Note at 2019: Some sensitive materials have been truncated in this public version. The tone in this document is harsher than usual - If I were to write it again today it may be rather different (which is also because I have learnt more stuff in the last year). Take it as a general guide and pointers to *directions of study*, not as a curse. Thanks!

-- BEGIN --

Before you begin

No matter what choice you made, there is one constant:

You need to learn how to learn.

Paramount to that is what people call “Google Skills” (Or google-fu), i.e. the ability to try the right keyword (with advanced filter used if it helps) to make Google your true friend. But that is only the beginning. The less tangible but equally important part of that research pipeline, is the ability to:

- Explore a new, unfamiliar **domain** - find out its territory, understand the context, learn its vocab, paradigm, and style
- Filter out good from outdated, bad, or simply inapplicable materials
- Consolidate insights and experience - into rule of thumbs, principles, heuristics that allow you to find directions to investigate on a problem, even if you can’t find anyone on the Internet who have faced the same problem already and posted solution (yes, this still happens). If you are lucky, you may even be able to directly intuit the right solution without consulting the Internet at all.
- Reflect appropriately to further enhance learning

For this reason I won’t fill you with a long list of links. But I will highlight the key parts of the landscape that you can use to bootstrap your journey.

(Remark: So if you are looking for learning resources, you may come across what people call “Bootcamp”. Critically evaluate whether they are a good fit - and don’t have the unreasonable expectation that you are a “real programmer” just after that camp. In the field of programmer at least, what you did trump what you say. Also with the abundance of learning materials, remember to use them correctly and selectively - check your learning style and optimize for that - Visual learner? Learning by doing? Reader? Vs Youtube video or coursera, codecademy or some exercise site, blog, articles, book, etc)

Entry Point

Reference:
https://insights.stackoverflow.com/survey/2018/
Also see: Glassdoor, website that provides data on average salary of various jobs (dig around a bit)

For the technical path - web dev:
https://engineering.videoblocks.com/web-architecture-101-a3224e126947
https://codeburst.io/the-2018-web-developer-roadmap-826b1b806e8d
https://hackernoon.com/how-it-feels-to-learn-javascript-in-2016-d3a717dd577f
(Luckily, this is 2018 - and enough people - even the super-elite ones, have been so annoyed by the unacceptable state that the JS ecosystem have converged a bit)

(Notice how the “medium” platform dominate these kinds of introductory things. But software development is a huge field, if you dig enough you will find a world beyond that)

Choose a/some Fields - what kind of technical work do you want to work with?

- Web Development
  “The web is eating the world, and Javascript is eating the web.” WebDev is the biggest technical field, where you basically “make (dynamic/custom) website” - the website can range from Content management system, platform (Instagram), SaaS (Software as a service), OLTP, to inhouse application. Because it is so big it is split into some subfields:
  - Backend: the code sitting behind a server that provides those functionality.
  - Frontend: Enrich what the designer made with even more Javascript (or language that Transpile to Javascript) to provide functionality right in the user’s browser. (The extreme here is Single page application and isomorphic app)
  - Fullstack: if you are badass enough to master both backend and frontend
  - Mobile: Mobile app is a specialty since you need to invoke the phone’s native API, but it is not exactly system programming either. Kotlin/Java for Android, Swift/Objective C for iPhone. May also use react-native and similar options if you do not want to actually be a mobile developer.
- Enterprise Information System
  Focus on system that deals with complex domain logic (usually because the institution it lives in has complex regulations, policy, and procedures). This day this one is mixed with Web Development a lots.

- Multimedia/Graphics/Game
  Video streaming, working with codec. 3D graphics, and so on.
- System/Embedded Programming
  Write driver for hardware(google: write your linux driver), write firmware for embedded device (say wireless router), Internet of Things, anything that involves using a low level language (C/Go/Rust) and interfacing with the underlying system directly. (Using assembly - beyond what you learnt in school, you may need to use processor-specific instructions, and advanced instructions like those for lock-free concurrency, data parallelism, etc)
- Data Scientist and Machine Learning/Artificial Intelligence
  An emerging field that rides on multiple hyped waves of "new tech". The summary version is that you start with "Big data" - website/information system that collected lots of data in the database (OLAP)/warehouse (with the traditional ETL). You need to build a data pipeline/infrastructure for that. (More recent incarnations are the hadoop cluster for batch, distributed processing, then lambda architecture, then stream processing via say Kafka and the Kappa architecture) Then you do lots of data cleaning. Then you do data analysis by building models, testing, fitting/training/validating them using "Machine Learning" (which is a computationally intensive version of statistics - In contrast with traditional statistics that assume data is sparse, here data is "big" but may not be of good quality). More recently neural network, deep learning, and reinforcement learning created a "third wave" where you may use even more powerful model (that are even harder to design, setup, and operate) to do AI stuff, like detecting things in pictures, translate text, chatbot, etc.

Next, you want to review your current background to see what your true starting point is.

(Don't worry or obsess too much. Programming skill is almost an infinite-dimensional vector space - you will always end up short in some part and that's okay. What we need is a minimal level of ability across a sufficiently broad area so that you can be productive without having to check google for the tiniest thing)

http://sijinjoseph.com/programmer-competency-matrix/

(And yes, this section is excruciatingly long because CS has grown exponentially, and not even Donald Knuth - Godfather - himself could follow the whole field, so what hope do we have as mere mortals? This journey is for beginners and veterans alike - they differ only at where they start, and restart.)

