OCTOBER 6, 2021 / #LEARNING TO CODE

How to Learn Programming – The Guide I Wish I Had When I Started Learning to Code

Jacob Stopak

Just the thought of learning to code can be very intimidating. The word **code** is mysterious by definition. It implies a technical form of communication that computers, and not humans, are meant to understand.

One way many people start learning to code is by picking a popular programming language and jumping in head first with no direction. This could take the form of an online coding course, a tutorial project, or a random book purchase on a specific topic.

Rarely do prospective developers start with a roadmap – a bird's eye view of the coding world that outlines a set of relevant programming concepts, languages, and tools that almost 100% of developers use every day.

In this article, I propose one such roadmap. I do this by outlining 14

I meticulously chose these 14 steps based on my own personal journey learning to code, which spans almost 20 years.

Part of the reason it took me so long to feel comfortable as a developer is that I would learn about specific topics without a broader context of the coding world.

Each of the steps in this article discusses a "coding essential" – something that I believe **is critical to at least know that it exists** at the start of your coding journey.

One final note before listing the steps in the roadmap: of course reading this article will not make you an expert programmer. It isn't meant to. The purpose of this article is to make you aware that each one of these topics exists, and hopefully give you a basic idea of how each one works so you can build on it intelligently going forward.

14 Step Roadmap for Beginner Developers

1. Familiarize Yourself with Computer Architecture and Data Basics
2. Learn How Programming Languages Work
3. Understand How the Internet Works
4. Practice Some Command-Line Basics
5. Build Up Your Text Editor Skills with Vim

8. Start Programming with JavaScript
9. Continue Programming with Python
10. Further Your Knowledge with Java
11. Track Your Code using Git
12. Store Data Using Databases and SQL
13. Read About Web Frameworks and MVC
14. Play with Package Managers

Without further ado, let's start at the top!

1) Familiarize Yourself with Computer Architecture and Data Basics

One of the wonderful things about modern programming languages is that they enable us to create fancy applications without worrying about the nitty-gritty details of the hardware behind the scenes (for the most part).

This is called **abstraction** – the ability to work with higher-level tools (in this case programming languages) that simplify and narrow down the required scope of our understanding and skills.

However, that doesn't mean it's useless to know the basics of the metal that your code is executing on. At the very least, being aware of a few tidbits will help you navigate workplace conversations about high CPU and memory usage.

Your computer's most important parts live on **microchips** (also known as **integrated circuits**).

Microchips rely on an electrical component called a **transistor** to function. Transistors are tiny electrical switches that are either off (0) or on (1) at any given time. A single microchip can contain millions or billions of tiny transistors embedded on it.

Most modern computers have a microchip called the **Central Processing Unit (CPU)**. You can think of it as the computer's brain. It handles most of the number crunching and logical tasks that the computer performs.

Each CPU has something called an **instruction set**, which is a collection of binary (zeros and ones) commands that the CPU understands. Luckily, we don't really need to worry about these as software devs! That is the power of abstraction.

If the CPU is the logical center of the brain, it is useful to have memory as well to store information temporarily or for the long term.

Computers have **Random Access Memory (RAM)** as "working memory" (or short-term memory) to store information that is actively being used by running programs.

RAM is made up of a collection of **memory addresses**, which can be used to store bits of data. In older languages like C, programmers do have access to working directly with memory addresses using a feature called **pointers**, but this is rare in more modern languages.

memory. A hard drive is an internal or external device that stores data that should persist even after the computer is turned off.

Before moving on to more details about programming languages, let's spend a second talking about data. But what exactly do we mean by the word **data**?

At a high level, we think of things like text documents, images, videos, emails, files, and folders. These are all high-level data structures that we create and save on our computers every day.

But underneath the hood, a computer chip (like a CPU or RAM chip) has no idea what an "image" or a "video" is.

From a chip's perspective, all of these structures are stored as long sequences of ones and zeros. These ones and zeros are called **bits**.

Bits are commonly stored in a set of eight at a time, known as a **byte**. A byte is simply a sequence of eight bits, such as `00000001`, `01100110`, or `00001111`. Representing information in this way is called a **binary representation**.

2) Learn How Programming Languages Work

In the previous section, we mentioned that most computers rely on a CPU, and a CPU can understand a specific set of instructions in the form of ones and zeros.

called **machine code**.

Sounds horrible to work with, doesn't it? Well it probably is, but I wouldn't know since I mostly use higher-level programming languages like JavaScript, Python, and Java.

A **higher-level programming language** provides a set of human-readable keywords, statements, and syntax rules that are much simpler for people to learn, debug, and work with.

Programming languages provide a means of bridging the gap between the way our human brains understand the world and the way computer brains (CPUs) understand the world.

Ultimately, the code that we write needs to be translated into the binary instructions (machine code) that the CPU understands.

Depending on the language you choose, we say that your code is either **compiled** or **interpreted** into machine code capable of being executed by your CPU. Most programming languages include a program called a **compiler** or an **interpreter** which performs this translation step.

Just to give a few examples – JavaScript and Python are interpreted languages while Java is a compiled language. Whether a language is compiled or interpreted (or some combination of the two) has implications for developer convenience, error handling, performance, and other areas, but we won't get into those details here.

Whatever type of programming you aspire to do, you'll run into situations where it helps to know how computers interact with each other. This typically occurs over the Internet.

The Internet is nothing more than a global collection of connected computers. In other words, it is a global network. Each computer in the network agrees on a set of rules that enable them to talk to each other. To a computer, "talking" means transferring data.

As we discussed in the previous section, all types of data – web pages, images, videos, emails, and so on – can all be represented as ones and zeros.

