Difference between High Level and Low level languages

High-Level Language

A high-level language is a programmer-friendly language that is designed to be easy to read, write, and understand. It is less memory efficient compared to low-level languages but simplifies debugging and maintenance. High-level languages are portable and can run on any platform, making them widely used in modern programming. They require a compiler or interpreter for translation into machine code. Due to their simplicity and ease of use, they remain the most common choice for software development today.

Low-Level Language

A low-level language is a machine-friendly language that is closely related to computer hardware. It is highly memory efficient but difficult to understand, making debugging and maintenance more complex. Unlike high-level languages, low-level languages are machine-dependent and non-portable, meaning they can only run on specific hardware. They require an assembler for translation rather than a compiler or interpreter. Due to their complexity, low-level languages are not commonly used in modern programming, except in specialized fields like embedded systems and operating system development.

Interpreted vs Compiled

Compiled Languages

Compiled languages are converted directly into machine code that the processor can execute. As a result, they tend to be faster and more efficient to execute than interpreted languages. They also give the developer more control over hardware aspects, like memory management and CPU usage.

Compiled languages need a “build” step – they need to be manually compiled first. You need to “rebuild” the program every time you need to make a change. In our hummus example, the entire translation is written before it gets to you. If the original author decides that he wants to use a different kind of olive oil, the entire recipe would need to be translated again and resent to you.

Examples of pure compiled languages are C, C++, Erlang, Haskell, Rust, and Go.

Interpreted Languages

Interpreters run through a program line by line and execute each command. Here, if the author decides he wants to use a different kind of olive oil, he could scratch the old one out and add the new one. Your translator friend can then convey that change to you as it happens.

Interpreted languages were once significantly slower than compiled languages. But, with the development of just-in-time compilation, that gap is shrinking.

Examples of common interpreted languages are PHP, Ruby, Python, and JavaScript.

Differences Between Scripting and Programming Languages

Scripting Language

A scripting language is a type of programming language designed to execute tasks within a specific runtime environment. It is commonly used for creating dynamic web applications and relies on different libraries to enhance functionality. Most scripting languages are interpreted rather than compiled, making them easier to learn and implement. They do not generate .exe files or create binary files, as they are run inside another program and require a host environment. Additionally, they are highly portable across various operating systems and require fewer lines of code, making them a beginner-friendly option. Examples of scripting languages include Bash, Ruby, Python, and JavaScript.

Programming Language

A programming language is a formal system used by humans to communicate with computers and write software applications. Programming languages are categorized into low-level, middle-level, and high-level languages. Unlike scripting languages, most programming languages are compiled, generating .exe and binary files, allowing them to run independently without a host. They are typically more code-intensive and require a significant amount of time to master, making them more challenging for beginners. Despite having a higher maintenance cost, they offer high-speed execution and greater flexibility in application development. Examples of programming languages include C++, Java, and PHP.

Open source vs not open source

Open Source

'open source refers to a computer program in which the source code is available to the general public for use and/or modification from its original design. Open-source code is meant to be a collaborative effort, where programmers improve upon the source code and share the changes within the community. Typically this is not the case, and code is merely released to the public under some license. Others can then download, modify, and publish their version (fork) back to the community. Today you find more forked versions, than teams with large membership.'

And therein lies the problem of open source. It's a nice model, but in practice, someone has to take a lead or it's just an unsupportable mess.

Not open source

also known as proprietary software, refers to software whose source code is not publicly available for use, modification, or distribution. Instead, it is controlled by an individual, organization, or company that holds exclusive rights. Users are typically required to purchase a license to use the software under specific terms and conditions. Unlike open-source software, proprietary software does not allow public collaboration or modifications. Updates and improvements are managed solely by the company or developers who own the software. While this model ensures stability, security, and official support, it limits customization and community-driven enhancements. Examples of proprietary software include Microsoft Windows, Adobe Photoshop, and macOS.

Difference Between Supporting OOP and Not Supporting OOP

Languages that support Object-Oriented Programming (OOP)

allow the use of objects, classes, inheritance, polymorphism, and encapsulation, making them highly modular and maintainable. They structure code into reusable objects and classes, improving organization and scalability. OOP languages use encapsulation to bundle data and methods together, promoting code reusability through inheritance and polymorphism. As a result, maintaining and scaling applications becomes easier. Examples of OOP-supported languages include Java, C++, Python, C#, and Ruby.

On the other hand, languages that do not support OOP

lack built-in object-oriented principles. Instead, they rely on functions and procedures to structure code, often using global variables or explicit data passing between functions. While code reusability is possible through functions, it lacks the structured approach of OOP, making maintenance and scalability more challenging. Examples of such languages include C, Assembly, Fortran, and Bash.