CS Fundamentals

http://matt.might.net/articles/what-cs-majors-should-know/
(Don't be scared - actually fulfilling his requirement verbatim will basically make you a da Vinci type universal genius, which is of course unrealistic.)

If we look only at the theory side, do you know:

- Data Structure and Algorithm?
  You don't actually need deep knowledge, unless you are an algorithm engineer (a rare niche). If you work in world-class company you may need to work on interesting problems in this area from time to time. If not, you only need to be able to apply the known stuff correctly. (Sometimes the less common thing like a trie, kd-tree, etc)
    - Interface for Map and List (JSON)
    - Performance characteristic for dynamic array vs linked list
    - Be able to choose the right implementation when you need to use graph and tree
    - Can do Big O analysis and use it to write code for data processing that scale reasonably
    - (Advanced) Also understand their variations, such as functional data structure, probabilistic ones, concurrent ones, and niche data structure like those for graphics (if you work in that field)
    - Apply moderately complex algorithm to solve problems like backtracking, finding the influential node in a graph, formulate model for optimization problem then apply gradient descent etc
    - Know when a problem is algorithmically intractable (NP-completeness) and requires meta-heuristic (e.g. simulated annealing) or other measures (such as algorithm that provably gives you an approximate answer guaranteed to be at least 90% optimal).
- Programming Language - Design, Semantics, Execution Model Concept and Implementation?
  (See also: SICP, and also its successor, HtDP)
  (One standard textbook is here; there are many others for various purposes)
  You will overwhelmingly likely be "just a user" of some programming language (at least at first). At an entry level you need to understand its syntax and semantics enough to be able to program in it. To progress, you want to learn the fundamental, universal concepts (high level features, low level protections) that are implemented to varying degree of success in the languages, and are able to apply them suitably in order to more powerfully express your idea/algorithms, so that they become more maintainable, understandable, clean and safe.
  So why would we need to know about execution model and implementation, things that should be done by a language implementor only? Well, all abstraction eventually leaks, and it is important to know how they work behind the scene in order to know why things are the way they are.

What about design? Not everyone can invent a new language, but almost everyone will eventually design and implement a mini-language, what we call **Domain specific language**. This is usually done subconsciously, and you definitely should steal the lessons won by people who designed much bigger languages, in order not to repeat the same mistakes.

- (Hard for a newbie but good exercise) Can you describe in full the semantics of a generic, make-believe procedural language?
- What is pointer, reference, and recursion?
- Can you explain Object Oriented Programming concepts?
  - Read up on the early history of OOP. Who invented OOP? What is his original vision? How does a programming model with message passing as the dominant primitive and “extreme late binding of everything” look like? (Further study: The actor model. Akka, Erlang, and Elixir)
  - Do you know about unconventional versions of OOP? Such as Prototype based ones - Javascript.
- What is the functional style?
  - Function as first class citizen, higher order function (map, filter, reduce, and their role in the Google MapReduce framework), anonymous functions. Partial application and currying. Closure. Continuation and continuation passing style.
  - Concept of purity. Why mutable variable is bad. Data transformation pipeline.
- (Advanced but really important) Theory of Pure Lambda Calculus. Church encoding of stuff. Modelling realistic programming language via lambda calculus style (see: http://lambda-the-ultimate.org/ which may be too niche, but it is named after a series of papers called “Lambda the ultimate” which is a classic). Advanced static typing via the Lambda cube.
- What is Static Typing?
  - How does advanced statically typed system look like.
  - Can you apply PLT (Programming Language Theory) as well as Category Theory(Math) to help clarify the complicated type classes such as Monad, Lens, etc.
- How does program get executed? What is Compiler vs Interpreter?
  - More advanced mode of execution, such as Just-in-time.
  - What is a Virtual Machine? Runtime environment and the nice thing, such as how Garbage Collection works?
- How does compiler/optimizer works?
  - What is the role of an intermediate representation (IR)? Give example of important IR in the procedural and functional world.
- What is abstraction and composition? Why are they important?
- (Advanced) What concurrency model are there? They are mostly incompatible, but equally important perspectives.
  - What is synchronous vs asynchronous?