Therefore, you can think of the Internet as a very large set of computers that can transfer ones and zeros amongst themselves, in a way that preserves the meaning of that data. The Internet is nothing more than a digital conversation medium.

If the Internet is just a big conversation arena, let's define the conversation participants.

First, an analogy: most human conversations require at least two participants. In most cases, one person initiates the conversation and the other person responds, assuming they are both present and available.

In Internet speak, the computer initiating the conversation is called the **client**. The computer responding or answering is called the **server**.

For example, let's say you open a web browser and go to

In a more abstract sense, YOU are the client because you are the one initiating the conversation. By typing "www.google.com" into the search bar and clicking <ENTER>, your browser is requesting to start a conversation with one of Google's computers.

Google's computer is called the server. It responds by sending the data required to display Google's web page in your browser. And voilà! Google's web page appears in front of your eyes. All Internet data transfers utilize this sort of client/server relationship.

4) Practice Some Command-Line Basics

The **Command Line** can be intimidating at first glance. It is often featured in movies as a cryptic black screen with incomprehensible text, numbers, and symbols scrolling by. It is usually associated with an evil hacker or genius techie sidekick.

The truth is that it doesn't take a genius to use or understand the command line. In fact, it allows us to perform many of the same tasks that we are comfortable doing via a point-and-click mouse.

The main difference is that it primarily accepts input via the keyboard, which can speed up inputs significantly once you get the hang of it.

You can use the Command Line to browse through folders, list a folder's contents, create new folders, copy and move files, delete files,

will give you a feel for working on the command line.

Once you open your terminal, a typical first question is "*Where am I*"? We can use the `pwd` command (which stands for "Print Working Directory") to figure that out. It outputs our current location in the file system which tells us which folder we are currently in.

Try it yourself:

How to Use the Command Line

If you're on a Mac, open the Terminal app, which is essentially a Unix Command Line terminal.

If you're running an operating system without a GUI (Graphical User Interface), like Linux or Unix, you should be at the Command Line by default when you start the computer. If your flavor of Linux or Unix does have a GUI, you'll need to open the terminal manually.

At the prompt, type `pwd` and press <ENTER>. The Command Line will print out the path to the folder that you're currently in.

By default, the active folder when opening the Command Line is the logged-in user's home directory. This is customizable in case you want the convenience of starting in a different location.

For convenience, the home directory can be referenced using the tilde `~` character. We will use this in a few examples going forward.

Now that we know what folder we're in, we can use the `ls` command

Type `ls` and press <ENTER>. The contents (files and subfolders) that reside in the current directory are printed to the screen.

Rerun the previous command like this `ls -al` and press <ENTER>. Now we will get more details about the directory contents, including file sizes, modification dates, and file permissions.

The hyphen in the previous command allows us to set certain flags that modify the behavior of the command. In this case we added the -a flag which will list all directory contents (including hidden files) as well as the -l flag which displays the extra file details.

Next, we can create a new folder using the `mkdir` command, which stands for "Make Directory". Below we create a new folder called "testdir".

Type `mkdir testdir` and press <ENTER>. Then type `ls` and press <ENTER>. You should see your new directory in the list.

To create multiple nested directories at once, use the `-p` flag to create a whole chain of directories like this: `mkdir -p directory1/directory2/directory3`

The Command Line isn't that useful if we can only stay in one location, so let's learn how to browse through different directories in the file system. We can do this via the `cd` command, which stands for "Change Directory".

First, type `cd testdir` and press <ENTER>. Then type `pwd` and press <ENTER>. Note the output now shows that we are inside the "testdir"

browse backwards to the parent directory.

Then type `pwd` and press <ENTER>. Note the output now shows that you are back in the original directory. We browsed backwards!

Next we'll learn how to create a new empty file in the current directory.

Type `touch newfile1.txt` and press <ENTER>. You can use the `ls` command to see that the new file was created in the current directory.

Now we'll copy that file from one folder to another using the cp command.

Type `cp newfile1.txt testdir` and press <ENTER>. Now use the `ls` and `ls testdir` commands to see that the new file still exists in the current directory and was copied to the "testdir" directory.

We can also move files instead of copying using the `mv` command.

Type `touch newfile2.txt` and press <ENTER> to create a new file. Next, type `mv newfile2.txt testdir` and press <ENTER> to move the file into the "testdir" folder.

Use the `ls` and `ls testdir` commands to confirm that the file has been moved into the "testdir" folder (it should no longer appear in the original location you created it, since it was *moved* not copied).

The `mv` command can also be used to rename files.

renamed.

Finally, we can delete files and folders using the `rm` command.

Type `rm cheese.txt` and press <ENTER> to remove the file. Use `ls` to confirm the file was removed.

Type `rm -rf testdir` and press <ENTER> to remove the "testdir" directory and its contents. Use `ls` to confirm the directory was removed.

Note that we need to use the `-rf` flags when removing directories. This forces the removal of the folder and all of its contents.

5) Build Up Your Text Editor Skills with Vim

At this point, we've covered the basics of the Command Line and seen a few examples of how we can work with files without a mouse.

Although we now know how to create, copy, move, rename, and delete files from the Command Line, we haven't seen how we edit the content of text files in the terminal.

Working with text files in the terminal is important because computer code is nothing more than text saved in an organized set of files.

Sure we could use a fancy text editor like Microsoft Word (or more likely specialized code editors like Sublime or Atom) to write and edit

There are several excellent text editors created specifically for this purpose, and I recommend learning the basics of one called Vim.

Vim is one of the oldest text editors around and it is a time-tested gem. Vim stands for "***VI*** i***M***proved" since it is the successor to a tool called ***Vi***.

