with a capital letter and with an open square bracket after its last letter. Such high-level descriptions are essential when trying to introduce novel concepts. I didn't go into the technicalities of functions, where and how they received arguments to do their work etc. This to me was too abstract for these kids. So I focused on what they could perceive visually. After showing them several examples of what an argument was in some code, they were already identifying and counting arguments – I had to show them that a function can have multiple arguments separated by a comma like FunctionName [arg1, arg2].

Sooner than they expected they were presented with some complexity, after identifying functions with simple arguments like strings and numbers, they were faced with functions that had other functions as arguments. This perplexed them at first but when I went ahead to show them how to understand nested things by showing them how to focus their eyes on the deepest comma, which signifies the separator between arguments, they could now detect nested structures.

So we didn't just count arguments on the root function, we counted arguments of any sub function that was contained in the root function. If we had a situation like FunctionOne[FunctionTwo[arg1, arg2], arg3], I would ask: how many arguments does FunctionOne have? They would respond with 2 as the answer. I would say name them? They would respond with FunctionTwo and arg3. Then to be exhaustive I would ask what about FunctionTwo? They would respond with arg1 and arg2. In the beginning of the questioning I would first of all ask them to name all the functions which they would do before we proceed to arguments.

This idea of nested functions and arguments was one that was quite difficult for them at first – so we spent some time on it till they were familiar enough. Wolfram programming lab contains a lot of examples that enabled us experience these nested stuff a lot. All the while they were still sharpening their code execution skills – with the most difficult aspect being using the touch pad of the laptop.

Sooner or later we encountered a function that had a List as an argument. This perplexed them a little as they didn't know how to count arguments again since the List was also a comma separated structure. As I have always said on the issue of understanding complexity, you must learn to ignore some parts while trying to understand some parts. I didn't go exhaustive on showing the kids all the comma separated structures possible – I just showed them the function. And when they wrongly counted the arguments in function because a list was included I went on to show them what a List was. In a standard programming textbook, you are shown a List datatype separately then depending on the programming language, if you can pass a list to the function, or just pass a variable representing the list etc. in the Wolfram language the List is an expression, and we work in expressions so we can just pass a whole List into a function without creating variables etc.

I explained that a list was anything that began with an open curly brace: {, and ended with a closing one:}, with everything in-between separated by commas. With this they were able to identify Lists and know that a list was something that could be an argument in a function, even though it had items that were separated by commas too. I also showed them that a List could also contain other Lists too – this was not very difficult for them to absorb because the idea of nested things was already solid in their heads.

When a new concept is introduced there will usually be a regular sessions of practicing until I felt they had reached a certain level of competency, before we moved on. There are usually more complexities to show but I delay the onslaught until the current one is mastered. We take any other thing we encounter as a given, meaning it is just what it is. The focus at this stage was to count arguments, name functions and execute code.

After a while we were presented with a language function called a Rule. Whenever I encountered this structure earlier, as an argument to a function, I refrained from making the kids count arguments and just tell them to execute the function or some other task. I wanted them to get used to earlier structures I had introduced before going into other details. When I was satisfied with their progress so far I introduced the rule structure by making them count arguments in a function that contained a single rule. They counted with an excess of one because a rule consists of two parts, an Expression, the Rule symbol -> and then another expression. So they counted both expressions individually resulting in the excess of one more argument to the function. I emphasized that a function had a square bracket [some stuff separated by commas, and another closing square bracket], and that anything you saw within the function that did not follow this convention was a single structure. So if you see a statement like FunctionName[arg1, arg2, RuleFirstpart -> RuleSecondPart], the last statement after the comma is a single argument. They learnt this very fast and we didn't have problems with this again.

A problem we encountered in naming functions was when we encountered a built-in symbol. In the Wolfram language you can have a symbol like Red, which simply stands for the color red. It starts with a capital letter and was mistaken by the kids to mean a function name. This was a time to introduce to them what a built-in symbol was and to emphasize that it was not a function because it wasn't followed by square brackets. This quickly differentiated a built-in Symbol from a built-in function and as soon as we had enough practice it was no longer a problem.

All these while we had been executing prewritten code and had not encountered a function definition. As soon as we encountered one I quickly explained it, knowing that it won't be understood sufficiently at the time, I didn't emphasize it so much. I only showed them that we can have our own functions and Wolfram language programmers use lower case as the first letter of the function name when defining their own functions to differentiate it from the built-in functions. I showed the key to identifying a function definition and that it is given by the signature userDefinedFunctionName[arg1, arg2]:= functionbody;. I told them that whenever they encounter a colon followed by an equal to, they were seeing a function definition. And that sometimes the programmer uses a capital letter in the beginning if they are sure that the name of the function will not clash with some already built-in name. The idea of a name clash was confusing to them so I gave them an example: if two people in your class have the same name and your teacher calls out with that name, what do you think will happen? They replied that both people bearing the same name will answer, and they will not know who exactly is being called. This was all I needed to explain name clashes, and they understood why they should try as much as possible to avoid defining names that will clash with the built-in names. I use these kinds of examples all the times to explain concepts that might be difficult for them to grasp at first.

There are other ways of managing name clashes in the Wolfram language like the package management system, but we can get away with that without knowing all the details of the Wolfram system which is enormous but enables you to use it starting with just a few primitives. There are other function definition modalities suited for different situations but knowing the basic is all that is required in the beginning.

The Wolfram programming lab is made up of explorations, a single notebook file with programs that deal with one topic after the other. In the bottom of the explorations are links to go further in the current exploration you are engaged in. This is especially good for older learners but would not be appropriate for the kids I was teaching. Not because it was too difficult but because it required us to type code and reason in higher concepts. The kids were still learning to locate keys on the keyboard so this was strenuous for them at this stage.

Initially I would work with the kids through each exploration, sharing tasks with them but at this stage it was time to let them do some independent work. So I would open an exploration for each kid and tell them to read the instructions and execute all the code contained following the instructions on modifying the code that was already specified before each code example. I would watch them to guide them on any difficulties they faced, and they excelled without much difficulty until they faced the problem of deploying a program in the cloud and running it through a browser.

I didn’t start a complex definition of what the cloud was, that would be completely incomprehensible to the kids at this stage. I simply stated that the Wolfram function CloudDeploy puts things in the cloud (described as a place where many computers where connected to each other and working as one which you can access with your browser and internet) and they took my statements as is. This was the most enlightening experience for me in the entire course of lecturing these kids, to see the level of automation that had been built into the Wolfram language and to imagine how deploying stuff to the cloud in any other language will come with some difficulties and that in the past even creating a simple website was some job more suited to an expert programmer with many different skills.
With a single function call they were able to deploy a function on the cloud and the output of that function was a URL which could be clicked to open a web browser. I didn’t explain these concepts I just told them to execute the code, they did and it returned a URL; I said put the cursor on that and they did and a hand appeared, I told them this means you can click that blue thing with an underline, this was all that was required to deploy a full interactive program on the cloud. Lunching interactive

programs on the cloud and clicking a link to access it through a browser was a very exciting experience for them.

The most fun filled aspect of their lectures was when they had to interact with the Wolfram system using natural language. The Wolfram notebook can accept natural language queries and produce appropriate results by activating the necessary underlying systems. All that is required is just using Ctrl + = on the keyboard, I think Ctrl is Command on Mac – we were using windows. After showing them examples of how to get the image of a cow using natural language, they were quick to try out names of their favorite cartoon characters and as the results came this inspired them to keep learning about programming.

Another exciting feature for the kids was the Manipulate function that produces some interactive graphical user interface that enabled them play with a program by just clicking buttons. By executing a single function, Manipulate, they were able to create this beautiful interactive GUI program without doing any GUI programming. Kids love clicking buttons and this added to their excitement. They were motivated to keep learning, with the hope that programming was exciting with many fun things to discover. It wasn't only the fancy functions that kept their motivation up, the output of some of the programs were graphical in nature, which was something that elicited much excitement.

After gaining sufficient practice, it was time to strengthen their keyboard skills. Without memorization, the were excepted to type out the code that was already written in Wolfram lab programs. This improved their keyboard skills and also gave their brain detailed exposure to code. Before now they were executing code, now they had to write. The issues they encountered here centered upon discovering keys on the keyboard, learning how to use Shift key to get at a second function of a key, etc.

This practice took a long period because their familiarity with the keyboard grew gradually, they had never been exposed to computer keyboards but had access to touch devices and what they did mostly was play games with their devices and not use the keyboard. While they struggled to search for keys they were typing code and gaining familiarity, which is the most efficient way to learn anything.
There is a useful code completion function of the Wolfram notebook, which came in very handy as they typed code. I encouraged them to utilize it as it enables them type faster. After a few glitches they became very conversant with this feature and utilized it appropriately to enhance their typing speed.

When I was comfortable with their typing speed, we went into documentation studies. I didn't make them type out every piece of code in the explorations, there are a lot and that will not be fair or creative. I gave them generous homework so that they could practice the essential skill of locating keys on the keyboard. They really got fast in locating keys even without using a standard typing tutor software. The goal for these kids is not typing large amounts of code, but getting to the level where they can write single function programs and thus proceed from there. This is another advantage of Wolfram language; you don't need enormous amount of code to do something really useful. A few lines of code can do something that would need a hundred lines in some other language making the Wolfram language the most appropriate for children. Children need constant feedback on their activities and if it takes too long for something to make sense, their patience will run out. I selected explorations where I would tell them to execute code; ask all kinds of questions about the code and in some cases make them type out the code they were seeing.

Although programming was about writing code most of the time I didn't want us to rush to this stage. Stephen Wolfram the creator of the Wolfram language has created an excellent book for learning how to program in the language, when it is time for them to proceed to that level they will just use the book, going chapter after chapter and trying their best to solve the problems in them. We are not in a hurry and I plan on taking them through Wolfram programming lab the second time after we have finished exploring some parts of the Wolfram documentation.

The exploration we are doing with the documentation is to make help them learn how to query the documentation and extract information that they need at the moment. I give them a quest like generate a random integer, then explain what an Integer is and what Random is. After this I task them with querying the documentation to find out information about random integers and how to generate them. I also help them navigate the documentation, filtering out results that they do not want to get at what they want. This is a skill that will take several months of practice to get right and it is very important and cannot be skipped. Most of programming involves consulting the documentation from time to time because the Wolfram system is a very large system and you cannot hope to memorize everything in it. It's better to build familiarization rather than brute memorization. If you are one of those memory masters, then go ahead and memorize the functions and their abilities, it will do you much good.
The art of querying the documentation is vital knowledge for every programmer and by asking the kids questions and telling them to search for the answers in the documentation when they cannot remember will aid them in the rest of their programming career.

After going through this documentation exploration, making sure that they have honed the skill to some degree we will start all over from the beginning of the explorations. There are a lot of exploration in the Wolfram programming lab and it will take us sometime to go through them. With the documentation exploration I think we will need a full year for this endeavor. Yes, they are young children and we have to go gradually understanding that their brain is still developing.

The importance of teaching kids programming:

Because they are learning programming at such an early age it will aid in all aspects of their reasoning, because they have drunk deep from programming, their ability to think rationally and systematically will be seriously enhanced. Their young brains are encoding the knowledge contained in all what is being taught at a deep level so subjects like mathematics especially at the level of calculus will come easier later on in life because they have been exposed to so much abstract thinking at such an early age.

People always underestimate the learning ability of children especially those under 10, they think that everything must be reduced to toys for these kids to learn. From my experience with these under 10 kids I conclude that rather than bring high concepts down to these kids by dumbing them down, rather bring the kids up to the higher concepts with clear explanations and learning to ignore inessential information and you will be surprised at how fast they will assimilate the concepts.

While I'm teaching these kids, I sometimes pause to listen to myself speak to them. I notice I am using many abstract words and having them fully comprehended by these kids. You can hear me say things like execute the code, followed by: change that first argument to an Integer between… I am usually amazed that my instructions are executed without hesitation and we have just been learning for a few months. This demonstrates the amazing power of the Wolfram language.

With their knowledge of Wolfram language and computational thinking they will be able to apply themselves more efficiently in many disciplines without much difficulty. Abstract reasoning is usually introduced by mathematics much later on in Algebra and eventually calculus but my argument is that if you introduce programming early on, because it is interactive like in the Wolfram language, then when you start dealing higher issues later on, the child has already been exposed to abstraction and the brain has adapted and will welcome those ideas easier. This skill is sure to carry on throughout the career and life of the individual.

With my current understanding from practical experience of teaching kids I now have the confidence that with sufficient patience and time I can even teach kids python programming or calculus. The problem with python and other languages is that they require so much extra structure to be built into them if you want to introduce them to younger individuals like under 10 kids. And learning calculus from a textbook is not fun enough for this kids. If you have to teach kids calculus then you can use Wolfram alpha, another product of Wolfram research to specify the problem and have the steps to the results returned in an interactive form that can be played with. I am confident that a 5-year-old can learn Wolfram language programming and get to the level of writing small programs in 2 years as far as the child is disciplined and stable enough. From experience a 6-year-old is usually disciplined enough to learn programming without much stress on the instructor.

You must also understand that many under 10 year olds are prone to loose attention from time to time while learn, so you should not make the lessons too long. I find out that 30 minutes at once is enough time to both capture their attention and impart sufficient knowledge. When there is a lot of class work that requires that they type code, etc. then 1 hour is ideal. Their attention will waver but they can be brought back to focus. One of the first code of learning I gave them was to tell them that a programmer must be able to focus on the task at hand for a long time. And whenever they are drifting away I remind them of this code of learning and they get back in shape. Sometimes I am concentrating on one student and so the other would drift away, I usually request that they learn from the person I am focusing on – but sometime they still drift and I let them. You have to be very patient, they are children and you should make the task fun and easy for them so they want to learn on their own later on.