        - Future and promise
        - Actor model, message passing
        - Stream, backpressure, etc
- Computer Architecture and Operating System?
  [Reference:
  http://lxr.linux.no/ - Online tool to view linux source code
  https://www.tldp.org/LDP/khg/HyperNews/get/khg.html - Hacker's guide
  https://notes.shichao.io/lkd/ch4/ - Walking through the source code
  https://www.tldp.org/LDP/lki/ - Official Linux Kernel Internal Docs]
  Everyone should know how the bare metal works, even if it is just a naive, simplified picture. Modern CS is differently from any other discipline in that it has a vertical, very steep stack from bottom (which is controversial, see remarks) to the top at your application. Because of this unique complexity, either top-down or bottom-down approach are inadequate by itself, so while the long term goal is to understand the stack end-to-end, one should begins (and gain confidence) by starting at both ends.
  Other reasons include being able to deal with performance issue (eventually, you will get hit with problem that can only be explained at the bare metal level, such as cache), understand the apparent "strangeness" of various language and libraries (which is really due to the constraints and limits at low layer), and to push software down to the hardware (they have more features than you think).
  No program lives in a vacuum - they are meaningful only when they finally produce an effect, whether it is printing on screen, outputting to file, connecting to network, and so on. None of that could happen without interacting with the OS. Hence to understand how programs work one cannot completely avoid understanding a program's true interface with the OS, and how the OS handle them.
  [Other technical types may have more reasons to learn these two fields.]
    - What is the von Neumann architecture?
    - What is RISC vs CISC? Do they matter?
    - What is the L1, L2 cache? What do they imply for the performance of Linked List?
    - How does the CPU execute a program?
    - What is SMP?
    - What is the call stack and heap? How are variable layout done?
    - How does C translates to assembly roughly?
    - What is call convention?
    - How does linking work? System call? What really happens when you get a Segmentation fault and why is it called like that?
    - How does the OS deals with multithreading/scheduling, interrupts, paging and virtual memory? It leverages underlying hardware such as the memory management unit - what features does it provide?
- Mathematics?
  There is only so much one person can learn. Unfortunately, mathematics still play a moderate role in CS (historically their ties are only surpassed by Math and Physics). As

math is so hard to learn, and both fields are so large, it is impossible to predict beforehand what kind of math one would need. However, there are some general directions that one can follow without being too wrong:

- Discrete math - it is not so much a set of topics but the skill and sense to be able to deal with formal proof and reasoning through a finite maze of possible states, like solving a sudoku puzzle by hand, that matters. Many phenomenon one want to study in CS can be modelled as some complicated finite process.
- Probability - once thing become too complicated to analyze exhaustively by hand, one can instead randomize to turn the discrete into the continuous. Then you can use a different/nicer set of tools. Useful for performance analysis.
- Optimization and Statistics - obviously, it's for the cool new kids of AI.

Additional fields for various specialist:

- Networking
  I don't know much about this field. But even networking has progressed in the years. Some things I've heard of: transiting to IPv6, software defined network, L2 vs L3, etc. (How about IoT and 5G? How about QoS, streaming, and the brave new protocol disrupting L5, such as Google SPDY and HTTP2 ?)
  (Another thing is that while school teach you the fundamentals, you will need to link them to the real world yourself. How does one setup a data center? How to design and configure a real network? What is the process for getting a public, fixed IP assigned to your server? How about DNS? NAT/bridging? Tunneling?)
- Computer Graphics
  Start with the usual purely mathematical/algorithmic stuff, then progress to mix with the style of image processing:
  - Vector algebra, linear algebra and coordinate systems, affine and other transform, quaternion for rotation, wireframe/mesh etc. Then go to do some Computational Geometry as a side branch.
  - Then do the classical round of topics, such as ray-tracing, shading, reflection, texture, particles, lighting and shadow, and more.
- Database
  https://use-the-index-luke.com/
  https://www.sqlite.org/atomiccommit.html
  The old guards here is of course the RDBMS. It is backed up by some elegant (and some ugly) theory. But the long reign of the dreaded ORM and impedance mismatch faces a plot twist: the arrival of NoSQL via a completely different concern, and then how they fight back with NewSQL. It is time to put down our prejudice and learn new theories.
  - Learn relational algebra and the third normal form.
  - Learn how physical file layout is done, and how hash/B-tree index works.
  - Learn the evaluation strategies of hash-join and merge-join, and their variants.
  - Learn about transaction and concurrency control and journaling. In other words, ACID.
    - Learn about the serialization graph, and lots of messy conditions.

        - Learn about 2 phase locking vs 2 phase commit.
    - Learn about distributed computing (which is really a field in and of itself)
    - When you have become an expert in the above field, you are worthy to deal with modern database. See this and this.
        - What is BASE vs ACID?
        - What is the CAP theorem? How is it relevant to this debate?
        - Learn how consensus algorithm works.
        - What is sharding and replication? Consistent hashing? Chord algorithm?
- AI

  I've probably talked about this too many times already. So here is a biased checklist:
    - What is supervised/unsupervised learning?
    - ~~What is the kernel trick in SVM?~~
    - Learn "Statistical Learning Theory" (VC dimension and all that) if you want to dig to the bottom, but even this only covers "classical machine learning" or "**early** modern AI".
    - What is bootstrapping, cross-validation, hyper-parameter tuning, boosting?
    - Important algorithm: naive Bayes, SVM, random forest; k-means, expectation-maximization, mean-shift; Principal component analysis.
    - Explain how does neural network and deep learning works.
    - What is the Stochastic gradient descent? Any benefit to making it stochastic?
    - How to detect/avoid over-fitting? (Why one should have a separate training/testing data set?)
    - What is the vanishing gradient problem?
    - Learn about the voodoo magic of neural network architecture only if necessary. On the other hand, there are many ways to improve their design without being a voodoo and those are worth pursuing. (LSTM, convolution, attention, transfer learning and other variations)
    - If you're into reinforcement learning, look at markov decision process, look at operation research, bellman equation, and dynamic programming for the theoretical formulation.
    - After that look at temporal difference, Q-learning, policy gradient.
    - Then look at the A3C framework.
    - You may also want to look at Natural Language Processing, one of the big frontier. It is all black magic, so don't say I haven't warned you.

