NSCT Programming (C++/Java/Python) — Practice MCQs

nsctexam.com · 1630 original questions · Answer key at the back. Independent; not affiliated with HEC or VU. Not real NSCT questions.

Programming Fundamentals MCQs for NSCT

1. Which translator converts an entire source program into machine code before the program is run? Understanding
   A. Interpreter
   B. Debugger
   C. Compiler
   D. Text editor

2. How does a pure interpreter run a program? Understanding
   A. It produces a standalone executable file, then runs it
   B. It sends the source code directly to the CPU without translation
   C. It checks the syntax only and never executes the program
   D. It translates and executes the source one statement at a time

3. What does the Java compiler (javac) produce from a .java source file? Understanding
   A. Native machine code for the current CPU in an .exe file
   B. Python bytecode in a .pyc file
   C. Assembly language that the programmer must assemble
   D. Bytecode in a .class file, run by the Java Virtual Machine

4. Which tool combines separately compiled object files and library code into one executable program? Understanding
   A. Linker
   B. Loader
   C. Preprocessor
   D. Assembler

5. In C++, which stage handles lines that begin with #, such as #include <iostream>? Understanding
   A. The linker, after compilation
   B. The loader, when the program starts
   C. The preprocessor, before compilation proper
   D. The CPU, while the program is running

6. Which line is a single-line comment in Python? Understanding
   A. // total marks
   B. # total marks
   C. /* total marks */
   D. <!-- total marks -->

7. A program breaks a grammar rule of its language, such as a missing closing bracket. What kind of error is this? Understanding
   A. Logic error
   B. Runtime error
   C. Syntax error
   D. Linker error

8. A program compiles and runs without crashing but prints the wrong total. What type of error is this? Understanding
    A. Syntax error
    B. Compile-time type error
    C. Logic error
    D. Linker error

9. Which method header does the JVM look for to start a standalone Java application? Understanding
    A. public static void main(String[] args)
    B. public void start()
    C. static int main()
    D. void run(String args)

10. Which statement best describes how the standard Python implementation (CPython) runs a .py file? Understanding
    A. It compiles the source ahead of time into a native executable, like a C++ compiler
    B. It hands the .py text straight to the CPU, which understands Python
    C. It compiles the source to bytecode, which the Python virtual machine then interprets
    D. It requires a separate linker step before every run

11. What is Just-In-Time (JIT) compilation? Understanding
    A. Compiling the whole program to machine code before it is shipped to users
    B. Compiling bytecode or frequently run code into native machine code while the program is running
    C. Checking syntax as the programmer types in the editor
    D. Linking library code only at the moment the program is installed

12. Which of these is a valid identifier (variable name) in C++, Java and Python alike? Understanding
    A. 2total
    B. total-marks
    C. class
    D. _total2

13. What does an assembler translate? Understanding
    A. High-level C++ code into assembly language
    B. Assembly language mnemonics into machine code
    C. Bytecode into Java source code
    D. Machine code back into readable source code

14. In standard C++, what happens if control reaches the end of main() without a return statement? Understanding
    A. It behaves as if main returned 0
    B. The program fails to compile
    C. The return value is undefined garbage
    D. The program throws an exception at exit

15. In a standard flowchart, which symbol represents a decision such as 'Is marks >= 50?' Understanding
    A. Rectangle
    B. Diamond
    C. Parallelogram
    D. Oval

16. In a Java program, which of these problems is reported by the compiler rather than at run time? Understanding

    A. Assigning a string literal to an int variable
    B. Dividing an int by a variable that holds zero
    C. Reading index 5 of a 3-element array
    D. Using < instead of <= in a loop condition

17. What does this Python program print? Application

```
def greet():
    print("Hi")

print("Start")
greet()
```

    A. Hi, then Start on the next line
    B. Only Start
    C. Only Hi
    D. Start, then Hi on the next line

18. What happens when this Python script is run? Application

```
print(square(3))

def square(n):
    return n * n
```

    A. NameError, because square is called before the def line has run
    B. It prints 9
    C. SyntaxError, so nothing runs
    D. It prints None

19. What does this Python program print? Application

```
x = 2
if x > 3:
    print("A")
print("B")
```

    A. A then B
    B. A
    C. B
    D. Nothing; it raises IndentationError

20. Why does this Java fragment fail to compile? Application

```
int count = 10
System.out.println(count);
```

    A. count must be declared as Integer, not int
    B. The first statement is missing its semicolon
    C. println cannot print an int
    D. Variables must be declared after they are used

21. What happens when you compile this C++ code? Application

```
#include <iostream>
int main() {
    int Total = 5;
    std::cout << total;
    return 0;
}
```

A. It prints 5
B. It prints 0, the default value
C. It compiles but crashes at run time
D. Compile error: total was not declared, because C++ is case-sensitive

22. What does this JavaScript code print? Application

```
console.log(sq(4));

function sq(n) {
  return n * n;
}
```

A. ReferenceError: sq is not defined
B. 16
C. undefined
D. NaN

23. What is the output of this Python script? Application

```
print("one")
x = 10 / 0
print("two")
```

A. Nothing is printed; the error is found before running
B. one and two are both printed
C. one is printed, then a ZeroDivisionError stops the program
D. two is printed, then the error

24. What does this Python line print? Application

```
a = 3; b = 4; print(a * b)
```

A. 12
B. SyntaxError, because Python does not use semicolons
C. 7
D. 34

25. This code is saved in a file named Main.java. What happens when you run javac Main.java? Application

```
public class Test {
    public static void main(String[] args) {
        System.out.println("Hi");
    }
}
```

A. It compiles and prints Hi
B. Compile error: public class Test must be declared in a file named Test.java
C. It compiles, but the JVM cannot find main at run time
D. It compiles to Main.class and prints Hi

26. What does this Python program print? Application

```
# print("A")
print("B")  # print("C")
"""print("D")"""
```

A. A B C D
B. B C
C. B
D. B D

27. What happens when this Java fragment is compiled? Application

```
int x = 5;
x = "five";
```

A. It compiles and x holds the text five
B. It compiles but throws an exception at run time
C. It compiles and x becomes 0
D. Compile error: incompatible types, String cannot be converted to int

28. What does this Python program print? Application

```
def a():
    return b() + 1

def b():
    return 10

print(a())
```

A. NameError, because b is defined after a
B. 11
C. 10
D. 1

29. What does this Python program print, in order? Application

```
def f():
    print("f")
    return 1

def g(x):
    print("g")
    return x + 1

print(g(f()))
```

A. g, then f, then 2
B. f, then g, then 2
C. f, then 2, then g
D. 2, then f, then g

30. This C++ program has no #include line. What happens when it is compiled? Application

```
int main() {
    std::cout << "Hello";
    return 0;
}
```

A. It prints Hello
B. Linker error only
C. It compiles but prints nothing
D. Compile error: std::cout is not declared

31. Line 3 is missing its colon. What happens when this Python script is run? Analysis

```
print("Ready")
x = 4
if x > 2
    print("big")
print("done")
```

A. Nothing is printed; Python reports a SyntaxError before any line runs
B. Ready is printed, then a SyntaxError is reported
C. Ready and done are printed; the bad line is skipped
D. Ready is printed, then an IndentationError

32. The file shapes.py contains this code. Another file runs import shapes. What is printed by the import? Analysis

```
def main():
    print("run")

if __name__ == "__main__":
    main()

print("end")
```

A. run, then end
B. run
C. end
D. Nothing

33. Why does this Java method fail to compile? Analysis

```
static void show() {
    System.out.println("A");
    return;
    System.out.println("B");
}
```

A. A void method is not allowed to contain return
B. System.out.println cannot appear twice in one method
C. The statement after return can never run, and Java reports an unreachable statement
D. It compiles; Java simply ignores the second print

34. What does this Python program print? Analysis

```
def f():
    return 1
    print("x")

print(f())
```

A. 1, then x
B. SyntaxError: unreachable code
C. None
D. 1

35. This function should return the mean of a list. average([80, 90]) returns 84.0. What kind of error is it, and what causes it? Analysis

```
def average(marks):
    total = 0
    for m in marks:
        total += m
    return total / len(marks) - 1

print(average([80, 90]))
```

A. A logic error: division happens before subtraction, so 1 is taken off the mean
B. A syntax error in the return line
C. A runtime error caused by dividing integers
D. A compile-time type error, because total is an int

36. A corrected average function passes tests with [70, 80] but crashes in use. Which input and which kind of error explain this? Analysis

```
def average(marks):
    return sum(marks) / len(marks)

print(average([]))
```

A. An empty list, causing a runtime ZeroDivisionError
B. A list of floats, causing a syntax error
C. A list with one item, causing a logic error
D. A very long list, causing a compile error

37. A team ships a desktop tool to customers. They want fast start-up, no need for customers to install a language runtime, and they do not want to ship readable source. Which approach fits best? Analysis
    A. Ship the .py source and ask customers to install Python
    B. Ship the source code and ask customers to compile it themselves
    C. Ship Java .class files and rely on whatever JVM the customer has
    D. Compile ahead of time to a native executable for each target platform

38. The same compiled Java .class file runs on both Windows and Linux, but a C++ program compiled to a Windows .exe will not run on Linux. Why? Analysis
    A. Java source is interpreted line by line, so it is never compiled
    B. The .class file holds JVM bytecode, while the .exe holds native code and a format tied to one operating system
    C. C++ compilers always produce bytecode for a virtual machine
    D. Linux cannot run any compiled programs

39. This C++ program compiles without errors but no executable is produced. At which stage does it fail, and why? Analysis

```
int square(int n);

int main() {
    return square(3);
}
```

    A. Preprocessing: there is no #include for square
    B. Compilation: square is used before it is declared
    C. Run time: square returns garbage
    D. Linking: square is declared but never defined, so the reference is unresolved

40. totl on line 3 is a typo for total. What happens when this Python script runs? Analysis

```
x = 5
if x > 10:
    print(totl)
print("done")
```

    A. It prints done with no error
    B. NameError before anything is printed
    C. SyntaxError, because totl is undefined
    D. It prints the value of x, then done

41. After the linker has produced an executable file, which part of the operating system copies it into memory and starts it running? Understanding
    A. The loader
    B. The assembler
    C. The preprocessor
    D. The compiler

42. What is source code? Understanding
    A. The binary machine instructions produced by the compiler
    B. The bytecode stored in a .class or .pyc file
    C. The executable file that the operating system loads
    D. The human-readable program text a programmer writes in a programming language

43. Which of these is a low-level programming language? Understanding
    A. Python
    B. Java
    C. JavaScript
    D. Assembly language

44. In what order do these phases of a typical compiler run? Understanding
    A. Lexical analysis, syntax analysis, semantic analysis, code generation
    B. Syntax analysis, lexical analysis, code generation, semantic analysis
    C. Lexical analysis, semantic analysis, code generation, syntax analysis
    D. Code generation, lexical analysis, syntax analysis, semantic analysis

45. What does the lexical analysis phase of a compiler produce from the source text? Understanding
    A. A parse tree that shows the grammatical structure
    B. Object code ready for the linker
    C. A stream of tokens such as keywords, identifiers, literals and operators
    D. A list of type errors in the program

46. In a standard flowchart, which symbol represents an input or output step such as 'Read marks' or 'Print result'? Understanding
    A. Parallelogram
    B. Rectangle
    C. Diamond
    D. Oval

47. What does the oval (terminal) symbol mark in a standard flowchart? Understanding
    A. The start or end of the algorithm
    B. A decision with Yes and No branches
    C. A calculation or assignment step
    D. Reading input from the user

48. Textbooks (following Knuth) list the properties every algorithm must have. Which of these is NOT one of them? Understanding
    A. It must finish after a finite number of steps
    B. Each step must be precisely and unambiguously defined
    C. It must be written in a programming language
    D. It must produce at least one output

49. In Python, which error is reported when the lines inside a block are not indented consistently? Understanding
    A. NameError
    B. TypeError
    C. IndentationError
    D. ValueError

50. When a Python program imports one of your own modules, where does CPython cache that module's compiled bytecode? Understanding
    A. In a .class file next to the .py file
    B. In a .o object file in the build folder
    C. In a native .exe file that replaces the .py file
    D. In a .pyc file inside a __pycache__ folder

51. What is a transpiler (source-to-source compiler)? Understanding
    A. A translator that converts source code in one high-level language into another high-level language, such as TypeScript into JavaScript
    B. A translator that converts assembly language into machine code
    C. A program that joins object files into one executable
    D. An interpreter that runs bytecode one instruction at a time

52. What is a cross-compiler? Understanding
    A. A compiler that accepts source code written in several languages at once
    B. A compiler that translates a program back from machine code into source code
    C. A compiler that runs on one platform but produces machine code for a different platform
    D. A compiler that checks code for errors without producing any output

53. In Java, int x = "hello"; uses valid tokens and valid grammar, yet javac rejects it. Which compiler phase detects this error? Understanding
    A. Lexical analysis
    B. Semantic analysis (type checking)
    C. Syntax analysis (parsing)
    D. Linking

54. Which Java comment style is read by the javadoc tool to generate API documentation? Understanding
    A. /** ... */
    B. // ...
    C. /* ... */
    D. # ...

55. What is true of dynamic linking compared with static linking? Understanding
    A. Library code stays in a shared file (.dll or .so) and is attached when the program loads or runs, so the executable is smaller
    B. Library code is copied into the executable at build time, so the executable is larger
    C. The program must be recompiled every time it runs
    D. It is only possible in interpreted languages such as Python

56. By convention, what does an exit status of 0 returned from main() mean to the operating system? Understanding
    A. The program crashed with an error
    B. The program finished successfully
    C. The program produced no output
    D. The program was stopped by the user

57. Which Python statement does nothing and is used where the grammar needs a statement but no action is wanted, such as an empty function body? Understanding
    A. continue
    B. break
    C. skip
    D. pass

58. Which statement about C++ block comments written as /* ... */ is correct? Understanding
    A. They can be nested to any depth
    B. They must start and end on the same line
    C. They are removed by the linker rather than by the compiler
    D. They cannot be nested; the first */ ends the comment

59. How does Java's import java.util.Scanner; differ from C++'s #include <vector>? Understanding

A. Both copy the library's source code into your file before compilation
B. import copies the class's bytecode into your .class file, while #include only records a reference
C. They are identical; Java simply uses a different keyword
D. #include copies the header's text into the source before compilation; import only tells the compiler where to find the class name and copies nothing

60. Which of these words is a reserved keyword in Java but NOT a keyword in Python? Understanding

A. if
B. return
C. class
D. static

61. Python is case-sensitive. Which of these is a Python keyword? Understanding

A. true
B. True
C. TRUE
D. null

62. In a C++ build, what is an object file (.o or .obj)? Understanding

A. A file that stores one C++ object (class instance) on disk
B. The final executable that can be run directly
C. The preprocessed source text with headers pasted in
D. The machine code for one source file, not yet linked with other files or libraries

63. What is pseudocode? Understanding

A. Code that compiles in every programming language
B. An informal, language-independent description of an algorithm's steps, written for people rather than a compiler
C. The bytecode produced by the Java compiler
D. Machine code written in hexadecimal

64. What does this Python code print? Application

```
print("a", "b", sep="-", end="!")
```

A. a b!
B. a-b
C. a-b-!
D. a-b!

65. What does this Python code print? Application

```
print("Hello" "World")
```

A. Hello World
B. A SyntaxError, because there is no comma
C. A TypeError, because strings cannot be joined without +
D. HelloWorld

66. What does this Python program print? Application

```
def show():
    print("one")

def show():
    print("two")

show()
```

A. one
B. one, then two
C. two
D. A SyntaxError, because show is defined twice

67. What does this Python code print? Application

```
print(print("hi"))
```

A. hi, then None on the next line
B. hi
C. hi, then hi again on the next line
D. None, then hi on the next line

68. What does this Python line print? Application

```
print("# total")  # shows the label
```

A. Nothing, because the line starts a comment
B. # total
C. total
D. # total # shows the label

69. This Python script is saved as app.py and run with the command python app.py red blue. What does it print? Application

```
import sys
print(len(sys.argv))
```

A. 2
B. 3
C. 1
D. 0

70. What does this Java fragment print? Application

```
System.out.print("A");
System.out.println("B");
System.out.print("C");
```

A. A, B and C on three separate lines
B. AB on the first line and C on the second
C. ABC on one line
D. A on the first line and BC on the second

71. What does this Java fragment print? Application

```
System.out.println("1" + 2 + 3);
System.out.println(1 + 2 + "3");
```

A. 6, then 6
B. 15, then 33
C. 123, then 123
D. 123, then 33

72. What happens when this Java fragment is compiled and run? Application

```
int a = 7, b = 0;
System.out.println(a / b);
```

A. It compiles, then throws an ArithmeticException at run time
B. The compiler rejects it with a division-by-zero error
C. It prints Infinity
D. It prints 0

73. What does this C++ program print? Application

```
#include <iostream>

int main() {
    std::cout << "A" << std::endl << "B";
    return 0;
}
```

A. AB on one line
B. A endl B
C. A on the first line and B on the second
D. B on the first line and A on the second

74. What does this JavaScript code print? Application

```
console.log(x);
var x = 3;
```

A. 3
B. undefined
C. A ReferenceError
D. null

75. What happens when this JavaScript code runs? Application

```
greet();

var greet = function () {
  console.log("hi");
};
```