What happens after sufficient documentation experience? We go back to the beginning with them opening a new empty notebook to start doing some real programming experience. The will look at the code in the exploration documentation but will be required to write out a modified version that does something else in the new notebook. This is the beginning of practical programming – they are now exposed to the real notebook environment not the exploratory notebook. They are free to look at the code while copying to modify but this time around I will give them a high level instruction. If we have code that generates a list of 10 random integers within a certain range, I will tell them to write code in their notebook that either increases the range of numbers generated or the length of the list. Sometimes I will require that they do both. After I have gone through most of the explorations giving them this kind of learning experience we will be ready to face Stephen Wolfram's book, an Elementary

Introduction to the Wolfram language which comes included in the Wolfram programming lab software package.

During this second round of exploring programming lab, we will also visit those further exploration areas that we skipped the first time around, I will assist them as they write code on their own to solve this section. They should be equipped with enough knowledge and experience at the end of this exercise to face the book.

With the book I will predigest most of the information in it, while also giving them the opportunity to read sections of it themselves then access their understanding before moving forward. We will treat each chapter slowly, gradually working section by section rather than chapter after chapter.

The book is loaded with so much instructive examples, but some might be too advanced for young learners. I will use my discretion to select what they can solve for now, or reduce the complexity of some others. We are not required to solve everything so we will choose what is fairly easy for now and face the more challenging ones at a later time.

As far as a child can read and do basic arithmetic, they are ready for programming in a highly engineered environment like the Wolfram programming lab. It is in my opinion that all children should be exposed to programming just the way they are exposed to language and arithmetic. Even though the children do not want to end up as software engineers, the computational thinking skills they derive will help them in any career of their choosing even if arts or poetry. With the built-in capabilities in the Wolfram language you can do all sorts of computations on all sorts of things, sometimes with a single function.

✻ ✻ ✻

42. COMPUTATIONAL X: COMPUTATIONAL THINKING IN HIGHER EDUCATION

Recently I got questioned by an academic, she wanted to know how Python programming could be useful to somebody who studied Biology or Linguistics. She had studied these disciplines the traditional way without augmenting her studies with computation. Recently she had been advised by someone to study Python programming and wondered how it would fit with what she already studied in the university.

This question got me thinking about what Stephen Wolfram had said some time ago about Computational X, the X being a placeholder for other traditional academic fields that were beginning to have computational aspects. For example, these days we have Computational Biology, Chemistry, etc.

The infiltration of computation into traditional academic fields have given birth to the Computational X's of today, as other academic fields start having computational aspects, we will witness computation sweeping over greater areas of the academic world. Soon we will have stuff like Computational law, etc.
In an earlier part of this work we went into details explaining the concept of computation and used that definition as a base for defining computational thinking.

In answering the individual who posed the question I talked about at the opening paragraph of this section, all I needed to do was define computational thinking and connect it to applications in traditional academic disciplines.

There is no field of academics that cannot benefit from computation, as far as there is information to manipulate. Computation is automation of manual information work, in the early days it was about numbers and mathematics but nowadays it's about every conceivable kind of data. The amount of information generated in various fields is daunting, terabytes of data are being generated every day from all kinds of scientific endeavors, not just the math-centric ones. In order to gain useful insight and make discoveries, we have to go beyond our manual tools of pen and paper or merely eyeballing printed data. We have to incorporate some kind of computational tools into the mix.

Experiments are performed in many fields of scientific endeavor and when these experiments are performed, be it in biology, chemistry or physics data is always generated. Traditionally the most computation scientist ever perform on these data is usually that which is packaged in a software tool like IBM SPSS etc. These tools enable the scientist do statistics on the experimental data and thus generate insights.

The limitation of these prepackaged systems of computation is that they limit the breadth and depth of enquiry, that is what kinds of questions can be asked. Some of these software packages are also scriptable and thus have some kind of programming embedded in them, but this scripting languages are more specialized to the environment of the software you are working on and thus limited to standard methods.

To probe information in a multidimensional way, we need a much more generalized tool like a computer programming language. It's just like in a calculator; an electronic calculator has a lot of prepackaged computation available for the user at the press of a button, but is limited to what the designers thought were important enough to include. This limitation is not all that obvious when dealing with a software tool like a statistics package, but it is there. The designers of the software package had a narrow goal and were not providing a generalized system for performing all kinds of computation but a high specialized environment for performing only a fixed class of computation.

All these limitation dissolve when you are working with a general programming language, you are not limited to anything. If the library of functions you are using does not contain all that you need, you can go ahead and design any custom nonstandard tools that you need.

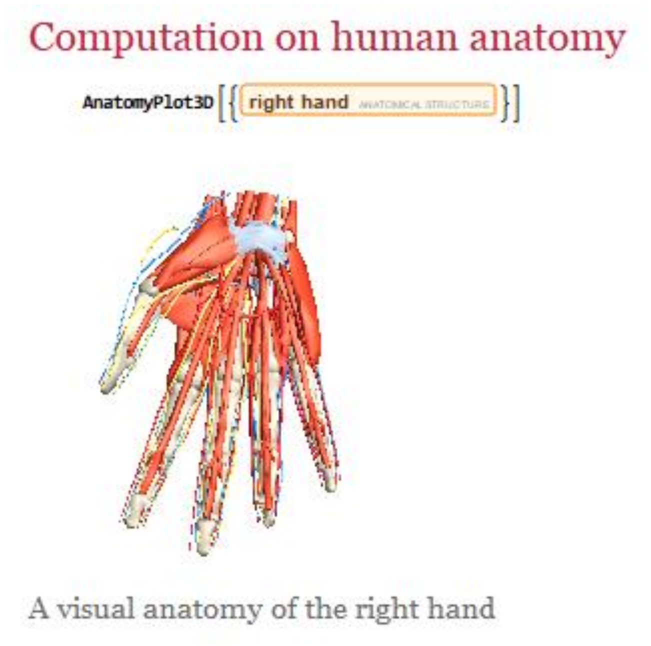
Most people in non-computer science disciplines do not want to write full blown software, so teaching them the general computer science will not be very useful for them. They want to be able to perform general computation in their fields and thus they need a kind of interactive environment where they can probe ideas, develop models and systematically analyze data using standard and even non-standard techniques.

Interactive computing, where you write a small piece of code that executes instantly without compilation in a notebook kind of environment is the exact kind of system that people in different academic disciplines need to incorporate computation into their workflows.

The Computational X's are not limited to scientific fields, computation can be applied to everything from the Arts to Law and even Philosophy. The mindset to hold when thinking of applying computational thinking to any field is this: as far as there is some information to deal with, be it numbers, words, images, videos, diagrams etc. then it is computable.

Any human endeavor can be made computable so it is pertinent that everybody in society learn a systematic language like a programming language to be able to go beyond natural language in their thinking about their respective fields.

Even fields like the study of Anatomy that one might think may not have any need for computation can benefit from the power of computation. On the next page is an image of the visual anatomy of the right hand. It can be manipulated symbolically in any way you deem fit.



Computation on human anatomy

```
AnatomyPlot3D[{right hand ANATOMICAL STRUCTURE}]
```

A visual anatomy of the right hand

Another powerful reason why academics should learn how to use the Wolfram language is to enable them take advantage of one of the most powerful publicly available tool for modelling systems of all kinds: from electrical systems to all kinds of engineering systems out there. The name of the tool is Wolfram System Modeler; the reason you need to know Wolfram language is because it is embedded in the System Modeler software system like a scripting language which you can use to automate your tasks when modelling systems.

Wolfram Mathematica is the most powerful platform for doing computational thinking in, this not a publicity campaign it is fact! Another big system used many scientist is the Python mega package called Anaconda. I am not going to do a pros and cons thing about these platforms, I love python and I cannot python bash because I am now using the large Mathematica system. All I can say is that in my opinion if you are an absolute beginner and you want to quickly start doing interactive computing as soon as possible, learning as little as possible about programming and computation, then you should go for the Mathematica system.

Mathematica started out as a system for doing computer algebra, the system is written in Wolfram language and has been around for about 30 years. The Mathematica system has about 6000+ functions for doing all kinds of computation in all kinds of domains. The functions are developed by advanced algorithm designers and make use of the latest research in the field of algorithms and of the associated domain which is being made computable.

The Mathematica system has an interactive environment called a notebook, if you have seen the graph I produced in the section of this book called Algorithm analysis, you must have experienced firsthand some of the visualization capabilities provided by the system. I usually include the graph and the code that generated it, it is usually a one liner and I bet that most people with programming experience in other languages but not the Wolfram language can already guess what the code does without having to know much about the language.

The Wolfram language is also minimal on syntax, you can start programming in just 3 days for an absolute beginner and in a few hours for a programmer. The simplicity makes it general and very powerful for expressing any kind of computation. While most languages where built to automate assembler, or automate a language like C, The Wolfram language was built from computational primitives.

By computational primitives I mean ideas harvested from the field of theoretical computing research, it expresses computation in the most succinct manner possible. Because of this computational sophistication it is able to express the computational structures contained in any domain from pure mathematical research to nuclear physics, economics, social sciences, visual arts, etc.

The problem with building computation in any other language is that, you will have to build in so much abstract structure into the program in order to capture the model you want to make computable, this is because of the way the languages are built. Other high level languages, even interactive ones where built to either automate some low level computer language like assembler, or to tame a very powerful but dangerous (to the novice programmer) language like C. Python was written with C, Java too. Because the philosophy behind these languages were to automate low level functionality, i.e. build up from a lower level abstraction, they were not built to express pure computational ideas and thus require a higher tower of abstraction to be built in order to capture the model/system you are trying to automate with software.

Primitive structures in most high level computer languages like the FOR looping construct where actually built to abstract and simplify the rigorous process of setting up looping constructs in lower level assembler which used much more primitive mechanisms like JUMP and GOTO. In one system of thought, the FOR statement might seem fundamental but it is not. The idea of looping, which involves repeatedly executing a region of code is fundamental to most computation but not its specific implementation in the FOR statement. In Wolfram Language FOR is a just a function like any other function in the system, it is not a primitive computational structure. An example of a primitive computation structure is the replace mechanisms for transforming expressions. From these primitives the entire Wolfram system was built.

From primitive ideas like functions we can build the so called FOR looping construct which is available as a primitive in other programming languages. You can write programs of any Scale in the Wolfram language without using a single FOR statement, using purely functional methods to transform your data. There are other iterative structures like DO in the Wolfram language if you are bent on doing procedural kind of programming.

The IF statement is another example of an important implementation of a fundamental requirement in computation which is decision making. But like other "fundamental" computational structures available in the Wolfram language alone like the NESTLIST and FOLDLIST functions it is just another standard function.

The key point here is that despite the fact that we build layers of abstraction by looking for definite low level patterns of consistencies, bundling and naming them to conquer them for easy reuse at the high level of abstraction we want to work in, we should look at these patterns of consistencies as just another layer of abstraction and not absolutely fundamental, this is because we could design programs in other styles that bypass these so called "fundamental" structures while still fulfilling our purposes.

The reason I am emphasizing the fundamental nature of the Wolfram language is because this capacity enables you to build the computational structures of any domain easier and in a much more straightforward manner without building too much abstraction above the basic language. In other languages you are forced to use some very limited structures to build your abstractions, these might require using computational structures in an indirect way to achieve your goals. It is this lack of straightforwardness in expressing computational ideas that makes it hard for the new programmer to get into that programmer state of mind due to the large knowledge barriers of conventions you have to understand.

In Wolfram language computation is much more explicit and straightforward with as little convention as possible governing programming activities. It is actually much more direct to just express ideas as they are rather than building your own bag of tricks from a limited pool of fixed structures before you go ahead to express your own ideas.

What makes the language the most suitable for computational thinking is also because of its highly integrated stack of functions, which you can use in almost the same way as you would build an expression in natural language. The naming convention and usage convention of the functions is consistent throughout the entire Mathematica Stack to ease expression of ideas. This is a major problem for a mega package like Anaconda. Anaconda allows you to install many libraries available for doing specialized computing in some field, the problem is that this code was created by different teams working independently who have differing ideas about how a computational interface should be designed. Because of this when you are working with any of these libraries which you import into your main Jupyter notebook you have to know the conventions of the libraries you are using.

The Wolfram functions are consistent across the whole Mathematica Stack making it easy to predict the usage convention of any function. The Wolfram system also express the ideas of meta algorithms, algorithms of algorithms. How this works is that if you are tasked with sorting a list of numbers or anything, you don't need to know the exact sorting algorithms to use, all you need to do is use the Sort function and depending on the nature or size of the input, an algorithm will be chosen on the fly that is the best for the kind of input you are presenting. The Sort function in this case is working like a meta-algorithm that uses other sub algorithms to perform its work. The code that analyzes your input and determines the best sort sub algorithm is the meta algorithm. While doing your own computation you can build your own meta algorithms speedily by wrapping a particular computational process that has become fundamental to your system in a high level function. Do not be deceived by the small size of Wolfram language programs, a simple 10-line program might be using algorithms that can be about 150,000 lines long in total.

To increase your relevance in your field and accelerate the rate of innovation you can produce I advise that you learn programming and computational thinking, you don't need to write full blown computer software. Just get your hand on a Wolfram system and start interacting with your notebooks to make discoveries on the fly.

The regular argument against the Wolfram Mathematica system is the cost of purchase. Although this was the case in earlier times, this is no longer an issue. The pricing has been streamline with very cheap single user educational licenses now available. Stephen Wolfram the CEO of Wolfram Research was once an academic and is an educator at heart so you know he is fully aware of the financial challenges students face. To get started, you don't need to purchase a license or install anything to start taking advantage of the enormous computational capabilities of the Wolfram language, all you need is a web browser and some internet connection to connect to the Wolfram Cloud and start doing computational thinking online. The cloud is free for entry level stuff but if you want more computational muscle you can simply buy extra cloud credits to expand your capacity.