Remarks:

- What is the "true bottom" of the computer stack? The boundary can get fuzzy. If you are a system person, you would say it ends at the physical artifacts such as CPU and RAM (and a conceptual model of how they work). But if you are a hardware person, or you need to do more intense interfacing, you would need to know how those integrated circuit works at the circuit, or even logic gate level. But if you are an electronic engineer, or you need to debug/workaround a design, you want to look at why the circuit doesn't

work, you would need to drill down to the physics of the semiconductor, look at the wrong timing diagram and voltage levels, etc. But if you are a research scientist, quantum computing is the future (to combat the chaos of multi-core), and you want to look at the electrons and replace them with qubits.

Practice - the mundane/supporting stuff

You now has to face this barrage of problem (it is a rite of passage):

Pick a language. Pick a framework. Pick a text editor. Pick an IDE. Pick a package manager. Pick a server. Pick a source control system. Pick pick pick pick pick…...

I thought of those “pick” as being repeatedly punched in my face, and by the end I’ve already begun to mistook “pick” as “punch”. Not fun.

The trick to get out of indecision, is the realization that if you are going for the long haul, then the choice mostly doesn’t matter. Usually you will have a large but finite set of choices, of which a smaller subset cover most of the market share (say > 90%). And there are two cases:

- None of them is good enough - eventually you will have to be at least half decent in **all** of them anyway.
- Any of them is good enough - in which case it is just a matter of personal taste and you really can “just pick any”.

Why don’t we have the middle case when some are good and not others, and the choice really matter? Well, in the long haul this doesn’t happen - by competition those that aren’t good enough hopefully dies out. The exception to this is if the market is young, then you should be selective.

Programming Language:

Assuming Web Development, we have:

- The classical: PHP (!), and Perl. Perl is famous for being good at string processing.
- The Enterprise: Two main family, Java (runs on JVM) vs Microsoft (C#, and they now push for F# which is statically typed style functional language). The main difference is that Java is monolingual but cross-platform (Win, Linux and Mac), while Microsoft is language polyglot but only on Window. Also because the language is owned by Microsoft, it can be a dictator and introduce innovation to the language at a much faster rate than Java (which have to go through the Java Community Process JCP). However Java’s ecosystem is **huge** , which is good if you want to always have an answer when you need a library.

    - Due to Java being so verbose and IMO abused, some JVM-hosted languages have appeared over the years, such as Groovy, Scala, Clojure, and the new cool kid, Kotlin.
- Hacker's Dynamic Language:
    - Python: Lots of people recommend it as a good first language for beginner. It does have some harder parts (mainly the tooling and environments), but overall is reasonably easy to learn as long as you have a C family background. The real pros is that it is going very strong due to it **winning multiple bets on the hype cycle in a row** - first it is good at scripting and become the go to choice for DevOps, then data science + machine learning + AI for scientists and engineers alike.
    - Ruby: Strong emphasis on **programmer-happiness-as-a-priority** and **community-must-be-friendly-to-beginner**. It is basically Lisp-with-syntax (although it works hard to not scare away people with this point by hiding it). Become popular due to the **Ruby-on-Rails** web framework many years ago. However popularity seems to have been waning and so progression path after you've become proficient is a potential problem.
    - Javascript: **Experiencing a lots of growing pain** as it is forced to become a full-fledged, mature general purpose language, when it is originally a hack written over two weeks only. Because of this the language core design is still messy (and plain counter-intuitive to even seasoned developer) and being sorted out through their yearly updates (ES201x). The one strong point for it is that it is ubiquitous on the web as it is **the only base language supported on modern browser**. In a sense it has become not a language but a runtime for other languages (transpiling languages). Watch out that WebAssembly may pose a threat by replacing it outright.
        - It can also work on backend via Node.JS (which has it claim to fame due to it supporting **asynchronous operation**, allowing high performance web server)
        - For the transpiling langs, TypeScript is popular due to Angular framework pushing for it, Dart is great but experienced a hiccup due to Google abandoning development and marketing effort suddenly before, although this year they're rebooting it. Elm is special in that it has great, understandable error message; and Purescript is the less popular cousin of Elm. Note that Coffeescript used to be popular but has become legacy (not because it's bad, but simply because it is invented right in the transitional period for Javascript, when jQuery + vanilla JS just won't cut it, but fully mature frontend framework hasn't yet appeared, instead they only have).
        - Be very careful about **Javascript Fatigue and Javascript Library/Framework Churn**. The second ones refer to the fact that because it is in a growing pain phase, and because of shortcomings of the original languages, people **frantically create new libraries** to fill

those hole all in **different ways**. As a result the knowledge you gain can rapidly become **useless if that library turns out to lose** in the market competition. In fact there is a joke that while you are reading this paragraph, N new JS libraries have just been created. The first one refers to how it is impossible to keep yourself up to date, and if you dare to try, **you will soon get tired**, while the elusive goal keep moving further away. The takeaway is that one must be **selective in what you learn** (instead of the blindly-read-everything approach), combine learning-with-practice with a **done is better than perfect** mentality, and learn the **universal principles over transient fads**.