A. It prints hi
B. It prints undefined
C. A TypeError: greet is not a function
D. A SyntaxError before anything runs

76. What happens when this JavaScript code runs? Application

```
console.log(y);
let y = 1;
```

A. A ReferenceError, because y is used before its let declaration runs
B. It prints undefined
C. It prints 1
D. It prints null

77. What does this Python program print? Application

```
def area(r):
    """Return the area of a square of side r."""
    return r * r

print(area.__doc__)
```

A. None
B. area
C. Nothing, because the docstring is a comment
D. Return the area of a square of side r.

78. What does this Python code print? Application

```
total = 1 + \
        2 + \
        3
print(total)
```

A. 1
B. A SyntaxError, because a statement cannot span lines
C. 123
D. 6

79. This Java class is run with the command java Main hello world. What does it print? Application

```
public class Main {
    public static void main(String[] args) {
        System.out.println(args.length);
    }
}
```

A. 3
B. 2
C. 1
D. 0

80. What does this C++ program print? Application

```
#include <iostream>
#define SQ(x) x * x

int main() {
    std::cout << SQ(2 + 3);
    return 0;
}
```

A. 11
B. 25
C. 10
D. It fails to compile

81. What does this Python code print? Application

```
print(10, end=" ")
print(20)
```

A. 10 and 20 on separate lines
B. 1020 on one line
C. 10 20 on one line
D. 10 20 followed by a space and no newline

82. What does this Python code print? Application

```
print("C:\\new")
print("C:\new")
```

A. C:\new on both lines
B. C:\\new on line 1 and C:\new on line 2
C. C:\new on line 1; C: on line 2 and new on line 3
D. C:new on both lines

83. This file is run directly with python report.py. What does it print? Application

```
def main():
    print("main")

print("top")

if __name__ == "__main__":
    main()
```

A. main, then top
B. top only
C. top, then main
D. main only

84. What happens when this C++ program is compiled and run? Application

```
#include <iostream>

int main() {
    std::cout << "A";
    return 0;
    std::cout << "B";
}
```

A. It fails to compile with an 'unreachable statement' error
B. It prints AB
C. It compiles and prints A
D. It prints B

85. What does this JavaScript code print, and why? Analysis

```
function total() {
  return
    5 + 5;
}
console.log(total());
```

A. undefined, because a semicolon is inserted automatically after return
B. 10, because the expression continues on the next line
C. A SyntaxError, because the expression is on its own line
D. 5, because only the first operand is returned

86. The function broken() is never called. What happens when this Python script is run? Analysis

```
print("start")

def broken():
    return 1 +

print("end")
```

A. start and end are printed, because broken() never runs
B. start is printed, then a SyntaxError is reported
C. A SyntaxError is reported and nothing is printed
D. start and end are printed, then a SyntaxError is reported

87. Why does this Java method fail to compile? Analysis

```
static int sign(int n) {
    if (n > 0) return 1;
    else if (n < 0) return -1;
}
```

A. An if statement without braces cannot contain a return
B. A path exists (n == 0) that reaches the end of the method without returning an int, so javac reports 'missing return statement'
C. A method returning int cannot have more than one return statement
D. It compiles; when n is 0 the method returns 0 by default

88. The same two arguments are passed to a Python script (python app.py a b) and a Java program (java App a b). Python's len(sys.argv) is 3 but Java's args.length is 2. Why? Analysis

    A. Python counts the word python itself as an argument
    B. sys.argv[0] holds the script name, while Java's args contains only the arguments after the class name
    C. Java drops the last argument by mistake unless it is quoted
    D. Python adds an empty string at the end of sys.argv

89. Two C++ files are built into one program. Each file compiles on its own without errors, but the build fails. At which stage, and why? Analysis

```
// util.cpp
int helper() { return 1; }

// main.cpp
int helper() { return 2; }
int main() { return helper(); }
```

    A. At the preprocessing stage, because there is no #include line
    B. At the compilation stage, because main.cpp cannot see util.cpp
    C. At the linking stage: helper is defined in two object files, giving a 'multiple definition' error
    D. At run time, because the program cannot decide which helper to call

90. What does this Java fragment do? Analysis

```
System.out.println(5.0 / 0);
System.out.println(5 / 0);
```

    A. Throws an ArithmeticException on the first line, before anything is printed
    B. Prints Infinity, then throws an ArithmeticException
    C. Prints Infinity twice
    D. Fails to compile, because both lines divide by zero

91. sum_to(5) should return 15 (1 + 2 + 3 + 4 + 5) but this Python program prints 10. What kind of error is this, and what causes it? Analysis

```
def sum_to(n):
    total = 0
    for i in range(1, n):
        total += i
    return total

print(sum_to(5))
```

    A. A syntax error: range needs three arguments
    B. A runtime error: total is used before it is assigned
    C. A logic error: range(1, n) stops at n - 1, so 5 is never added
    D. A logic error: total should start at 1 instead of 0

92. main.cpp includes both shapes.h and point.h, and shapes.h itself also includes point.h. Neither header has an include guard. What happens when main.cpp is compiled? Analysis

```
// point.h
struct Point { int x; int y; };

// shapes.h
#include "point.h"

// main.cpp
#include "shapes.h"
#include "point.h"
int main() { Point p{1, 2}; return p.x; }
```

    A. A compile error: struct Point is defined twice in the same translation unit; #pragma once or an #ifndef include guard fixes it
    B. A linker error: Point has two definitions in different object files
    C. It compiles, because the preprocessor ignores a header it has already included
    D. A runtime error when the second copy of Point is loaded

93. A Python script runs for ten minutes and then crashes with NameError because of a misspelled variable in a rarely used branch. What would happen to the same typo in an equivalent Java program? Analysis

    A. javac would reject it with a 'cannot find symbol' error before the program ever ran
    B. The Java program would also crash after ten minutes, with a NullPointerException
    C. The JVM would silently create the misspelled variable with a default value
    D. The Java program would run, and the typo would only be caught by the linker

94. What happens when this Python script runs? Analysis

```
debug = False

if debug:
    def log(msg):
        print(msg)

log("hi")
```

    A. It prints hi, because every def is processed before the program runs
    B. A SyntaxError, because a def cannot appear inside an if block
    C. It prints nothing, because calls to log are skipped when debug is False
    D. A NameError, because the def inside the false if block never runs, so log is never defined

95. A Java student meets three bugs: (i) a missing semicolon, (ii) reading index 10 of an array of length 10, (iii) using + where * was needed in a formula. How are they classified, in order? Analysis

    A. Runtime error, compile-time error, logic error
    B. Compile-time error, logic error, runtime error
    C. Compile-time (syntax) error, runtime error, logic error
    D. Compile-time error, compile-time error, logic error

96. A student converts the Python line print(7 // 2) to C++ as shown. What happens when it is compiled? Analysis

```
#include <iostream>

int main() {
    std::cout << 7 // 2;
    return 0;
}
```

A. It prints 3, because // is integer division as in Python
B. It fails to compile: // starts a comment, so the statement loses its semicolon
C. It prints 3.5
D. It prints 7, because the rest of the line is a comment and the code is otherwise fine

97. Why does CPython usually run a tight numeric loop much more slowly than the same loop in C++ compiled with optimisation? Analysis

A. CPython recompiles the loop's source text on every iteration
B. Python integers are stored as text strings
C. The C++ compiler removes the loop and runs it on the GPU
D. CPython interprets bytecode instruction by instruction with type checks at run time, while the C++ loop is already native machine code

98. This C++ program fails to compile, although the same arrangement works in JavaScript with a function declaration. Why? Analysis

```
#include <iostream>

int main() {
    std::cout << twice(4);
    return 0;
}

int twice(int n) { return 2 * n; }
```

A. The linker cannot find twice because it is in the same file as main
B. C++ requires a name to be declared before it is used; twice is declared after main, so add a prototype int twice(int n); above main
C. Functions in C++ must be defined inside main
D. twice must be declared static to be called from main

99. The rule is: a mark of 50 or more passes. Which test input reveals the bug in this Python function? Analysis

```
def is_pass(mark):
    return mark > 50
```

A. 49
B. 50
C. 51
D. 100

Data Types & Variables MCQs for NSCT

100. How many bits does an int occupy in Java? Understanding
    A. 32
    B. 16
    C. 64
    D. It depends on the platform

101. What is the size of the char type in Java? Understanding
    A. 8 bits
    B. 32 bits
    C. 16 bits
    D. 1 bit

102. Which size rule is actually guaranteed by the C++ standard? Understanding
    A. int is always 32 bits
    B. long is always 64 bits
    C. double is always 4 bytes
    D. sizeof(char) is always 1

103. How do Python programmers normally mark a value as a constant? Understanding
    A. By naming it in UPPER_CASE, a convention the interpreter does not enforce
    B. With the const keyword
    C. With the final keyword
    D. By declaring it inside a constant block

104. Which Java keyword makes a variable's value unchangeable after it is assigned? Understanding
    A. const
    B. static
    C. final
    D. immutable

105. What happens when this C++ code is compiled? Understanding

```
const int x = 5;
x = 6;
```

    A. It compiles and x becomes 6
    B. It compiles, but x stays 5 at run time
    C. Compile error: assignment of a read-only variable
    D. It throws an exception at run time

106. What is the range of the Java byte type? Understanding
    A. -128 to 127
    B. 0 to 255
    C. -127 to 127
    D. -256 to 255

107. Which of these is NOT a primitive type in Java? Understanding
    A. boolean
    B. char
    C. String
    D. double

108. What is the largest value a Python 3 int can hold? Understanding

A. 2,147,483,647
B. 9,223,372,036,854,775,807
C. 1.8 × 10^308
D. There is no fixed limit; it is bounded only by available memory

109. Python is described as dynamically typed. What does this mean? Understanding

A. Every variable must be declared with a type before use
B. Values are converted automatically between any types, so "5" + 3 gives 8
C. Types are fixed at compile time and cannot change
D. Types belong to values and are checked at run time, so a name can be rebound to a value of a different type

110. Which Java declaration compiles without an explicit cast? Understanding

A. int i = 3.5;
B. double d = 10;
C. float f = 2.5;
D. long n = 2.5;

111. In Java, a variable is declared inside the braces of an if block. Where can it be used? Understanding

A. Only inside that block, from its declaration to the closing brace
B. Anywhere in the method after the if
C. Anywhere in the class
D. Only in the condition of the if

112. In Java, what is true about initial values? Understanding

A. An int field of a class defaults to 0, but a local int variable must be assigned before it is read
B. Both int fields and local int variables default to 0
C. Neither has a default; both hold garbage
D. Local int variables default to 0, but fields must be assigned

113. In Java, what is the type of the literal 3.14 written on its own? Understanding

A. float
B. BigDecimal
C. double
D. It depends on the variable it is assigned to

114. In what order does Python search for a name used inside a nested function? Understanding

A. Global, Local, Enclosing, Built-in
B. Built-in, Global, Enclosing, Local
C. Local, Global, Built-in, Enclosing
D. Local, Enclosing, Global, Built-in

115. Which statement about Java's boolean type is correct? Understanding

A. true is stored as 1 and can be added to an int
B. It cannot be converted to or from int, even with a cast
C. Any non-zero int can be used where a boolean is expected
D. It can be cast to int with (int) true

116. What does this Python code print? Application

```
x = 7 / 2
print(type(x).__name__, x)
```

A. int 3
B. float 3.5
C. float 3.0
D. int 3.5

117. What does this Python code print? Application

```
print(int("42") + 8)
```

A. 428
B. TypeError
C. 50
D. 42 8

118. What does this Python code print? Application

```
print(int(-3.99))
```

A. -3
B. -4
C. -3.99
D. -4.0

119. What happens when this Python code runs? Application

```
print("5" + 3)
```

A. TypeError: Python will not add a str and an int
B. It prints 8
C. It prints 53
D. It prints 5 3

120. What does this Java code print? Application

```
int a = (int) 9.99;
System.out.println(a);
```

A. 10
B. 9.99
C. 9
D. Compile error

121. What does this Java code print? Application

```
byte b = (byte) 130;
System.out.println(b);
```

A. 130
B. -126
C. 127
D. 2

122. What does this Java code print? Application

```
int x = Integer.MAX_VALUE;
x = x + 1;
System.out.println(x);
```

A. 2147483648
B. -2147483648
C. 2147483647
D. An ArithmeticException is thrown

123. What does this Java statement print? Application

```
System.out.println('A' + 2);
```

A. C
B. A2
C. Compile error
D. 67

124. What does this Python code print? Application

```
print(True + True + 1)
```

A. TrueTrue1
B. 3
C. 2
D. TypeError

125. What does this JavaScript code print? Application

```
console.log("5" * 2, "5" + 2);
```

A. 10 7
B. 10 52
C. 52 52
D. NaN 52

126. What does this Python code print? Application

```
x = 10

def f():
    x = 20
    return x

print(f(), x)
```

A. 20 10
B. 20 20
C. 10 10
D. 10 20

127. What does this JavaScript code print? Application

```
let x = 1;
{
  let x = 2;
}
console.log(x);
```

A. 2
B. undefined
C. SyntaxError
D. 1

128. What does this JavaScript code print? Application

```
if (true) {
  var y = 5;
}
console.log(y);
```

A. ReferenceError: y is not defined
B. undefined
C. 5
D. true

129. What does this Python code print? Application

```
n = 1

def bump():
    global n
    n += 1

bump()
bump()
print(n)
```

A. 1
B. 3
C. 2
D. UnboundLocalError

130. What happens when this Python code runs, and why? Analysis

```
count = 0

def inc():
    count += 1

inc()
```

A. It runs and the global count becomes 1
B. NameError: count is not defined anywhere
C. SyntaxError: += cannot be used inside a function
D. UnboundLocalError: count += 1 makes count local, and it is read before it has a value

131. This C++ code prints 3, not 3.5. Why? Analysis

```
int a = 7;
double b = a / 2;
std::cout << b;
```

A. cout always rounds doubles down
B. b is stored as an int
C. a / 2 is integer division, done before the result is converted to double
D. double cannot hold 3.5 exactly

132. Why does line 2 compile even though a is final? Analysis

```
final int[] a = {1, 2, 3};
a[0] = 9;
System.out.println(a[0]);
```

A. final only applies to primitive types, never to arrays
B. final fixes the reference a, not the contents of the array it points to
C. Line 2 does not compile
D. final values can be changed once, inside the same method

133. What does this Python code print, and what does it show about constants? Analysis

```
PI = 3.14
PI = 3
print(PI)
```

A. 3.14: the interpreter blocks changes to UPPER_CASE names
B. An error, because PI is a constant
C. 3: UPPER_CASE is only a naming convention, so reassignment is allowed
D. 3.14: the second assignment creates a separate variable

134. What happens when this Python code runs? Analysis

```
print(int("3.5"))
```

A. It prints 3
B. It prints 4
C. It prints 3.5
D. ValueError: int() cannot parse the text 3.5 as an integer

135. Why does line 3 fail to compile in Java when line 2 compiles? Analysis

```
char c = 'A';
c += 2;      // line 2
c = c + 2;   // line 3
```

A. char variables cannot be changed after they are declared
B. Line 2 also fails; both lines are errors
C. 2 must be written as '2' when added to a char
D. c + 2 is an int, and assigning an int to a char needs an explicit cast; += includes an implicit cast

136. A programmer wants the number of microseconds in a day. What does this Java code print? Analysis

```
long micros = 24 * 60 * 60 * 1000 * 1000;
System.out.println(micros);
```

A. 500654080
B. 86400000000
C. -500654080
D. An ArithmeticException is thrown

137. What does this Python code print? Analysis

```
x = "10"
y = x * 2
print(y)
```

A. 20
B. 1010
C. TypeError
D. 10 10

138. Assume unsigned int is 32 bits. What does this C++ code print? Analysis

```
unsigned int u = 0;
u = u - 1;
std::cout << u;
```

A. -1
B. 0
C. Undefined behaviour, so no fixed output
D. 4294967295

139. What happens when this Python code runs, and why? Analysis

```
list = [1, 2]
print(list("ab"))
```

A. TypeError: the name list now refers to a list object, which is not callable
B. It prints ['a', 'b']
C. SyntaxError: list is a reserved keyword
D. NameError: list is not defined

140. A Java program must store a population count of 5,000,000,000. Which primitive type is the smallest that can hold it without overflow? Understanding

A. long
B. int
C. short
D. char

141. In Python 3, what is the type of the value None? Understanding

A. NoneType
B. null
C. void
D. object only, with no type of its own

142. Roughly how many significant decimal digits of precision do Java's float and double give? Understanding

A. About 16 for float and about 32 for double
B. Exactly 4 for float and exactly 8 for double
C. About 7 for float and about 15 to 16 for double
D. Both give unlimited precision

143. What is the wrapper class for the primitive type int in Java? Understanding

A. Int
B. Number
C. IntWrapper
D. Integer

144. What is autoboxing in Java? Understanding

A. Automatically casting a double to an int when precision would be lost
B. Grouping several variables into one object to save memory
C. The automatic conversion of a primitive to its wrapper object, such as an int to an Integer when it is added to an ArrayList<Integer>
D. The automatic conversion of a String to an int when it is printed