* * *

43. WOLFRAM LANGUAGE: A VERY POWERFUL SYMBOLIC LANGUAGE

Of all the kinds of programming languages I mentioned earlier: Functional, procedural and Object oriented, there is one kind of programming language that I skipped mentioning and that is Symbolic languages. I skipped it on purpose because I was going to introduce some ideas earlier before going into it.

Wolfram language is a symbolic language; this is the most general any high level language can get to. We build up programs from symbolic expressions. Rather than talking in abstract terms let's get down to some concrete explanations.

There is this classification system I derived while learning programming languages. I call it levels of freedom. The levels of freedom a language gives you determines how forgiving it will be when you violate hard fixed rules of your natural thinking.

Programming is about thinking, we are trying to solve a computational problem, and we do this in a programming language. Before it used to be mostly in math, but now with languages like the Wolfram language we can just go straight to solving problems in the programming language rather than in math. You can actually do math in

Wolfram language making it more general that the general systematic language of math itself.

When you are working in Assembler you are in a very unforgiving environment. If you have mostly worked in high level interpreted languages, then getting to assembler environment will mystify you. Most of the freedom you experience when dealing in the highly cushioned environments presented in the interpreted languages are taken away.

Things get better when you move up a level to the C programming language but you are still faced with some very hard constraints when expressing programs. If you have worked in a language that provides garbage collection, then that freedom will be taken from you and when you work in C you will have to do your own memory management, which is a skill on its own.

C is very powerful and since it is mostly a wrapper over assembler, actually you can still include assembler in C code, it gives you direct access to the machine. This is good if you want to control resources by yourself, which is a black belt kind of skill in computer programming.

As a beginner your focus on learning a programming language might be getting to compute things with the computer doing all other low level tasks automatically. Except your future career direction is actually writing system software then you will need to master controlling resources by hand.

C++ even adds more complication upon C. you have access to the underlying machine like in C with the added complexity of classes. I usually recommend that you do not start learning programming with C++. You will just make your life unnecessarily difficult. It is better you start with a very high level language like the Wolfram language that enables you do computation at the highest possible level until a time when you have developed sufficient systematic thinking skills, then you go lower and lower and eventually even assembler will be tolerable. Unlike other things you will learn in life where you have to start from the most concrete to the most abstract, like in math you start with arithmetic, then algebra and later on more complex stuff like calculus. In programming you can't follow the same logic – you have to start with the highest level stuff that hides the underlying concrete machine as much as possible and deal in pure computation, before you start going low and eventually end-up at the lowest level.
You could still start by climbing the abstraction ladder, but don't try to force yourself to master anything that is too difficult when using this second path. Climb up loosely

and reach the top, then reverse your direction to start mastering what you have already experienced before.

After C++ is C# and Java – these are more forgiving languages with far more freedom of expression than the previous C, C++ or assembler. In these languages even though you have reached a level with a lot of automation, you are forced to think in only one paradigm, which is Class based thinking.

Object oriented programming is great but I think when you are doing this kind of programming, although the proponents will talk about how it simplifies the solving of problems, I think you are doing more of project management than actual computation. And sometimes these enforced project management features are not really necessary even when you are writing large scale programs. It is just a style of programming that appeals to some people.

At the level of C# and Java you have more freedom from the machine but with the added over head of classes. In a textbook on object oriented programming you will experience some toy programs that demonstrate the use of Classes in defining computational problems, but in the real world class hierarchies are usually deep and very complex. Some features of Java like Interfaces really freak me out! Or Abstract classes. These are features of the language that make up for the shortcomings of the philosophy of the language itself and have no real bearing on the real computational issues that bring us to write computer programs in the first place.

In my opinion, object oriented programming style is only good for describing Graphical User Interfaces. This is because the abstraction of Classes enables us to specify features of a GUI easier than just functions because most GUI structures have data and behavior, which is well suited to the object oriented paradigm. Java is so strict in its object oriented design that a program starts with the definition of a Class. Even if the task you want to do is adding numbers, you will have to do it by specifying a class and main method and doing so many stuff like creating variables with types before actually computing the sum of two numbers.

Another personal issue I have is with static typing. A typed language requires that you specify the type of an object while defining it. If you were creating an integer variable in a language like Java or C you would have to declare that it is an int or a float if it is floating point, etc. typing has its benefits in compiled languages, but when computation is what you're interested in and not making life easier for the compiler then dynamic typing in languages like Python or Wolfram language is the most optimal way to write programs. In Java if you specified a variable as an Integer type, you will be careful about the bounds of the Integer. If the number grows beyond a certain magnitude your computation will be erratic. So you have to choose variable type carefully. When a type is created certain amount of memory is allocated. If you know before time that you will only be needing a number between a certain range, then you can choose your type accordingly. Although eventually you will get used to the idea of typing when you use the language for a while, this is usually an issue that should be left to the machine. In Wolfram you can deal with numbers of any size, of course limited by your computer hardware, and not think for a moment what type you need to use.

Python comes to free us further from the chains of strict object orientation by enabling us work in many kinds of programming styles. You could do C like procedural programming or LISP like functional programming. It also has Class mechanisms which I think are superior to that of Java. I fell in love with python before my Wolfram days because it allows more freedom of expression than all the languages I mentioned before it. You could actually just type 2 + 2 and compute it straight ahead without much boilerplate stuff like import I/O (Input/Output) libraries and call on long function names to do simple stuff like printing results.

Python is a language that is very flexible and allows you focus more on the computational problem you want to solve and less on the machine. Unlike C/C++, C#/Java, it is dynamically typed, meaning rather than doing something like int i = 2, to define an integer variable in the languages I mentioned, which are statically typed language, you do something like i = 2, and the type of the variable will be determined automatically. You could define a new list by declaring a variable aList = [“firstItem”,”secondItem”] instead of doing something like ArrayList aList = new ArrayList() in something like Java.

The ability to do many things on the fly knowing that the system will handle much of the details makes you focus more on the problems you want to solve. In python scoping is by indenting which is more natural than keeping track of curly braces, which indicate the beginning and end of a block of code. And also the language is interpreted so you don’t need to do compilation before executing the code. You could write code, run it get results and then go further to instantly modify your code and run it again if you are not satisfied rather than doing a compilation step.

Most people engaging in modern programming activities like AI and Data Science use python programming language because of the ease and flexibility. AI programming is abstract and complicated so therefore you will need a language that was not too constrained in expression. Data science requires a lot of experimentation so you need a language that will enable you recover from mistakes and wrong theories faster than a compiled language. Python allows more exploratory programming, the kind of programming where you don't know beforehand the path you want to take towards solving some problem and you just have to hack your way at solution after solution till you find what you are looking for usually in an interactive environment.

The default python installation is good for pedagogical purposes, that is when you are learning the language. It has a lot of built-in functionality which is a good thing because you don't want to import a function to do some basic tasks. The standard library is also extensive and has additional functionality that you can also import into your programs when you need them, but these do only the most basic tasks. For some tasks that are nonstandard, you will have to download and install a package written by someone else and hosted externally on some web based repository. The python package index is one of such code repositories for locating packages for doing all kinds of specialized tasks. So it is a good idea to explore the documentation for what you want to do, and then proceed to the python package index, before deciding to reinvent the wheel.

Building your own python system by installing packages you need is a tedious task because of version clashes and other problems. Some packages require the installation of other packages and sometimes they go as far as specifying the exact version number of the packages required and if you have another version installed that you are using for another program that requires it and then you have a program requiring another version, conflicts can emerge.

In your early days of learning Python, you might not encounter these problems, but as you get advanced and have to build stuff that requires a large amount of external libraries these problems will crop up. Another common issue is the version of the python installation required to run a program. Many programs need to be run on a special version of python, for example on Python 2 or Python 3. Even a python mega system like anaconda cannot save you from python version requirements, there are different anaconda installation for either Python 2 or 3.

Anaconda is mega python system, which unlike your default python installations comes with a lot of external libraries installed for doing all kinds of things. Apart from having a huge number of preinstalled libraries with the ability to install a new one by simply ticking a list with the downloading and installing automated, anaconda also provides a fully featured IDE and interactive environments where you can do both the regular native python IDLE kind of interactivity of go for a full notebook environment for more powerful features.

Anaconda reduces the time and effort required to set up a python environment for specialized task like scientific computation, data science and AI. Doing such a setup on a stock python installation can usually be difficult for the newbie so it is advised that you just go straight to the pre stocked anaconda installation to get started on bigger projects.

Python will give you so much freedom but JavaScript will give you another level of flexibility. I wasn't really into JavaScript in the early days because my interests were different from the use cases JavaScript was usually applied to, but nowadays you can't get by with web programming without JavaScript somewhere. The language is also flexible and nowadays can be used for more backend stuff, away from the browser. Like I have said previously knowledge of JavaScript is pertinent on your journey towards programming mastery and especially if you are going to work on stuff that has something to do with a browser. It has some very useful language structures of which my most favorite is its definition of objects. You should take some time out to explore JavaScript even if you won't have to use it anytime soon, it's an enlightening experience.

A cool language that people don't use that much these days but was once the best to learn deep ideas in Computer programming is the LISP language. LISP was well designed but too constrained. Just like Java constrains your thinking to classes, LISP constrains your thinking to lists. This kind of hyper constraining of your thinking is not something I like when thinking about the solutions to computational problems.

When you use a particular language often your thinking will adapt to it, meaning when you are thinking about some computational problem, your brain will be creating structures in that language. Although you can try to consciously adjust your brain to think in any language, but it is usually fixed on one kind of language, your favorite. In my early days when faced with a problem, I will think of it in python terms. What are the functions I need to define etc. this happened for a long time – it's hard to think abstractly about a computational problem without constraining it to a particular programming language, even if you know multiple languages. I know some people could be thinking in pseudocode or even mathematics, but my training allows me to think mostly in the computer language I am most engaged it.

Although you can solve any computational problem in any general programming language, some languages allow more flexibility in your thinking than others. If you are thinking of the solution to a problem in C, you would actually have to include solutions to memory management issues as well as other quirks of the language, but if you were using a language like Python or Wolfram, then you would focus more on the computation you want to do, maybe like finding the best model for some data, than on issues of memory management. Wolfram language makes this even easier by freeing your mind more away from not just the physical computer but much of the abstract machinery that makes it possible to model a particular domain in a language.

When I talked about abstraction in languages I told you that we abstracted away from the physical machine right up to the level of the high level languages and stopped when structures like the function were arrived at – although some people went up to wrap highly related functions and data together in a Class, this only eased up program management and was not in any way fundamental to core computational requirements. A function is the most necessary high level abstraction needed to compute anything on a general purpose computer.

Now there is another dimension of abstraction which is much subtler to define, it is the area of building abstractions to represent the domain that you are solving a problem in. This is actually the programming itself, but it is a more specialized construction of abstract structures in the language to represent a particular domain. This is not the more generalized abstractions from low level to high level languages that we have previously encountered but the abstractions generated from this kind of specialized domain stuff is what programming is all about.

A high level programming language offers some basic structures which are abstractions from the underlying level. To do something like IF…ELSE in a high level language, you get a bunch of assembly instructions together and the compiler is responsible for translating your high level constructs to your low level machine instructions. But when you now have a basic structure like IF…ELSE, to solve some problem in a particular domain, you will have to build more specialized abstractions using them alongside other high level structures to represent the problem you are solving.

To solve a computational problem in a high level language, you will first of all have to model the solution using the structures in that language. Structures like IF…ELSE…, Functions, etc. will have to come together in certain ways to enable build the abstract representation of the model. This is what programming is about and this is what understanding a program really means. Understanding how high level programming structures work, that is the syntax of the language is not really a problem in learning a programming language – unfortunately this is what most programming books helplessly focus on. That is why I say you cannot learn programming from books alone.

Understanding a program is actually about unraveling the abstract model another programmer has created, that is why documenting code is so vital. If you don't understand the domain the program is trying to solve, then you will find it harder to understand the code. Most programs are domain specific and require some knowledge about the field to solve them. If it's a program on finance, then a variable might represent an account name or account itself. If you are creating a Banking application in an object oriented programming language, you might have a class that represent the bank itself; another class for representing employees and still other for representing customers. Apart from people you can also represent ideas like an Account with a class and even an Account type can also be a class. All the transactions the bank performs, from which teller received a deposit to how much was deposited, etc. will be modeled abstractly in the program and from there physical actions will be represented by several combinations of structures - this is what programming is about. Don't think that because programs are abstract and that they cannot affect the real world, apart from devices like robots that can affect the real world from commands encoded in a program. When you transfer money from one bank account to another, there are a stack of programs that make that possible and thus an abstract entity like a computer program is able to influence the world physically.

If we are writing a program to control a nuclear power plant, we will have to model aspects of the physical reactor in the program and have program controlled mechanisms that capture certain physical variables like temperature in a sensor, and another part of the program which is constantly alive monitoring things will receive this sensor data, perform some computation on it and determine whether it should send a command to reduce the temperature, keep it stable or otherwise.

A currency trader using automatic trading with a computer program will take tick data from the markets, perform analysis and then determine the best decision to take, either to BUY or SELL. Within the trading program itself might by hierarchies of abstractions created using the base facilities of the high level language used. These abstractions could be modelling market dynamics and other structures of the market being dealt with and can provide solutions in the form of decisions which in turn will guide the program in trading activities. In Metatrader, one of the most popular platform for currency trading offered by many brokers, the language used is Metaquotes language (MQL), It looks like C in its syntax structure.

The expressiveness of a programming language depends on how much complication is required to build the necessary abstractions that model a problem from a particular domain in the language. When you start writing code, you are building these abstractions and the language you are using will determine how much code is required to model the solution to a problem.