- Advanced Statically Typed Language:
    - Haskell: the old guard. Highly academic and emphasis true functional purity (you need to use the dreaded IO Monad, which is not a Monad, if you want side effect. Unfortunately reading tutorial will not help you understand Monad - the only reliable way to truly understand Monad is to study enough advanced Mathematics to become a professional grade Mathematician, and then combine it with expert level insight into the true nature of computation - and this is why proficient Haskell programmer are so highly sought after. However after that, you still have N advanced typeclasses to learn. Also beware of language extensions which range from the exotic to the absolutely crucial). Suitable as a research language and for pushing the state-of-the-art of programming language theory forward (but see also “dependently typed languages”)
        - Recommended to look at some resources carefully before you start (if you somehow end up choosing this… the programmer equivalent of choosing nightmare level difficulty on a computer game): the full roadmap, “Learn yourself Haskell for great good”, “Real World Haskell”, and this (haven’t vetted this resource, but still).
    - Standard ML and OCaml. Support comparably advanced static typing as Haskell, but does not insist on purity. Very nice for learning/teaching.
    - The spirit of these language (Haskell in particular) ended up influencing many of the more “pragmatic” language, such as the transpiling JS languages above.

There are also specialty/niche languages:

- Concurrency/Distributed Language: Erlang is the undisputed king here with its BEAM virtual machine, and the OTP platform - you can use it to write system that just keeps running even if some component fails. However it is pretty old and the syntax hard. So we now have the “happy version” - Elixir.
- Contender in the system programming landscape
    - For a long time if you do system programming, the only choice is C (maybe C++). Now we have some more choice.
    - Go: Invented by Google to deal with the demand for high performance and scalability (so you need good concurrency). It also has a pretty uniform style that makes it highly readable even if it is written by someone you’ve never met in life.

    Be careful that it uses garbage collection though so if you want hard guarantee about latency (i.e. real time computing), and you (or your boss) are dogmatic, that could be a deal-breaker.
  - Rust: Can be thought of as further evolution of C++ (although C++ is also finally moving forward with new features after decades of stability), target somewhat lower level than Go, and has **safe, manual** memory management. (Compare also with D)
- C++. It is rather special in that while it can be used to enhance C with OOP, another even better use is for writing generic, high performance numerical library.
- Smalltalk. The original OOP language (which is completely different from what we have today - ask Alan Kay about his “message passing model” taken to its logical conclusion) (also look at the metaclass). Aside from that what made it famous is its IDE - you don’t edit source code file - the program plus the runtime is the IDE itself. See also Pharo for a modern re-incarnation.

Another random point. What is an ecosystem? It is a set of mutually reinforcing “things” surrounding/protecting a core technology that make it possible to be productive in it, basically a symbiosis. You need to organically grow it.

Now if you are a true beginner, you may be worried that you can’t easily learn one, let alone so many language. Fear not, unlike natural language, most programming languages are context-free and hence “easy” compared to human language, and those within the same family are all pretty similar.

The usual learning path is:

- Learn the data types
- Learn some syntax
- Learn the procedural portion (control flow) of the language
- Learn the (recently introduced) functional elements
- Learn how their flavor of OOP works
- Learn the standard libraries for the kind of work you need
- (Optional) Learn special features, such as Annotation in Java, generators in Python, async and promise in Javascript, block in Ruby, template meta-programming in C++, and macros in the Lisp family.

I don’t know what core technology you will pick, but the following choices are kind of technology-agnostic:

Command Line:

Not a choice, but you should learn how to operate one. Aim for the following:

- Basic navigation (ls, pwd, escaping from a process, working with files and directory, and a whole bunch of things) become your reflex and instinct.
- The slightly complex ones are in your memory, you only need to Google for the details (e.g. netstat -anp, you remember a few standard variations for the arguments, and google if you need something else)
- The really advanced ones (and OS dependent commands, say system service/daemon differ depending on your distro, ubuntu vs redhat/centos vs debian, etc) you know how to dig out.

Text editor:

- Learn the old UNIX way - nano, vim (recommended), or emacs.
  They may appear old, but their value is that you will (almost) always have access to them as long as you can ssh to that Linux machine - say if you need to do production rescue then that's the way to go.
    - Nano is very basic and only suitable for beginner.
    - Vim is reasonably easy to learn and also reasonably powerful (Be very careful that vi is a primitive version of vim and shouldn't be used - arrow key doesn't work and will break the file you're editing and you have to use asdw instead. Well unfortunately some very old Linux system only has vi).
    - Emacs is a mysterious one with a "time-bending learning curve", that have gained a cultish, religious following. Perhaps it is more accurate to say that you are not really using Emacs, but your own personalized Emacs, constructed symbiotically through years of interactively, incrementally writing your own plugins/core so as to bend it to your will (It is extremely flexible and customizable). And this is just like in Harry Potter (wand choosing you) or even Star Wars (making your own lightsaber).
- Then pick a modern one - Sublime text, Atom (recommended), VS code.
  They all emphasis being a text editor that have powerful capabilities and more powerful plugins that can replace a full IDE.
    - Sublime text is sort of like Notepad++ but with a better range of plugins.
    - Atom is pretty popular now, it is based on the electron engine.
    - VS Code is Microsoft's response to the challenge of those text editors against Visual Studio (they work especially well with hacker culture, dynamic language). Whether it can really breakthrough the limitation of its bias, you will have to try it out and judge yourself.

Integrated Development Environment:

This depends somewhat on language, but some outstanding ones are: IntelliJ (Java), PyCharm (Python). Note that the good one usually cost money, which is why learning a modern text editor really matters.