145. Which of these Java conversions is a widening conversion that happens automatically, without a cast? Understanding

A. double to float
B. int to short
C. int to char
D. long to float

146. In Java 10 and later, what does var count = 10; declare? Understanding

A. A dynamically typed variable that can later hold a String
B. A variable of type Object
C. A global variable visible in every class
D. A local variable whose type, int, is inferred at compile time and cannot change afterwards

147. In C++11 and later, what type does x have after auto x = 5.0f;? Understanding

A. double
B. int
C. float
D. auto, a special type that can hold any value

148. In JavaScript, which keyword declares a block-scoped variable that cannot be reassigned? Understanding

A. var
B. let
C. static
D. const

149. A C++ function declares static int calls = 0; inside its body. How do the scope and the lifetime of calls compare? Understanding

A. Both its scope and lifetime are the whole program, so any function can use it
B. Both its scope and lifetime end when the function returns
C. Its scope is the function's block, but its lifetime is the whole run of the program, so it keeps its value between calls
D. Its scope is the whole file, but it is destroyed when the function returns

150. In Java, which kind of variable has one copy shared by every object of the class? Understanding

A. A static field (class variable)
B. An instance field
C. A local variable
D. A method parameter

151. What is the nonlocal keyword used for in Python? Understanding

A. To rebind a variable that belongs to the enclosing function's scope from inside a nested function
B. To rebind a module-level (global) variable from inside a function
C. To declare a variable that cannot be changed
D. To make a local variable visible to other modules

152. In C++, what is an advantage of const double PI = 3.14159; over #define PI 3.14159? Understanding

A. The const is replaced by the preprocessor, so it uses no memory
B. The const has a type and obeys scope rules, so the compiler can check how it is used
C. The const can be changed at run time if needed
D. The #define version cannot be used in arithmetic

153. Which of these Python types is immutable? Understanding

A. tuple
B. list
C. dict
D. set

154. In JavaScript, what type does let n = 5; give n? Understanding

A. int, a 32-bit integer
B. number, a 64-bit IEEE 754 floating-point value; JavaScript has no separate integer type for ordinary numbers
C. long, a 64-bit integer
D. A string, until it is used in arithmetic

155. Why does 0.1 + 0.2 == 0.3 evaluate to False in Python (and the same comparison fail in Java and JavaScript)? Understanding

A. Python rounds every float to one decimal place before comparing
B. 0.1, 0.2 and 0.3 cannot be stored exactly in binary floating point, so the stored values are close approximations that do not add up exactly
C. == compares the memory addresses of floats, not their values
D. The + operator joins the numbers as text, giving 0.10.2

156. Which of these Python literals is a float? Understanding

A. 1000
B. "1e3"
C. 1e3
D. 0x3E8

157. Which is the conventional way to declare a class-level constant in Java? Understanding

A. const int MAX_USERS = 100;
B. public final static var maxUsers = 100;
C. #define MAX_USERS 100
D. public static final int MAX_USERS = 100;

158. A Python function is written def area(r: float) -> float:. What checks these type hints? Understanding

A. The interpreter, which raises TypeError if area receives a str
B. Only external tools such as mypy or an IDE; the Python interpreter stores the hints but does not enforce them at run time
C. The interpreter, which converts the argument to float automatically
D. The compiler, which refuses to create bytecode for a wrong call

159. What is the value of a after this Java fragment runs, and why? Understanding

```
int a = 5;
int b = a;
b = 9;
```

A. 9, because a and b refer to the same memory
B. 0, because a is reset when b is assigned
C. 5, because assigning a primitive copies its value, so a and b are independent
D. It does not compile, because a value cannot be assigned twice

160. What is the range of an 8-bit unsigned char in C++? Understanding

A. -128 to 127
B. 0 to 255
C. 0 to 127
D. -255 to 255

161. What is a Python float, in terms of storage? Understanding

A. A 64-bit IEEE 754 double-precision number
B. A 32-bit single-precision number
C. An arbitrary-precision decimal with no rounding
D. A 16-bit half-precision number

162. Which Java primitive type is unsigned? Understanding

A. byte
B. short
C. char
D. int

163. A Java class has a field x, and its constructor has a parameter also named x. How does the constructor refer to the field? Understanding

A. x
B. super.x
C. this.x
D. Main.x

164. What does this Python code print? Application

```
print(bool(""), bool("False"))
```

A. False False
B. True False
C. True True
D. False True

165. What does this Python code print? Application

```
print(int("1010", 2))
```

A. 1010
B. 2
C. A ValueError
D. 10

166. What does this Python code print? Application

```
a, b = 5, 9
a, b = b, a
print(a, b)
```

A. 5 9
B. 9 9
C. 5 5
D. 9 5

167. What happens when this Python code runs? Application

```
s = "Java"
s[0] = "L"
print(s)
```

A. It prints Lava
B. It prints Java
C. A TypeError, because strings are immutable
D. It prints LJava

168. What does this Python code print? Application

```
for i in range(3):
    last = i * 10
print(i, last)
```

A. 3 30
B. 2 20
C. A NameError, because i and last only exist inside the loop
D. 0 0

169. What does this Java statement print? Application

```
System.out.println(0x1F + 010);
```

A. 39
B. 41
C. 0x1F010
D. 25

170. What does this Java fragment print? Application

```
char c = 'a';
c++;
System.out.println(c);
```

A. 98
B. a1
C. It fails to compile
D. b

171. What does this Java statement print? Application

```
System.out.println((int) 'A' + " " + (char) 66);
```

A. A 66
B. 65 B
C. 65 66
D. A B

172. What does this Java statement print? Application

```
System.out.println(Math.round(2.5) + " " + Math.round(-2.5));
```

A. 3 -3
B. 3 -2
C. 2 -2
D. 2 -3

173. What does this Python code print? Application

```
print(round(2.5), round(3.5))
```

A. 3 4
B. 2 3
C. 2 4
D. 3 3

174. What does this JavaScript code print? Application

```
console.log(typeof null, typeof NaN);
```

A. object number
B. null NaN
C. null number
D. object NaN

175. What does this JavaScript code print? Application

```
console.log(parseInt("12px"), Number("12px"));
```

A. 12 12
B. NaN NaN
C. 12 NaN
D. NaN 12

176. What does this C++ fragment print? Application

```
int x = 3.9;
char c = 66;
std::cout << x << c;
```

A. 3.966
B. 3B
C. 466
D. 4B

177. What does this Java fragment print? Application

```
float f = 0.1f;
double d = 0.1;
System.out.println(f == d);
```

A. true
B. false
C. It fails to compile, because float and double cannot be compared
D. 0.1

178. What does this Python code print? Application

```
print(isinstance(True, int), True == 1)
```

A. False True
B. True True
C. False False
D. True False

179. What does this Python program print? Application

```
def outer():
    n = 0
    def inc():
        nonlocal n
        n += 1
    inc()
    inc()
    return n

print(outer())
```

A. 0
B. 1
C. 2
D. An UnboundLocalError

180. What happens when this JavaScript code runs? Application

```
const k = 1;
k = 2;
console.log(k);
```

A. It prints 2
B. A TypeError: Assignment to constant variable
C. It prints 1
D. It prints undefined

181. What does this C++ program print? Application

```
#include <iostream>

int counter() {
    static int n = 0;
    return ++n;
}

int main() {
    counter();
    counter();
    std::cout << counter();
    return 0;
}
```

A. 1
B. 0
C. 3
D. 2

182. What does this Java fragment print? Application

```
long big = 1L << 40;
int small = (int) big;
System.out.println(small);
```

A. 1099511627776
B. 2147483647
C. 0
D. -1

183. What does this JavaScript code print? Application

```
console.log(9007199254740993 === 9007199254740992);
```

A. false
B. A RangeError
C. undefined
D. true

184. What does this Java fragment print? Analysis

```
Integer a = 127, b = 127;
Integer c = 128, d = 128;
System.out.println((a == b) + " " + (c == d));
```

A. true true
B. false false
C. false true
D. true false

185. What does this Python code print? Analysis

```
a = b = []
a.append(1)
print(b)
```

A. [1]
B. []
C. [[1]]
D. A NameError, because b is never assigned

186. What does this Python code print? Analysis

```
grid = [[0] * 2] * 2
grid[0][0] = 1
print(grid)
```

A. [[1, 0], [0, 0]]
B. [[1, 1], [0, 0]]
C. [[1, 0], [1, 0]]
D. [[1, 1], [1, 1]]

187. What happens when this Java fragment is compiled? Analysis

```
int total = 0;
for (int i = 0; i < 3; i++) {
    total += i;
}
System.out.println(i);
```

A. A compile error: i is declared in the for header, so it does not exist after the loop
B. It prints 3
C. It prints 2
D. It prints 0, the default value of an int

188. What does this C++ program print? Analysis

```
#include <iostream>

int x = 1;

int main() {
    int x = 2;
    {
        int x = 3;
        std::cout << x << ::x;
    }
    std::cout << x;
    return 0;
}
```

A. 312
B. 322
C. 333
D. 321

189. What does this JavaScript code print? Analysis

```
var fns = [];
for (var i = 0; i < 3; i++) {
  fns.push(() => i);
}
console.log(fns.map(f => f()).join(","));
```

A. 0,1,2
B. 3,3,3
C. 2,2,2
D. undefined,undefined,undefined

190. A student adds 0.1 ten times and checks the result. What does this Python code print, and why? Analysis

```
total = 0.0
for _ in range(10):
    total += 0.1
print(total == 1.0)
```

A. False, because each 0.1 is a binary approximation and the small errors add up to 0.9999999999999999
B. True, because ten times 0.1 is exactly 1.0
C. True, because Python stores floats as exact decimals
D. A TypeError, because a float is compared with an int-like literal

191. Which line does a C++11 compiler reject, and why? Analysis

```
int a = 3.7;   // line 1
int b{3.7};    // line 2
```

A. Line 2 only, because brace (list) initialisation forbids narrowing conversions such as double to int
B. Line 1 only, because C++ never converts double to int implicitly
C. Both lines, because a double can never initialise an int
D. Neither line; both store 3

192. 16777217 is 2^24 + 1. What does this Java fragment print? Analysis

```
float f = 16777217;
System.out.println(f == 16777216);
```

A. true
B. false
C. It fails to compile, because an int cannot be assigned to a float without a cast
D. 16777217.0

193. The user types 20 and presses Enter. What happens when this Python code runs? Analysis

```
age = input()
print(age + 1)
```

A. A TypeError, because input() returns a str and Python will not add a str and an int
B. It prints 21
C. It prints 201
D. A ValueError, because 20 is not a valid input

194. What does this JavaScript code print? Analysis

```
let a = "5";
let b = 5;
console.log(a == b, a === b);
```

A. false false
B. true false
C. true true
D. false true

195. Which line of this Java fragment fails to compile? Analysis

```
final int limit;      // line 1
limit = 5;            // line 2
limit = 6;            // line 3
System.out.println(limit);
```

A. Line 1, because a final variable must be given a value where it is declared
B. Line 2, because a final variable cannot be assigned after its declaration
C. None; it prints 6
D. Line 3, because a final variable can be assigned only once

196. What does this Python code print? Analysis

```
price = "250"
qty = 2
print(int(price) * qty)
```

A. 250250
B. A TypeError, because price is a string
C. 2502
D. 500

197. What does this Python code print? Analysis

```
x = "global"

def outer():
    x = "enclosing"
    def inner():
        return x
    return inner()

print(outer())
```

A. global
B. An UnboundLocalError
C. A NameError
D. enclosing

198. Assume int and unsigned int are 32 bits. What does this C++ fragment print? Analysis

```
int i = -1;
unsigned int u = 1;
std::cout << (i < u);
```

A. 1
B. 0
C. -1
D. It fails to compile, because signed and unsigned cannot be compared

Operators & Expressions MCQs for NSCT

199. Which of these operators has the highest precedence in C++, Java and Python? Understanding

A. +
B. ==
C. =
D. *

200. What does the // operator do in Python 3? Understanding

A. Floor division: divides and rounds the result down to the nearest whole number
B. Starts a comment
C. True division that always returns a float
D. Integer division that truncates toward zero

201. What does the % operator return for two positive integers, as in 17 % 5? Understanding

A. The remainder after integer division, here 2
B. The percentage, here 3.4
C. The quotient, here 3
D. The result rounded to the nearest multiple of 5, here 15

202. In C++ and Java, how is a = b = c = 7 evaluated? Understanding

A. Left to right: a = b first, then b = c
B. It is a compile error
C. Only a is assigned 7
D. Right to left: c = 7 first, then b = c, then a = b

203. In Python, how is 2 ** 3 ** 2 grouped? Understanding

A. (2 ** 3) ** 2, because all operators are left-associative
B. It raises a SyntaxError without brackets
C. 2 ** (3 ** 2), because ** is right-associative
D. (2 ** 3) * 2

204. Which pair correctly writes logical AND in Java and in Python? Understanding

A. and in Java, && in Python
B. & in both languages
C. && in Java, and in Python
D. && in both languages

205. What does short-circuit evaluation mean for && and ||? Understanding

A. Both operands are always evaluated, from right to left
B. The result is converted to a short integer
C. The right operand is not evaluated if the left operand already decides the result
D. The expression is evaluated at compile time

206. In C++ and Java, what is the difference between x++ and ++x when used inside an expression? Understanding

A. x++ yields the old value then increments; ++x increments then yields the new value
B. x++ adds 1 and ++x adds 2
C. There is no difference in any context
D. ++x increments x only after the whole statement finishes

207. In C++, Java and Python, what does the ^ operator do? Understanding

A. Raises a number to a power
B. Logical AND
C. Bitwise exclusive OR (XOR)
D. Bitwise NOT

208. For a non-negative integer x, what is x << 3 equivalent to, assuming no overflow? Understanding

A. x multiplied by 3
B. x divided by 8
C. x plus 3
D. x multiplied by 8

209. What happens if you write x++ in Python? Understanding

A. It is a SyntaxError; Python has no ++ operator
B. x increases by 1
C. x increases by 2
D. x is unchanged but the expression returns x + 1

210. In Java, a and b are boolean. How does a & b differ from a && b? Understanding

A. & is a compile error with boolean operands
B. & always evaluates both operands, while && skips b when a is false
C. & returns an int, while && returns a boolean
D. They differ only in precedence; both short-circuit

211. Which operator has lower precedence than the relational operators (<, >, <=, >=) in Java? Understanding

A. &&
B. *
C. +
D. unary !

212. In Python, what does the expression 1 < x < 5 mean? Understanding

A. (1 < x) < 5, comparing a True/False value with 5
B. It is a SyntaxError
C. 1 < x and x < 5, with x evaluated only once
D. 1 < x or x < 5

213. What does Java's >>> operator do? Understanding

A. Shifts right and copies the sign bit into the vacated bits
B. Shifts left by three places
C. Compares whether one value is much greater than another
D. Shifts right and fills the vacated left bits with zeros, regardless of sign

214. What does == compare in Python, and what does is compare? Understanding

A. == compares object identity; is compares values
B. They are identical in every case
C. is compares types; == compares values
D. == compares values; is checks whether both names refer to the same object

215. What does this Python code print? Application

```
print(2 + 3 * 4)
```

A. 20
B. 14
C. 24
D. 9

216. What does this Python code print? Application

```
print(10 // 3, 10 % 3)
```

A. 3.3333333333333335 1
B. 3 3
C. 1 3
D. 3 1

217. What does this Python code print? Application

```
print(-7 // 2, -7 % 2)
```

A. -4 1
B. -3 -1
C. -3 1
D. -4 -1

218. What does this Java statement print? Application

```
System.out.println(-7 / 2 + " " + -7 % 2);
```

A. -3 -1
B. -4 1
C. -3 1
D. -3.5 -1

219. What does this Java code print? Application

```
int x = 5;
int y = x++;
System.out.println(x + " " + y);
```

A. 6 6
B. 6 5
C. 5 5
D. 5 6

220. What does this C++ code print? Application

```
int x = 5;
int y = ++x * 2;
std::cout << x << " " << y;
```

A. 6 10
B. 6 12
C. 5 12
D. 5 10

221. What does this Python code print? Application

```
print(6 & 3, 6 | 3, 6 ^ 3)
```

A. 2 7 1
B. 6 3 5
C. 2 7 5
D. 1 7 5

222. What does this Python code print? Application

```
print(5 << 2, 20 >> 3)
```

A. 10 6
B. 7 17
C. 20 2.5
D. 20 2

223. What does this Python code print? Application

```
print(~5)
```

A. -5
B. -6
C. 2
D. 250

224. What does this JavaScript code print? Application

```
console.log(1 + 2 + "3", "1" + 2 + 3);
```

A. 6 6
B. 33 123
C. 123 33
D. 33 6

225. What does this Python code print? Application

```
x = 7
print("odd" if x % 2 else "even")
```

A. even
B. odd
C. 1
D. SyntaxError

226. What does this Python code print? Application

```
x = 10
x -= 3 * 2
print(x)
```

A. 14
B. 1
C. -4
D. 4

227. What does this Python code print? Application

```
def check():
    print("checked")
    return True

x = 0
if x != 0 and check():
    print("yes")
print("end")
```