A well designed object system can represent most problems very well and the idea of using Classes to represent entities in a system and Methods of Classes to represent the actions they can perform is a neat theoretical way to solve computational problems. But in reality, the real computational work is done inside the methods.

What is real computation? The object oriented purist will try to boil down every computational problem to agents and actions but this is not super fundamental. At the most fundamental level, computation is transforming expressions. These expressions might be data expressions or symbols representing operations.

In most programming language there is a hard separation between data and the code that operates on the data. It is this idea that has matured into the object oriented paradigm of programming where Classes and Methods represent code and then you have dumb data they perform their operations on. The first language that showed us that data need not be dumb and you can have some kind of active data was LISP. In LISP the lines between what is data and the operators that work on this data in very blurry. You can view everything in a program as data or as operators. This is what gave LISP so much of its applicative power in the field of artificial intelligence research. LISP consumed a lot of memory, which was very scarce at the time and people complained about the excess parentheses that came with LISP programs especially big ones, but because of the expressive power of LISP, it was very compact and very large LISP program can require at most a 100 lines of code to describe - the equivalent in other languages would require tens and thousands of lines.

I always tell people that Wolfram language came to correct the errors of LISP. Wolfram language is so well designed and so compact that it could capture thousands of lines of programs in other languages in code that is less than a 100 lines. It is the highest level language ever designed – this is not some false promotional statement, yes I am affiliated with Wolfram research but I am not just promoting a “product”. I am trying to let people know that there is a simpler and more elegant way to solve their computational problem and that many large organizations are using this tool to leverage their innovative potential. Before I got attracted to the Wolfram language I had experienced many other languages, many I haven’t mentioned in this work. When I found the Wolfram language I was amazed that such a language existed and was used by gigantic organizations around the world and many people in the programming public didn’t know about it. I was shocked at this, I later found out that this was basically because the language initially came bundled in Mathematica, a gigantic software system for doing all kinds of computation and since Mathematica cost some money people didn't get exposed to the language that made Mathematica possible.

This has all changed now as you can have instant free access to the Wolfram language on the cloud – even Stephen Wolfram's book: An Elementary Introduction to the Wolfram Language is out there on the web for free.

In the Wolfram language if you type something like: avar at the interactive prompt, you do not need to attach it to any variable name for it to make sense. If you execute avar it will just return itself as a symbol. In most language it will be assumed you are trying to create a variable and thus if you try to execute it you will get an error message, something like: Uninitialized variable. We don't get such errors in the Wolfram language due to its symbolic nature. We build expressions from symbols in the Wolfram language so something like avar or an expression like (a + b)^2 is just taken as it is.

You can also create variables in the Wolfram language using the various assignment structures. In most languages the equal sign is represented by a single = while in the Wolfram language it is double like this == because the single version has other uses in the creation of variables.

You can create a variable in Wolfram language by doing something like avar = 10. Once you have assigned a variable to a value like we did in the above expression, we can go ahead to perform other operations on it.

There is another kind of assignment operator, which is the delayed assignment. In the above we saw what is actually immediate assignment or instant assignment where we gave the name avar the value of 10. That value remains until we give avar another value. In the other kind of assignment which I call dynamic but which is officially called delayed, we assign a value to a variable but the assignment doesn't take place until we use the name in some kind of expression or evaluate it.

Before we demonstrate delayed assignment, let us see some other concepts first. When you speak in a natural language like English, we use words. Each word has a meaning which we can usually find out in a dictionary. When working in other programming languages, there is this feeling that you are dealing with a very rigid unforgiving machine, because most languages are just direct abstractions of the underlying machine. In Wolfram language you are working at the highest possible abstraction of the underlying machine so, rather than thinking in terms of the machine you are actually thinking in terms of communicating your ideas to humans and the machine.

We might think that when we are writing a computer program we are actually instructing a machine to do some task, which is true but in reality we are communicating with other programmers and eventually ourselves the solution we are expressing in the programming language.

The Wolfram language is designed in such a way that when expressing the solution to some computational problem, we are writing in what looks like natural language. Apart from being very direct with the underlying machine, it is very communicative with other humans. The code is sometimes indistinguishable from some natural language script written for human consumption.

All programming languages have what we call functions. A function is an encapsulation of an integrated piece of code, without going into further formalities if you have a task you want to solve like generate random numbers – when you write code that actually does this, to make it usable in other parts of a program you encapsulate this code in a function, which has a name. Wherever you need a random number all you need is to call on this function. Wolfram language has about 6000+ functions for doing all kinds of things and you don't need to do any kind of importing like you would do in other languages to bring in additional names to the namespace. All you need to use these functions is just to mention their name in a program you are creating and there you have it.

Almost any basic task in computation you can think of has already been thought about and a function designed for it and included in the Wolfram language system. Your language documentation is just like a dictionary – when you are reading some text in natural language and you come across words that you are not familiar with it is customary that you turn to your dictionary for reference. Similarly, when you are faced with some Wolfram language code and you see some built-In function you don't understand simply turn to the documentation. When you are coding and are faced with some task for which you don't know a function that solves it, before you start writing your own solution take some time first of all to search for it in the documentation. The documentation is structured in sections corresponding to different fields of computational and technical endeavor – seek the field you are working in and drill down and you will find something that does your task for you or something similar and more primitive which you can use as a basic element in constructing your own code.
As an example if faced with the task of finding a random number, we can go to the documentation and search for random number generation and we will see the search results containing list of functions for generating random numbers. If it is random

integers we seek to generate, we will see a function like RandomInteger[], and all we need to do is to read how to use it and then go ahead to use it in our programs.

To demonstrate the idea of delayed assignment I talked about we will use RandomInteger[]

```
avar = RandomInteger[10]
3
avar
3
```

We see that we call the RandomInteger[] function and give it a range of 10. This tells it to generate random integers between 0 and 10. It generates one, that is 3 in this case we assign it to avar. Every time we request for the value of avar we get 3 because we used immediate assignment.

```
avar1 := RandomInteger[10]
avar1
10
avar1
4
```

We create a new variable called avar1 and we used delayed assignment to attach it to the RandomInteger[] function. But this time around when we request for the value of avar1 we get a new value every time. This is because the delayed assignment doesn't actually give a value to avar1 when we do the assignment. But calls RandomInteger[] anew every time we request a value for avar1.

There are a lot of programming scenarios where we might need different kinds of assignment to cope with different situations. In most languages you get only immediate assignment but with Wolfram you get both and some other types which are quite advanced for a book like this. If you have to use delayed assignments in other programming languages, you would need to come up with a lot of wizardry to make that possible, definitely not for a beginner.

In earlier programming language the array was the standard container data type, that is a datatype that holds other datatypes. But arrays are limited in many ways – they only allow you to hold only one datatype. An integer array will hold only integers and a character array will hold only characters. A string in a language like C is actually a character array. When we came to languages like Python, we had this list which is an important modification of the array. The Python list enables you to store different kinds of data types individually or in combination with each other. Due to the dynamically typed nature of python you don't need to specify what type of stuff a list contains.

Wolfram language takes the idea of lists much further. The most fundamental difference between lists in python and in Wolfram is what we call indexing. Indexing is a way of selecting elements from a list by their position. A list of strings like this in python [“oranges”, “bananas”, “mangoes”, “guava”] has the index, i.e. Position of the first element as 0 and the second as 1. This is an idea inherited from the ancestor of python, the C language. Most programming languages also follow this convention. Many programmers think that there is something fundamental about this way of indexing but actually there isn't. it is just a convention inherited from the early days of programming. In this scenario the first element of a list is 0 and the last in n-1, where n is the number of elements on the list. So the index for “guava” will actually be 3 not 4. You can actually get used to this convention if you use it from time to time, but for me it is annoying.

In the Wolfram language a four element list like the one above will actually have index starting from 1 and ending in 4. The first element has index 1 and the last element has index 4. No complication like n – 1 for the last index. We make lists in the Wolfram language with curly braces like this { }. This simplifies your thinking a lot as it removes most of the complications involved in thinking about 0 as the first element.

Wolfram language is also a functional language. If you recall I told you that the function was the highest level of abstraction needed to build up any computation, this is fully proven in theoretical computer science. The classes in object oriented programming are a good abstraction too but it's much of a stylistic kind of abstraction. There are ways of doing object oriented programming even in a functional/symbolic language like the Wolfram language but as you get into actual programming you will realize you don't really need the philosophy of object orientation to solve computational problems.

In Wolfram language we build computations by using either a function and supplying it with appropriate arguments, or with a combination of functions with some nested within others. Like other programming languages there are also ways of defining your own functions. Recall our discussion of different styles of making assignments – we mostly showed how different kinds of assignments were done with a variable returned from a function call but most of the magic of having alternative forms of assignments happens when we make function definitions using them.

Although there is a standard built-in Factorial function in the Wolfram language, let's see an example of assignments in function definition using the recursive definition of factorial:

```
factorial[0] = 1;
factorial[1] = 1;
factorial[n_] := n factorial[n − 1]
factorial[0]
1
factorial[1]
1
factorial[2]
2
factorial[3]
6
```

Wolfram language passes arguments into functions using pattern matching. If you have experienced any other programming language at all you will see that what we did in factorial[0] = 1 would seem very weird to you but is very normal to us Wolfram language programmers. This is because in most languages when you define a function that takes arguments you must create a name for that argument. It is during function calling that you supply the value of that argument to call the function with, but in Wolfram language you can do something like we have in the first line of the above code.

What you are actually doing with factorial[0] = 1 is that you are specifying a pattern, that is factorial[0] and saying that wherever you see that pattern replace it with 1. This is a very powerful feature of Wolfram language programming and when you get into programming in it you will see how much expressive power such a feature provides.

A function definition in Wolfram language is just a pattern and when you call a function you are requesting that whatever matches this pattern return what is on the right hand side. In the first and second lines of the code we use immediate assignments which we saw earlier. In line 3 of the code above, factorial[n_]:= n factorial[n − 1], we see another example of delayed assignment. If you look inside the square brackets after factorial you will see something like n_ which is also a pattern object. The actual pattern is just an underscore _ which represents anything, while the letter n represents what was matched by the underscore. In Wolfram language parlance, n_ is read as match anything and name it n.

Why do we name it? This is so that we can use it in the body of the function definition which is after the colon-equal, := sign, n factorial[n − 1]. These are all very powerful features that makes Wolfram language the most general of the general programming languages. The primitives upon which the language is built are super fundamental giving you so much more expressive power than you have in other programming languages. This expressiveness gives you way more mental freedom when thinking about all kinds of computational problems.

Recursion:

In the factorial example above we witness an example of a recursive function definition. Although this is usually introduced later in other programming languages, after you have done basic function definition in Wolfram language, it is so much fun and simple that I introduced it first.

The recursive part of our definition is n factorial[n − 1]. Although a typical computer science book would go into all sorts of detail about recursion, I will not do that. Recursion is complex and getting to the stage where you can write your own recursive function is not beginners stuff but I can show you a simple way of understanding our factorial example above.

The space after n in n factorial[n − 1] is actually indicating multiplication – and the statement factorial[n - 1] states that where ever you see the pattern factorial[n] compute n multiplied by factorial[n − 1]. A concrete trace of the execution of this code will aid you. Let's manually compute factorial[3].

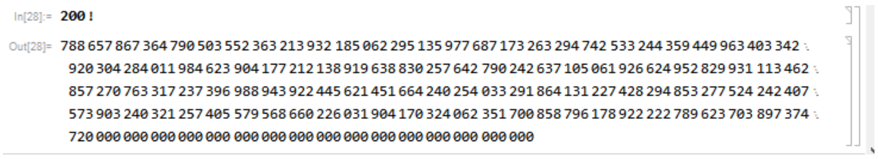
At first the pattern matcher finds the definition factorial[n_] := n factorial[n − 1] and then when it encounters factorial[3], the underscore _ matches 3 and it is assigned to n. Then the expression on the right hand side becomes 3 factorial[3 − 1]. The computer is very detailed in its execution of anything and that is what we are trying to mimic with this execution trace of our making. The next step in the execution involves transforming 3 factorial[3 − 1] into 3 factorial[2]. All these results are stored in the computer memory. In the next step the pattern matcher finds factorial[2] again and expands it like it did for the first factorial[3]. The right side becomes 3 * 2 * factorial[2 − 1], this is further transformed into 3 * 2 * factorial[1]. Finally the pattern matcher finds factorial[1] which it remembers has an immediate assignment in the code which results in 1. Our final computation then is 3 * 2 * 1 == 6.

You just went through the execution trace of a simple recursive function which I guess was a very enlightening experience because it was for me.

The recursive problem we defined above has a subtle problem, each time through the execution it recalculates everything it has ever calculated, you might not fully understand what I just said until you have more experience but you should just know that because of that recalculation when we get to higher factorials like above factorial[20] the program becomes slower. There is a way to fix this problem in the code with what is called memoization. By simply including a single modification to the code resulting in, factorial[n_] := factorial[n] = n factorial[n − 1]. By including the code factorial[n], every computed result is stored so that they are not recomputed. This makes it possible to compute bigger factorials like factorial[200] and above.

There is a convention amongst Wolfram language programmers that we use lowercase letters in the beginning of our user defined functions to avoid name clashes with built in functions. In this case of finding factorials, there is a built-in function with the first letter as a capital, Factorial. If we had defined our function with a capital letter it would have clashed with the built-in function. There are built-in functions about 6000+ of them as of version 11.3 for doing almost any computation you might want to do. Try to spend some time with the documentation, especially with the index of functions to get familiar with them. It will save you a lot of problems in your development effort.