Source Control System:

- No questions - must be a Distributed Version Control System this days - either git (highly recommended), or mercurial if you don't mind being niche (it does have a simpler, more user friendly interface). Only learn the legacy generation (SVN, CVS) if you are the poor sod forced to maintain an old project.
- Learn to rely on the command line interface.
- Learn the concept of branching and merging (and how git makes it so easy and low friction - in the old days merging is a delicate operation that could fail and mess up your repository in a way that the only solution is to nuke it and start over)
- Learn the distributed nature of git, and the various integration models invented by different organizations and societies. (here and here)

Package Manager/Build Toolchain:

While they are *a priori* distinct concept, for convenience many modern language provide a bundled, package deal.
However some languages, due to historical trauma, still has fragmentation in this space.

Usually they should have these functions:

- Central repository and dependency manager: Some community sponsored website host authoritative versions of various libraries, and you just specify the libraries you want, then the tool will: 1) Resolve "Transitive dependency" (You use library A, "A" may depend on Library B that you will also need even if you don't use it yourself and didn't state in your dependency declaration file), 2) Fetch the libraries automatically and add them to your project.
- Application Template and Scaffolding: Instead of having to create all the folders and config file manually, you can just type a command and have it do these routine task for you.
- Build and packaging: Even in the old days of C (see remarks below), you use a compiler to build object files, then link using a linker (although the compiler has a flag to do both steps at once). With modern languages building is a complex task consisting of a pipeline. Packaging adds more problem - sometime you need more than a single executable, and how do you pack static assets say, in a form suitable for release/distribution become a big headache. The tool here should provide both powerful primitive operations as well as being either configurable (if they follow the declarative

school of thought) or programmable (if they follow the procedural/scripting school of thought).
    - For the scripting style tool, they can be used standalone as a general purpose “Task Executor”, say to watch your source files for change and rerun unit-test upon updates.
- Plugins: to further extend its power.

This picture used to be a popular summary of the options out in the market.

The mainstream ones are:

- Python: pip (easy_install is deprecated). Just beware that binary packages (those that have C source code and requires compilation for high performance) are more troublesome to deal with. If you can use a pre-compiled version and it works, use it.
    - **You also need pipenv/virtualenv to isolate dependencies. (other language does it elsewhere, e.g. npm local package vs -g)**
    - They also have their own terminologies: setuptools, eggs, wheels...
- Javascript: npm + webpack. (The not-so-old days are the absolute horror story of Grunt, Gulp, Bower; and babel, yeoman; and CommonJS, requireJS, AMD, ES-whatever, we also have backend JS with Node.JS)
- Ruby: bundler (rake and gem).
- Java: Maven (recommended) vs Ant is the old way. Then come gradle ;), and other JVM based languages (I use clojure, not a common choice).

Extra: Platform version Manager

To round up the fun, the language itself sometimes have to evolve in non-backward compatible ways, or it wants to phase out support for ancient, well entrenched version in order to be able to modernize and innovate. This present a crisis because language is the core tech itself and all the tools and ecosystem depends on it. So now instead of just installing the language core, then install the tools, we should first install a platform version manager, and then install multiple versions of the same language (yes, this is needed because we are in a transition period for all the mainstream languages - the new ones are unstable while the old ones doesn’t have that cool new feature). You invoke command for this version manager to switch between using which version of the language.

The downside is that now just setting up a language and working with it becomes a real pain-in-the-rear.

What we have now:

- Python: virtualenv can serves as a temporary solution. For the serious, use anaconda (though it is more like a specially prepared installation targeted for data scientist etc). Otherwise try pyenv.
    - Due to the Python2 vs Python3 rift.
- Javascript: nvm.
- Ruby: rvm.
- Java: jenv for mac or Linux, jabba if you're in window (oh poor!)
    - Due to the recent decision to change to an agile release cycle every 6 months. The old ones are Java 7, Java 8 is the first modernized version (lambda!). Then it is Java 9, 10, 11 (upcoming *fast*).

Remark: Well, the C world has make, configure, autoconf, and more recent innovation to catch up with modern developments.
Remark: Even with that solution, such transition risk **fragmenting the language community** if not done carefully.
Remark: Even C++ now has those C++11 and up features.

Practice - the meat

Welcome. This has been a pilgrimage, and you have proved your fervor and dedication. Now you can truly initiate.

For Web Development:

(General purpose references:
https://developer.mozilla.org/en-US/docs/Web/Guide/Introduction_to_Web_development
MDN

https://developers.google.com/web/fundamentals/primers/service-workers/
Google Web Fundamentals Series

https://developers.google.com/training/
Google Developer training
)

Fundamentals:
The Great Vision of the World Wide Web is to open up new opportunity for everyone by having a global, equal platform. To do so one of its design goal is to have anarchic distribution/scalability - the ability for people to cooperate without having to coordinate. This is accomplished through open standards.