As mentioned, Vim is a text editor that was built to run directly in the terminal, so we don't need to open a separate window to work in or use a mouse at all. Vim has a set of commands and modes that allow us to conveniently create and edit text content using only the keyboard.

Vim does have bit of a learning curve, but with a little bit of practice, the skills you learn will pay dividends throughout your coding career.

Vim is installed by default on many operating systems. To check if it's installed on your computer, open the Command Line and type `vim -v`.

If Vim opens in your terminal and shows the version, you're good to go! If not, you'll need to install it on your system. (Note that you can quit Vim by typing `:!q` and pressing <ENTER>). For more information on installing Vim, see https://vim.org.

In my opinion, the quickest and easiest way to learn how to use Vim is to use their built-in tutorial, the **VimTutor**. To run it, ensure that Vim is installed on your system, open the Command Line, type `vimtutor`, and press <ENTER>.

It is such a good tutorial that there is no reason for me to waste time

If you still have energy left after you've completed the VimTutor, check out these 7 Vim commands that will dramatically improve your productivity as you get started with Vim.

6) Take-up Some HTML

You can think of HTML – short for **H**yper**T**ext **M**arkup **L**anguage – as the bones of a web page. It determines the structure of the page by specifying the elements that should be displayed and the order that they should be displayed in.

Every web page that you've ever visited in your browser has some HTML associated with it. When you visit a web page, the web server hosting the web page sends over some HTML to your browser. Your browser then reads it and displays it for you.

Most web pages contain a fairly standard set of content, including a title, text content, links to images, navigation links, headers and footers, and more. All of this information is stored as HTML that defines the structure of the page.

One thing to keep in mind is that HTML is not technically a programming language, although it is often referred to as "HTML code".

As we'll see later, other programming languages enable us to write code that **does stuff**, such as running a set of instructions in sequence. HTML doesn't **do** anything. We don't run or execute HTML. HTML just

page should look like, nothing more.

So how do you write HTML? HTML uses a standard set of **tags** (basically just labels) to identify the available elements that make up a web page. Each tag is defined using angle brackets.

For example, the **title** tag is defined as `<title>My Page Title</title>` and the **paragraph** tag is defined as `<p>A bunch of random text content.</p>`.

Each HTML element is made up of a starting tag and an ending tag. The starting tag is just the tag label in between angle brackets, like this:

```
<tagname>
```

This opens the new HTML tag. The ending tag is essentially the same, but it uses a forward slash after the first angle bracket, to mark it as an ending tag:

```
</tagname>
```

Any text between the two tags is the actual content that the page will display.

Let's cover a couple of the most common tags in use. The first is the `<html>` tag. This defines the start of an HTML page. A corresponding `</html>` tag (note the forward slash) defines the end of the HTML page. Any content between these tags will be a part of the page.

defines the end of the HEAD section.

Previously, we saw the `<title>` tag. It defines the title of the web page, which the browser will display in the browser tab. This tag needs to be placed inside the `<head>...</head>` section.

Next is the `<body>` tag. All content inside this tag makes up the main content of the web page. Putting these four tags together looks something like this:

```
<html>

    <head>
        <title>My Page Title</title>
    </head>

    <body>
        <p>A bunch of random text content.</p>
    </body>

</html>
```

The simple HTML snippet above represents a simple web page with a title and a single paragraph as body content.

This example brings up a point we didn't mention in the last section. HTML tags can be nested inside each other. This just means that HTML tags can be placed inside other HTML tags.

HTML provides many other tags to provide a rich set of content to web users. We won't cover them in detail here, but below is a short list

- `<h1>` : A page heading usually used for page titles.
- `<h2>` : A section heading usually used for section titles.
- `<hx>` : Where *x* is a number between 3 and 6, for smaller headings.
- `<img>` : An image.
- `<a>` : A link.
- `<form>` : A form containing fields or inputs for a user to fill out and submit.
- `<input>` : An input field for users to enter information, usually within a form.
- `<div>` : A content division, used to group together several other elements for spacing purposes.
- `<span>` : Another grouping element, but used to wrap text phrases within another element, usually to apply specific formatting to only a specific part of the text content.

7) Tackle Some CSS

A web page without CSS – or **C**ascading **S**tyle **S**heets – is like a cake without frosting. A frosting-less cake serves its purpose, but it doesn't look appetizing!

CSS allows us to associate style properties such as background color, font size, width, height, and more with our HTML elements.

Each style property tells the browser to render the desired effect on

Let's see how to associate CSS styles with our HTML elements. There are three pieces to this puzzle:

The CSS selector: Used to identify the HTML element or elements we want the style to apply to.

The CSS property name: The name of the specific style property that we want to add to the matched HTML elements.

The CSS property value: The value of the style property we want to apply.

Here is an example of how these pieces come together to set the color and font size of a paragraph:

```
p {
  color: red;
  font-size: 12px;
}
```

Let's start at the beginning, before the curly braces. This is where the CSS selector goes. In this case, it is the letter **p** which indicates the `<p>` (paragraph) HTML tag. This means that the styles inside the curly braces will apply to all `<p>` tags on the web page.

Let's move on to what goes inside the curly braces – the styles we want to apply to the targeted elements.

values of these properties (in this case "red" "12px") are on the right. A semicolon ends each property/value pair.

You can probably see how this works. The snippets of CSS code above tell the browser to use red, 12px size letters for all the text placed inside `<p>` tags.

So how does an HTML page know to include these CSS styles? Enter the `<link>` HTML tag. Usually, CSS styles are created in separate files (**.css** files) from the HTML. This means we need some way to import them into our HTML files so the browser knows that the styles exist.