A. end
B. checked, then end
C. checked, yes, then end
D. yes, then end

228. What does this Python code print? Application

```
print(-2 ** 2)
```

A. 4
B. -4
C. 0.25
D. -0.25

229. Java evaluates operands left to right. What does this Java code print? Analysis

```
int a = 5;
int b = a++ + ++a;
System.out.println(a + " " + b);
```

A. 7 11
B. 6 12
C. 7 13
D. 7 12

230. What does this Python code print? Analysis

```
print(not 1 == 2)
```

A. True
B. False
C. SyntaxError
D. TypeError

231. What does this Java statement print? Analysis

```
System.out.println(1 + 2 + "3" + 4 + 5);
```

A. 15
B. 339
C. 12345
D. 3345

232. x is 4, which is even. What does this C++ code print, and why? Analysis

```
int x = 4;
if (x & 1 == 0) std::cout << "even";
else std::cout << "odd";
```

A. even, because 4 & 1 is 0
B. odd, because == binds tighter than &, so the test is x & (1 == 0)
C. A compile error, because & cannot be used in an if
D. odd, because & is logical AND and x is non-zero

233. d is 0. Which condition is safe to evaluate, and why? Analysis

```
n, d = 10, 0
print(d != 0 and n / d > 2)
print(n / d > 2 and d != 0)
```

A. n / d > 2 and d != 0, because Python evaluates the right side first
B. Both are safe, because Python ignores division by zero inside conditions
C. d != 0 and n / d > 2, because and stops before the division when d is 0
D. Neither is safe, because and always evaluates both operands

234. What does this Python code print? Analysis

```
print(0 or "x", 3 and 0)
```

A. True False
B. 0 3
C. x 0
D. x 3

235. What does this JavaScript code print? Analysis

```
console.log(0 == "0", 0 === "0", null == undefined);
```

A. false false false
B. true false true
C. true true true
D. false false true

236. What does this Python code print? Analysis

```
x = 5
print(--x, -(-x))
```

A. 4 5
B. 4 6
C. 5 5
D. SyntaxError

237. What does this Python code print? Analysis

```
print(1 << 2 + 1)
```

A. 5
B. 6
C. 8
D. 4

238. What does this Java code print? Analysis

```
int i = 5;
i = i++;
System.out.println(i);
```

A. 5
B. 6
C. Compile error
D. Undefined output

239. Which operator in C++ and Java takes three operands? Understanding

A. ?: (the conditional operator)
B. && (logical AND)
C. += (compound assignment)
D. [] (array subscript)

240. In Python 3, what does the expression 7 / 2 evaluate to? Understanding

A. 3, an int
B. 3.5, a float
C. 4, an int
D. 3.0, a float

241. Which of these is a relational (comparison) operator in C++, Java and Python? Understanding

A. =
B. +=
C. >=
D. &

242. For an integer n, what does the condition n % 2 == 0 test in C++, Java and Python? Understanding

A. Whether n is odd
B. Whether n is divisible by 4
C. Whether n is positive
D. Whether n is even

243. In Python, what does the * operator do when one operand is a string and the other an int, as in "ab" * 3? Understanding

A. Repeats the string that many times
B. Raises a TypeError
C. Multiplies the character codes
D. Converts the string to a number first

244. What does Python's in operator test, as in x in items? Understanding

A. Whether x is an instance of the type items
B. Whether x is a member of the collection items
C. Whether x and items are the same object
D. Whether x is less than every element of items

245. In Java, what is the type and value of the expression 5 + 2.0? Understanding

A. int, 7
B. float, 7.0f
C. double, 7.0
D. String, "52.0"

246. Python has no ++ operator. Which statement increases x by 1? Understanding

A. x++
B. ++x
C. x =+ 1
D. x += 1

247. What is the difference between operator precedence and order of evaluation? Understanding

A. Precedence decides how operands are grouped with operators; order of evaluation decides which operand is computed first at run time
B. They are two names for the same rule
C. Precedence applies only to arithmetic operators; order of evaluation applies only to logical operators
D. Order of evaluation decides grouping; precedence decides which function call runs first

248. In C++ and Java, how is the integer expression 100 / 10 / 5 grouped, and what is its value? Understanding

A. 100 / (10 / 5), which is 50
B. (100 / 10) / 5, which is 2
C. (100 / 5) / 10, which is 2
D. It is a compile error without parentheses

249. What does Python's := operator (the walrus operator, Python 3.8+) do? Understanding

A. Compares two values for identity
B. Declares a constant that cannot be reassigned
C. Assigns a value to a name and also returns that value, so it can be used inside an expression
D. Assigns only if the name is not already defined

250. What does Java's instanceof operator produce? Understanding

A. The class name of the object as a String
B. A new object cast to the given type
C. An int code identifying the object's type
D. A boolean saying whether an object is an instance of the given type or one of its subtypes

251. In C++, sizeof is best described as which of these? Understanding

A. A compile-time operator that yields the size in bytes of a type or expression
B. A library function that counts the elements in an array at run time
C. A function that returns the length of a string
D. A keyword that allocates memory

252. In Python, what is the precedence order of the logical operators, from highest to lowest? Understanding

A. and, then or, then not
B. not, then and, then or
C. or, then and, then not
D. All three have the same precedence and group left to right

253. In JavaScript, when does a ?? b return b? Understanding

A. When a is any falsy value such as 0, "" or false
B. When a and b are not strictly equal
C. Only when a is null or undefined
D. Only when a is NaN

254. In Java, what is the value of the int expression Integer.MAX_VALUE + 1? Understanding

A. An ArithmeticException is thrown
B. Integer.MAX_VALUE, because the value saturates
C. 2147483648, because Java widens the result to long
D. Integer.MIN_VALUE, because int overflow wraps around in two's complement

255. In C++ and Java, what does x *= y + 1; do? Understanding

A. x = x * (y + 1)
B. x = x * y + 1
C. x = x * y, then y = y + 1
D. x = x + (y * 1)

256. In C++ and Java, how is a || b && c grouped? Understanding

A. (a || b) && c, because operators group left to right
B. a || (b && c), because && has higher precedence than ||
C. It is a compile error without parentheses
D. (a && c) || (b && c)

257. In Java, s1 and s2 are String objects with the same characters, built separately at run time. Why can s1 == s2 be false? Understanding

A. == compares lengths, not characters
B. == is not allowed on Strings and always returns false
C. == compares object references, so it is true only if both refer to the same object; equals() compares the characters
D. == compares hash codes, which differ for every object

258. In C++, what does the expression (a, b) produce when , is the built-in comma operator? Understanding

A. It yields a, after evaluating b
B. It yields a pair made of the two values
C. It yields a + b
D. It evaluates a, discards the result, then evaluates b and yields b

259. In Java, byte b = 10; b += 5; compiles, but b = b + 5; does not. Why? Understanding

A. A compound assignment includes an implicit cast back to the variable's type, while b + 5 is an int that needs an explicit cast to become a byte
B. += works only on byte, and + works only on int
C. b + 5 overflows a byte, but b += 5 does not
D. b = b + 5 reads b before it is initialised

260. In two's complement arithmetic (used by Java ints, by C++20 and conceptually by Python ints), which expression equals -x for any integer x where nothing overflows? Understanding

A. ~x - 1
B. ~x + 1
C. x ^ 1
D. x >> 1

261. Bits are numbered from 0 at the right. Which expression gives x with bit k cleared to 0 and every other bit unchanged? Understanding

A. x | (1 << k)
B. x ^ (1 << k)
C. x & ~(1 << k)
D. x & (1 << k)

262. In C++, how do 6 & 3 and 6 && 3 differ? Understanding

A. Both give 2; && is just the short-circuit version of &
B. 6 & 3 gives true and 6 && 3 gives 2
C. 6 && 3 does not compile because && needs bool operands
D. 6 & 3 is a bitwise AND giving 2; 6 && 3 is a logical AND giving true because both operands are non-zero

263. What does this Python code print? Application

```
print(8 - 3 - 2)
```

A. 3
B. 7
C. -3
D. 13

264. What does this Python code print? Application

```
print("ab" * 3 + "c")
```

A. abcabcabc
B. abababc
C. ab3c
D. ababab

265. What does this Python code print? Application

```
x = 4
x **= 2
x //= 3
print(x)
```

A. 16
B. 5.333333333333333
C. 5
D. 2

266. What does this Python code print? Application

```
print(7.5 // 2)
```

A. 3
B. 3.75
C. 4.0
D. 3.0

267. What does this C++ code print? Application

```
int x = 10;
x %= 4;
x <<= 1;
std::cout << x;
```

A. 4
B. 2
C. 5
D. 20

268. What does this Python code print? Application

```
print(2 ** 10, 2 ^ 10)
```

A. 1024 1024
B. 1024 8
C. 20 8
D. 1024 12

269. What does this Java statement print? Application

```
System.out.println(7 / 2 * 2.0);
```

A. 7.0
B. 6
C. 6.0
D. 7

270. What does this Java code print? Application

```
char c = 'a';
System.out.println(c + 1);
```

A. b
B. a1
C. 97
D. 98

271. What does this Java code print? Application

```
char c = 'a';
c += 2;
System.out.println(c);
```

A. c
B. 99
C. a2
D. Nothing; it is a compile error because an int cannot be assigned to a char

272. What does this Python code print? Application

```
print(-8 >> 1, 8 >> 1, -9 >> 1)
```

A. -4 4 -4
B. -4 4 -5
C. -3 4 -4
D. -4 4 4

273. What does this Python code print? Application

```
print(True + True * 3)
```

A. 6
B. True
C. 4
D. 2

274. What value is stored in r by this C++ code? Application

```
int a = 5, b = 2;
double r = a / b;
```

A. 2.5
B. 3.0
C. 0.4
D. 2.0

275. What does this Java code print? Application

```
int x = 7;
String s = x > 5 ? x > 10 ? "big" : "mid" : "small";
System.out.println(s);
```

A. mid
B. big
C. small
D. Nothing; nested ?: without parentheses does not compile

276. What does this Python code print? Application

```
a, b = 3, 5
a, b = b, a + b
print(a, b)
```

A. 5 10
B. 5 8
C. 3 8
D. 8 5

277. What does this Python code print? Application

```
x = 5
x += x * 2
print(x)
```

A. 20
B. 10
C. 25
D. 15

278. What does this Python code print? Application

```
x = 0b10110
print(x & -x)
```

A. 2
B. 22
C. 16
D. 0

279. What does this JavaScript code print? Application

```
console.log(null ?? "d", 0 ?? "d", 0 || "d");
```

A. d d d
B. d 0 d
C. null 0 d
D. d 0 0

280. Java saves the value of the left operand of += before it evaluates the right side. What does this Java code print? Application

```
int x = 3;
x += x++ * 2;
System.out.println(x);
```

A. 10
B. 11
C. 9
D. 8

281. What does this Python code print? Application

```
print(5 if 0 else 3 if None else 1)
```

A. 5
B. 3
C. None
D. 1

282. Java evaluates operands left to right. What does this Java code print? Application

```
int a = 10, b = 3;
System.out.println(a++ / b + --a % b);
```

A. 4
B. 5
C. 3
D. 6

283. A student wants the average of a and b. What does this Python code print, and why? Analysis

```
a, b = 4, 8
avg = a + b / 2
print(avg)
```

A. 6.0, because the expression is evaluated left to right
B. 8.0, because / binds tighter than +, so it computes 4 + 4.0
C. 6, because both operands are ints
D. 8, because / on ints truncates

284. What happens when Python runs this code? Analysis

```
x = 3
if x = 5:
    print("five")
```

A. It prints five, because x is set to 5 and 5 is truthy
B. It prints nothing, because 3 is not equal to 5
C. A SyntaxError is raised before anything runs, because = is a statement and cannot be a condition
D. It prints five, and x stays 3

285. This Java code is meant to store 3.5 in avg but stores 3.0. Which change fixes it? Analysis

```
int total = 7;
double avg = total / 2;
```

A. Write (double) (total / 2)
B. Declare avg as float instead of double
C. Write total / 2 + 0.5
D. Write total / 2.0

286. What does this JavaScript code print, and why? Analysis

```
console.log(0.1 + 0.2 === 0.3);
```

A. false, because 0.1 and 0.2 have no exact binary representation, so their sum is 0.30000000000000004
B. true, because === compares values only
C. false, because === also compares types and the sum has a different type
D. true, because JavaScript rounds results to 15 digits before comparing

287. What does this Python code print, and why? Analysis

```
x = [1, 2]
y = x
y += [3]
print(x)
```

A. [1, 2], because += creates a new list for y
B. [1, 2, 3], because += on a list extends it in place and x and y name the same list
C. [1, 2, [3]], because += appends the right-hand list as one element
D. A TypeError, because lists do not support +=

288. a is 3 and b is 4. What does this C++ code print, and why? Analysis

```
int a = 3, b = 4;
std::cout << (a < b < 2);
```

A. 0, because 4 is not less than 2
B. Nothing; it does not compile, because comparisons cannot be chained
C. 1, because a < b gives true, which converts to 1, and then 1 < 2 is true
D. 0, because C++ reads it as a < b && b < 2

289. What does this Python code print, and why? Analysis

```
print(10 / 5 == 2, type(10 / 5).__name__)
```

A. True int
B. False float
C. False int
D. True float

290. What does this Java code print, and why? Analysis

```
int c = 0;
if (c > 0 & ++c > 0) { }
System.out.println(c);
```

A. 1, because & on booleans evaluates both sides, so ++c runs
B. 0, because the left side is false and the right side is skipped
C. Nothing; it does not compile, because & needs int operands
D. 2, because ++c runs once for each comparison

291. Unary + converts its operand to a number. What does this JavaScript code print? Analysis

```
console.log(+"", +"3a", +"42");
```

A. NaN NaN 42
B. 0 NaN 42
C. 0 3 42
D. NaN 3 42

292. What does this Java code print? Analysis

```
int i = 10;
i -= i - 3;
System.out.println(i);
```

A. -3
B. 7
C. 3
D. 13

293. What does this Python code print? Analysis

```
def f(v):
    print(v, end=" ")
    return v

print(f(0) or f(2) and f(0))
```

A. 0 2 0
B. 2 0 2
C. 0 0
D. 0 2 0 0

294. x is 4. The C++ statement std::cout << (x & 1 == 0); is written in Python as below. What does each version print? Analysis

```
x = 4
print(x & 1 == 0)
```

A. Python prints True; C++ prints 0
B. Python prints True; C++ prints 1
C. Python prints False; C++ prints 0
D. Python prints False; C++ prints 1

295. What does this Java code print, and why? Analysis

```
int lo = 2_000_000_000, hi = 2_100_000_000;
int mid = (lo + hi) / 2;
System.out.println(mid);
```

A. 2050000000, because Java widens the sum to long
B. -97483648, because lo + hi overflows int and wraps to a negative number before the division
C. An ArithmeticException is thrown
D. 1073741823, because the sum is capped at Integer.MAX_VALUE

296. What does this Java code print, and why? Analysis

```
System.out.println(true ? 1 : 2.0);
```

A. 1
B. 2.0
C. 1.0
D. Nothing; it does not compile, because the branches have different types

297. Java evaluates operands left to right. What does this Java code print? Analysis

```
int x = 5;
x = x++ * 2 + x;
System.out.println(x);
```

A. 15
B. 17
C. 6
D. 16

Control Structures MCQs for NSCT

298. In C++ and Java, which loop always runs its body at least once, even when the condition is false from the start? Understanding

A. do-while
B. while
C. for
D. Range-based for (for-each)

299. In C++ and Java, what happens when a case inside a switch statement has no break? Understanding

A. The compiler reports an error
B. Execution falls through into the statements of the next case
C. The switch exits automatically after that case
D. The next case is skipped and the default runs

300. In Python, what does the continue statement do inside a loop? Understanding
    A. Exits the loop completely
    B. Restarts the loop from its first iteration
    C. Returns from the enclosing function
    D. Skips the rest of the current iteration and moves to the next one

301. Which Python statement does nothing and is used where a block is syntactically required but no action is needed? Understanding
    A. continue
    B. break
    C. skip
    D. pass

302. In Python, when does the else block attached to a for loop run? Understanding
    A. Only when the loop body never runs
    B. When the loop finishes normally without executing break
    C. Only when break is executed
    D. After every iteration of the loop

303. Which type of expression can a standard C++ switch statement test? Understanding
    A. An integral or enumeration type, such as int, char or an enum
    B. Any type, including std::string
    C. Floating-point types such as double
    D. Only int, not char or enum

304. Since Java 7, a Java switch statement can test a value of which of these types, which a C++ switch cannot? Understanding
    A. double
    B. boolean
    C. String
    D. long

305. Which pair of C++ loops are both entry-controlled, meaning the condition is tested before the body runs? Understanding
    A. do-while and while
    B. do-while and for
    C. do-while only
    D. for and while

306. In C++ and Java, an else written without braces after two nested if statements pairs with which if? Understanding
    A. The outermost if
    B. The first if in the function
    C. Neither; the code is a compile error
    D. The nearest preceding if that does not already have an else

307. How does Python's match statement (Python 3.10+) differ from a C++ or Java switch without break? Understanding
    A. Only the first matching case runs; there is no fall-through
    B. It falls through to the next case unless you write break
    C. It runs every case whose pattern matches
    D. It requires a default case or raises an error