In the Wolfram environment there is a far swifter way of getting factorials with just specifying the number with an exclamation mark:



```
In[28]:= 200!

Out[28]= 788 657 867 364 790 503 552 363 213 932 185 062 295 135 977 687 173 263 294 742 533 244 359 449 963 403 342
          920 304 284 011 984 623 904 177 212 138 919 638 830 257 642 790 242 637 105 061 926 624 952 829 931 113 462
          857 270 763 317 237 396 988 943 922 445 621 451 664 240 254 033 291 864 131 227 428 294 853 277 524 242 407
          573 903 240 321 257 405 579 568 660 226 031 904 170 324 062 351 700 858 796 178 922 222 789 623 703 897 374
          720 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
```

This is what 200 factorial looks like – it is customary to handle such large numbers in the Wolfram environment and you don't need to think of what datatype to use for the number like integer or double in typed languages.

More about patterns:

The pattern language within the Wolfram language is capable of being a full programming language itself. Actually it is much more primitive that the other structures that we have seen about the Wolfram language. The Wolfram language transforms the expressions which you give it into simpler forms until no more transformations can be done on them again - this is basically how the Wolfram language works.

We have seen the basic pattern symbol representing any one thing which is the underscore _ but there are other pattern constructs. The next we consider is the double underscore _ _ which represents at least one or more things and _ _ _ which represents zero or more things. There are a lot of patterns types for representing all kinds of things in the Wolfram language and since this work is not about teaching you the entire language you will have to read the tutorials contained in the Wolfram documentation for more. My goal here is to highlight what I think are the most interesting and distinguishing features of the language to whet your appetite as a new programmer.

More about functions:

There is another kind of function that is different from what we have encountered in the factorial example above. It is called a pure function; a simple demonstration will make this clear. These are usually called lambda expressions in Python.

```
#^2&
#^2&[4]
16
square = #^2&
square[4]
16
```

In the first line #^2& we define a pure function. The # is called a slot which is a kind of place holder for the argument to be supplied when the function is called. In the second line we call the pure function with 4 as argument, #^2&[4], which returns 16 as the result of squaring 4. In the next line we assign the pure function to a name, square, which we now call in the next line with the argument of 4 to return 16 as expected.
Pure functions exist not just for elegance, there are scenarios in practical programming where you will need to supply a function as an argument or for other purposes. Defining a function somewhere else might not be the best solution all the time in these kinds of scenarios so we go for pure functions.

Rules

This is one powerful feature of the Wolfram language that you will not find in many other programming languages. Below are two lines of code to demonstrate the power of rules.

```
{one, two, seven, four}/. seven -> three
{one,two,three,four}
```

In the first line we have a list, indicated by the opening and closing curly brace { }. Following that is the Slashdot operator /. Which indicates that following the expression on the left hand side is a rule specification on the right hand side. A rule is indicated by the dashgreater operator ->, with the left hand side indicating what pattern to search for and the right hand side what pattern to replace with. In this example we are replacing the symbol seven with the symbol three anywhere we find it in the preceding list.

```
1 + x^2 + x^4 /. x^p_:>f[p]
1 + f[2] + f[4]
```

In the first line of the program above we have the algebraic expression 1 + x^2 + x^4 followed by the Slashdot /. Operator. On the right side of the Slashdot is a rule. The rule says that: scan the expression on the left hand side of the /. Operator where ever you find the pattern x^p_ replace it with f[p]. The p_ in x^p_ is a pattern. If you remember from the section on patterns an underscore _ is a pattern operator indicating anything. The pattern p_ says find anything _ and give it the name p. so x^p_ says find anything in the expression of the form x to the power an exponent named p, use that exponent as the argument to the function f – and thus the result of the transformation on the second line.

Data and Knowledge

Another thing that makes Wolfram language unique is its large data repository and built in knowledge. You have ready and instant access to data across a wide range of domains for instant computation, curated data you can work with instantly. Whether its access to all the words in the English dictionary; the full text of Alice in the Wonderland or the US declaration of independence; pre-trained neural network models; financial and socio-economic data; air traffic data; genomic data, etc. with just tiny bits of code you have this data available.

You don't even have to worry about memory management when dealing with very large datasets, the Wolfram system knows just how to manage the underlying memory architecture in a way that enables you work as efficiently as possible.

You can even make full scale knowledge queries and get computable data with built-in access to the computational knowledge engine called Wolfram alpha, the stuff that powers some of the backend computational capabilities of apple Siri.

In summary with these and numerous other features, Wolfram language should be the ideal entry point for beginning programmers, it's very simply and expressive and also powerful enough to power the computation done in many large institutions.

✻ ✻ ✻

44. VIRTUALIZATION AND CONTAINERIZATION

As a programmer you will sometimes want to run several operating systems on one computer. You can usually do this by dual booting the system, that is installing one or more operating systems on one computer so that when you want to use the one or the other operating systems you will have to shut down one and boot the other.

This can be a clumsy affair and sometimes you might want to work in multiple environments without having to shut down an operating system. You might have windows installed by default on your computer and you want to be able to switch to the Linux environment without shutting down the Windows. A typical scenario is that you want to test some multiplatform software on more than one operating system.

An elegant solution to this kind of problem is provided by a software system known as a virtualizer. A virtualization software will usually allow you to run multiple operating systems side by side on one computer without requiring that you shut any down.

There are several virtualization solutions out there of which the most popular are Vmware and Virtualbox – you get an installation package for your host computer operating system, this is the operating system that is running by default on the computer you are using. After installing this virtualization software, you can now start creating what are known as virtual machines running on your host operating system. A virtual machine is a software abstraction of a computer system, everything from RAM memory to hard disk capacity is specified as a software abstraction. Each virtual machine will have its own operating system and hardware resources that it has access to. Although the underlying physical hardware of the computer stays the same, the hardware resources provided to the virtual machines is an abstraction of the underlying physical resource and nothing concrete – but eventually the virtual machine will be limited by what is available on the physical machine.

To start a virtual machine, you will have to open the management interface of your virtualization software, and start the virtual machine. This usually boots up like a normal operating system running on bare metal will do the only thing different is that it is in the software environment of the host operating system.

The virtual machine can access your hardware resources and in some management software there are methods for even allowing different operating systems share files directly with each other. This is usually a more elegant solution than dual booting a system. You should dual boot when you want only one operating system at a time to have exclusive access to the entire hardware system.

When running a virtual machine, you should understand that the operating system you are running as the guest utilizes physical resources like RAM and CPU cycles apart from what the host is using. so you will need adequate physical resources on the computer that is running the system. If you don't have enough physical resources, then the system will run very slowly and you will be inefficient.

Virtual machines ease the burden running different operating systems on one computer which usually comes in handy when testing software on different platforms. In enterprise environments, were compute capacities are large you can have several virtual machines running different software and presenting a unified coherent interface for clients to access the system for various purposes.

There is another way to run multiple software stacks on the same host computer. These software stacks are packaged with their own operating systems in what is called software containers. When software is provided in a container, it is fully packaged with its own operating system files and runs on the host system like native code. With virtualization you will have to create an entire virtual machine to run just one software. This virtual machine is the entire operating system itself, which usually comes with some software baggage that you might not need at the moment. The virtual machine will require that some disk space be reserved for it during creation stage and when running some RAM has to be made available to it exclusively. In Software containers only essential parts of the guest operating system is required and not the full system.

The most notable software system for creating containers is the Docker software system. When Docker is installed in a host system, it enables you create software containers, packages of software with kernels from various popular operating system. The system operates seamlessly as if you had multiple operating systems installed.

While software might have to be modified to run on different virtual machines created from different versions of the operating system, software containers run without any modification on different host computers with the container management software.

Different situations require different solutions. Sometimes you want to create a full virtual machine because your task is specific to that, sometimes all you might need is a software container. As you become a programmer and understand your task better you will know when to use one solution or the other. The thing to keep in mind is that containers are leaner than virtual machines. Containers use less memory and resources in generally. While virtual machines can be in the gigabytes, typical containers are usually in the megabyte range.

You should learn how to use both containerization and virtualization as a programmer, you don’t know when you will be called upon to demonstrate your skill.

✻ ✻ ✻

45. HOW A PROGRAMMER LEARNS

So far in this work you have seen various topics that you will have to learn and you might have become overwhelmed about becoming a programmer, do not be. I have outlined core skills like learning programming languages but I have also introduced you to some technologies. There are many technologies that I have not and will not cover in this book, many of which you will encounter in the tech world at large.

A programmer must learn a lot and must be constantly learning new things to stay afloat. You are advised to learn a lot more programming languages than those listed here in the core learning path. Exposing your mind to other programming languages, even if you will never use them in your work helps improve and deepen your understanding of computation. Different programming languages have different philosophies behind them and have their own following, although you already have the language you think in a new language will give you a different perspective on how to handle computation and will broaden your perspectives. This will lead to quicker and more elegant thinking when it comes to solving computational problems, your mind will have different modes to access when faced with a computational problem.

You don't have to be an expert at everything you study, of course you have your core field of expertise or what your job requires but you must check out a lot more stuff. When you know a lot of stuff, you will find out that if circumstances require that you expand into a certain field, because you have encountered it before, all you will be doing now is revision and not learning afresh.

You have to master the art of mastering complexity, just like I showed you in an earlier section about navigating large projects, your entire career as a programmer/technologist is a large project with a large number of parts. You will learn how to focus on one small thing at a time, get to a certain level of mastery and then go to another all the while mastering your core skills which are the foundations for everything you will eventually do.

Don't be afraid of new learning requirements, most of the time it's just built on certain primitives you have already encountered. There is no field that demonstrates this requirement to learn new stuff like JavaScript programming. There is the core JavaScript language itself and then there are JavaScript libraries which simplify the art of developing JavaScript applications. There are libraries like JQuery, Angular, React, node etc. there are so many libraries that one might wonder how they will master all of them, but the thing is if you have learnt the core JavaScript language then learning a library is just learning a new modality of access to the language and not learning a new language itself. Mastering the use of any one of these libraries will take extensive time and practice but getting to know them and even use them to build a useful application or modify one created in them will take just some hours or days at most.

I personally cannot keep track of all the technology out there but due to time and experience I have seen the foundations upon which most software technologies are built and I have shared them with you in this work. You must always be learning new technologies, sometimes you just need a cursory observation of the new technology be it a new JavaScript library, some new productivity tool, a new code editor or IDE, a new programming language, etc. sometimes it is enough just knowing what the technology is about and asking yourself if this resonates with you or not. You cannot know if it resonates with you without exploring the technology itself and then realizing if you like it or not. Although I am diehard fan of the Wolfram language, that doesn't stop me from exploring R programming language, a language for doing statistics or Ruby. The more you know of the general field of programming, and software technologies the better you get at what you really want to do.

The art of copy and pasting:

With sites like stackoverflow.com you can ask a question when faced with some coding problem and have someone answer it for you with a code sample. There are many online forums where you can ask questions when facing a problem and have others answer it for you, most of the times these people are more knowledgeable and have more experience in the field. They provide the solution and what you have to do is just copy and paste it into your code, with a little bit of adjustments sometimes. Some companies frown about this practice thinking that they hired programmers to just vomit every solution from their heads without any assistance at all. This is quite primitive thinking because you must be knowledgeable enough to ask the right questions to get the right solutions and incorporate it into the project you are doing. As a programmer I will advise you that if you are faced with a non-trivial problem, it is better that you seek expert advice online rather than try to hack out your own solution. If it is some exploratory project, then you can hack out a solution but if it some mission critical stuff you should first of all seek libraries that have your solution.

You can get more than code from forums and communities sometimes you might get pointers to some research paper or library that solves your problem, something you can't always get with direct search on a search engine except you are an expert at structuring queries. Join the communities of the various technologies you are using and when you are stuck go there and ask for help. Programming is about getting solutions to problems not always mental gymnastics.

Algorithmic thinking:

Many people trying to get into programming always say they just want to learn coding. When you bring up the idea of learning about algorithms, if they have explored the field a little they will say that all the standard algorithms have already been developed. I use the Wolfram language most of the time and this is truer for this language than many others. There are a lot of standard algorithms already developed, you will never need to know all the kinds of sorting algorithms out there or when to implement which depending on the dataset you are dealing with. All you need to do in Wolfram language is Sort[data], it will automatically determine which sorting algorithms to use for the dataset based on enormous research in algorithms done by the Wolfram guys. So why will a guy like me that uses a super-automated stack of pre-designed standard algorithms be telling people to learn about algorithms? This is not really about designing production standard algorithms like we have in Wolfram language, this is more about learning to think in a certain way.

If all you have known is your life is natural language, then your thinking is still very mushy when it comes to thinking about problems of a computational nature. Exposure to higher mathematics will also enhance your ability to think systematically in general but learning programming is one of the most powerful ways to develop systematic thinking. When we are writing a program, we are specifying a solution to a computational problem. This is like providing a recipe for some meal, you have to be very definite in your specification or else you will end up with a meal that tastes wrong. These recipes are called algorithms.

The field of algorithms is large and I have talked about it earlier in this work, the reason I am talking about algorithmic thinking again is to highlight another perspective. Even if you will never design algorithms yourself it is very essential to study them and understand them. As you spend time studying algorithms your capacity to think systematically will increase, this is what differentiates technical people from non-technical people. Natural language is weak when it comes to expressing computational solutions, mathematical thinking takes things a step ahead from natural language, while computational thinking with a programming language takes things to the peak. Computational thinking is another name for algorithmic thinking, the slight difference is that while we are working at the most primitive level with raw algorithms in algorithmic thinking in computational thinking we are developing meta-algorithms – we are using primitive algorithms as the fundamental building blocks of our meta-algorithms.

In algorithms we solve problems that are more oriented towards the problems of computation, basic problems like sorting are computation related problems but why do we want to sort some data? The why is usually a computational problem that has to deal with the solutions to human centered problems. We might want to sort a list of ages of our friends in our address book application to find out who is the oldest, how the sorting is done at a primitive level using merge sort or quicksort algorithms is not really what we are about but using sorting as a primitive in some larger application, that is computational thinking at its core.