The `<link>` element exists for this purpose. We include `<link>` elements in the `<head>` section of HTML files which allow us to specify the external CSS files to import:

```
<head>

    <title>My Page Title</title>

    <link rel="stylesheet" type="text/css" href="/home/style.css">

</head>
```

In this example, we are importing the CSS styles specified by the **href** attribute, in this case the file */home/style.css*.

In the next 3 sections, we'll (finally) dive into some more technical programming languages!

features and example code so you can hopefully get a well-rounded understanding of the basics of all three.

8) Start Programming with JavaScript

Let's start by answering the following question: if we can use HTML to build the structure of a web page and CSS to make it look pretty, why do we need JavaScript?

The answer is that we technically don't. If we are happy with a static site that sits there and looks pretty, we are good to go with just HTML and CSS.

The keyword here is "static". If, however, we want to add dynamic features to our web pages, such as changing content and more complex user interactions, we need to use JavaScript.

What is JavaScript?

So what exactly is JavaScript? JavaScript is a programming language that was created specifically for websites and the Internet. As we mentioned in section 2, most programming languages are either compiled or interpreted, and programs are typically run in a standalone manner.

JavaScript is somewhat unique in this respect in that it was designed to be executed directly inside web browsers. It allows us to write code

extension or inside `<script>` tags directly in the HTML.

For many years, JavaScript code was primarily relegated to running inside web browsers. But the **Node.js** project changed this paradigm by creating a standalone JavaScript environment that could run anywhere.

Instead of being trapped in a browser (that is, client-side), Node.js can be installed locally on any computer to allow the development and execution of JavaScript code. You can also install Node on web servers which allows you to use JavaScript as backend code for applications instead of simply as web browser frontend code.

Now that we've covered some background, let's dive into a few basics of the JavaScript language.

Variables and Assignment in JavaScript

Variables possibly represent the most fundamental concept in programming. A variable is simply a name or placeholder that is used to reference a particular value.

The word **variable** implies that the stored value can change throughout the execution of the program.

You can use variables to store numbers, strings of text characters, lists, and other data structures that we will talk more about in a minute.

All programming languages use variables, but the syntax varies

our code. This enables us to check their values as needed and perform different actions depending on how the variable's value changes.

In JavaScript, we declare variables using the `let` keyword, like this: `let x;`.

This declares x as a variable that we can use in our code. Note that we added a semicolon at the end of the line. In JavaScript (and many other languages) semicolons are used to specify the end of each code statement.

Now that we have created the variable *x*, we can assign a value to it using the equals sign, also called the **assignment operator**: `x = 10;`

Here we assigned the number 10 to the variable named *x*. Now any time we use *x* in our code, the value 10 will be substituted in.

Both variable declaration and assignment can be done in one line as follows:

```
let x = 10;
```

Data Types in JavaScript

In the last section, we stored an integer (whole number) value in the variable named *x*. You can also store decimal numbers, or **floating-point numbers** as they are known. For example, we could write: `let x = 6.6;`.

floating-point numbers), but we are just scratching the surface. We can store text data in variables as well.

In coding terminology, a piece of text is called a **string**. We can store a string value in our variable x by surrounding it in either single or double quotes:

```
let x = 'Hello there!';

let y = "Hey bud!";
```

The next data type we'll discuss is the **boolean**. A boolean can only hold one of two values, `true` or `false` – and they must be all lowercase. In JavaScript, true and false are two keywords used specifically as values for boolean variables:

```
let x = true;

let y = false;
```

Note that the values `true` and `false` don't appear within quotes the way strings do. If we surround them with quotes, the values would be strings, not booleans.

We often use booleans to control the flow of programs in conditional (if/else) statements which we'll learn about next.

Now that we have an understanding of variables and the basic JavaScript data types, let's take a look at some things we can do with them.

Variables aren't that useful without being able to tell our code to do something with them. We can make our variables do things by using **statements**.

Statements are special keywords that allow us to perform some action in our code, often based on the value of a variable we have defined. Statements let us define the logical flow of our programs, as well as perform many useful actions that will dictate how our programs work.

If / Else Statement

The first statement we'll discuss is the `if` statement. The `if` statement allows us to perform some action only when a desired condition is true. Here is how it works:

```
let x = 10;

if ( x > 5 ) {
    console.log('X is GREATER than 5!');
} else {
    console.log('X is NOT GREATER than 5!');
}
```

We defined a variable called *x* and set its value to 10. Then comes our `if` statement. After the keyword `if`, we have a set of parentheses containing the condition to evaluate, in this case, `x > 5`. We just

Since the condition in the parentheses is true, the code between the curly braces will be executed, and we will see the string "X is GREATER than 5!" printed to the screen. (We didn't discuss the meaning of `console.log()`, so for now just know that it prints the value in the parentheses to the screen).

In the same example, we also included an `else` statement. This allows us to execute specific code in the event that the condition in the condition is `false`.

While Loops

The next type of statement we'll discuss is the **while loop**. Loops enable us to repeat a block of code as many times as we desire, without copying and pasting the code over and over again.

For example, let's assume we need to print a sentence to the screen 5 times. We could do it like this:

```
console.log('This is a very important message!');
console.log('This is a very important message!');
console.log('This is a very important message!');
console.log('This is a very important message!');
console.log('This is a very important message!');
```

This works fine for only 5 messages, but what about 100, or 1000? We need a better way to repeat pieces of code multiple times, and loops allow us to do this. In coding terminology, repeating a piece of code multiple times is called **iteration.**

```
let x = 1;

while ( x <= 100 ) {

    console.log('This is a very important message!');

    x = x + 1;

}
```

In this example, we initialize *x* to the value of 1. Then we write a `while` loop. Similar to the `if` statement, we add a condition in parentheses. In this case the condition is `x <= 100`. This condition will be `true` as long as *x* is less than or equal to 100.