308. What is the Python equivalent of the C++/Java conditional expression cond ? a : b? Understanding

A. cond ? a : b
B. if cond then a else b
C. a if cond else b
D. iif(cond, a, b)

309. In Java, outer labels the outer of two nested loops. What does break outer; do when executed inside the inner loop? Understanding

A. Ends only the inner loop
B. Jumps back to the label and restarts the outer loop
C. Ends the labelled outer loop, so both loops stop
D. Nothing; Java does not support labelled break

310. Python has no do-while loop. Which pattern gives the same 'run at least once' behaviour? Understanding

A. A for loop with an else clause
B. while condition: with continue as the first statement
C. A while loop whose condition is negated
D. while True: with the body first and if not condition: break at the end

311. In JavaScript, how does a switch statement compare the switch value with each case value? Understanding

A. Strict equality (===), with no type conversion
B. Loose equality (==), converting types first
C. By converting both values to strings
D. By Object.is, so NaN matches NaN

312. In Java, why is if (p != null && p.size() > 0) safe even when p is null? Understanding

A. Java ignores method calls on null references inside if conditions
B. && evaluates its right side only when the left side is true, so p.size() is never called on null
C. && evaluates both sides but catches the exception
D. The compiler reorders the conditions to check null first

313. A loop variable i is declared in the loop header (for (int i = 0; ...) in C++/Java, for i in ... in Python). Which statement about using i after the loop is correct? Understanding

A. i is still usable after the loop in all three languages
B. i is out of scope after the loop in all three languages
C. C++/Java: i is out of scope after the loop; Python: i still exists and holds its last value
D. C++/Java: i keeps its last value; Python: i is deleted

314. What does this Python code print? Application

```
for i in range(2, 11, 3):
    print(i, end=" ")
```

A. 2 5 8 11
B. 2 3 4 5 6 7 8 9 10
C. 2 5 8
D. 3 6 9

315. How many times does count += 1 run in this Python code, i.e. what is printed? Application

```
count = 0
for i in range(5):
    for j in range(i):
        count += 1
print(count)
```

A. 25
B. 10
C. 15
D. 5

316. What does this Java code print? Application

```
int x = 2, y = 0;
switch (x) {
    case 1: y += 1;
    case 2: y += 2;
    case 3: y += 3; break;
    default: y += 10;
}
System.out.println(y);
```

A. 2
B. 5
C. 15
D. 6

317. What does this Python code print? Application

```
for i in range(1, 10):
    if i % 2 == 0:
        continue
    if i > 6:
        break
    print(i, end=" ")
```

A. 1 3 5 7
B. 1 3 5 7 9
C. 2 4 6
D. 1 3 5

318. What does this C++ code print? Application

```
int n = 5, c = 0;
while (n-- > 0) c++;
cout << c << " " << n;
```

A. 5 0
B. 4 -1
C. 4 0
D. 5 -1

319. What does this Python code print? Application

```
i = 1
while i < 50:
    i *= 3
print(i)
```

A. 27
B. 243
C. 81
D. 50

320. What does this Python code print? Application

```
for n in [3, 5, 7]:
    if n % 2 == 0:
        print("even found")
        break
else:
    print("no even")
```

A. even found
B. even found then no even
C. no even
D. Nothing

321. What does this JavaScript code print? Application

```
let i = 10;
do {
  i++;
} while (i < 5);
console.log(i);
```

A. 10
B. 5
C. Nothing; the loop body never runs
D. 11

322. What does this Python code print? Application

```
count = 0
for i in range(3):
    for j in range(3):
        if j == i:
            break
        count += 1
print(count)
```

A. 9
B. 3
C. 6
D. 0

323. What does this Python code print? Application

```
marks = 85
if marks >= 50:
    grade = "C"
elif marks >= 70:
    grade = "B"
elif marks >= 80:
    grade = "A"
print(grade)
```

A. C
B. A
C. B
D. ABC

324. What does this C++ code print? (Indentation is misleading on purpose.) Application

```
int a = 5, b = 3;
if (a > 10)
    if (b > 1) cout << "X";
else cout << "Y";
cout << "Z";
```

A. Z
B. YZ
C. XZ
D. XYZ

325. What does this JavaScript code print? Application

```
const v = "1";
switch (v) {
  case 1: console.log("number"); break;
  case "1": console.log("string"); break;
  default: console.log("none");
}
```

A. number
B. string
C. none
D. number then string

326. What does this Python code print? Application

```
for i in range(3):
    print(i, end=" ")
    i = 10
```

A. 0 10
B. 0
C. 0 11 12
D. 0 1 2

327. What does this Java code print? Application

```
int count = 0;
for (int i = 1; i <= 100; i *= 2) count++;
System.out.println(count);
```

A. 7
B. 6
C. 100
D. 50

328. This Python code is meant to print 1 to 5. What actually happens? Analysis

```
i = 1
while i <= 5:
    print(i)
i += 1
```

A. It prints 1 to 5
B. It prints 1 once and stops
C. It prints 1 forever, because i += 1 is outside the loop body
D. It raises an IndentationError

329. What does this C++ code print, and what is x afterwards? Analysis

```
int x = 5;
if (x = 0) cout << "zero";
else cout << "nonzero";
```

A. zero, and x becomes 0
B. nonzero, and x stays 5
C. Nothing; it is a compile error in C++
D. nonzero, and x becomes 0

330. What does this Java code print? Analysis

```
for (int i = 0; i < 3; i++);
{
    System.out.println("hi");
}
```

A. hi three times
B. hi once
C. Nothing
D. Nothing; it does not compile

331. This Python code should report whether nums contains any negative number. With nums = [-1, 4, 2] what does it print, and why? Analysis

```
nums = [-1, 4, 2]
found = False
for x in nums:
    if x < 0:
        found = True
    else:
        found = False
print(found)
```

A. True, because -1 is negative
B. False, because the loop never reaches -1
C. False, because the else branch resets found on each later non-negative value
D. True, because found can only change from False to True

332. What does this JavaScript code print, and why? Analysis

```
for (var i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 0);
}
```

A. 0 1 2, because each callback captures the value of i at that time
B. 0 0 0, because the callbacks run before i changes
C. 3 3 3, because var gives one shared i that is 3 when the callbacks run
D. undefined three times, because i is out of scope

333. Why does this C++ loop never terminate? Analysis

```
for (unsigned int i = 5; i >= 0; i--) {
    cout << i << " ";
}
```

A. i is unsigned, so i >= 0 is always true; decrementing 0 wraps round to the largest unsigned value
B. It does terminate, after 6 iterations
C. It does terminate, after 5 iterations
D. i becomes -1, which is undefined behaviour and crashes

334. This Python code is meant to print 5 4 3 2 1. What does it print, and why? Analysis

```
for i in range(5, 0):
    print(i, end=" ")
```

A. Nothing, because the default step is +1 and start is already past stop
B. 5 4 3 2 1
C. 5 4 3 2 1 0
D. It raises a ValueError

335. What does this Python code print? Analysis

```
n = 10
while n > 0:
    n -= 3
    if n == 4:
        break
else:
    print("done")
print(n)
```

A. 4
B. done then 4
C. -2
D. done then -2

336. With score = 30, what does this C++ code print, and why? Analysis

```
int score = 30;
if (score > 50);
    cout << "Pass";
```

A. Nothing, because 30 > 50 is false
B. Pass, because the semicolon ends the if, so the output line always runs
C. Pass twice
D. It does not compile

337. Which Python expression gives r the same value as this nested if/else for every pair of integers a and b? Analysis

```
if a > 0:
    if b > 0:
        r = 1
    else:
        r = 2
else:
    r = 3
```

A. 1 if a > 0 and b > 0 else (2 if b <= 0 else 3)
B. 1 if a > 0 and b > 0 else (2 if a > 0 else 3)
C. 1 if b > 0 else (2 if a > 0 else 3)
D. 1 if a > 0 or b > 0 else (2 if a > 0 else 3)

338. Which loop is the usual choice when the number of iterations is known before the loop starts? Understanding

A. A for loop
B. A do-while loop
C. A while (true) loop with a break
D. A switch statement

339. In C++ and Java, what does break do when it runs inside a loop? Understanding

A. Skips to the next iteration of the loop
B. Exits the innermost enclosing loop immediately
C. Exits every enclosing loop at once
D. Ends the whole program

340. The structured programming theorem (Böhm and Jacopini) says any algorithm can be built from which three control structures? Understanding

A. Input, processing and output
B. Functions, classes and modules
C. Sequence, selection and iteration
D. if, switch and goto

341. In Python, which keyword adds another condition to an if chain, playing the role of else if in C++ and Java? Understanding

A. else if
B. elseif
C. case
D. elif

342. In a C++ or Java switch statement, when does the code under the default label run? Understanding

A. When no case label matches the switch value, or when an earlier case falls through into it
B. Always, after the matching case finishes
C. Only when the switch value is 0
D. Before any case is tested

343. Which C++/Java loop header creates an infinite loop? Understanding

A. for (int i = 0; i < 10; i++)
B. for (;;)
C. while (false)
D. for (int i = 10; i > 0; i--)

344. In Python, which values does range(5) produce? Understanding

A. 1, 2, 3, 4, 5
B. 0, 1, 2, 3, 4, 5
C. 0, 1, 2, 3, 4
D. 5, 4, 3, 2, 1

345. Which statement ends the current function at once, even when it runs inside a nested loop? Understanding

A. break
B. continue
C. pass
D. return

346. Why does if (1) { ... } compile in C++ but not in Java? Understanding

A. Java requires an if condition of type boolean, while C++ converts any non-zero number to true
B. Java does not allow literals in conditions
C. C++ treats 1 as a label, not a value
D. Java requires braces around the condition

347. In Python, which value makes if value: take the else branch? Understanding

A. "0" (a string holding a zero)
B. [] (an empty list)
C. [0] (a list holding a zero)
D. " " (a string holding a space)

348. How does a Java 14+ switch with arrow labels (case 1 -> ...) differ from a classic switch with colon labels? Understanding
    A. Arrow labels work only with String values
    B. Arrow labels run every case after the matching one
    C. Arrow labels never fall through, so no break is needed
    D. Arrow labels require default to be the first case

349. What happens if a C++ or Java switch statement has two case labels with the same constant value? Understanding
    A. Both cases run, in order
    B. Only the first matching case runs
    C. Only the last matching case runs
    D. It is a compile-time error

350. In Java, outer labels the outer of two nested for loops. What does continue outer; do when it runs inside the inner loop? Understanding
    A. Ends the current inner loop and starts the next iteration of the outer loop
    B. Exits both loops completely
    C. Restarts the inner loop from its first iteration
    D. Skips the rest of the current inner iteration and continues the inner loop

351. In C++, v is a std::vector<int>. Why does for (int x : v) x *= 2; leave v unchanged? Understanding
    A. Range-based for loops make v read-only
    B. x is a copy of each element; writing for (int& x : v) binds x to the elements so the changes stick
    C. x *= 2 is evaluated only after the loop finishes
    D. The loop body runs zero times for a vector

352. In a Python 3.10+ match statement, what does case _: do? Understanding
    A. It matches only None
    B. It matches only an empty string
    C. It matches any value, acting like default in a switch
    D. It raises an error if it is reached

353. In a while loop the counter is incremented at the bottom of the body, and a continue statement sits above that line. What is the risk? Understanding
    A. continue exits the loop, so later iterations never run
    B. continue makes the loop test its condition twice
    C. There is no risk, because continue always runs the increment first
    D. When continue runs, the increment is skipped, so the loop can repeat forever with the same counter value

354. In C++ or Java, which task suits an if/else-if chain better than a switch statement? Understanding
    A. Choosing a grade from score ranges such as marks >= 80
    B. Choosing an action from a menu number 1 to 5
    C. Mapping a char 'a' to 'e' to a message
    D. Dispatching on an enum value

355. In Python, what does for i, x in enumerate(items): give on each iteration? Understanding
    A. The element x and the number of times it occurs, i
    B. The index i and the element x
    C. Two consecutive elements, i and x
    D. The key i and value x of a dict

356. In Java, how does a switch on a String decide which case matches? Understanding

A. It compares references with ==, so only interned strings can match
B. It compares only the first character
C. It compares lengths first and picks the case with the same length
D. It compares the value with each case label using equals(), so content matters, and a null value throws NullPointerException

357. In Python 3.10+, limit = 5 was set earlier. Inside match n:, what does the pattern case limit: do? Understanding

A. It matches any value and binds it to the name limit, because a bare name is a capture pattern
B. It matches only when n equals 5
C. It raises a NameError because limit is already defined
D. It matches only when n refers to the same object as limit

358. Why is removing items from a list inside for x in lst: risky in Python? Understanding

A. Python raises RuntimeError whenever a list changes during iteration
B. The loop advances by position, so removing an item shifts later items left and the next one is skipped
C. The removed items are re-added at the end of the loop
D. The loop iterates over a copy, so the removals have no effect

359. In Java, what happens when a statement is placed directly after break; in the same block? Understanding

A. The statement runs after the loop ends
B. The statement runs once, then the loop exits
C. A compile-time error: unreachable statement
D. Only a warning is shown and the statement is skipped

360. In C++, a switch has case 1: int y = 5; ... followed by case 2: with no braces around the first case's code. Why does this fail to compile? Understanding

A. Variables cannot be declared anywhere inside a switch
B. A declaration must be preceded by break
C. int variables are not allowed in a switch on an int
D. Jumping to case 2 would bypass the initialisation of y, which is still in scope; wrapping the case body in braces fixes it

361. What does this Java code print? Application

```
int sum = 0;
for (int i = 1; i <= 5; i++) sum += i;
System.out.println(sum);
```

A. 10
B. 15
C. 14
D. 21

362. What does this Python code print? Application

```
x = 0
if x:
    print("yes")
else:
    print("no")
```

A. yes
B. yes, then no
C. no
D. Nothing; it raises a TypeError

363. What does this C++ code print? Application

```
int i = 0;
while (i < 10) i += 4;
std::cout << i;
```

A. 8
B. 10
C. 16
D. 12

364. What does this Python code print? Application

```
total = 0
for ch in "a1b2c3":
    if ch.isdigit():
        total += int(ch)
print(total)
```

A. 6
B. 123
C. 3
D. 0

365. What does this JavaScript code print? Application

```
let out = "";
for (let i = 0; i < 5; i++) {
  if (i === 3) break;
  out += i;
}
console.log(out);
```

A. 0124
B. 012
C. 0123
D. 01234

366. What does this Java code print? Application

```
int x = 5;
String s = "";
switch (x) {
    default: s += "D";
    case 1:  s += "1"; break;
    case 2:  s += "2";
}
System.out.println(s);
```

A. D
B. D12
C. D1
D. 1

367. What does this Python code print? Application

```
c = 0
for i in range(4):
    for j in range(4):
        if i >= j:
            continue
        c += 1
print(c)
```

A. 10
B. 16
C. 4
D. 6

368. What does this C++ code print? Application

```
int n = 100, steps = 0;
while (n > 1) {
    n /= 2;
    steps++;
}
std::cout << steps;
```

A. 6
B. 7
C. 5
D. 50

369. What does this Python code print? Application

```
i = 0
while True:
    i += 1
    if i % 3 == 0:
        continue
    if i > 7:
        break
    print(i, end=" ")
```

A. 1 2 4 5 7 8
B. 1 2 4 5 7
C. 1 2
D. 1 2 3 4 5 6 7

370. What does this Java code print? Application

```
int n = 0;
do {
    n += 2;
} while (n < 7);
System.out.println(n);
```

A. 6
B. 7
C. 8
D. 2

371. What does this Python 3.10+ code print? Application

```
cmd = "move"
match cmd:
    case "go" | "move":
        print("walking")
    case "stop":
        print("halt")
    case _:
        print("unknown")
```

A. unknown
B. walking, then unknown
C. halt
D. walking

372. What does this JavaScript code print? Application

```
let s = "";
for (const k of [1, 2, 3]) {
  s += k;
  if (k === 2) continue;
  s += "-";
}
console.log(s);
```

A. 1-23-
B. 1-2-3-
C. 1-2
D. 1-3-

373. What does this C++ code print? Application

```
int a = 7;
std::cout << (a % 2 == 0 ? "even" : a > 5 ? "big odd" : "small odd");
```

A. even
B. big odd
C. small odd
D. Nothing; nested ?: needs more parentheses to compile

374. What does this Python code print? Application

```
lst = [4, 8, 15]
for i in range(len(lst) - 1, -1, -1):
    print(lst[i], end=" ")
```

A. 15 8
B. 4 8 15
C. 15 8 4
D. 8 4

375. What does this Java code print? Application

```
int count = 0;
outer:
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        if (j == 2) continue outer;
        count++;
    }
}
System.out.println(count);
```

A. 6
B. 1
C. 9
D. 3

376. What does this Python code print? Application

```
x = 0
for i in range(3):
    for j in range(i, 3):
        x += i * j
print(x)
```

A. 7
B. 9
C. 5
D. 3

377. What does this C++ code print? Application

```
int i = 0, s = 0;
for (; i < 10; i += 3) {
    if (i % 2) continue;
    s += i;
}
std::cout << s << " " << i;
```

A. 6 9
B. 6 12
C. 18 12
D. 0 12

378. What does this Java code print? Application

```
int k = 3, t = 0;
switch (k) {
    case 1:
    case 3: t += 3;
    case 5: t += 5; break;
    case 7: t += 7;
}
System.out.println(t);
```