The most important thing is learning to think systematically about things, which is a skill that we are not naturally born with and have to learn as we engage in some kind of discipline that requires rational systematic thinking.

Brute memorization vs familiarization:

Many aspiring programmers always ask me what is the best approach for learning something, should they memorize it in a brute force manner or should they familiarize themselves. I tell them that you have to mimic deep learning in your approach to learning, you have to expose yourself to a large amount of examples to learn the stuff except you have natural photographic memory abilities.

If you are like the rest of us with average memories, then you have to familiarize yourself with a topic by going over multiple implementations rather than just using some memorization techniques to memorize it. It’s better to get 3 or more books on the same programming language than spend all your effort trying to memorize all the code in one. The more code you expose your mind to, the better you get at coding your own solutions. Except is in some poorly designed interview, you are not required to memorize exactly how some algorithm is implemented. If you expect to be questioned about designing binary search trees you can surely cheat at an interview by brute memorizing the implementation, but to really know it you have to study it in various forms and trying out small simulations of it, and in the end you will see it as a method of approach to solving problems even if you cannot instantly recall an implementation on the spot. The most important thing is to be able to decide when to use it when confronted with a problem, this is the true power of knowledge. Seeing some computational problem and deciding that the best data structure to use in solving it will be binary search trees rather than B-trees or AVL trees. Even if you have a full mental picture of the implementation of an algorithm or data structure without truly knowing it you will not know where it will be appropriate to use.

If you don’t have money to purchase different books teaching the topic it is better to repeat the book you have till you understand it rather than forcing yourself to memorize it. In the long run you are better off knowing these things and understanding them at their cores rather than force feeding your brain with undigested information.

Focus and memory:

Although a good memory is a powerful asset for a programmer, because you will be able to absorb concepts faster, improving your memory is not very easy task. There are supplements out there that may be appropriate for you and you should use them if you can, but apart from pharmaceuticals and herbs other methods for memory improvement are usually too cumbersome. Memory techniques that involve hard memorization of facts, lists and numbers could help you in some examination where you are required to regurgitate facts but it will not help you much in the field of programming. Programming is a creative activity, it requires more of focus than hard memorization.

A programmer should develop her ability to focus for very long periods of time, because this the way to be the most productive in such a challenging field. Focus is developed by focusing so what you need is actually the discipline to focus on the job at hand. Caffeine helps at the biological level as well as nutrition and exercise to keep the brain functioning at a high level but discipline is more important. You could have a healthy brain and body and a weak will, if you are this type then programming will be a hard discipline for you to engage in. If you have a deficiency of focus and you have made sure that biological factors are eliminated by taking care of them, then you should learn how to discipline yourself to focus on the task at hand. It usually takes some effort but it can be mastered to a very high degree with practice and persistence.

Sometimes when developing software, you will be faced with mountains of code to write or read, books to reference, teams to collaborate with etc. this can be very tasking on the individual and requires that your ability to maintain calm in the face of these storms is highly developed. Most people have this naturally but you can discipline yourself and also develop these traits to a high degree, sometimes higher than those who have it naturally and are thus unconscious of it. Programming is an art as well as a kind of mental sport, you can develop yourself both in the actual coding exercise and the supporting abilities that make it easier for you to accomplish your goals in the most optimal time possible.

✻ ✻ ✻

46. SOFTWARE SECURITY

I name this section software security instead of computer security because the entire field of computer security is much broader. As a programmer when you start writing out software your goal is majorly to get code out that works, then make it faster. Many new programmers do not think about secure code when writing software. But eventually someone will try to exploit your code and that is why we have to think of securing our code.

Computer security is a very broad field which can eventually involve physically securing the facilities where high value computation is done to prevent intrusion. Companies that handle a lot of data usually have high end physical security systems to physically secure the data of their customers. A data breach in this case would sometimes involve someone getting away with physical hard disks that might contain important personal data. Servers have to be physically secured from intrusion too because you don't want to have someone getting into your server farm and installing some malware directly on your system.

After securing the physical location you would also have to secure network connections, this is where firewall software come in. Firewalls scrutinize incoming traffic and block traffic from sources that might be dangerous. They also prevent already installed software from making unauthorized connections with the outside world. A firewall system usually has some list of programs that require internet access and allows you control access.

Another level of security is to control the actual access to the computer. This is usually provided by your login credentials that you use to login to your computer. Choosing a good password can prevent intrusion at this level. So far we can see that securing computer systems is in levels. From security the physical site with fences and armed guards, to securing you network connections with firewall software and eventually securing access to your computer with a password. Another level to this system is data security.

Data security:

Sometimes we want to transfer data across a network connection like the internet. This data can include sensitive things like bank account information or health care information. If you are writing software that must transfer this kinds of sensitive data across a network connection, you will have to know all about encryption. Encryption is a way of scrambling data so that no one else except the intended recipient has access to it. If you have sensitive data on your computer, like private research data there are also tools to help you encrypt it.

Writing secure software is a different thing and was not really taken seriously until recently. There are very few books talking about it. Most of the times programmers just write software without bulletproofing it and just hope that people will be nice enough to not attempt breaking their software for malicious purposes. Malware writers always try to find weakness in a program by putting it through all kinds of tests, these are usually called penetration tests. They try different ways to break your code by exploiting weak programming patterns that crop up in code from time to time.

There is a lot that applies to writing secure software and I am not yet an expert in that field, I am here directing you as a new programmer about certain areas of focus which you should look into as you progress in your career as a programmer. Ethical hacking is a thing that you need to learn about – it involves using certain tools to try to penetrate your software or network and find weaknesses that can be exploited usually in order to patch them.

Ethical hackers use tools that many other bad hackers use to exploit your systems but they are ethical because they do this so that they can suggest patches than can be done to improve the system. It is unfortunate that the term hacking has the connotation of gaining unauthorized access to the computer, this is an error of history encouraged by a wrong perception promoted by the 1995 movie "Hackers". Eric S. Raymond has made it clear that people who try to gain unauthorized access to other peoples' computer systems are called crackers but the public has stuck with the term hackers. Anyway you know what I mean when I say hacking, I actually mean cracking.

Most ethical hacking is done on a flavor of the Linux operating system called Kali Linux. It is generally advised that you learn how to use the Linux operating system with another flavor like Ubuntu or Fedora not with Kali Linux. Kali Linux is very advanced and you run risk of getting arrested or breaking your system if you do not know what you are doing so gaining some good Linux OS experience before delving into the dark waters of Kali is required. It is also recommended that you run Kali as a virtual machine on some virtualization platform like Vmware or Virtualbox, etc. this offers you some degree of insulation and isolates some of your mistakes from your main system in the sandbox of the virtual machine. But this doesn't protect you from accidentally hacking someone's computer if you do not know what you are doing. There are good books out there on the topic of Penetration testing or Pen testing as it is popularly called, it will be wise to avail yourself of some.

✻ ✻ ✻

47. SOFTWARE ENGINEERING WISDOM

Straight from the frontlines of software engineering comes these words of wisdom. Excerpted from the notes section of the NKS book, notes for chapter 2 section name: Intuition from practical computing:

1. General-purpose computers and general-purpose programming languages can be built:

In the beginning, we talked about general purpose programming languages which proves that this statement is true.

2. Different programs for doing all sorts of different things can be set up:

This might seem obvious because we are actually in the field of computer programming to write programs that do all kinds of things. But if we look deep in the statement we will actually notice some deep implications – we usually look at programming as an art where we create stuff with direct human applications but in general we can write programs that do anything whatsoever as far as it obeys syntax of the language you're using.

3. Any given program can be implemented in many ways:

There is no single way to solve a computational problem, there are always multiple solutions in code for the same problem. Some solutions are more effective than others thus the mantra: we can always do better.

4. Programs can behave in complicated and seemingly random ways - particularly when they are not working properly:

When a program doesn't work as expected we say it has a bug. For simple errors like syntax errors we can easily use the tools in the editor to find the exact line of code that has problem and correct it. But for semantic errors it is not usually straightforward to pinpoint the source of the error and more advanced debugging tools and methods must be applied to spot the errors. When a program has some semantic errors it will run but produce unexpected behavior.

5. Debugging a program can be difficult:

Like I mentioned in 4 above, finding simple syntax errors in a program is easy but finding semantic errors, those ones that result when the program runs but is producing unexpected error, can be very difficult. For small programs it can be faster to find the source of errors especially when you have built some experience as a programmer. But for very large projects with hundreds of thousands to millions of lines of code, debugging can be a horrifying experience much more difficult than actually writing code. There are lots of books that teach you how to debug and they are needed but actually working in the field will reveal more wisdom than any book will every teach you. Debugging is one of the pains of programming just like athletes suffer sore muscles and joint pain, programmers suffer from the pain of debugging code. As the interactions between different parts of a piece of software increase the total amount of errors that can result also increase.

6. It is often difficult to foresee what a program can do by reading its code:

This is a point I will like to emphasize for the new programmer. Programming is about reading code and writing code and like I have said before some pieces of code can be deceptively simple but produce far more complicated output than expected. This is where experimentation come in, in order to understand some pieces of code you would have to design mini experiments where you take out fragments of code and run them in some kind of interactive environment so you can probe the code in different ways to gain fuller understanding of it. In some large compiled languages like Java writing experiments can be so time consuming many coders just assume that they know what the code is doing while in reality no one can be certain about this. This is the source of many bugs in large so called object oriented languages. Another issue that adds another layer of complexity is the fact that your program requires a lot of other programs to run and also might come with its own complex inheritance hierarchies. This makes it much more difficult to do mini experiments. It is much easier to perform mini experiments in a compact language like the Wolfram language because tiny pieces of code can do great things plus the highly sophisticated notebook environment for doing interactive computing, you can actually probe any piece of code down to single functions and rebuild the entire thing to gain a complete understanding of a program.

7. The lower the level of representation of the code for a program the more difficult it tends to be to understand:

This boils down to what we have talked about in the earlier sections of this work about the abstraction hierarchy. The lower the level of abstraction you are working in, the much closer to the machine's way of thinking you are and the farther away from a human's way of thinking. One of the major reasons for building higher level languages is to abstract away from the way a machine requires things to be done to more of the way a human would want to think about things. Although in the end all programs will eventually be converted to the lowest level representation ever, which is machine language, no one is forcing us to use machine language to describe complicated software. The computer is a very simple machine, as complicated as our software systems written in high level languages are, at the lowest level the computer only operates on some very simple instructions for doing simple things like moving data from one place to the other; adding one piece of data to another; performing simple logic etc. from extremely simple operations like the jump instructions and others are combined to build up abstractions like functions which as I have said earlier can express any kind of computation. While many other high level languages were built up step by step as a reactive response to programming needs and other built on some abstract philosophy of the way the world works, the Wolfram language redefines computing by building on deep computational ideas not just on knee jerk reaction to wanting to work in an easier environment. The Wolfram language is built from computational atoms like pattern matching and others and enables you to easily do even mathematics in an elegant way. In order to do math in other languages you would have to build too much abstraction to represent your mathematical ideas and thus building mathematical systems like neural networks always require too much high level structures to be built first before you get to the level where you can do something useful.

8. Some computational problems are easy to state but hard to solve:

This also applies to other things in life, some problems are easier to state than solve. You can never know how much code you have to write beforehand before you actually start designing away or hacking at your program. This is not meant to scare you but maybe excite you. Making the decision to be a programmer sometimes implies that you are the kind of person who loves tackle challenging problems so therefore you are not scared of how much effort might be required to solve some problem. A typical example of a simply stated problem that is hard to solve, note I said simply stated not simple, the problem of intelligence. The problem statement is easy, design a system that demonstrates human thinking, the solution has taken us many decades to get to our current machine learning systems which just simulates our pattern recognition abilities using brute force computation.

9. Programs that simulate natural systems are among the most computationally expensive:

All programs are basically simulations of some kinds of systems. This is similar to what I have been calling abstraction where we summarize the basic features of an underlying system and model it at a higher level. In programming languages, we are abstracting away from the underlying machine while when solving some problem like designing an address book we are abstracting the problem of representing the address book in code. In essence all we are really doing is simulating abstract systems when we solve computer-software-centric problems. But the computer is a universal machine, meaning it can also be used to simulate any machine. Science works because somehow the universe obeys mathematical laws, which can be simulated on a computer so therefore we can simulate natural systems on a computer because computers can simulate any system that can be systematically specified. In most cases it is quite computationally cheap to run computer-software-centric simulations on a computer. Programs like address books, database systems, word processors, compilers, etc. are cheaper to run meaning they require less computational resources. Programs that directly simulate natural systems like liquid flow, aerodynamics, quantum mechanics, etc. can be computationally expensive because despite the fact that a computer is universal and in theory it can simulate any system, natural systems usually have a lot of things going on that it is quite expensive to represent most of the variables involved in a system on a practical computer. This brings us to the question of modelling. When we create a model of a system, we usually pick out the core variables and constants in the system for this purpose, this is what is called a simple model – it can pick out the major features but might sometimes be hard to trust. We sometimes times want a model that represents as much of the features of the natural system as possible so that we can have a more realistic representation of the system. The more features we try to represent and the greater the number of interactions the more computationally expensive it will be to simulate the system on a computer. It is hoped that quantum computers will make it much easier to simulate natural systems with enormous amounts of variables.

