Types of programming languages:

LOW LEVEL:

Low level language abbreviated as LLL, are languages close to the machine level instruction set. They provide less or no abstraction from the hardware. A low-level programming language interacts directly with the registers and memory. Since, instructions written in low level languages are machine dependent. Programs developed using low level languages are machine dependent and are not portable.

Low level language does not require any compiler or interpreter to translate the source to machine code. An assembler may translate the source code written in low level language to machine code.

Programs written in low level languages are fast and memory efficient. However, it is nightmare for programmers to write, debug and maintain low-level programs. They are mostly used to develop operating systems, device drivers, databases and applications that requires direct hardware access.

Low level languages are further classified in two more categories – Machine language and assembly language.

HIGH LEVEL:

A high-level language is a programming language that is designed to make it easier for humans to understand and write. It is closer to natural language and uses commands and instructions that are more abstract and less dependent on the specific details of the underlying computer hardware.

High-level languages are used in programming because they allow programmers to write code that is more readable, maintainable, and portable. These languages provide a higher level of abstraction, allowing developers to focus on solving problems rather than dealing with low-level details of the computer system.

Some examples of high-level languages include Python, Java, C++, JavaScript, Ruby, and C#. These languages have syntax and structures that are closer to human-readable language, making them easier for programmers to understand and use.

High-level languages differ from low-level languages in their level of abstraction. While high-level languages provide a higher level of abstraction and are closer to human-readable language, low-level languages are closer to the machine code and are specific to the underlying hardware architecture.

Using high-level languages offers several advantages. They are easier to learn and understand, reducing the learning curve for new programmers. High-level languages also provide built-in functions and libraries, increasing productivity. Additionally, these languages offer portability, allowing code to be run on different platforms with minimal modifications.

High-level languages can be either compiled or interpreted. Compiled languages, like C++ or Java, are translated into machine code before execution. Interpreted languages, like Python or JavaScript, are executed line by line through an interpreter without the need for compilation.

High-level languages typically handle memory management automatically through features like garbage collection. This means you don't have to manually allocate and deallocate memory as you would in a low-level language. The high-level language's runtime environment takes care of managing memory for you, reducing the chances of memory leaks and other memory-related errors.

INTERPRETED:

An interpreted language is a programming language that executes instructions directly, without the need for a separate compilation step. The instructions are translated and executed line by line, making it easier and quicker to develop and test code.

In an interpreted language, the code is executed line by line, while in a compiled language, the entire code is converted into machine language before execution. This means that interpreted languages offer more flexibility in terms of modifying and testing code on the fly.

Some popular interpreted languages include Python, JavaScript, Ruby, Perl, and PHP. These languages are widely used in web development, scripting, and automation tasks due to their ease of use and quick development process.

One advantage is that you can write code and see the results immediately, making it great for prototyping and iterative development. Interpreted languages also tend to have simpler syntax and are often easier to learn compared to compiled languages.

Yes, most interpreted languages are designed to be platform-independent, meaning they can run on different operating systems such as Windows and Linux. This makes it easier to develop cross-platform applications that can be used by a wider audience.

While most interpreted languages share common characteristics, there are differences in syntax, available libraries, and use cases. For example, Python is known for its readability and versatility, while JavaScript is widely used for web development.

Absolutely, it's common to use both interpreted and compiled languages in a project. For instance, you might use an interpreted language like Python for scripting tasks and a compiled language like C++ for performance-critical parts of your application.

COMPLIED:

A compiled language is a programming language where the source code is translated into machine code and the machine code is stored in a separate file. A compiled language tends to give the developer more control over hardware aspects like memory management and CPU usage. However, the compiled code is hardware dependent.

In a compiled language, the program needs to be rebuilt whenever you make a change.

Some examples of compiled languages are C, C++, and Haskell.

Advantages

All compiled programs are faster as compared to any interpreted code. This is because the code does not need to be compiled while the program is running.

A compiler gives a list of all the compilation errors during compilation. A programmer can fix the errors and execute the code again.

Disadvantages

The entire code needs to be compiled before testing. This increases the overall run time of execution.

The machine code depends on the platform it is running.

PROGRAMMING:

A programming language is a system of notation for writing computer programs.[1] Programming languages are described in terms of their syntax (form) and semantics (meaning), usually defined by a formal language. Languages usually provide features such as a type system, variables, and mechanisms for error handling. An implementation of a programming language is required in order to execute programs, namely an interpreter or a compiler. An interpreter directly executes the source code, while a compiler produces an executable program.

Computer architecture has strongly influenced the design of programming languages, with the most common type (imperative languages—which implement operations in a specified order) developed to perform well on the popular von Neumann architecture. While early programming languages were closely tied to the hardware, over time they have developed more abstraction to hide implementation details for greater simplicity.