Next we specify the block of code to execute in the curly braces. First, we print out our message to the console. Then we increment *x* by 1.

At this point the loop attempts to re-evaluate the condition to see if it's still `true`. Variable *x* now has a value of 2 since it was incremented in the first loop run. The condition is still `true` since 2 is less than 100.

The code in the loop repeats until *x* gets incremented to the value of 101. At this point, *x* is greater than 100 so the condition is now `false`, and the code in the loop stops executing.

The HTML <script> Tag

Now that we've introduced JavaScript, let's discuss how to add

This is similar to the `<link>` element that we used to add CSS files to our HTML, except that the `<script>` element is specifically for JavaScript.

Let's say we saved one of the previous JavaScript examples we discussed in a file called `customscript.js` in the same folder as our HTML file. We can add this JavaScript file to our HTML by adding the following HTML tag into the `<head>...</head>` section of our HTML:

```
<script type="text/javascript" src="customscript.js"></script>
```

This will load in the JavaScript code from the file, which will execute when the web page is displayed in the browser.

Once you get comfortable with your JavaScript skills, you can try building some of these fun beginner-friendly projects to practice.

9) Continue Programming with Python

Now that you've learned some basic JavaScript, it will be useful to jump into another programming language – Python.

Many programming languages provide a similar set of functionality, including variables, arithmetic operators, if/else statements, loops, and functions.

(the way the code is written) varies from language to language.

What is Python?

First we'll cover a little bit of background information on Python. Like JavaScript, Python is a high- level programming language that prioritizes ease of development over the speed of execution.

In my opinion, Python is one of the best languages for beginners to learn. The syntax is clean and intuitive and it is a very popular language in the open-source and business spheres.

Earlier we talked about compiled languages versus interpreted languages. Python is an interpreted language. Each time we want to run a Python program, the **Python interpreter** actively processes your code and executes it line by line on your machine.

This is different than compiled languages, in which we would first use a compiler to process the code into a more optimized form (an executable), and then execute it later.

Unlike JavaScript, Python was not built to be run directly inside web browsers. Python was created to be a convenient *scripting language* – a language that can be used to write code for arbitrary tasks that usually execute on a user's local computer.

Python code can be executed on any computer that has the Python interpreter installed on it. It is still a commonly used scripting language but is also used extensively for data science and server-side applications.

simply use the equals sign to create and assign variables as needed:

```
x = 10
y = "cheese"
```

There are two differences between the syntax for defining variables in Python and JavaScript. In Python, we don't need the `let` keyword and we also don't need a semi-colon at the end of each line.

Python uses a set of syntax rules based off of whitespace and indentation. This removes the need for line terminating characters like the semi-colon, and block scoping using curly braces.

Data Types in Python

Python also has a set of data types that we can assign to our variables. These include integers, floating-point numbers (decimals), strings, lists, and dictionaries.

Integers, floating-point numbers, and strings are essentially the same as their JavaScript counterparts, so we won't repeat that information here.

In Python, booleans are very similar to those in JavaScript, except that the keywords True and False must be capitalized:

```
x = True
```

Program Flow Control Statements

Like in JavaScript, Python has as similar set of flow control statements, but with slightly different syntax.

If / Else Statement

This is the Python equivalent of the `if/else` example we saw in the JavaScript section:

```
x = 10

if ( x > 5 ):
    print('X is GREATER than 5!')

else:
    print('X is NOT GREATER than 5!')
```

We defined a variable called *x* and set its value to 10, followed by our `if` statement. Since the condition in the parentheses evaluates to `True`, the code indented after the `if` statement will be executed, and we will see the string 'X is GREATER than 5!' printed to the screen.

In Python, we use the `print()` function for printing information to the screen.

Also note the `else` statement above, which will print an alternative string to the screen if *x* if the condition is `False`.

There are two main differences between the Python code above and

In addition, the indentation of the `print()` function actually matters in Python. In JavaScript, the indentation or white space between statements doesn't matter since JavaScript identifies code blocks using curly braces and identifies the end of a statement using a semi-colon. But in this Python example, there are no semi-colons and no curly braces!

That is because Python actually uses the white space and newline characters to identify the end of statements and code blocks.

The colon tells the Python interpreter that the `if` block is starting. The code that makes up the `if` block must be indented (1 tab = 4 spaces is the convention) for the Python interpreter to know that it is a part of the `if` block. The next unindented line will signal the end of the `if` block.

While Loops

Next we'll discuss loops in Python. The `while` loop in Python is essentially the same as we saw in JavaScript, but with the Python syntax:

```
x = 1

while ( x <= 100 ):
    print('This is a very important message!')
    x = x + 1

print('This is not in the loop!')
```

- We removed the `let` when defining our variables.
- We removed line-ending semicolons.
- We replaced the curly braces with a colon.
- We made sure that the code in the loop is indented with a tab.

We printed an additional message outside of the loop to show that unindented lines of code are not a part of the loop and won't be repeated.

For beginner Pythonistas, I recommend taking a peek at the Zen of Python, which is a list of 20 rules-of-thumb for writing Pythonic code.

And when you get comfortable with the basics, try building some of these fun beginner-friendly Python projects.

10) Further Your Knowledge with Java

Now that we've worked with a couple of higher-level programming languages, let's take it one step lower with Java.

Unlike JavaScript and Python which execute source code in real time using an interpreter, Java is a compiled language. This means a compiler is used (instead of an interpreter) to convert Java source code into a form the computer can understand.