10. It is possible for people to create large programs - at least in pieces:

There is no hard limit on the maximum size a program can reach. You can keep writing code until your problem is solved. Programs grow in size as requirements increase and having a solid method of managing complexity is a must in very large programs. This calls for a much more modular design where individual pieces of a program are completely isolated from other parts. If a program is too tightly knit with too many dependencies, then it becomes very difficult to extend it as requirements increase and bugs also follow. Even if you are hacking out a solution to a problem without much planning, when you have solved the core problem it might be time to design the system properly if you hope to keep building on the initial idea sustainably. This might involve massive rewrites of the entire system but it's a much healthier adventure than waiting for your hacked out code to grow out control and then trying to tame the beast then. Many companies maintain large code bases, Google has about 2 billion lines of code running and Wolfram Mathematica is a miracle of software engineering with tens of millions of lines of code which result in over 6000+ functions in version 11.3, you can instantly use any built-in function without a perceptible time lags. The only time lags you experience is that actually spent running of your algorithms.

11. It is almost always possible to optimize a program more, but the optimized version may be more difficult to understand:

When you start learning programming from programing books most of the code you will be exposed to will be designed for pedagogical purposes and thus will be very easy to read and understand. But in the real world you will be faced with highly efficient code which is sometimes difficult to read because most of descriptive structure has been optimized away. We said earlier that programming is not just communicating with the computer but also communicating with humans that will eventually have to read your code. No matter now difficult to read a piece of code is, as far as it is correct and passes all the valid tests it will still be executed without trouble by the computer. But if there is some subtle bug in this piece of code because of its highly optimized nature it might be very difficult to understand and thus debug. Hence in programming we have to walk the noble midway path between hyper optimized code and highly readable code.

12. Shorter programs are sometimes more efficient, but optimizations often require many cases to be treated separately, making programs longer:

A short program will often have little memory requirements and depending on what you are trying to solve less compute requirement. Optimizing a program involves removing unnecessary stuff from a program and making a program more specific to solving certain anticipated scenarios rather than making it more generic. Even though the decision pathways to reaching a goal can be theoretically infinite we usually sometimes know beforehand the program will not explore all the available pathways and follow only a few – so we optimize it by writing additional code that will handle each pre-anticipated execution pathway separately. This produces longer code even though the resulting program will be much more optimal. In summary some generic piece of code meant to handle all possible scenarios equally might have to make too many decisions to choose a particular execution path, and we know that decision making is one of the most expensive operations in computer programs. In reality this is not an optimal design as we know that certain execution pathways will be favored more than others. To optimize the program will require that we treat the cases we anticipate most by writing code that deals with them which in turn will produce a longer program.

13. If programs are patched too much, they typically stop working at all:

No matter how carefully we conceive and design our programs there will always be things that we miss out and wish to include in the final version of the software. This is what is usually referred to as patching software. Most of the updates you perform on your operating system are usually patches. The need for software patches can also come from expanded requirements for the product for which we are not ready or do not find it necessary to do a full rewrite of the entire program. Patching software is like applying Band-Aid it is not a permanent solution but fixes a specific solution awaiting when a more general solution is implemented. Some problems in a software product might be general enough to require and entire rewrite of all of the product or some major component but sometimes this is not feasible and might be time consuming so a patch is released on time until the final solution is designed. A timely patch is usually required when a security vulnerability is found in the code that requires immediate attention and there is no time to do a full overhaul of too many components of the product. If the need for patching is too much the entire system might need a rewrite but due to certain factors the team might be complacent and continue patching, but if this continues the system will become so convoluted that it eventually stops working completely. Patches are usually local solutions to local seeming problems that might have global reach to the entire software system. They are good in many cases but should not be relied on as a permanent solution, after a while the core structure of the program should be redesigned to get rid of the majority of patching requirements.

* * *

48. THE RIGHT PERSPECTIVE ON AI

Deep learning should be understood as a bunch algorithms based on mathematical research that enables us find patterns in data, and also help us with appropriate caution extrapolate future patterns too. It is a tool, but unlike the deterministic tool that a calculator is, it should be viewed as a non-deterministic tool, meaning it could work really well of fail terribly and hence its results should be looked at with sufficient caution.

Deep learning and its variations can help us see what we might normally miss, it might help us spot patterns in financial data and make some predictions, but it shouldn't be relied on to make all final decisions on trade no matter our confidence level because it can fail catastrophically.

It will find great utility in scientific endeavors like genomics to help us comb through enormous amounts genetic data. In health research for spotting out disease patterns that investigators no matter how skilled might miss. In astronomy for finding hidden cosmic objects etc. It has found great use in physics and recently a researcher Daniel George made some discovery about gravitational waves using deep learning, he is a friend of mine.

If you want to apply deep learning think of it as a tool to expand your pattern recognition and prediction capacity. The way it does it will remain mysterious to us, but the results can be tested for accuracy using other methods.

There are other modern AI technologies that are making waves and deserve a mention: Recurrent Neural Networks (RNNs) are generative in nature, meaning that they create new information that makes sense sometimes. While deep learning is more predictive, RNNs are more creative. RNNs are great algorithms and will find great application in the areas of artificial creativity.

Reinforcement learning has come back again from the past where it was one of those methods that we usually refer to as GOFAI (Good Old Fashioned AI), it is just an algorithm, a powerful one used in strategy and planning.

✻ ✻ ✻

49. THE FUTURE OF AI

The real AI work is going on in the research community from mathematics to algorithms and psychology. The future algorithms that will power the world are being baked up there from the ingenuity, skill and creativity of the researchers there. Also the silent heroes of the computer industry, the microprocessor designers are forging ahead, making more powerful chips.

Since it has become ridiculously hard to miniaturize further, we have resorted to massive multi-core systems. We will keep at this for some time bringing more and more computation power to increase the applicability of many algorithms. The basic algorithm in deep learning, back propagation was invented by Geoffrey Hinton, et al., in the 70s but it only showed promise when hardware capacity was ready.

Effort to push towards quantum computing, the room temperature type will continue. The quantum computer will make the current AI algorithms work at a near perfect level and also a very high speeds, but the true heroes of the quantum world are being created in mathematics and code right now in various laboratories around the world.

AGI is rearing its head again and many people are talking about it, but after going through some of the papers, I am disappointed. It's not really a shortage of smart people to tackle these problems that is lacking but the thing is that we have not yet properly defined what we seek to create. Our inability to properly define that which we seek to create stems from the fact that our natural language is not definite enough and different words could mean different things. The biggest suitcase word in the world now is intelligence which actually means different things depending on who you ask. This is not because we are naturally mushy headed beings but because we have been handed natural languages that are weak when it comes making non-ambiguous statements about things.
In the future there will be this powerful stack of computation out there, ubiquitous and immersed everywhere and it will be possible with a well-written article using a

symbolic discourse language to actually create an executable that could bring an AGI to life.

When we write a computer program, we specify code in text that is to be executed by the computer. But when we execute the code a software image which is a bunch of 1s and 0s for controlling the computer hardware is created to represent the text code we have typed in.

This software image contains instructions on how we want the problem we are specifying to be solved. In order for this software image to produce desired results we must specify in excruciating detail how we want our problem to be solved on the computer. We usually build a hierarchy programming language above the machine to ease the task of specifying the software in a textual form, far away from the raw mechanism of the digital computer.

With a symbolic discourse language, which will be capable of being executed automatically by a computer, rather than think about how to specify a solution to a computational problem like creating an AGI, the programmer will be tasked with writing an essay in the text of the language about what the AGI is, no different from the way you would describe it to a human being and the AGI will be automatically created. This is like a 5G programming language. In current programming languages we define a task and tell the computer how to solve it, it the future we will not describe the method of solving it like we do now in code, we will only specify using a symbolic discourse language how we think the final result should "look" like and the entire stack of computation out there in some computing cloud connected to our local client, maybe running some aspects on quantum computers and others on serial computers will just generate a running software image representing our ideas of how things should be. This is still far off and in time clear lines of actions will emerge.

Misunderstanding the capabilities of current AI technologies can lead to disasters that we cannot fully comprehend. One that easily comes to mind is assigning the current AI the task of running some critical facility like a nuclear reactor without some system of human assistance, or putting current 2018 grade AI in charge of the driver seat in driverless cars and maybe soon pilotless planes.

It is not yet time to give AI increasingly larger executive control of many areas of our lives and civilization. This is not because we should be scared of them taking over, but merely because when they fail they do so catastrophically. Humans fail but our failures are quite understandable especially when we have many years of training in the field. We have errors of functioning in our brains especially when we make the kinds of extrapolation that have led to the over hyping and anthropomorphism of current AI. We over extrapolated too in the great depression and the tulip mania of the 1600 and the 2000s dot-com crash.

Humans fail but we have evolved to understand our failures and we have safety nets that make sure we have those who always rescue us when the majority of us take the plunge of foolishness. While some are polluting the environment, some others are creating solutions to end the pollution. Human failure always has a backup of humans to fix, but AI failure might be so fast that humans may not be able to back-up the failure.

There is no problem in producing some shiny looking report of some product or technology, but when it starts shinning too much beyond reality; triggering hopes that are not real then we have to stop such marketing. We should not market AI too hard, chugging it down the throats of consumers and business people. The technology is promising and we should use it to improve our systems but we should also tell people about its weaknesses so that we have a balanced view of the risks involved.

We should proceed with caution. Deep learning and related “AI” technologies are great tools to aid us in discovery and improve what we are currently doing in our research and work, it will not become a super intelligent being that will want to wipe the world.

✻ ✻ ✻

50. THE FUTURE OF PROGRAMMING

As a programmer you might be considering if this new endeavor you are engaging in has any future at all. The industrial revolution saw the destruction of many manual workers who were replaced by machines, recently there has been rumors and even declarations by a CEO of a major company that artificial intelligence will soon replace human programmers at many tasks. So what shape will the future of programming take? This is the question I seek to tackle in this section.

The history of computing has seen the building of one layer of abstraction above the lower, shielding the human being away from the details of how the computer performs its task and shifting the cognitive requirements towards computational thinking itself. In the early days we controlled computers by manipulating electro-mechanical switches, from then we proceeded to punch card systems and so forth into the future.

In these early times we had to control computers by directly working in the language of the machine. We had to turn our high level conception of a solution into a representation that was convenient enough for the machine. Then came assembler which gave us some high level ability to express solutions in a more human friendly way of typing out characters into the system that would be further processed into the machine language representation. This enabled us to build up away from the machine and towards the human way of conceiving things, but at these level things were still primitive.

Then came the so called high level languages with conditionals, loops, etc. These languages just stepped up a bit away from assembler and even allowed you to include inline assembler code in C programs. C allowed you to express programs with functions as the highest level of abstraction necessary. There were other languages enjoying a parallel evolution beside C, like LISP even much older than C. But the point being specified here is not the history of programming languages but more about the climbing the abstraction ladder. From lines of assembly code, we went to functions and finally object oriented programming languages. Each of the levels took the human far away from the underlying machine as possible, enabling to human to express their concepts in more human friendly terms rather than restricted to the machine's way of representation, but eventually this high level structures will be converted to low level machine language to be executed by the computing hardware.

Symbolic languages like the wolfram language took a different path and focused more on using fundamental ideas from the world of theoretical computing to build a language that could express highly abstract ideas with as few layers of abstraction away from the machine as necessary. When you are writing a program with functions or classes, you are building abstractions with these features to enable you model your idea. The more complex the idea the higher the abstraction you will have to build; this is why reading some programs are difficult. The abstraction hierarchy of machine language, assembler, functional languages and object oriented languages are what I refer to as the root abstractions. When we get to the level of functions, we are the highest necessary level of this root abstractions, it has been proven in theoretical computer science that the function is the highest level abstraction needed to express any computational idea. Object orientation promises to make it easier to represent solutions to problem as a system of objects sending messages to each other but this is just a philosophy and is not fundamental to computing.

Solving a problem using a program requires modelling the solution using the programming language of choice by building the necessary abstractions. The abstractions we can refer to as secondary abstractions because they use fundamental building blocks like the function to model the system in question. Some domains require that we build another enormous mountain of abstraction to model the system we are trying to solve, especially in the domain of natural systems. A good programming language is one that enables us build the solution model with as little extra abstraction as possible. If the tower of abstraction is too high the program will be very hard to understand. Class based object oriented programming adds so much extra fluff to expressing the solution to a problem that it becomes hard to distinguish features that are necessary for the expression of the important algorithms and those that exist to manage the complexity that is being generated in the attempt to model the system. Sometimes the added complexity of managing class inheritance hierarchies is more expensive than the problem you are trying to solve. There is some argument out there that object oriented design aids management of large software pieces but this is not really a fundamental argument. The Linux operating system is a large software system and it was written in C, which is a procedural programming language. I am not against object oriented programming or any other programming paradigm, I am just clearing out the air so that new programmers might be aware of the possibilities.

With a language like Wolfram, the abstraction ladder required to model the solution to a problem is very low because most of the fluff found in other programming languages have been stripped away, leaving just the essentials required to express the solution to computational problems.

As concerning the issue of the future of programming what do we expect? The function is capable of building any tower of abstraction needed to solve a problem so what other abstractions are needed?

Human Intelligence:

What really is human intelligence? I am not a fan of lifting other peoples' definitions unless they resonate with me. People can make definite statements of certain things like in physics where we know that the definition of force is given by the equation F = MA, which is not revocable no matter how you contest it. It is a fact stated in math. It's a fact as strong as 2 + 2 = 4. You cannot create another answer for 2 + 2 = 4 using the standard understanding of the basic arithmetic operator. You can create another operator that defines addition in a different way but the basic arithmetic operator cannot be changed.

This does not apply to definitions in nonsystematic natural languages like English language and others. I mention English because it is my standard means of communicating in natural language. All natural languages are nonsystematic like English. In English a word usually stands for a certain thing, but certain words can stand for many things, this ambiguity of natural languages makes it difficult to ascertain what is fact because in many cases one person's fact is not usually that of another person. People hear the same word and imagine many things.

This doesn't mean that there are no English words that are not definite in what they mean, like the word red means the red color.

