**Use Cases:**

Machine code is natively understood by hardware at the most basic level and is the building block for all programming. No matter what the language, if broken down into it’s most basic parts, is 0’s and 1’s. Assembly language abstracts the machine code into basic functions like add, load, move, stop, branch, etc. This makes coding loops, conditional logic, and algorithms much easier, and also makes reading the code much easier. As programs and codebases grow in size, utilize more complex data structures, code that is easier to construct, and understand is a big help. High-level programming languages like Java, Python, Go abstract commands at an even higher level to language that is composed of code that resembles written language. What takes hundreds of lines in assembly can be done in one line of Python. You can create complex functions and extensive libraries for anything from math to naturalization to button creation.

One would think that high-level languages eliminate the need for knowledge of machine code and assembly, but that’s just not the case. Analysis of how functions and data structures process instructions involve looking at the time and space complexity of code, put another way how quickly and how much memory is used by a program to work through all instructions. Doing this on a deeper level involves understanding how the underlying assembly or machine code moves and stores data.

In some ways things change, in other ways they stay the same.

**Performance Trade-offs:**

The more abstraction, the more layers you need to pass through to get to machine code, the more time it takes to process instructions. Machine code is directly accessible by hardware, and doesn’t need a compiler, the instructions are literally just 1 bit in size, and is extremely fast. Assembly is still fast but has to convert instructions into machine code. High level languages need compilers and offer ease of use and readability, but at the sacrifice of speed and direct access to hardware.