Most compilers generate one or more executable files made up of

But Java is somewhat special in that it compiles the Java source code into an intermediate form called **bytecode**. This is different than the machine code that most other compiled languages produce. Java bytecode is intended to be executed by something called the **Java Virtual Machine (JVM)**.

You can think of the JVM as a program that you install on your computer, which allows you to run Java programs by executing Java bytecode. When people talk about "*whether or not Java is installed on a computer*," they are usually asking whether or not the **JVM** is installed on the computer.

The JVM serves a similar function to the interpreters we discussed in previous chapters. But instead of taking source code (which is stored in .java files) as an input, it takes compiled bytecode.

The benefit of this setup is that it allows bytecode compiled on particular operating systems and platforms to be executed by a JVM on any other platform.

For example, imagine we have a file of Java code that was written and compiled to bytecode on a computer running the Windows operating system. This bytecode can be executed (that is, the program can be run) by a JVM on any platform, including Windows, Mac OS, Linux, and so on.

This is not the case with most compiled executables in other programming languages, which can only execute in the environment which they were compiled for.

so far (Python and JavaScript) is that Java is a **statically typed** language.

This means that the data types of our variables must be known and established at the time the program is compiled.

Each time we create a variable in Java code, we need to explicitly specify the data type of that variable, such as an integer, string, and so on. This is called variable **declaration**.

Once we declare a variable's data type, it can only hold that type of data throughout the execution of the program.

This is very different from JavaScript and Python, where variable data types are established during program execution, also known as **run time**. Languages like JavaScript and Python are therefore referred to as **dynamically typed** languages – we don't explicitly state variable data types in our source code and can easily reassign a variable to any type on the fly.

In Java, we create variables using this syntax:

```
Datatype name = value;
```

Here the `Datatype` is the type of data that the variable will store, such as Integer, String, and so on. Next, the `name` represents the name of the variable we are defining so we can use it in our code. The `value` is

Data Types in Java

In Java, the basic built-in data types are called the **primitive** data types and they will look very familiar based on what we have seen in higher-level languages like Python and JavaScript. The main primitive types are:

- Integer `int` : Stores whole numbers between −2,147,483,648 and 2,147,483,647.
- Float `float` : Stores decimal numbers between 3.4x10^−038 to 3.4x10^038.
- Boolean `bool` : Stores one of the two boolean values `true` or `false` .

Note that there are a few other primitive types (short, long, byte, and double) that we won't be covering here since they aren't used as often as the others. Here is how we initialize these data types:

Integer: `int x = 100;`

Float: `float pi = 3.14;`

Char: `char middleInitial = 'T';`

Boolean: `bool isHuman = true;`

I do want to reiterate that once the data type of a variable is declared, that variable can only hold values of the specified data type.

For example, an error would be thrown if our program tried to store a

The next data type we'll discuss is the string – a sequence of characters, numbers, or symbols represented as textual data.

Strings in Java are a non-primitive data type, which means they are built up from smaller parts. To declare a string variable we use the String data type and place the assigned value in double-quotes:

```
String name = "Harry Potter";
```

Program Flow Control Statements in Java

Like JavaScript, Java uses curly braces to define code blocks for `if` statements, loops, and functions. We'll examine the same program control statements as in the previous chapters and update the examples to use the Java syntax.

If / Else Statement

Here is the Java if/else statement that mirrors the examples in the previous sections:

```
int x = 10;

if ( x > 5 ) {
    System.out.println("X is GREATER than 5!");
} else {
    System.out.println("X is NOT GREATER than 5!");
}
```

The only differences are we declared the datatype of *x* to be int and we using `System.out.println()` instead of `console.log()` to print out our message.

Next, we'll move on to loops in Java. Since Java and JavaScript syntax are quite similar, the `while` loop in Java is essentially the same as we saw in JavaScript:

```
int x = 1;

while ( x <= 100 ) {

    System.out.println("This is a very important message!");

    x = x + 1;

}
```

This `while` loop will print out the specified message 100 times.

This concludes our sections on specific programming languages. It may have been a bit repetitive since we covered the same set of concepts in 3 languages, but hopefully this helped hammer in these basic but fundamental ideas.

Now we'll round out this article with a few in-between topics that you might not otherwise start learning right away.

We'll talk about an essential collaboration tool called Git. Then we'll learn to store and access data in a database. Next we'l briefly touch on

11) Track Your Code Using Git

Git is the most popular Version Control System (VCS) in use today. It allows multiple developers to collaborate on software together. In this section we'll learn what Git is, how it works, and how to use its basic commands.

Before jumping straight into Git, let's flesh out some concepts common to most programming projects.

The full set of directories and files that make up a software project is called a **codebase**. The **project root** is the highest-level folder in the project's directory tree. Code files can be included directly in the project root or organized into multiple levels of folders.

When the codebase is ready for testing or deployment it can be **built** into the program that will run on your computer. The **build process** can include one or more steps that convert the code written by humans into an executable that can be run on your computer's processing chips.

Once the code is built, your program is ready to run on your specific operating system, such as Linux, Mac OS, or Windows.

Over time, developers update the project code to add new features, fix bugs, implement security updates, and more. In general, there are three ways developers can make these changes to a software project:

1. Add new files and folders to the project

As projects grow and new features are added, the number of files and folders (as well as the amount of code within them) increases. Large projects can grow up to hundreds of thousands of files containing millions of lines of code.

To support this growth, the number of developers on large project teams typically increases. Large software projects can have hundreds or even thousands of developers all working in tandem.

This begs the question: "*How the heck do all these developers, who may be geographically spread out all around the world, keep track of their software project code in such a way that they can work together on a single project?*"

Development teams need a way to keep track of exactly what changes were made to the code, which files or folders were affected, and who made each change. Each developer also needs to be able to obtain updates from all other developers.