A. 3
B. 15
C. 8
D. 5

379. What does this Python code print? Application

```
n = 0
while n < 5:
    n += 1
    if n == 3:
        continue
else:
    print("done", n)
```

A. Nothing is printed, because continue cancels the else
B. done 3
C. done 6
D. done 5

380. What does this JavaScript code print? Application

```
let count = 0, i, j;
for (i = 0, j = 10; i < j; i += 2, j -= 3) count++;
console.log(count, i, j);
```

A. 2 4 4
B. 3 6 1
C. 2 2 7
D. 3 4 4

381. This Python code is meant to add the numbers 1 to 10. What does it print, and why? Analysis

```
total = 0
for i in range(1, 10):
    total += i
print(total)
```

A. 55, because range(1, 10) includes 10
B. 45, because range(1, 10) stops before 10
C. 10, because total is overwritten on each pass
D. 36, because range(1, 10) stops at 8

382. x is -2. What does this Java code print? Analysis

```
if (x > 0)
    System.out.println("positive");
    System.out.println("checked");
```

A. Nothing
B. positive, then checked
C. checked
D. positive

383. What does this Python code print? Analysis

```
i = 10
while i < 5:
    print(i)
    i += 1
print("end")
```

A. 10, then end
B. It loops forever, printing 10, 11, 12, ...
C. Nothing
D. end

384. How many lines does this C++ code print? Analysis

```
for (int i = 0; i <= 5; i++)
    std::cout << i << "\n";
```

A. 6
B. 5
C. 4
D. 7

385. What does this Python code print, and why? Analysis

```
nums = [2, 4, 5, 6]
for n in nums:
    if n % 2 == 0:
        nums.remove(n)
print(nums)
```

A. [5], because every even number is removed
B. [4, 5], because removing an item shifts the list and the next item is skipped
C. [4, 5, 6], because only the first even number is removed
D. A RuntimeError, because the list changed size during iteration

386. What does this Java code do? Analysis

```
int i = 0;
while (i < 5) {
    if (i == 2) continue;
    System.out.print(i + " ");
    i++;
}
```

A. It prints 0 1 3 4
B. It prints 0 1 2 3 4
C. It prints 0 1 and then loops forever
D. It prints 0 1 and then the loop ends

387. This Python code was meant to stop at the first pair adding up to 4. What does it print, and why? Analysis

```
found = 0
for i in range(1, 4):
    for j in range(1, 4):
        if i + j == 4:
            found += 1
            break
print(found)
```

A. 1, because break stops both loops after the first pair
B. 9, because break has no effect inside a for loop
C. 2, because i = 3 has no partner
D. 3, because break exits only the inner loop, so every i finds one pair

388. This Python code should print the total of all values, 10. What does it print, and why? Analysis

```
for row in [[1, 2], [3, 4]]:
    total = 0
    for v in row:
        total += v
print(total)
```

A. 7, because total is reset to 0 at the start of each row
B. 10, because total accumulates across both rows
C. 3, because only the first row is summed
D. 4, because total holds the last value added

389. What happens when this Python code runs? Analysis

```
n = 10
while n != 0:
    n -= 3
print(n)
```

A. It prints 0
B. It never ends, because n jumps from 1 to -2 and never equals 0
C. It prints 1
D. It prints -2

390. A student used pass to skip printing 1. What does this Python code print? Analysis

```
for i in range(3):
    if i == 1:
        pass
    print(i, end=" ")
```

A. 0 2
B. 0
C. 0 1 2
D. Nothing

391. What does this JavaScript code print, and why? Analysis

```
const arr = [1, 2, 3];
let sum = 0;
for (const i in arr) sum += i;
console.log(sum);
```

A. 6
B. 3
C. 123
D. 0012

392. A guard was added to stop a runaway loop. What does this Java code print, and why? Analysis

```
int n = 0;
for (double d = 0.0; d != 1.0; d += 0.1) {
    n++;
    if (n > 20) break;
}
System.out.println(n);
```

A. 21, because adding 0.1 ten times gives 0.9999999999999999, so d != 1.0 stays true
B. 10, because d reaches exactly 1.0 after 10 steps
C. 11, because the loop test runs one extra time
D. 20, because the guard breaks before n is incremented

393. Which while loop prints exactly the same values as this Python for loop? Analysis

```
for i in range(2, 22, 5):
    print(i)
```

A. Start i at 2; while i <= 22, print i and then add 5
B. Start i at 2; while i < 22, print i and then add 5
C. Start i at 2; while i < 22, add 5 and then print i
D. Start i at 0; while i < 22, print i + 2 and then add 5

394. What does this Python code print? Analysis

```
for n in range(2, 10):
    for d in range(2, n):
        if n % d == 0:
            break
    else:
        print(n, end=" ")
```

A. 2 3 4 5 6 7 8 9
B. 4 6 8 9
C. 2 3 5 7
D. 3 5 7

395. What does this Java code print? Analysis

```
int s = 0;
for (int i = 0; i < 4; i++) {
    switch (i % 3) {
        case 0: s += 1;
        case 1: s += 10; break;
        default: s += 100;
    }
}
System.out.println(s);
```

A. 112
B. 11
C. 121
D. 132

Functions / Methods MCQs for NSCT

396. What is the difference between a parameter and an argument? Understanding

A. An argument is the variable in the definition; a parameter is the value passed in the call
B. They are two names for exactly the same thing in every context
C. Parameters belong to methods and arguments belong to free functions
D. A parameter is the variable in the function definition; an argument is the value passed in the call

397. How does Java pass arguments to methods? Understanding

A. Primitives by value and objects by reference
B. Always by reference
C. Always by value; for an object, the value copied is the reference
D. By value for methods and by reference for constructors

398. Which description of Python argument passing is correct? Understanding

A. Every argument is copied, so the caller never sees any change
B. It works like C++ references: assigning to the parameter changes the caller's variable
C. The parameter is bound to the same object as the argument; mutating a mutable object is visible to the caller, rebinding the parameter is not
D. Immutable objects are passed by reference and mutable objects are copied

399. Which C++ function declaration lets the function change the caller's int variable directly? Understanding

A. void f(int x);
B. void f(const int x);
C. void f(const int &x);
D. void f(int &x);

400. In a recursive function, what is the base case? Understanding

A. The first call made from main
B. A condition under which the function returns without calling itself
C. The call with the largest input
D. The line that contains the recursive call

401. What happens in CPython when a recursive function never reaches its base case? Understanding

A. It runs forever without any error
B. It returns None after the limit is reached
C. A SyntaxError is reported before the program runs
D. A RecursionError is raised once the recursion limit (about 1,000 frames by default) is exceeded

402. In Java, which of these can distinguish two overloaded methods with the same name? Understanding

A. Their return types alone
B. The number, types or order of their parameters
C. Their parameter names alone
D. Their access modifiers alone

403. What happens when a Python module defines two functions with the same name but different numbers of parameters? Understanding

A. Python keeps both and chooses by argument count, like Java
B. Python raises a SyntaxError for the duplicate name
C. The second def replaces the first, so only the later version exists
D. typing.overload makes both run as separate implementations

404. What is the purpose of a function prototype in C++? Understanding

A. It allocates memory for the function's local variables
B. It declares the function's name, return type and parameter types so calls can be checked before the definition appears
C. It provides the function body
D. It forces the compiler to inline the function

405. Which rule about default parameter values holds in both C++ and Python? Understanding

A. Defaults may be given to any parameters in any order
B. Only the first parameter may have a default
C. Once a parameter has a default, every parameter after it must have one too
D. Default values are allowed only in methods, not in free functions

406. In Python, def f(*args): collects extra positional arguments into which type? Understanding

A. A tuple
B. A list
C. A dict
D. A set

407. When is the default value [] in def add(item, bag=[]): evaluated in Python? Understanding

A. Once, when the def statement runs
B. Every time add is called without bag
C. On the first call only, then cached per caller
D. Each time the module is imported by another file

408. What does a Python function return if it ends without a return statement? Understanding

A. 0
B. An empty string
C. It raises an error
D. None

409. How is a static method in Java usually called? Understanding

A. Only through an object of the class
B. Only from inside main
C. Through the class name, without creating an object
D. Only from a subclass

410. Which statement about tail recursion is correct? Understanding

A. The recursive call is the last action, but CPython and the standard JVM do not eliminate tail calls, so depth is still limited
B. Python automatically turns tail recursion into a loop
C. A tail-recursive function never needs a base case
D. Java guarantees tail-call optimisation for static methods

411. A variable is declared inside a function body in C++, Java or Python. Which statement is correct? Understanding

A. It is shared by all calls, like a global
B. It keeps its value between calls automatically
C. Functions called from this function can read it by name
D. It is local: each call gets its own copy, which disappears when the call returns

412. What does this Python code print? Application

```
def f(a, b=2, c=3):
    return a + b * c

print(f(1, c=4))
```

A. 7
B. 9
C. 11
D. Error: c must be passed by position

413. What does this Python code print? Application

```
def add(x):
    x.append(4)

nums = [1, 2, 3]
add(nums)
print(len(nums))
```

A. 3
B. 4
C. 1
D. None

414. What does this Python code print? Application

```
def change(x):
    x = [9]

nums = [1]
change(nums)
print(nums)
```

A. [9]
B. [1, 9]
C. None
D. [1]

415. What does this Java code print? Application

```
static void swap(int a, int b) {
    int t = a; a = b; b = t;
}
// in main:
int x = 1, y = 2;
swap(x, y);
System.out.println(x + " " + y);
```

A. 2 1
B. 2 2
C. 1 1
D. 1 2

416. What does this C++ code print? Application

```
void swap(int &a, int &b) {
    int t = a; a = b; b = t;
}
// in main:
int x = 1, y = 2;
swap(x, y);
cout << x << " " << y;
```

A. 1 2
B. 2 2
C. 2 1
D. 1 1

417. What does this Java code print? Application

```
class Box { int v; }

static void f(Box b) {
    b.v = 5;
    b = new Box();
    b.v = 9;
}
// in main:
Box x = new Box();
x.v = 1;
f(x);
System.out.println(x.v);
```

A. 5
B. 1
C. 9
D. 0

418. What does this Python code print? Application

```
def f(n):
    return 1 if n <= 1 else n * f(n - 1)

print(f(5))
```

A. 120
B. 24
C. 15
D. 720

419. What does this Python code print? Application

```
def p(n):
    if n == 0:
        return
    p(n - 1)
    print(n, end=" ")

p(3)
```

A. 1 2 3
B. 3 2 1
C. 0 1 2 3
D. 3 2 1 0

420. How many times is fib called in total, including the first call, when this Python code runs fib(5)? Application

```
def fib(n):
    if n < 2:
        return n
    return fib(n - 1) + fib(n - 2)
```

A. 5
B. 15
C. 9
D. 8

421. What does this Python code print? Application

```
x = 5

def f():
    global x
    x = x + 1
    return x

f()
print(f(), x)
```

A. 6 6
B. 7 7
C. 7 6
D. 6 5

422. What does this Python code print? Application

```
def add(item, bag=[]):
    bag.append(item)
    return bag

add(1)
print(add(2))
```

A. [2]
B. [1]
C. []
D. [1, 2]

423. What does this JavaScript code print? Application

```
function g(obj) {
  obj.n++;
  obj = { n: 100 };
}
let o = { n: 1 };
g(o);
console.log(o.n);
```

A. 2
B. 1
C. 100
D. 101

424. What does this Python code print? Application

```
def f(a, *args, **kw):
    return len(args), len(kw)

print(f(1, 2, 3, x=4))
```

A. (3, 1)
B. (2, 0)
C. (4, 0)
D. (2, 1)

425. What does this C++ code print? Application

```
void inc(int *p) {
    (*p)++;
}
// in main:
int a = 5;
inc(&a);
cout << a;
```

A. 5
B. The address of a
C. 0
D. 6

426. This Python function should return 1 + 2 + ... + n. What happens when total(3) is called? Analysis

```
def total(n):
    if n == 0:
        return 0
    n + total(n - 1)

print(total(3))
```

A. It returns 6
B. A TypeError, because the recursive branch returns None and a level above adds an int to None
C. It returns None without any error
D. A RecursionError, because the base case is never reached

427. This Java method is meant to clear an array. What does the code print, and why? Analysis

```
static void reset(int[] a) {
    a = new int[a.length];
}
// in main:
int[] arr = {1, 2, 3};
reset(arr);
System.out.println(arr[0]);
```

A. 0, because the array was replaced with zeros
B. 1, because assigning a new array to the parameter does not affect the caller's reference
C. It throws a NullPointerException
D. It does not compile, because arrays cannot be parameters

428. What happens when this Python code runs? Analysis

```
def count_down(n):
    if n == 0:
        return
    print(n)
    count_down(n - 2)

count_down(5)
```

A. It prints 5, 3, 1 and stops
B. It prints 5, 3, 1, 0 and stops
C. It prints 5, 3, 1, then raises RecursionError because n skips past 0
D. It raises RecursionError before printing anything

429. def add(item, bag=[]) keeps items from earlier calls. Which change gives each call a fresh list when bag is omitted? Analysis

A. Use bag=None and write if bag is None: bag = [] inside the function
B. Use bag=list() as the default
C. Declare bag as global inside the function
D. Use bag=[] and call bag.clear() at the end of the function

430. What does this Java code print, and why? Analysis

```
static void up(String s) {
    s = s.toUpperCase();
}
// in main:
String t = "pk";
up(t);
System.out.println(t);
```

A. pk, because strings are immutable and s = ... only rebinds the parameter
B. PK, because objects are passed by reference
C. null, because the parameter is reassigned
D. It does not compile

431. For which non-negative inputs n does this Python function fail to terminate normally? Analysis

```
def g(n):
    if n == 0:
        return 0
    return g(n - 2)
```

A. Every even n
B. Only n = 1
C. Every odd n, because n steps over 0 and keeps going negative
D. None; it always returns 0

432. What happens when this Python code runs? Analysis

```
count = 0

def inc():
    count += 1

inc()
print(count)
```

A. count becomes 1, because globals are visible inside functions
B. It prints nothing and count stays 0
C. UnboundLocalError, because the assignment makes count local to inc
D. A SyntaxError when the function is defined

433. What happens when this Java code is compiled? Analysis

```
static void f(int a, double b) { System.out.println("A"); }
static void f(double a, int b) { System.out.println("B"); }
// in main:
f(1, 2);
```

A. Compile error: the call is ambiguous because both methods apply and neither is more specific
B. It prints A
C. It prints B
D. It prints AB

434. A naive recursive Python fib(n) takes a very long time for n = 40. What is the main cause, and what fixes it? Analysis

A. Python's recursion limit slows each call; raising the limit fixes it
B. The same subproblems are recomputed exponentially many times; memoisation or a loop makes it linear
C. Integer overflow in large Fibonacci numbers; using floats fixes it
D. Printing inside the function; removing print fixes it

435. What does this JavaScript code print? (Contrast it with Python's mutable default.) Analysis

```
function add(item, bag = []) {
  bag.push(item);
  return bag;
}
add(1);
console.log(add(2).length);
```

A. 2, because the default array is shared between calls
B. 0, because push returns a new array
C. 1, because a default parameter is evaluated afresh on each call
D. undefined, because bag is not passed

436. In C++, what does the return type void in void show(int n) mean? Understanding

A. The function returns 0 by default
B. The function returns a null pointer
C. The function may return a value of any type
D. The function returns no value to the caller

437. In Python, what does the expression lambda n: n * 2 create? Understanding

A. An anonymous function object whose body is a single expression
B. A named function that can contain several statements
C. A new class with one method called n
D. A generator that yields n * 2 repeatedly

438. What is stored in a function call's activation record (stack frame)? Understanding

A. That call's parameters, local variables and return address
B. The machine code of every function in the program
C. Only the global variables of the program
D. The objects that the call allocates with new

439. In Python, def f(**kwargs): collects extra keyword arguments into which type? Understanding

A. tuple
B. list
C. set
D. dict

440. Function f calls g, and g calls f, until a base case stops them. What is this called? Understanding

A. Mutual (indirect) recursion
B. Tail recursion
C. Function overloading
D. Nested functions

441. Which description fits a pure function? Understanding

A. It takes no parameters
B. It returns nothing
C. It always returns the same result for the same arguments and has no side effects
D. It is defined inside a class

442. In Python, a function ends with return total, count. What does the caller receive? Understanding

A. Only total; count is ignored
B. A SyntaxError, because return takes one value
C. A list holding both values
D. One tuple holding both values

443. In Java, what is method overriding? Understanding

A. A class defines two methods with the same name but different parameter lists
B. A method calls itself until a base case is reached
C. A method changes only its return type to replace another method
D. A subclass gives its own body for an inherited method with the same signature

444. In Java, static int sum(int... nums) is declared. Inside the method, what is the type of nums? Understanding

A. List<Integer>
B. int[]
C. int
D. Iterator<Integer>

445. What does it mean that functions are first-class objects in Python? Understanding

A. They always run before any other code in the file
B. They can be stored in variables, passed as arguments and returned from other functions
C. They cannot be defined inside other functions
D. They must be declared at the top of the file before use

