What does an interpreter do?



www.yzokul.com.tr

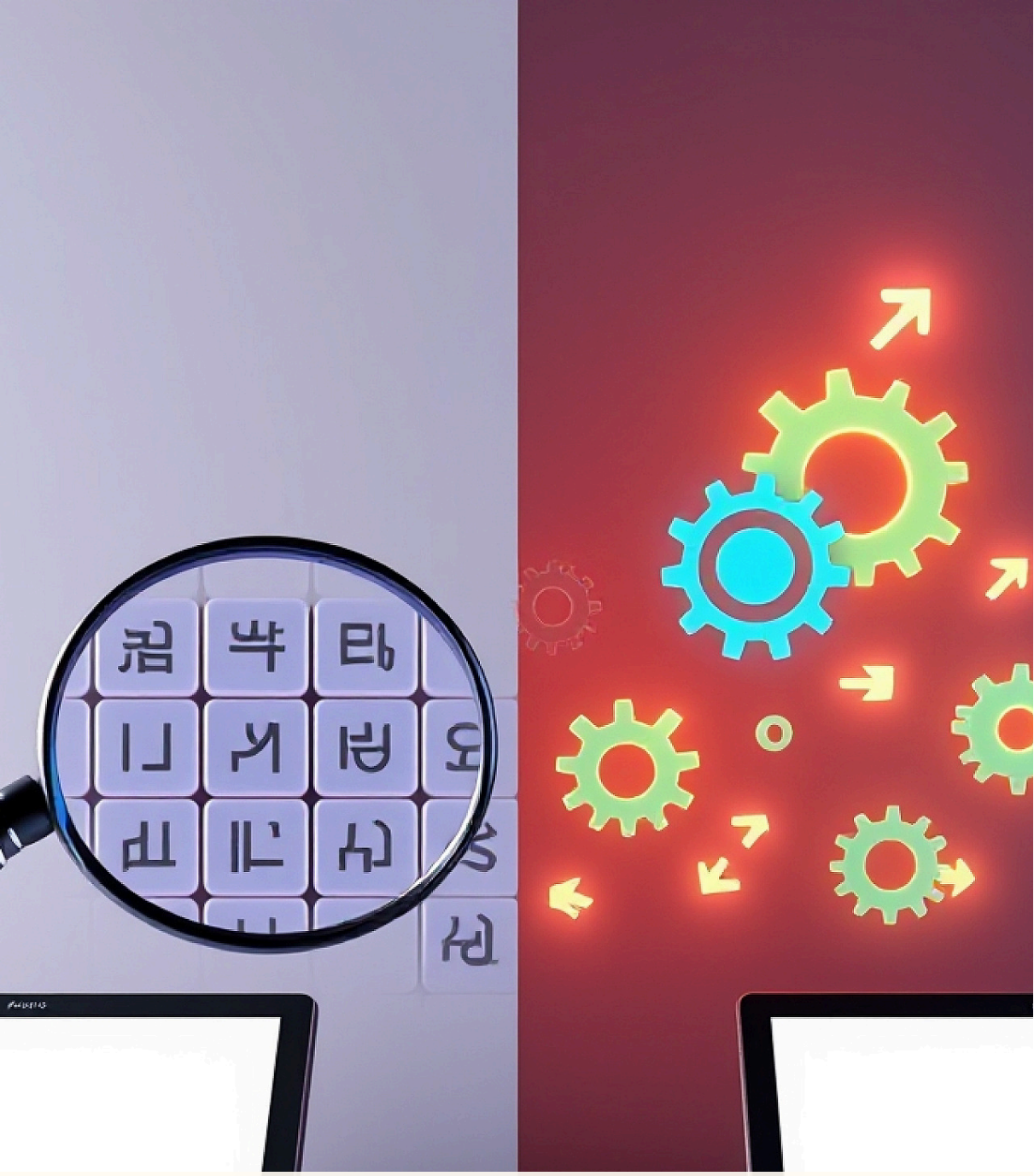
What does an interpreter do?