This process is called **versioning** or **version control**. Developers use special tools called **Version Control Systems (VCS)** to track, manage, and share the versions of software projects. Here are a few popular version control systems that are actively used these days:

- Git
- Subversion (SVN)
- Mercurial (Hg)

Git forms the core of popular web-based VCS platforms like GitHub and Bitbucket. Git is an essential tool for any well-rounded developer to add to their skill set.

Basic Git Commands

Git creates and stores information about our software projects in something called a **Git repository**. A Git repository is just a hidden folder on your computer that Git uses to store data about the code files in a software project.

Each software project we work on typically has its own Git repository for storing information related to that project. This way, code related to different projects on a single computer can be tracked separately.

There are two main ways to create a Git repository on your computer. The first is to create a brand new Git repository in an existing folder on your file system.

To do this, simply open up the Command Line, create a new folder somewhere convenient like on your Desktop, and browse into it:

```
cd ~/Desktop

mkdir testgit

cd testgit/
```

```
git init
```

You should see some output similar to the following:

```
Initialized empty Git repository in /Users/me/Desktop/testgit/.git/
```

All of the Git commands we'll run start with the word `git` followed by a space and then the specific Git command we would like to run. Sometimes we'll add flags and arguments after the Git commands as well.

The `git init` command creates a hidden folder called `.git` in the current directory. This folder is the Git repository we mentioned above. You can see this by running the command `ls -al`.

The second way to get a Git repository on to your computer is to download one from somewhere else, like Bitbucket or GitHub.

Bitbucket and Github are websites that allow people to host open source projects that can be downloaded to your computer.

If you browse to a project you find interesting on Bitbucket or GitHub, you'll see a button labeled Clone. This button will provide you a command and URL that you can copy and paste into the command line terminal. It will look something like this:

The `git clone` command downloads the repository from the specified URL into a new folder on your computer. The URL can either be a web URL as in the example above or an SSH URL as follows:

```
git clone git@bitbucket.org:jacobstopak/baby-git.git
```

After running the git clone command, you should see a new folder created. If you browse into it, you'll see all of the files and subfolders that make up the project you downloaded.

The next command we'll mention is `git add <filename.ext>`. The `git add` command is used to tell Git which files we want it to track, and to add changes in already tracked files to Git's `staging area`.

Once new or changes files have been staged, they can be committed to the repository by using the command `git commit -m "Commit message"`. This will store the changes in all staged files in the Git repository.

The `git status` and `git log` commands are handy for reviewing the current state of the working directory and the commit history of your project.

We barely scratched the surface here. Git has many more essential commands which are definitely worth getting comfortable with.

A database is a program specifically designed to efficiently store, update, retrieve, and delete large amounts of data. In a nutshell, we can think of a database as a container for a set of tables.

You have probably worked with tables in Microsoft Excel. A table is just a set of columns and rows containing data. We can set up tables in a database to store the information that our programs need to work properly.

Whether we are writing programs in JavaScript, Python, Java, or some other language, we can tell our programs to interact with databases as needed.

We can retrieve data from the database to display to our users on a web page. We can accept a web sign-up form from a user and store that user's information in a database for later use.

Our programs can interact with databases in real-time as events transpire in our application. To do this, most databases speak a language called **SQL**, short for **Structured Query Language**.

SQL is a programming language specifically created for databases. It allows us to tell databases what to do.

A chunk of SQL code is called a **query**. We can write SQL queries to fetch the data we need at a particular time or to insert new data into a specific table. Roughly speaking there are two main types of SQL queries: **read-SQL** and **write-SQL**.

On the other hand, a write-SQL query either inserts new data into a table, updates existing data, or deletes existing data. We'll learn how to write some basic read-SQL queries in this section.

Before writing a query, it helps to know what we are querying! Traditional databases contain **tables** made up of columns and rows. When we write a read-SQL query, our goal is usually to retrieve a subset of those rows and columns.

For example, let's say we have a table called `PERSON` with 4 columns, `FIRST_NAME` and `LAST_NAME`. We can use the following query to select all the data from only the `FIRST_NAME` column:

```
SELECT FIRST_NAME FROM PERSON;
```

The SELECT keyword tells the database that we want to retrieve data. It is followed by the name of the column – FIRST_NAME – that we want to get.

Then we use the FROM keyword to tell the database which table we want to get the data from, in this case, the PERSON table. Also, note that all SQL commands are terminated by a semi-colon.

One of the most common requirements we have with data is to filter it. Filtering means restricting the result set based on a specified condition.

For example, we might only want to select rows from the `PERSON`

```
SELECT * FROM PERSON WHERE FIRST_NAME = 'PHIL';
```

This query would return all columns in the `PERSON` table since we used an asterisk `*` in the `SELECT` clause instead of listing specific column names. Only rows in the `PERSON` table where the `FIRST_NAME` is set to "PHIL" would be retrieved.

Lastly, we'll talk about sorting. There are many times when we'd like to see our query results sorted in a particular order. We can use the `ORDER BY` clause for this:

```
SELECT *
FROM PERSON
ORDER BY LAST_NAME;
```

This will return all columns in the `PERSON` table sorted alphabetically by last name.

By default, the results will be sorted in ascending order, from A to Z. We can add the optional ASC or DESC keyword, to specify whether to sort in ascending or descending order:

```
SELECT *
FROM PERSON
ORDER BY LAST_NAME DESC;
```

13) Read About Web Frameworks and MVC

Oftentimes, we'll find ourselves writing code for very common types of applications. Web applications (or **web apps**) are applications that rely on the Internet in order to function. Webapps are some of the most commonly created types of software applications.