446. In C++, what does the parameter declaration const string &s give the function? Understanding

A. A private copy that the function may not change
B. No copy of the argument, and no permission to modify it
C. Direct access to change the caller's string
D. A global string shared by all calls

447. What does the Python keyword nonlocal do inside a nested function? Understanding

A. It makes the variable global for the whole module
B. It creates a fresh local variable that hides the outer one
C. It lets the inner function rebind a variable of the nearest enclosing function
D. It deletes the variable from the enclosing function

448. Following Python's LEGB rule, in what order is a name looked up? Understanding

A. Local, Enclosing, Global, Built-in
B. Global, Local, Enclosing, Built-in
C. Built-in, Global, Enclosing, Local
D. Local, Global, Enclosing, Built-in

449. In Python, def f(a, /, b, *, c): is defined. Which call is valid? Understanding

A. f(a=1, b=2, c=3)
B. f(1, 2, 3)
C. f(1, 2, c=3)
D. f(1, c=3)

450. Java has no default parameter values. What is the usual way to get the same effect? Understanding

A. Write = value after the parameter, as in C++
B. Declare the parameter as final
C. Overload the method: a shorter version calls the full one with the default filled in
D. Make the parameter static

451. In C++, void f(int a[]) is called as f(arr). What does the function actually receive? Understanding

A. A full copy of the array
B. A copy of only the first element
C. The array's length and a copy of its elements
D. A pointer to arr's first element, so changes to a[i] affect arr

452. What does marking a C++ function inline do? Understanding

A. It forces every call to be replaced by the body
B. It makes the function run in a separate thread
C. It asks the compiler to expand the body at the call site; the compiler may ignore the request
D. It makes the function visible only inside its own file

453. A non-tail recursive Python function sums a list of n numbers with one recursive call per element. What is its extra space use? Understanding

A. O(1), because each call finishes immediately
B. O(n), because n stack frames are alive at the deepest point
C. O(log n), because the list is halved each time
D. O(n^2), because each frame copies the whole list

454. In Java, which of these is NOT allowed directly inside a static method? Understanding

A. Calling another static method of the same class
B. Declaring a local variable
C. Creating a new object of the same class
D. Using this or reading an instance field without an object

455. A recursive function does O(1) work and then makes two calls on n - 1, so T(n) = 2T(n - 1) + O(1). What is its time complexity? Understanding

A. O(2^n)
B. O(n)
C. O(n log n)
D. O(n^2)

456. In a Python def, where may **kwargs appear in the parameter list? Understanding

A. Anywhere, as long as it comes after self
B. Only as the first parameter
C. Only as the last parameter
D. Before *args, so keyword arguments are bound first

457. In C++, a file defines both void f(int x) { } and void f(const int x) { }. What happens? Understanding

A. A compile error: top-level const on a by-value parameter does not change the signature, so f is defined twice
B. Both compile, and f(5) picks the const version
C. Both compile, and the choice depends on whether the argument is a constant
D. Both compile, and f(5) is ambiguous only at run time

458. What does the Java compiler do with this method? Understanding

```
static int sign(int n) {
    if (n > 0) return 1;
    else if (n < 0) return -1;
}
```

A. Compiles it and returns 0 when n is 0
B. Compiles it and throws an exception at run time when n is 0
C. Rejects it with a missing return statement error
D. Compiles it and returns null when n is 0

459. What does this Python code print? Application

```
def area(w, h=1):
    return w * h

print(area(h=3, w=2))
```

A. 2
B. 3
C. TypeError
D. 6

460. What does this Python code print? Application

```
def f():
    return 1, 2

x = f()
print(type(x).__name__)
```

A. list
B. int
C. tuple
D. NoneType

461. What does this Python code print? Application

```
sq = lambda n: n * n
print(sq(sq(2)))
```

A. 4
B. 16
C. 8
D. 256

462. What does this Python code print? Application

```
def show(*args):
    print(len(args))

show(1, [2, 3], "ab")
```

A. 3
B. 5
C. 4
D. 1

463. What does this C++ code print? Application

```
int sq(int x) { return x * x; }
// in main:
cout << sq(3) + sq(sq(2));
```

A. 13
B. 18
C. 25
D. 81

464. What does this Java code print? Application

```
static int sum(int... nums) {
    int s = 0;
    for (int v : nums) s += v;
    return s;
}
// in main:
System.out.println(sum() + sum(1, 2, 3));
```

A. 0
B. A compile error, because sum() needs at least one argument
C. 6
D. 7

465. What does this Python code print? Application

```
def add(a, b):
    return a + b

nums = [3, 4]
print(add(*nums))
```

A. 7
B. [3, 4]
C. TypeError
D. 34

466. What does this Python code print? Application

```
def f(a, b, c):
    return a - b - c

d = {"c": 1, "a": 10, "b": 2}
print(f(**d))
```

A. -11
B. 7
C. 9
D. TypeError

467. What does this Python code print? Application

```
def make():
    n = 0
    def inc():
        nonlocal n
        n += 1
        return n
    return inc

a = make()
b = make()
a()
a()
print(a(), b())
```

A. 1 1
B. 3 1
C. 3 3
D. 2 1

468. What does this Python code print? Application

```
def r(n):
    if n > 0:
        print(n, end=" ")
        r(n - 1)
        print(n, end=" ")

r(3)
```

A. 3 2 1 3 2 1
B. 3 2 1
C. 1 2 3 3 2 1
D. 3 2 1 1 2 3

469. What does this Python code print? Application

```
def g(a, b):
    return a if b == 0 else g(b, a % b)

print(g(48, 18))
```

A. 18
B. 6
C. 3
D. 2

470. What does this Java code print? Application

```
class C {
    static int n = 0;
    C() { n++; }
}
// in main:
new C();
new C();
C c = new C();
System.out.println(c.n + C.n);
```

A. 3
B. 2
C. A compile error, because c.n is static
D. 6

471. What does this C++ code print? Application

```
int f(int a, int b = 10, int c = 100) {
    return a + b + c;
}
// in main:
cout << f(1) << " " << f(1, 2);
```

A. 111 103
B. 111 3
C. 1 3
D. 103 111

472. What does this C++ code print? Application

```
void p(int x)    { cout << "int "; }
void p(double x) { cout << "double "; }
void p(char x)   { cout << "char "; }
// in main:
p(5); p(5.0); p('5');
```

A. int double int
B. int double char
C. int int int
D. double double char

473. What does this C++ code print? Application

```
int counter() {
    static int c = 0;
    return ++c;
}
// in main:
counter();
counter();
cout << counter();
```

A. 1
B. 0
C. 3
D. 2

474. What does this Python code print? Application

```
fs = [lambda: i for i in range(3)]
print([f() for f in fs])
```

A. [0, 1, 2]
B. [0, 0, 0]
C. [2, 2, 2]
D. NameError, because i no longer exists

475. What does this JavaScript code print? Application

```
function f(a, b = 5) {
  return a + b;
}
console.log(f(1), f(1, undefined), f(1, null));
```

A. 6 6 6
B. 6 6 1
C. 6 NaN 1
D. 6 NaN NaN

476. What happens when this Python code runs? Application

```
def f(a, b):
    return a + b

print(f(1, a=2))
```

A. TypeError: f() got multiple values for argument 'a'
B. It prints 3
C. It prints 4
D. TypeError: f() missing 1 required positional argument: 'b'

477. What does this Python code print? Application

```
x = 10

def f(a=x):
    return a

x = 20
print(f())
```

A. 20
B. 10
C. None
D. NameError

478. What does this Python code print? Application

```
def flat(x):
    if not isinstance(x, list):
        return [x]
    out = []
    for e in x:
        out += flat(e)
    return out

print(len(flat([1, [2, [3, 4]], [[5]]])))
```

A. 3
B. 4
C. 6
D. 5

479. This Python function was meant to give back the larger number. What does the code print? Analysis

```
def biggest(a, b):
    if a > b:
        print(a)
    else:
        print(b)

r = biggest(3, 7)
print(r)
```

A. 7, then 7
B. Only 7
C. None, then 7
D. 7, then None

480. After setTen(x), x is still 3. Which change makes setTen set the caller's x to 10? Analysis

```
void setTen(int n) {
    n = 10;
}
// in main:
int x = 3;
setTen(x);
```

A. Change the parameter to int &n
B. Add return n; at the end
C. Declare n as static int n
D. Change the parameter to const int n

481. This Python function should add up a list. What happens when s([1, 2, 3]) runs, and why? Analysis

```
def s(lst):
    if not lst:
        return 0
    return lst[0] + s(lst)

print(s([1, 2, 3]))
```

A. It returns 6
B. RecursionError, because the recursive call passes the same list, so it never gets shorter
C. It returns 1
D. IndexError, because lst[0] fails on an empty list

482. A student says this Python code raises UnboundLocalError. What actually happens? Analysis

```
total = 0

def add(n):
    return total + n

print(add(5))
```

A. UnboundLocalError, because total is not declared global
B. NameError, because functions cannot see module variables
C. It prints 0, because the function gets a copy of total
D. It prints 5, because the function only reads the global and never assigns to it

483. What happens when this Java method is called as f(3)? Analysis

```
static int f(int n) {
    return n * f(n - 1);
}
```

A. It returns 6
B. It compiles, then throws StackOverflowError, because there is no base case
C. It returns 0, because f(0) multiplies by 0
D. It fails to compile, because a method cannot call itself

484. raw changes after this call, which the caller did not want. Which fix keeps raw unchanged and still returns the scaled list? Analysis

```
def normalize(scores):
    for i in range(len(scores)):
        scores[i] = scores[i] / 10
    return scores

raw = [50, 80]
scaled = normalize(raw)
print(raw)
```

A. Build and return a new list, e.g. return [x / 10 for x in scores]
B. Add global scores at the top of the function
C. Pass raw as normalize(raw) without assigning the result
D. Rename the parameter so it differs from raw

485. You must add 1 + 2 + ... + 1,000,000 in Python. Why is a loop (or sum(range(...))) better than a simple recursive function here? Analysis

A. Recursion gives a different, wrong answer for large n
B. A loop is O(1) while recursion is O(n) in time
C. The recursion would need about a million nested calls, far past CPython's default limit of about 1000, so it raises RecursionError
D. Recursive functions cannot return integers that large

486. For which input does this Python factorial fail, and how? Analysis

```
def fact(n):
    if n == 1:
        return 1
    return n * fact(n - 1)
```

A. fact(1) returns 0
B. fact(5) returns 24
C. fact(0) recurses past 1 into negatives and raises RecursionError
D. fact(2) raises TypeError

487. A student worries that the local name sum breaks the built-in sum(). What does this Python code print? Analysis

```
def total(items):
    sum = 0
    for x in items:
        sum += x
    return sum

print(total([1, 2]), sum([3, 4]))
```

A. 3, then TypeError: 'int' object is not callable
B. TypeError before anything is printed
C. 0 7
D. 3 7

488. A C++ file has overloads f(long) and f(double). What happens with the call f(5)? Analysis

```
void f(long x)   { cout << "long"; }
void f(double x) { cout << "double"; }
// in main:
f(5);
```

A. A compile error: the call is ambiguous, because int to long and int to double are both standard conversions of the same rank
B. f(long) runs, because long is closer to int
C. f(double) runs, because double can hold any int
D. It compiles and the choice is made at run time

489. What does this Java code print? Analysis

```
class A { static String n() { return "A"; } }
class B extends A { static String n() { return "B"; } }
// in main:
A obj = new B();
System.out.println(obj.n());
```

A. B
B. A compile error, because static methods cannot be redefined
C. AB
D. A

490. What does this Java code print? Analysis

```
static void f(Object o) { System.out.println("Object"); }
static void f(String s) { System.out.println("String"); }
// in main:
f(null);
```

A. String
B. Object
C. A compile error: ambiguous call
D. A NullPointerException

491. This Python power function calls itself twice in the even case. What is its running time in terms of e, and what is the fix? Analysis

```
def pw(b, e):
    if e == 0:
        return 1
    if e % 2 == 0:
        return pw(b, e // 2) * pw(b, e // 2)
    return b * pw(b, e - 1)
```

A. O(e); store pw(b, e // 2) in a variable and square it to get O(log e)
B. O(log e) already, because e is halved each time
C. O(2^e); replace the recursion with a loop
D. O(e log e); add a base case for e == 1

492. This tail-recursive Python function raises RecursionError for n = 5000. Which change keeps the result and removes the error? Analysis

```
def total(n, acc=0):
    if n == 0:
        return acc
    return total(n - 1, acc + n)
```

A. Add @functools.lru_cache to the function
B. Move the recursive call so it is the last line, making it tail-recursive
C. Rewrite it as a loop: while n > 0: acc += n; n -= 1
D. Return acc before the if statement

493. What does this Java code print? Analysis

```
static void inc(Integer n) {
    n++;
}
// in main:
Integer a = 5;
inc(a);
System.out.println(a);
```

A. 6
B. 5
C. A compile error, because ++ cannot be used on Integer
D. A NullPointerException

Input / Output Handling MCQs for NSCT

494. In Python 3, what type does input() return? Understanding
A. int when the user types digits
B. Whatever type the typed text looks like
C. Always str
D. bytes

495. In C++, cin and cout are objects of which classes, declared in which header? Understanding
A. istream and ostream, declared in <iostream>
B. ifstream and ofstream, declared in <fstream>
C. FILE and FILE*, declared in <cstdio>
D. string and stringstream, declared in <string>

496. What does opening a file with mode "a" do in Python? Understanding
A. Empties the file and starts writing at the beginning
B. Opens it read-only
C. Opens it for writing at the end, creating it if it does not exist
D. Raises an error if the file does not exist

497. Which Python open() mode creates a new file and raises FileExistsError if the file already exists? Understanding
A. "w"
B. "a"
C. "r+"
D. "x"

498. What is the main benefit of with open("data.txt") as f: in Python? Understanding

A. Reading becomes faster
B. The file is closed automatically when the block ends, even if an exception occurs
C. The file is opened in binary mode
D. Other programs are locked out of the file

499. In Java's java.io package, which class families handle character (text) data rather than raw bytes? Understanding

A. InputStream and OutputStream
B. DataInputStream and DataOutputStream
C. Reader and Writer
D. ObjectInputStream and ObjectOutputStream

500. In C++, what is the difference between cout << endl and cout << '\n'? Understanding

A. '\n' flushes the stream and endl does not
B. They are identical in every way
C. endl works only on Windows and '\n' only on Linux
D. endl writes a newline and also flushes the stream; '\n' only writes a newline

501. Which C++ standard stream object writes to standard error and is unbuffered by default? Understanding

A. clog
B. cerr
C. cout
D. cin

502. Why do I/O libraries use buffering? Understanding

A. To encrypt data before it is written
B. To make sure every byte reaches the disk immediately
C. To cut the number of slow system calls by collecting data in memory and transferring it in larger blocks
D. To convert text between character encodings

503. Which Java Scanner method reads the rest of the current line, including spaces? Understanding

A. next()
B. nextLine()
C. nextInt()
D. hasNext()

504. A Java class can be used as a resource in try-with-resources only if it implements which interface? Understanding

A. AutoCloseable (Closeable extends it)
B. Serializable
C. Runnable
D. Iterable

505. Which C++ statement reads a whole line, including spaces, from the keyboard into the string s? Understanding

A. std::getline(std::cin, s);
B. std::cin >> s;
C. std::cin.get(s);
D. std::cout >> s;

506. What are the default values of the sep and end parameters of Python's print()? Understanding

A. sep="" and end="\n"
B. sep=" " and end="\n"
C. sep="," and end=""
D. sep="\t" and end=" "

507. What is the difference between opening a file in text mode ("r") and binary mode ("rb") in Python? Understanding

A. Binary mode is faster but otherwise returns the same str objects
B. Text mode works only with ASCII files
C. Binary mode converts the file to 0s and 1s before returning it
D. Text mode decodes bytes to str using an encoding and translates line endings; binary mode returns raw bytes unchanged

508. Which C++ header declares ifstream and ofstream? Understanding

A. <iostream>
B. <cstdio>
C. <fstream>
D. <sstream>

509. Which Python file method returns a list of all the remaining lines? Understanding

A. read()
B. readline()
C. lines()
D. readlines()

510. What does this Python code print? Application

```
print("a", "b", sep="-", end="!")
print("c")
```

A. a b!c
B. a-b! c
C. a-b-c!
D. a-b!c

511. The user types 5 and presses Enter. What does this Python code print? Application

```
x = input()
print(x * 3)
```

A. 555
B. 15
C. 5
D. TypeError

512. What does this Python code print? (Spaces matter.) Application

```
print(f"{3.14159:.2f}|{42:5d}|")
```

A. 3.14159|42|
B. 3.1|42 |
C. 3.14| 42|
D. 3.14|42 |

513. What does this Python code print? Application

```
with open("t.txt", "w") as f:
    f.write("ab\ncd\n")
with open("t.txt", "a") as f:
    f.write("ef")
with open("t.txt") as f:
    print(len(f.readlines()))
```

A. 2
B. 1
C. 4
D. 3

514. What does this Python code print? Application

```
with open("n.txt", "w") as f:
    f.write("hello")
with open("n.txt", "w") as f:
    f.write("hi")
with open("n.txt") as f:
    print(f.read())
```