- The HTTP protocol
    - The basics of how it works: URL, HTTP method (GET, POST, PUT, etc), how parameters are passed (get param vs. form post param vs json encoded param)
    - Headers and Cookies, and how this relates to the implementation of Session in application
    - How browser does caching
    - The REST architectural style (if you want to go deep, read Roy Fielding's thesis where this all started), see the four level of adoption. In practice it is actually more like JSON based RPC (still better than web service through XML). Understand his vision of web-scale, *anarchic* system integration.
        - See also the tooling surrounding it such as Swagger.
        - In recent years the GraphQL architecture (Facebook) supplement this approach.
- HTML5 and CSS3
  (An important reference is the Mozilla Developer Network - should use it over W3School which is not affiliated with the standard setting organization w3c, and had its share of inaccurate reference information. I heard this problem improved in recent years though. Another good one is SitePoint. For online sandbox I use https://jsfiddle.net/ - alternatives include CodePen and JSBin)
    - Learn about the DOM (Document Object Model).
    - Learn their new features, such as flexbox.
    - Learn to do complex layout using CSS (not easy!) - use this tool to check compatibility with browsers.
    - Learn about web best practice such as "separation of content and presentation", "Semantic HTML", "Progressive Enhancement and Graceful Degradation", accessibility, and "Responsive UI" (i.e. change layout when screen width change through CSS media query).

Basic (Programming) Architecture:

- Learn about the MVC model for the first/front layer of a web application. (where Controller ~= routes)
    - Learn about its variation such as MVVM or MV* for modern frontend application.
- Learn about the 3-tier/3-layer structure: presentation layer, domain/business layer, and data access/persistence layer.

Follow the stack (for backend) (not LAMP I presume):

- The client-server / request-response model
- How does a web server work?
    - Learn to operate some fronting server (as opposed to web application server). Apache httpd and nginx.
- The integration interface with web application:
  This is under-appreciated but I think it is important - similar to the central role of the Internet Protocol, sitting in the middle L3, providing a single uniform interface for both the

provider(web server) below it, and the services(web framework) above it.
Because of this, I believe newbie should try to program directly using this interface (just something simple is okay) to understand better the difference between various concepts. CS is unique for having a steep stack and if you're not level-headed enough, you can easily confuse/mix up layers.
    - CGI for the old people
    - WSGI for Python
    - Servlet for Java
- Also these interfaces usually have concepts of routes and filter//middleware.
- (Backend) Web Framework
  They provide sensible defaults for many thing, and package many commonly used libraries together (or make it themselves) so that you don't have to repeat elements that are common to any web application.
    - Choices in different languages:
        - Ruby: Ruby-on-Rail. It is famous for two ideas: Convention over Configuration, and ActiveRecords.
        - Python: more choices, such as Django.
        - Java: Structs in the old days, Spring(MVC) is mainstream and a bit old now, Play is the new comer. (Also have "dependency injection")
    - You should also know some common libraries (may or may not be mandated/fixed by the framework) in the language you've chosen.
        - HTML Templating. Many have their own script-like language that can interleave with HTML code. Though it is tempting - leave/extract domain logic out of the template file - they should be for presentation only.
        - Working with (serialization) data formats - JSON, XML.
        - HTTP client libraries - the most common way (hopefully) of doing system integration.
        - Object Relational Mapping (ORM) framework - see Database section below.
- Database
    - How to operate a Relational DB (MySQL/Postgresql/Oracle/Microsoft SQL Server etc)
    - How to use SQL (select, join, aggregation via group by, complex query eg correlated subquery)
        - (Optional) Advanced SQL/DB feature such as Materialized view, trigger, Common table expression
    - How to do transaction
        - Knows that despite the ~~brainwashing~~ heavy marketing on ACID, the default transaction isolation level is not what you expect - how to override it if you want
        - How to control locking manually
    - How to use Index to improve performance
    - How to design a schema

        - How to denormalize, add generic column for future extension, model dynamic fields, and all sorts of trick they don't teach you in school
    - How to talk to the DB programmatically
        - Using primitive interface
        - Using an ORM. A good one is Python's SQLAlchemy.
            - How to use the libraries query language//query builder methods to perform complex query
            - How to annotate a class with relational mapping
            - How to create/update/delete objects
            - Understand some limitations/problem of ORM such as the "N+1 problem"
    - How to operate and use a NoSQL Database
        - How to design (could be implicit) schema that fits with application usage

Other concerns:

- Web Security
  We're not expert in this area, but there are still something we ought to do:
    - Common Sense. Use ssh and not telnet, use PGP in email if you are hardcore. Always use HTTPS (HSTS).
    - Guarding against famous attack
      The most important ones are:
        - Cross Site Request Forgery (CSRF)
        - Cross Site Scripting (XSS)
        - SQL Injection
    - To guard against SQL Injection, use Parameterized Query. For XSS, always sanitize user input using a battle-proved library. CSRF is the hardest to defend, but you can start by reviewing your Cross Origin Resource Sharing (CORS) policy, and by adding anti-CSRF token into HTML forms.
    - Using Cryptography
      Learn the two/three basic laws:
        - The security of a cryptosystem must depends solely on the secrecy of the keys - it must not be compromised even if every details of the system's design is leaked (i.e. no Security through obscurity)
            - Knows how to secure the secret key - DO NOT store it in source control (Yes this happen a lots)
        - Always assume an adversarial model - the attacker could be highly resourceful and determined (think NSA level)
        - DO NOT design your own cryptographic system - no matter how secure you think it is, it will be broken fast by some unknown hacker in Nigeria. The only way that is okay (if you really want) is to publish a design to the public and let it be subject to scrutiny by all the experts for years (sometimes decades). Instead use existing, battle-hardened system.