A web app is essentially a more functional and robust version of a website. Most web apps implement some backend code that resides on a web server and performs logic behind the scenes to support the application's functionality.

Common programming languages to use for a web app's backend code include Python, Java, and JavaScript, among others.

Some functionalities common to most web apps include:

- Providing a convenient way to dynamically alter content on web pages
- Performing secure user authentication via a login page
- Providing robust application security features
- Reading and writing data to a database

A web framework is a set of code libraries that contain the common functionalities that all web apps use out of the box. Web frameworks provide a system for developers to build their applications without having to worry about writing the code for many of the behind the scenes tasks common to all web apps.

For example, if we don't need to connect to a database in a particular web app, we can just ignore the database features and use the other features that we do need.

We still have the full ability to customize the web pages that make up our application, the user flow, and the business logic. You can think of a web framework as a programming tool suite that we can use to build web apps.

Each programming language we covered in this article has one or more popular web frameworks currently in use. This is great because it gives development teams the flexibility to use the framework of the language that they are the most proficient in.

Java has the **Spring Framework** that's made especially convenient via **Spring Boot**. Python has the **Django Framework**. JavaScript has the **Node.js** runtime environment with the multiple framework options including **Express.js** and **Meteor.js**. These frameworks are all free and open-source.

14) Play with Package Managers

The final topic that we'll cover in this guidebook is the **package manager**. Depending on the context, a **package** can either represent a standalone program that is ready to install on a computer or an external code library that we want to leverage in one of our software projects.

Since our applications often depend on these external code libraries,

dependencies of a system or software project. By "maintain" we mean installing, updating, listing, and uninstalling the dependencies as needed.

Depending on the context, the package managers we'll discuss can either be used to maintain the programs we have installed on our operating systems or to maintain the dependencies of a software project.

Mac OS X: Homebrew

Homebrew is the most popular package manager for the Mac OS X operating system. It offers a convenient way to install, update, track, list, and uninstall packages and applications on your Mac.

Many applications that can be installed via downloaded .dmg files can also be downloaded and installed using Homebrew.

Here is an example of installing the `wget` package via Homebrew:

```
brew install wget
```

Linux: Apt and Yum

Since Linux was built around the Command Line, it's no surprise that package managers are the default way to install programs.

Most mainstream flavors of Linux ship with a built-in package

RedHat Linux distribution.

Here is an example of installing Vim using APT:

```
sudo apt-get install vim
```

And using Yum:

```
sudo yum install vim
```

JavaScript: Node Package Manager (NPM)

Now that we have seen how some OS-level package managers work, let's take a look at some programming language-specific package managers. These can help us manage the software libraries that many of our coding projects depend on. **Node Package Manager (NPM)** is installed by default with Node.js.

One difference between NPM and the previous package managers we have seen is that NPM can be run in **local** or **global** mode. Local mode is used to install a package only within a particular project/directory we are working on, while global mode is used to install the package on the system.

By default, packages are installed locally, but you can use the `-g` flag to install a package globally:

```
npm install request -g
```

Python: Pip

Python also has a package manager called **Pip**. Pip may already be installed on your system as it comes prepackaged with recent versions of Python. Pip allows us to easily install packages from the **Python Package Index** using the `pip install <package-name>` command:

```
pip install requests
```

Java: Apache Maven

Apache Maven (usually referred to as simply **Maven**) is a free and open-source tool suite that includes dependency management.

Maven is mostly used for Java projects although it does support other languages as well. Maven usage is a bit more complicated and it can do a lot of things, so we won't get into the weeds here.

Summary

In this article, I introduced a set of essential coding concepts and tools with the intention of presenting a bird's eye view of software development that I wish I had when I started learning to code.

I covered topics including the Internet, several programming

Next Steps

If you enjoyed this article, I wrote a book called the Coding Essentials Guidebook for Developers which has 14 chapters, each covering one of the topics discussed in this post.

In the book I go into even more depth on these 14 subjects, so that might be a good resource for you to check out if you got value out of this article.

After reading this, you may feel drawn to a particular language, tool, or concept. If this is the case I encourage you to dive deeper into that area to further your learning.

If you have any questions, suggestions, or concerns about this book, I would love to hear from you at jacob@initialcommit.io.

Jacob Stopak

Read more posts by this author.

If you read this far, tweet to the author to show them you care.

Tweet a thanks

Learn to code for free. freeCodeCamp's open source curriculum has helped more than 40,000 people get jobs as developers.

Get started

freeCodeCamp is a donor-supported tax-exempt 501(c)(3) nonprofit organization (United States Federal Tax Identification Number: 82-0779546)

Our mission: to help people learn to code for free. We accomplish this by creating thousands of videos, articles, and interactive coding lessons - all freely available to the public. We also have thousands of freeCodeCamp study groups around the world.

Donations to freeCodeCamp go toward our education initiatives, and help pay for servers, services, and staff.

You can make a tax-deductible donation here.

Trending Guides

Zoom Screen Sharing

Decimal Place Value

How to Get Into BIOS

String to Int in C++

What is msmpeng.exe

Facetime Not Working

Desktop Icons Missing

C++ Vector

What is CPU

IPV4 vs IPV6

What is IPTV

HTML Font Size

Change Mouse DPI

How to Make a GIF

How to Open .dat Files

Record Calls on iPhone

Ascending vs Descending

HTML Email Link Tutorial

Python List Comprehension

Password Protect Zip File

Restore Deleted Word File

Software Engineering Guide

How to Find Your IP Address

How to Find iPhone Download

Our Nonprofit