1. Write a simple algorithm for finding the maximum of three numbers using pseudo code.

* Start
* Enter three numbers into a, b, c
* If a>b and a>c , then print a is greatest
* If not
* Else If b>a and b>c , then print b is greatest
* If not
* Print c is greatest
* End

1. Compare and contrast two different programming languages, highlighting their strengths and weaknesses.

| **Python** | **Java** |
| --- | --- |
| Dynamically typed (type checking at runtime) | Statically typed (type checking at compile time) |
| Readable and concise | More verbose, C+±like syntax |
| Interpreted (instant conversion from human-readable to machine-readable code) | Compiled and interpreted (source code compiled to bytecode, runs on the JVM) |
| Cross-platform (compatible with Windows, macOS, Unix, Linux) | Write once, run anywhere (JVM-based, platform-independent) |
| Scientific computing, web development, data science | Enterprise applications, large-scale systems |
| Active community, extensive third-party libraries | Mature ecosystem, widely used in enterprise |
| Beginner-friendly, easy to read and write | Learning curve due to strict rules and OOP concepts |
| Slower execution due to interpretation | Generally faster due to bytecode compilation |
| Moderate | Can be memory-intensive due to JVM overhead |
| Limited by Global Interpreter Lock (GIL) | Supports multithreading, but GIL limitation |

1. Explain the compilation process and how it differs from interpretation.

**Compilation Process:**

1. **Source Code**: It all starts with the programmer writing human-readable code in a high-level programming language (like Python, C++, Java, or Rust). This code is often referred to as the “source code.”
2. **Compiler**: The source code is then fed into a **compiler**. The compiler is a specialized tool that translates the high-level source code into low-level machine code (specific to the target architecture). This translation involves several steps:
   * **Lexical Analysis**: The compiler breaks down the source code into smaller units called tokens (such as keywords, identifiers, operators, etc.).
   * **Syntax Analysis (Parsing)**: The compiler checks whether the tokens form valid expressions according to the language’s grammar rules.
   * **Semantic Analysis**: The compiler verifies the correctness of the program’s logic and type consistency.
   * **Intermediate Code Generation**: Some compilers generate an intermediate representation (like an abstract syntax tree or bytecode) before producing machine code.
   * **Code Optimization**: The compiler may perform various optimizations to improve the efficiency of the resulting machine code.
   * **Code Generation**: Finally, the compiler produces machine code (usually in the form of an executable binary or object files).
3. **Object Files and Linking**: If the program consists of multiple source files, the compiler generates separate object files for each. These object files contain the compiled machine code for individual modules. The linker then combines these object files into a single executable (or a library) by resolving references between them.
4. **Execution**: The user can now run the compiled executable directly. The compiled code interacts directly with the hardware, making it efficient in terms of execution speed.

**Interpretation Process:**

1. **Source Code**: Similar to compilation, the programmer writes source code in a high-level language.
2. **Interpreter**: Instead of using a compiler, an **interpreter** processes the source code line by line during runtime. Here’s how it works:
   1. **Lexical Analysis**: The interpreter tokenizes the source code.
   2. **Syntax Analysis (Parsing)**: It checks the syntax and structure of the code.
   3. **Execution**: The interpreter executes the code directly, interpreting each statement and performing the associated actions.
   4. **Dynamic Typing**: Interpreted languages often allow dynamic typing, meaning variable types can change during runtime.
   5. **Slower Execution**: Since interpretation happens line by line, interpreted programs are generally slower than compiled ones.
3. **No Intermediate Files**: Unlike compilation, there’s no intermediate machine code or object files. The interpreter reads the source code directly and executes it.

**Key Differences:**

* **Compilation**:
  + Produces an independent executable file.
  + Requires a separate compilation step before execution.
  + Generally faster execution.
  + Examples: C, C++, Rust.
* **Interpretation**:
  + Executes the source code directly.
  + No separate compilation step.
  + Typically slower execution.
  + Examples: Python, JavaScript, Ruby.

1. Create a flowchart for a program that calculates the factorial of a given number.
   1. Start
   2. Input : enter a positive integer n
   3. Initialize result =1
   4. Set counter i=1
   5. If i<=n ?
   6. If yes -> multiply result by i
   7. And increment i
   8. Now go to step 6
   9. If no -> display result
   10. End

5. Write a function in your preferred programming language to calculate the area of a rectangle

def rectangle\_area(length, width):

area = length \* width

return area

length\_input = float(input("Enter the length of the rectangle: "))

width\_input = float(input("Enter the width of the rectangle: "))

result = rectangle\_area(length\_input, width\_input)

print(f"The area of the rectangle is {result:.2f} square units.")