    - Learn the attack models in web communication: Eavesdropper and Man-in-the-middle.
    - Learn the cryptographic primitives
        - Hash. (Cryptographic) Hash is among the least well understood (in terms of how much the academics know about its property) of all primitives.
            - Understand the Rainbow table attack and why simply hashing and then storing user password is insecure
            - Application: Hash based message authentication code. (Beware of the length extension attack)
        - Pseudo-Random number generator (PRNG). Know why the standard ones aren't secure and you should use the cryptographic version (CSPRNG) if it must be unpredictable.
        - Symmetric Cipher. Should know that correctly using it is actually pretty subtle and requires care. (Advanced: name the attack that could happen if you're not careful for each of these details)
            - Padding scheme?
            - Mode of operation? (For block cipher)
            - Initialization vector?
        - Asymmetric Cipher. Usually involves some number theory/math that you can skip if not interested. (RSA // elliptic curve)
            - Application: Encryption vs Authentication
            - How does digital signature + public key infrastructure works.
        - Diffie-Hellman Key Exchange and its application in the TLS protocol
    - Just how dangerous the online world is
        - The story of breaking the hash (why we cannot use MD5 anymore, and even SHA-1 is not really secure against a sufficiently rich entity)
        - The story of breaking SSL, the old HTTPS protocol (POODLE) - why we must use (sufficiently new and good version of) TLS only now. (Use this tool to check - POODLE is so severe that vulnerability to it is an **instant F grade**) Also the Heartbleed exploit for the OpenSSL implementation.

Frontend Development:

- First generation
    - Using jQuery + vanilla Javascript
    - Use CSS framework such as Bootstrap for layout
    - Using CSS preprocessor such as SASS and LESS so that you gain access to more programmatic ability in CSS instead of always having to do copy and pasting code
    - Using AJAX
    - Using the browser's debugger/console effectively
- Current generation (Single Page Application/Frontend framework)
    - (Optional) Some niche but powerful one: Meteor Framework
    - Concept of web component

- The big three:
  - AngularJS (Google) (See also MEAN stack): in my opinion the most Enterprisy one and hence suitable in those environment. Be careful that Angular 1 is completely different from Angular 2 and above and should not be used as of 2018. It has a fairly steep learning curve and Typescript is recommended to make using it less painful. (Just be careful not to mix up the layers in a deep stack, e.g. typescript import VS Angular dependency injection)
  - React (Facebook). The champion for the **functional-reactive architecture**. Recommended if you want functional purity or need its huge ecosystem.
    - Virtual DOM and JSX.
    - What does reactive UI means. (UI as pure function of some input, instead of manual DOM manipulation)
    - Technically, React is just the view layer. The full architecture is called Flux. An important implementation is redux.
  - Vue.js. The little guy who came over after both of them messed up (Google because it suddenly announce Angular 2 and simply discarded Angular 1, also it changes Angular 2 to a rapid release schedule; Facebook because of the licensing controversy where they added a patent clause allowing them to revoke your patent right to all of their software the moment you sue them over patent). Combine the best of both world with reactive view by default, while still allowing state mutations. Recommended for Greenfield project as long as you don't need a huge ecosystem support.

For computer graphics:

(Just a sample ref:
https://www.reddit.com/r/opengl/comments/88hf40/whats_a_good_path_to_learn_3d_graphics/
http://erich.realtimerendering.com/)

Basically for fun and to get started gently, use Unity. You can also try WebGL + Three.JS (I *told you* Javascript is eating the web and the world).

For something more comprehensive (plus game engine), consider Unreal (and its competitors - I played with OGRE when I was still a teenager). You may also want to get your hand dirty on 3D modelling softwares (even though they may not be just programming per se) - blender (general purpose) and Wings 3D (allow creating small scale mesh in an efficient manner) say.

After that you get to the real deal by learning, of course, OpenGL (cross platform) and DirectX (Windows).

Also note that this field is special in that the most important academic event is actually an industry conference: the SIGGRAPH.

For Big Data/machine learning/AI:

- Big data is pretty chaotic with lots of frameworks and no clear winner. The SMARK stack (with apology to the old LAMP) could be a starting point for finding your ways.
- The field of machine learning/AI is dominated by Python because it successfully marketed itself to scientists that doesn't know about high level programming language (but might know C) and rode the wave with excellent timing. The basic library is scikit-learn. We also have, at a lower/foundational layer, Scipy + numpy to replace matlab, sympy for something like Mathematica, matplotlib for plotting graphs, and pandas for statistics.
    - Numpy is special. On one hand it is just a numeric library and provide the same feature as Matlab. On the other hand numpy, just like matlab, ultimately calls BLAS/LAPACK for performance - but due to the way numpy packaged things, it can also serves as a "nice interface" for higher level library to do the neural network stuff.
- The lower level neural network libraries (not literally about neural network per se, but that's their primary use case) - Tensorflow (Google), PyTorch (Facebook), Mxnet (Apache project), CNTK (Microsoft).
    - Unfortunately, if you want to scale up, you will need to use GPU to accelerate, and here it is a one-sided Nvidia monopoly + the CUDA + cuDNN lib.
- Higher level/user oriented ones - Keras. Also note Google's Firebase.

-- THE END --