What does an interpreter do? An interpreter is a system that carries out a program's operations while the program runs. It takes the instructions and performs them. For example, when you run a Python script, the interpreter executes each operation. The short answer: an interpreter is a system that carries out a program's operations during execution.



A program is just a list of instructions, like a recipe. Each instruction tells the computer to do something, such as add two numbers or show text. Someone or something must actually follow those instructions to make anything happen. For example, a recipe for cake tells you steps, but you must do them. The interpreter is what does the steps for a program.



Reading instructions is not the same as carrying them out. You can read a recipe without cooking. An interpreter does the carrying out, not just the reading. For example, reading a program's code is like reading a recipe; the interpreter is like the cook who actually makes the dish. It performs the actions the program describes.



Execution means performing each operation the program describes, one after another, while the program is running. The interpreter does this execution. For example, if the program says to print a word, the interpreter makes the word appear on the screen. If the program says to add numbers, the interpreter does the addition. Execution happens during the run, not before.



Picture the interpreter as a worker who takes the program's instructions and performs them. The worker acts during the run, not before it. For example, think of a builder following a blueprint: the builder does the work while building. The interpreter is like that builder, doing each operation as the program runs. It is not just planning; it is doing.



One kind of interpreter takes the original source code the programmer wrote. Source code is the text of the program in a programming language. The interpreter works directly from that text. For example, a simple interpreter for a language like Python reads the source code and executes it. It does not need a separate conversion step first.



Another kind of interpreter first receives code that was already converted into a simpler intermediate form. Intermediate code is a version of the program that is easier for the interpreter to execute. The interpreter then executes that intermediate code. For example, some systems convert source code to bytecode, which is a kind of intermediate code, and then interpret the bytecode.



A common belief is that interpreters always read one line at a time. That description is not true for all interpreters. Some interpreters process larger chunks or the whole program at once. For example, an interpreter might read an entire function before executing it. So, do not assume that line-by-line reading is the definition of an interpreter.



People believe the line-by-line idea because some simple interpreters do work that way. For example, a basic interpreter for a teaching language might read and execute one line at a time. But that is only one possible design, not the definition. Many modern interpreters work differently. The key is that they carry out operations during execution, however they read the code.



Some systems first convert the code into intermediate code and then execute it. The conversion happens before the operations are carried out. For example, a system might take source code and translate it into a simpler form, then an interpreter executes that form. This two-step approach can make execution faster or more portable. The interpreter still does the execution.