There are more facts in mathematics because mathematics is a definite and systematic language. The world of mathematics is finite, it is defined by an axiom systems and all mathematical truths or fact must be derived from these basic axiom systems. When a mathematical statement is made, a proof of it can be derived in the axiom system of mathematics. The proof to be correct must mean what it means, that is it cannot mean another thing. Once a statement is true, it is true in the axiom systems of mathematics and thus it is fact.

Although the field of mathematics has its contradictions like the one exposed by Kurt Gödel in the majority of cases, mathematical results are unequivocally true and as such are a solid fact.

Closer to mathematics is the field of the physical science known as physics. Physics explores the physical laws of nature and states its results in a factual form called an equation. An equation explicates the relationships between physically measureable variables represented with symbols. Science is the systematic exploration of knowledge and Physics is a science where facts can be easily obtained through experimentation and verification of theories, thus we have facts in physics. These facts have the further backing that they are written in the language of science which is mathematics.

When we move to the biological sciences and eventually to the social science we find that getting definite equations for the behavior of these systems becomes very difficult, making it increasingly difficult to derive facts that hold in all situations like we are used to in physics. Thus in the social sciences we have few facts and lots of conventions because we are dealing with elements whose nature cannot always be known to a high degree of accuracy like in physics or chemistry.

When we step out of science and enter the mushy world of human languages then we are faced with increasing difficulties and it becomes very hard to make definite descriptions of facts. We now tend to rely more on convention than on hard facts as can be observed in the sciences.

This article is on human intelligence, the most misunderstood field in the world right now. We never really thought seriously about the nature of human intelligence until the advent of the modern computer. Philosophers of all ages had contemplated the issue of human intelligence like they have contemplated all things before it, of which includes the nature of God, the universe etc.

When faced with some physical object like a stone, if we ask the question, what is a stone? Without science we can describe what a stone is by its action, a small piece of rock that we throw at things or just leave there. But with our science we can define a stone as a rock, then include information about how it was formed, what type of rock it is and its chemical composition. This is what we call scientific knowledge. We reduce things to what we can contemplate.

When thinking about things we cannot objectify our brains rely on its faculty for speculation using its knowledge base much more freely than when we are dealing with an objective fact. The nature of the development of our brains, its structure and the history of our experiences will determine the kinds of speculations we derive from it.

The work is not an exploration of the nature of human intelligence itself, that is a huge topic for psychology and philosophy. What I am interested in is in the question of Artificial intelligence. How should we go about it in the most efficient of ways?

If we cannot appropriately define the word intelligence at least as it occurs in humans, we cannot build a machine that encodes that intelligence. The problem with artificial intelligence is not lack of human ingenuity to solve the problem, it is more of the problem of not defining the problem properly. How can we define intelligence using our intelligence? That is what do we code?

Natural language understanding

The problem with natural language is that it is so ambiguous that if we examine a statement like: "understanding what intelligence is…" we find out that I used a bunch of words whose meaning is not really definite and can mean different things to different people. This is the problem we face in natural language. You must have glanced over those words with the assumption that you understood what it meant but if I ask you what understanding really meant you will maybe reach for a dictionary and produce some definition, or define it based on your memory of the word as it is defined in the dictionary which uses other words too whose meaning we cannot really understand in a formal definite way.

If it is difficult to get a definite meaning of what the words "understanding" means, then what do we mean by "meaning"?

When natural language is used to represent things that can be easily observed in the physical world, like apple, it is very useful. We use the suitcase word apple to represent that distinct piece of the physical environment which we can identify as a separate apple. But when we start using natural language to talk about concepts that we cannot point a finger at, the world of mushiness begins.

In human interactions, this mushiness can be tolerated. We are social animals and have the need to communicate. When this communication is about physically observable things, it is direct and efficient. But when we start having concepts for which you cannot point a finger at, natural language must then rely on conventions, or accepted definitions.

Science would not be possible without the definite systematic language of mathematics. An objective way to symbolize the things we observe in the physical world, things that can move a needle on some kind of measuring device. If asked what is force? Since we cannot point at it, without mathematics such concepts will be difficult to define. With natural language we can go on to use other words, which will still not give a person some direct contact with the concept of force.

The next level of abstraction:

Now that we understand the need for a systematic language in order to express ideas definitely, we now fully appreciate the impact of programming languages in enhancing the expressive capacities of the human mind. Programming languages like the Wolfram language are more general than mathematics in expressing systematic knowledge, you can actually do mathematics in the Wolfram language.

When programming we tell a computer "how" to perform a task. We model the solution to the programming problem with the language telling it step by step using an algorithm how to solve a particular problem. The next level of abstraction we need to create is completely removed from the other abstractions we have witnessed; it will be a layer above the current programming method of describing how to solve a problem. It will be about telling the computer "what" you want to do and letting it decide how to solve it.

So those CEOs predicting the end of programming as a career overtaken by computers are not really understanding the situation properly. It is the end of mechanical programming and the beginning of creative programming where the predominant problem is not how to solve a problem, but a proper description of the problem in a kind of symbolic discourse language.

Although we say we are communicating with a computer when programming, this is not very accurate. The computer is a kind of tabular rasa general machine, it's our job as programmers to create the environment in which to solve our own problems. We do this by modelling abstraction upon abstraction, first of all abstracting away from the machine and then modelling the problem and finally executing to produce our results. In all these steps the computer is just there like a bicycle, just a raw mechanism, we are the one doing all the work from bottom to the top. In the next level of programming or meta programming it will be about describing what we want to accomplish without specifying the how, that is the algorithms to use. The computer will find its own way of doing it, either by using existing algorithms or inventing one on the fly.

This is similar to what the deep learning neural network paradigm does. We set up our network and then train it with examples in the form of thingA -> class1; thingB -> class2; thingC -> class1, etc. after training the network if we present it with thingM which falls into class2 and it will do the correct classification provided thingM really falls into class2, even if the system has never encountered thingM. This is similar to the kind of metaprogramming I describe because we do not write a program to tell the computer exactly how to handle any of the cases, thingA, thingB, etc. the neural network architecture extracts whatever patterns it can find and then assigns a class to whatever it encounters after training, the metaprogramming platform just like the neural network will be capable of writing programs on how to solve the problem while the human will be tasked with the job of describing the problem to the system the same way an ML expert describes a classification problem to a neural network system by preparing training data, building the netmodel and then training. Do not take the analogy I describe too directly, I used machine learning because it is the closest to what the future system will be and not because I am expecting the current ML paradigm of deep learning to be able to write programs now.

Recurrent neural networks are generative systems that produce new output from training input, much like our creativity works. This paradigm will also be important in building a metaprogramming platform but might just be part of the philosophy of the final system.

The need for a symbolic discourse language:

I have talked about the mushiness of natural language earlier and how systematic languages like mathematics and computer programming help us think about and express concepts much clearer. Stephen Wolfram has talked about the need for a symbolic discourse language as a way of producing automatically executable legal contracts with no ambiguity. This ambiguity is the root of the problem in natural language, which is why non-ambiguous languages like math and programming surmounted natural language as the mode of expression in the sciences and other fields.

Future programming will involve a programmer communicating with a computer just like you do with a human individual about a problem and the computer in return producing solution after solution until it meets the programmer’s requirement. This will be very difficult to do in natural language because it was not constructed from the ground up to be a non-ambiguous form of communication. Many things have multiple meanings in natural language and things cannot be explicitly defined properly in natural language.

The symbolic discourse language will enable us define everything perfectly so that when we say a word in this language to the computer the computer will know what it means and with this language we will be able to converse with a computer just in the same way we converse with a programmer on the nature of the software we want to build. The computer with its enormous mountain of knowledge and algorithms, including fully built software projects out there in the open source domain, will be able to execute our wills when it comes to creating any kind of software. This is akin to the scenario with artificial general intelligence but it is much more narrowed down to the problem of creating software. Just like current paradigms like deep learning help with image recognition tasks and general pattern recognition and discovery, this kind of metaprogramming will be able to churn solutions to programming problems in executable code.

Will the code they produce be readable to humans? Our modern programming languages have been designed to fit well with the way humans conceive problems, we even have functions and classes named in a way that is easy for humans to consume – but there is really no intrinsic requirement by the computer for this kind of convenience. Human programs are often written in a way that is easy for other humans to comprehend with lots of documentation to aid that purpose but the computer doesn't really need these conveniences to read and execute a program. The abstraction mountain we have created only aids human programmers to write computer programs, it has nothing to do with making the computer understand the code. As far as the code is correct the computer will scrape away all our human artefacts and just produce the much needed machine code that it needs to run on its circuitry.

Compilers even peel away the documentation that is provided in code during the compilation stage and also optimizes variable names to shorter ones and a whole lot of other optimizations. These extra fluff are only there to ease the labor of humans and nothing else. Metaprogramming systems could take human commands like: write a browser software; and directly produce machine code without the intermediate stages of producing human readable code. This will be akin to the way compilers currently work, taking code specified in a language and compiling it to object code. In the metaprogramming framework, we will issue the highest level command possible and have our computers churn out object code immediately.

Code inspection tools to inspect the output of a metaprogramming system need not produce text based code. It could produce graphical diagrams detailing the entire software structure to the most minute detail, enabling the human being to investigate the work that has been produced.

In my opinion this is just one possible future of programming, until a god-like AGI appears in the world humans will remain relevant as developers of software products. Even though the metaprogramming platform can in theory churn out the solution to any well-defined software problem specified in the highly descriptive symbolic discourse language, the problem of actually specifying the software will be the domain of humans for a long time to come and may not be as simple as "churn out a browser". The system will be able to create any kind of software that is required but will need a human to give it detailed specifications and observe the output and make corrections interactively till the goal is met. Humans will shift from the mechanically specifying software to designing specifications for software in a symbolic discourse language to be automatically executed by a metaprogramming system.

Conclusion

In this last chapter we explored the question of what programming is really about. All a human being can do is think and act with their bodies. The thinking aspect must be done in some kind of language; language puts a peg on otherwise mushy looking concepts. Language makes things concrete so that concepts can be built and solutions to various problems produced. Language is an interface to the mushy neural activity in our brains, language systematizes the otherwise chaotic patters going on in our brain which we call thought. When we put a word tag on a concept, which is represented by a neural pattern in our brains, we put a grip on that particular pattern so that whenever we recall the concept via the word tag associated with it we trigger those patterns.

Human languages have evolved from very earlier forms as scratches on a cave wall, to ways of counting livestock and to the fuller natural languages like German, English, French, Mandarin, etc. our languages were designed to aid in our own conceptualizations and eventually transferring these concepts to others at first by word of mouth and eventually through the aid of writing.

There were some individuals who were engaged in a more systematic study of natural phenomena and decided that the natural language that had evolved from the common need to think and share human-centric concepts was not adequate for their need to systematize the knowledge they obtained from their experiments. They found mathematics, a language that begun from simple counting and generalized further by individuals like Al Khwarizmi the inventor of algebra. Mathematics and its system of symbols with variables, constants and operators was the modus operandi of recording the consistent patterns discovered along the course of scientific investigations. This consistent patterns or so called laws of nature were made definite in what is called formulas. These formulas are like programs that need to be executed by a human being to produce desired results.

When given the formula for finding the force acting on an object which is F = MA, with F representing the force; M representing the mass of the object and A representing the acceleration of the object. The human computer who has been giving some numbers representing the variables in the equation, M and A, will go ahead and mechanically computer/calculate the results. By convention when you see two variables joined together without a space between, they represent the operation of multiplication. To show this formula in full explicit form we would write F = M * A, where the * represents multiplication operator.

Mathematics further branched into two main branches: Pure mathematics concerned with the development of mathematics itself and Applied mathematics concerned with applying mathematics to solve human problems.
As mathematics evolved the calculations became immense and the human effort required also increased. Large number of humans mostly women were employed in the pre-computer days to do the job of computing formulas required for certain grand engineering projects. The need to mechanize this work increased research work towards the creation of a programmable electronic computer. With the creation of the electronic computer and with a programming language like FORTRAN (Formula Translator), the repetitious job of doing calculations mechanically by human hands was taken over by computers. Computer programming evolved from then to what we have today with Deep neural networks and automated trading systems, etc.

So what is programming really about? It is basically about systematic thinking. Behind all the tasks we accomplished in human history from the most basic to the highly technical has always been some form of language. Programming goes ahead of other languages to being a medium where our thoughts about computational things can find proper expression. A good programming language is one that is designed in such a way that it eases the human mind from worrying so much about the details of the machine and concentrating more on the problem being solved. Many programming languages do their best to isolate the underlying machine as best as possible, in my own opinion the Wolfram language does the best job at this task because rather than trying to directly build upon the structure of the machine, it builds upon pure and powerful ideas from the world of theoretical computation.

While other languages might require that you build an enormous stack of abstraction to model problems, most times with effort equivalent to creating a new language, the Wolfram language with its solid primitives reduces the height of extra abstraction that is needed making for programs that are both compact and faster to execute.

Many people argue that object oriented programming languages are the best way to model systems and especially natural systems, but that only sounds bright as a philosophy not really in practice. The amount of extra fluff required to manage a large program in an object oriented language like Java can take as much as 80% percent of the effort put into engineering the software with core computation taking like 20%. In Wolfram language you are computing directly with as little fluff as possible, any fluff you see is mostly to aid you the programmer in your task.

In a certain sense Wolfram language code looks like the Abstract Syntax Trees (ASTs) of Python programming language. An abstract syntax tree is kind of an intermediate form of code that is produced as a result of parsing the input text of Python programs. It also looks like MIT scheme, a flavor of LISP for educational purposes. This is where the analogy ends.
As a new programmer, who has never written a single line of code I would strongly recommend that the first language you learn is the Wolfram language, it will affect your thinking about computation in the best possible way for the rest of your career, even if you work in other languages because of your job or other requirements.

Thank You
Jackreece Ejini ©2018
jackreeceejini@gmail.com
+2347084869518

* * *