Thousands of programming languages—often classified as imperative, functional, logic, or object-oriented—have been developed for a wide variety of uses. Many aspects of programming language design involve tradeoffs—for example, exception handling simplifies error handling, but at a performance cost. Programming language theory is the subfield of computer science that studies the design, implementation, analysis, characterization, and classification of programming languages.

SCRIPTED:

Scripting languages can be an effective tool for programmers, engineers, and other developers to create systems and software. Learning a scripting language is an excellent introduction to coding and programming. They are relatively easy to learn and can be an effective jumping-off point to pursue your hobbies or career interests further.

Learning a scripting language may open up new personal and professional opportunities. Discover the differences between server-side scripting languages and the pros of learning them to decide if this is a good path for you.

Scripting languages can be an effective tool for programmers, engineers, and other developers to create systems and software. Learning a scripting language is an excellent introduction to coding and programming. They are relatively easy to learn and can be an effective jumping-off point to pursue your hobbies or career interests further.

Scripting vs programming

People often refer to scripting languages and programming languages interchangeably. However, they are not the same. All scripting languages are programming languages, but not all programming languages are scripting languages. Programming languages are a way for coders to communicate with computers using compiled languages—source code compiled to convert into machine code.

Difference between scripting and programming

Scripting languages are a type of programming language that is interpreted rather than requiring compilation. These are languages designed for specific runtime environments to provide additional functions, integrate complex systems, and communicate with other programming languages. One example is JavaScript, which you can use to display messages, perform calculations, and integrate elements of user interfaces for web pages.

Scripting language vs. markup language

Markup languages are not programming or scripting languages because you don’t use them to perform actions. Instead, you use them to structure and present data.

As you research possible scripting languages to learn, you’ll see that the different types fall into two main categories: server-side scripting language and client-side scripting language. The main difference is that server-side scripting gets processed through a server, and client-side scripting runs scripts on client machines using browsers without interacting with the server.

Examples of popular scripting languages: server-side

Server-side scripting works in the back end—what happens behind the scenes that website users don't see but makes it possible for them to use the site. You can customize web pages and create dynamic websites with these scripting languages. Common server-side scripting languages include:

PHP: Popular for use on the web

ASP.net: A web application language that Microsoft developed

Node.js: A scripting language you can use on multiple platforms including Unix, Windows, Mac, and Linux

Java: A scripting language used in just about everything, including consumer Bluetooth devices and applications used by NASA.

Ruby: A dynamic scripting language that focuses on simplicity

Python: A popular language that uses shorter code, making it easier for beginners to learn

OPEN SOURCE & NOT OPEN SOURCE:

Difference Between Open Source and Proprietary (Closed Source) Programming Languages

The key difference between Open Source and Proprietary (Closed Source) programming languages lies in access to the source code, modification rights, and licensing.

1. Open Source Programming Languages

Source Code is Publicly Available: Anyone can view, modify, and distribute the code.

Community-Driven Development: Many open-source languages evolve through contributions from developers worldwide.

Usually Free: Most open-source languages are free to use and distribute.

Examples: Python, JavaScript, PHP, Ruby, Go.

2. Proprietary (Closed Source) Programming Languages

Source Code is Restricted: Only the company or organization that owns the language can access and modify the code.

Limited Development Access: Changes and updates are controlled by the company, making them less flexible.

Often Requires Licensing Fees: Some proprietary languages require paid licenses to use.

Examples: MATLAB, Visual Basic, Swift (originally closed-source but later partially open-source).

SUPPORT OOP & NOT SUPPORT OOP:

Programming languages can be categorized based on whether they support Object-Oriented Programming (OOP) or not.

1. Languages that Support OOP

These languages support the core principles of Object-Oriented Programming: Encapsulation, Inheritance, Polymorphism, and Abstraction.

Examples of OOP-Supporting Languages:

Fully OOP Languages: Java, C#, Smalltalk, Ruby

Multi-Paradigm Languages (Support OOP & Other Paradigms): Python, C++, JavaScript, Swift

2. Languages that Do Not Support OOP

These languages follow procedural, functional, or other paradigms instead of OOP. They may not have built-in support for classes and objects.

Examples of Non-OOP Languages:

Procedural Languages: C, Pascal, COBOL

Functional Languages: Haskell, Lisp, Prolog

Scripting Languages (without OOP focus): Bash, AWK

Some languages, like C++ and Python, support both OOP and other paradigms, making them multi-paradigm.

__ Resources:

- CodeForWin
- Lenovo
- how.dev
- WIKIPEDIA
- Coursera