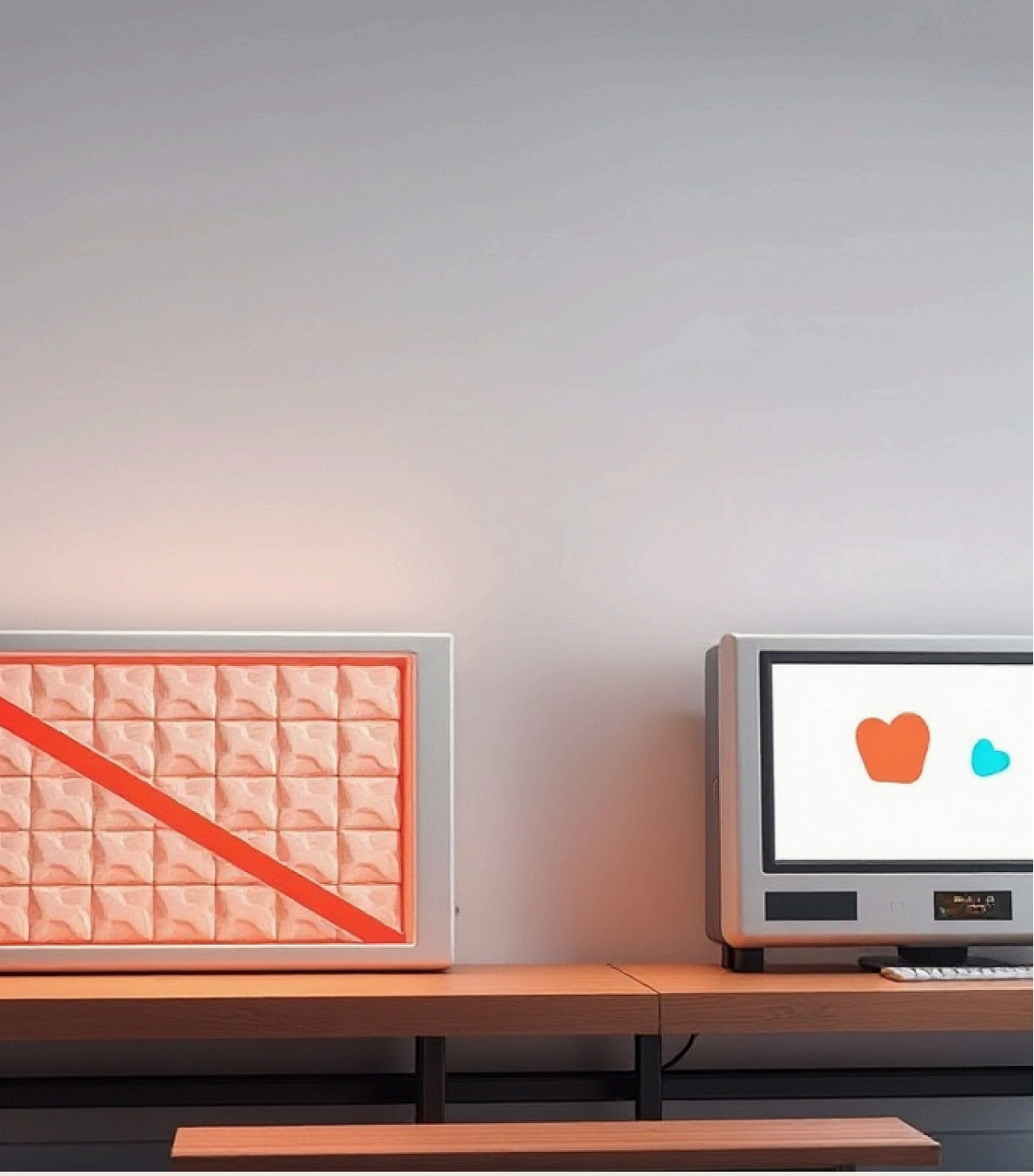
Compilation means turning code into another form ahead of time. This is different from carrying out the operations right away. For example, a compiler might turn source code into machine code that the computer can run directly later. The compiler does not execute the program; it prepares it. An interpreter, in contrast, executes the program's operations.



Interpretation and compilation can be used together in modern language implementations. A system may convert source code to intermediate code (a form of compilation) and then interpret that intermediate code. For example, many language runtimes compile to bytecode and then interpret the bytecode. So, the two jobs can be combined in one system.



However the input is prepared, the interpreter's job stays the same: carry out the program's operations during execution. Whether it works from source code or intermediate code, it performs the actions. For example, whether you give it original text or a converted form, the interpreter executes the steps. The goal is always to run the program.



The biggest misunderstanding is thinking every interpreter reads line by line. The correct view is that it depends on the implementation. Some interpreters do read line by line, but many do not. For example, an interpreter might read a whole expression or block before executing. So, avoid saying that interpreters always work line by line.



A second misunderstanding is thinking interpretation and compilation never mix. In modern implementations, they often work together. For example, a system might compile source code to intermediate code and then interpret it. So, interpretation and compilation are not opposites; they can be parts of the same process. Many languages use both.



An interpreter carries out a program's operations during execution, from source or intermediate code, and is not always line by line. Remember that interpretation and compilation can be combined. For example, Python uses both compilation to bytecode and interpretation of that bytecode. The core idea: an interpreter executes the program's instructions while it runs.