A. hellohi
B. hihello
C. hi
D. hillo

515. data.txt contains the three lines one, two and three, each ending in a newline. What does this Python code print? Application

```
with open("data.txt") as f:
    f.readline()
    print(repr(f.readline()))
```

A. 'two\n'
B. 'two'
C. 'one\n'
D. 'one\ntwo\n'

516. What does this JavaScript (Node.js) code print? Application

```
process.stdout.write("x");
process.stdout.write("y\n");
console.log("z");
```

A. x, y and z on three separate lines
B. xyz on one line
C. z on the first line, then xy
D. xy on the first line, then z on the second

517. The user types 42 with spaces before and after it, then presses Enter. What does this Python code print? Application

```
print(int(input()) + 1)
```

A. ValueError
B. 421
C. 43
D. 42

518. The user types 3 4 on one line. What happens when this Python code runs? Application

```
a = int(input())
print(a)
```

A. It prints 3
B. It prints 7
C. It prints 34
D. ValueError, because int() cannot convert the string "3 4"

519. The user types 7 8 on one line. What does this Python code print? Application

```
a, b = map(int, input().split())
print(a * b)
```

A. 56
B. 15
C. 78
D. ValueError

520. The user types Ali Khan and presses Enter. What does this C++ code print? Application

```
string s;
cin >> s;
cout << s.length();
```

A. 8
B. 7
C. 3
D. 0

521. The input is 25 on the first line and Ali on the second. What does this Java code print? Application

```
Scanner sc = new Scanner(System.in);
int age = sc.nextInt();
String name = sc.nextLine();
System.out.println("[" + name + "]");
```

A. [Ali]
B. []
C. [25]
D. It throws InputMismatchException

522. What does this C++ code print? Application

```
cout << fixed << setprecision(2) << 3.14159;
```

A. 3.14159
B. 3.14
C. 3.1
D. 3

523. What does this Python code print? Application

```
nums = [1, 2, 3]
print(*nums, sep=", ")
```

A. [1, 2, 3]
B. 1 2 3
C. 1, 2, 3
D. 1,2,3

524. What happens when this Python code runs? Analysis

```
with open("out.txt", "w") as f:
    for i in range(3):
        f.write(i)
```

A. The file contains 012
B. TypeError, because write() needs a str, not an int
C. The file contains 0, 1 and 2 on separate lines
D. ValueError, because the file is not open for writing

525. msg.txt contains exactly hello with no newline. What does this Python code print? Analysis

```
with open("msg.txt") as f:
    a = f.read()
    b = f.read()
print(len(a), len(b))
```

A. 5 0
B. 5 5
C. 0 5
D. 10 0

526. What happens when this Python code runs, assuming a.txt exists? Analysis

```
with open("a.txt") as f:
    pass
print(f.read())
```

A. ValueError: I/O operation on closed file
B. It prints the file's contents
C. It prints an empty string
D. NameError, because f exists only inside the with block

527. In this Java code, nextLine() returns an empty string instead of the name. Which fix is correct? Analysis

```
Scanner sc = new Scanner(System.in);
int age = sc.nextInt();
String name = sc.nextLine();
```

A. Call System.out.flush() before nextLine()
B. Call sc.nextLine() once after nextInt() to consume the leftover newline
C. Close the Scanner and create a new one on System.in
D. Replace nextLine() with nextInt()

528. answers.txt holds the lines yes, no and yes, and the last line has no trailing newline. What does this Python code print? Analysis

```
with open("answers.txt") as f:
    count = sum(1 for line in f if line == "yes")
print(count)
```

A. 1
B. 2
C. 0
D. 3

529. This C++ code reads a number and then a full name, but name comes out empty. Which line, added before getline, fixes it? Analysis

```
int n;
string name;
cin >> n;
getline(cin, name);
```

A. cin.clear();
B. cout.flush();
C. cout << endl;
D. cin.ignore(numeric_limits<streamsize>::max(), '\n');

530. Why is while (fin >> x) preferred over while (!fin.eof()) { fin >> x; ... } when reading numbers from a file in C++? Analysis

A. eof() is slower because it reads the whole file first
B. eof() becomes true only after a read has already hit the end, so the eof loop processes one extra, failed read
C. while (fin >> x) can read only one number
D. The eof version cannot compile with ifstream

531. A Java program writes lines with a BufferedWriter, but the output file is empty after the program ends. What is the most likely cause? Analysis

A. BufferedWriter cannot write to files
B. The writer was never flushed or closed, so the data stayed in its buffer
C. The file was opened in read mode
D. Java writes files only after the JVM restarts

532. You must count error lines in a 10 GB log file in Python with little memory. Which approach suits best? Analysis

A. for line in f: (iterate over the file line by line)
B. data = f.read() then data.split("\n")
C. lines = f.readlines() then loop over lines
D. Read the file twice, once to size it and once to process it

533. You need to read a file's existing contents and then update it without losing them. Why is mode "r+" right and "w+" wrong? Analysis

A. "r+" cannot write; "w+" can
B. "w+" opens the file in binary mode only
C. They are identical; both keep the content
D. "w+" truncates the file when it is opened, so there is nothing left to read; "r+" keeps the content

534. In Python, which exception does open("missing.txt") raise when the file does not exist? Understanding

A. FileNotFoundError
B. EOFError
C. ValueError
D. KeyError

535. In Java, what does BufferedReader.readLine() return when the end of the stream is reached? Understanding

A. An empty string
B. null
C. -1
D. It throws EOFException

536. In Python, what does the file argument in print("Error!", file=sys.stderr) do? Understanding

A. It saves the text to a file named sys.stderr
B. It raises the text as an exception
C. It sends the printed text to the given stream instead of standard output
D. It reads input from the given stream

537. In Python, which pair of file methods moves the read/write position and reports the current position? Understanding

A. read() and write()
B. open() and close()
C. flush() and fileno()
D. seek() and tell()

538. In C++, what does opening a file with ofstream out("log.txt", ios::app); do? Understanding

A. The file is erased before writing
B. The file is opened for reading only
C. The file is opened in binary mode
D. Every write is added to the end, keeping the existing contents

539. Which Java method prints formatted output using a format string such as "%.2f"? Understanding

A. System.out.println
B. System.out.print
C. System.out.printf
D. System.out.flush

540. A program is run as python app.py < in.txt. Where does input() read from? Understanding

A. The keyboard, ignoring in.txt
B. It writes its prompt into in.txt
C. The lines of in.txt, instead of the keyboard
D. It raises an error, because input() needs a terminal

541. In C++, what is >> called in cin >> x? Understanding

A. The insertion operator
B. The right-shift assignment operator
C. The extraction operator
D. The scope resolution operator

542. In Python, what happens if you open a file that does not exist with mode "w"? Understanding

A. FileNotFoundError is raised
B. The file is created
C. FileExistsError is raised
D. The file is opened read-only

543. IOException is a checked exception in Java. What does that require of a method that calls FileReader's read()? Understanding

A. It must catch IOException or declare throws IOException
B. Nothing; checked exceptions are only logged
C. It must run the call in a new thread
D. It must close the file before reading

544. In Python, what happens when input() is called and standard input has no more data (end of file)? Understanding

A. It returns an empty string
B. It returns None
C. It waits for ever
D. It raises EOFError

545. What is the role of InputStreamReader in Java I/O? Understanding

A. It reads whole lines faster by buffering them
B. It writes characters out as bytes
C. It parses numbers from text tokens
D. It bridges a byte stream to a character stream by decoding bytes with a charset

546. Why is it good practice to write open(path, encoding="utf-8") in Python? Understanding

A. It makes the file open faster
B. It switches the file to binary mode
C. Without it, text mode uses a platform-dependent default encoding, so non-ASCII text may fail or read wrongly on another machine
D. It is required, or open() raises an error

547. A file contains the bytes a, \r, \n, b. What does Python's text mode (the default newline=None) give for f.read()? Understanding

A. "a\r\nb" unchanged
B. "a\nb", because \r\n is translated to \n
C. "ab", because line endings are dropped
D. "a\rb", because only \n is removed

548. In C++, how long does the setw manipulator's width setting last? Understanding

A. For the rest of the program
B. Only for the next output item; then the width resets to 0
C. Until setprecision is called
D. For the rest of the current line

549. Which difference between sys.stdin.readline() and input() is correct in Python? Understanding

A. readline() keeps the trailing newline, while input() strips it
B. input() keeps the trailing newline, while readline() strips it
C. Both strip the newline and differ only in speed
D. readline() converts the line to an int

550. In Java, what does the true argument do in new FileWriter("log.txt", true)? Understanding

A. It enables auto-flush after every write
B. It creates the file only if it does not exist
C. It opens the file in append mode, so new text goes after the old
D. It opens the file for reading and writing

551. In Python's json module, what is the difference between json.dump and json.dumps? Understanding

A. dump writes JSON to a file object; dumps returns the JSON as a string
B. dump returns a string; dumps writes to a file
C. dump reads JSON; dumps writes JSON
D. dump handles lists; dumps handles dicts

552. A Java program prints messages with System.out and errors with System.err, and is run as java App > out.txt. What appears on the screen? Understanding

A. Nothing; both streams go to out.txt
B. Only the System.err messages; System.out goes to out.txt
C. Only the System.out messages
D. Both streams, because > copies output

553. Why is a FileReader usually wrapped in a BufferedReader in Java? Understanding

A. BufferedReader converts bytes to characters, which FileReader cannot do
B. FileReader cannot be used without a wrapper
C. BufferedReader lets the file be written as well as read
D. The buffer reads large chunks at once, cutting slow system calls, and adds readLine()

554. In C++, int x; cin >> x; reads the input abc. What is the state afterwards? Understanding

A. x holds the ASCII code of 'a' and reading continues normally
B. cin is in a fail state, the letters stay in the buffer, and later >> reads do nothing until cin.clear() is called
C. The program throws an exception and ends
D. cin skips abc automatically and waits for a number

555. A Python script runs print("Working...", end="") and then a 10-second computation in a terminal. The text appears only after the computation. Why? Understanding

A. print with end="" does not output anything until the program ends
B. Terminal stdout is line-buffered, and no newline was written, so the text waits in the buffer; flush=True shows it at once
C. The terminal hides text that has no newline
D. Python runs the computation before the print statement

556. What do ios::sync_with_stdio(false) and cin.tie(nullptr) do in a C++ program? Understanding

A. They make cin and cout thread-safe
B. They stop syncing C++ streams with C stdio and stop flushing cout before each cin read, which speeds up heavy I/O
C. They switch cin and cout to binary mode
D. They make cout unbuffered so output appears at once

557. In C++ code that uses <cstdio>, why does scanf("%d", &x) take &x while printf("%d", x) takes x? Understanding

A. & converts x to a string for scanf
B. printf also needs &x, but the compiler adds it automatically
C. & makes scanf skip whitespace
D. scanf must store a value into x, so it needs x's address; printf only needs the value

558. What does this Python code print? Application

```
print("Total:", 5, "items")
```

A. Total:5items
B. "Total:" 5 "items"
C. ('Total:', 5, 'items')
D. Total: 5 items

559. What does this Python code print? Application

```
print("a" + "b", "c", sep="")
```

A. ab c
B. a b c
C. ab,c
D. abc

560. What does this Python code print? Application

```
with open("w.txt", "w") as f:
    n = f.write("hello\n")
print(n)
```

A. 5
B. 6
C. 1
D. None

561. What does this C++ code print? Application

```
cout << "A" << 1 + 2 << "B";
```

A. A3B
B. A12B
C. A1+2B
D. A 3 B

562. What is the output of this Java code? Application

```
System.out.print("x");
System.out.println("y");
System.out.print("z");
```

A. x, y and z each on its own line
B. xyz on one line
C. xy on the first line and z on the second
D. x on the first line and yz on the second

563. What does this Python code print? Application

```
print(str(3) + "4", 3 + 4)
```

A. 7 7
B. 34 34
C. TypeError
D. 34 7

564. What does this Python code print? Application

```
with open("w.txt", "w") as f:
    f.writelines(["a", "b", "c"])
with open("w.txt") as f:
    print(f.read())
```

A. abc
B. a\nb\nc on three lines
C. ['a', 'b', 'c']
D. a b c

565. What does this Python code print? Application

```
with open("s.txt", "w") as f:
    f.write("abcdef")
with open("s.txt") as f:
    f.seek(2)
    print(f.read(3))
```

A. bcd
B. cde
C. abc
D. cdef

566. What does this Python code print? Application

```
print(f"{7:03d}|{'ab':<4}|{2.5:6.1f}")
```

A. 007|ab | 2.5
B. 7|ab|2.5
C. 007| ab|2.50
D. 7|ab |2.5

567. What does this C++ code print? (Spaces matter.) Application

```
cout << setw(5) << 42 << setw(3) << 7 << "|";
```

A. "42 7|"
B. " 42 7|"
C. " 42 7|"
D. " 427|"

568. What does this C++ code print? Application

```
cout << setfill('*') << setw(6) << left << 12 << "|";
```

A. ****12|
B. 12****|
C. 12|
D. **12**|

569. What does this Java code print? (Spaces matter.) Application

```
System.out.printf("%5.2f|%-4d|%s%n", 3.14159, 7, "ok");
```

A. "3.14|7|ok"
B. " 3.14| 7|ok"
C. "3.14159|7 |ok"
D. " 3.14|7 |ok"

570. The user types 10,20,30 and presses Enter. What does this Python code print? Application

```
a = input().split(",")
print(a[1] + a[2])
```

A. 2030
B. 50
C. 60
D. TypeError

571. The input is 12 abc. What does this Java code print? Application

```
Scanner sc = new Scanner(System.in);
int n = sc.nextInt();
String s = sc.next();
System.out.println(s + n);
```

A. abc12
B. 12abc
C. abc 12
D. An InputMismatchException

572. What does this Node.js code print? Application

```
console.log("a", 1, [2, 3]);
```

A. a1[2,3]
B. a 1 2,3
C. a 1 [2, 3]
D. a,1,[2,3]

573. What does this Python code print? Application

```
with open("p.txt", "w") as f:
    print("a", "b", file=f, sep=",")
with open("p.txt") as f:
    print(repr(f.read()))
```

A. 'a b\n'
B. 'a,b\n'
C. 'a,b'
D. ''

574. h.txt contains exactly hello. What does this Python code print? Application

```
with open("h.txt", "r+") as f:
    f.write("J")
    f.seek(0)
    print(f.read())
```

A. Jhello
B. J
C. hello
D. Jello

575. ab.txt contains ab. What does this Python code print? Application

```
with open("ab.txt", "a+") as f:
    print(repr(f.read()))
    f.write("c")
    f.seek(0)
    print(repr(f.read()))
```

A. 'ab' then 'abc'
B. 'ab' then 'c'
C. '' then 'c'
D. '' then 'abc'

576. What does this Python code print? Application

```
with open("r.txt", "w") as f:
    f.write("ab\ncd\n")
with open("r.txt") as f:
    f.read(1)
    print(repr(f.readline()))
```

A. 'b\n'
B. 'ab\n'
C. 'cd\n'
D. 'b'

577. The user types 12abc and presses Enter. What does this C++ code print? Application

```
int n;
string s;
cin >> n >> s;
cout << n << "|" << s;
```

A. 0|12abc
B. 12abc|
C. 12|abc
D. Nothing, because cin fails on the letters

578. What does this Python code print? Application

```
print(print("x"))
```

A. x on the first line and None on the second
B. x only
C. None only
D. x twice

579. The user types 20. What goes wrong in this Python code, and what fixes it? Analysis

```
age = input("Age: ")
if age > 18:
    print("Adult")
```

A. It prints Adult; nothing is wrong
B. TypeError, because input() returns a str; use age = int(input("Age: "))
C. ValueError, because 20 is not a valid input
D. SyntaxError, because the prompt string is not allowed

580. This Python loop should save all five results, but results.txt ends up with only the last one. Why? Analysis

```
for i in range(5):
    with open("results.txt", "w") as f:
        f.write(f"{i}\n")
```

A. write() only keeps one line per file
B. The with block does not save data to disk
C. Each open with "w" empties the file, so each pass erases the previous result
D. range(5) runs the loop only once

581. nums.txt holds one integer per line. What happens when this Python code runs, and what is the fix? Analysis

```
with open("nums.txt") as f:
    print(sum(f))
```

A. TypeError, because sum() tries to add the int 0 to line strings; use sum(int(line) for line in f)
B. It prints the total of the numbers
C. It prints the number of lines
D. ValueError, because each line ends with \n

582. Each run of this C++ program should add a line to log.txt, but only the latest run's line is ever in the file. What is the fix? Analysis

```
ofstream log("log.txt");
log << "run finished\n";
```

A. Call log.flush() after writing
B. Use ifstream instead of ofstream
C. Add endl instead of \n
D. Open it with ofstream log("log.txt", ios::app);

583. If readLine() throws, the file is never closed. Which rewrite closes it on every path? Analysis

```
BufferedReader br = new BufferedReader(new FileReader("in.txt"));
String line = br.readLine();
br.close();
```

A. Call br.close() as the first line of the method
B. Declare br as static
C. Catch the exception and ignore it
D. try (BufferedReader br = new BufferedReader(new FileReader("in.txt"))) { ... }

584. This Python loop should stop when the user types done, but it never stops. Why? Analysis

```
import sys
while True:
    line = sys.stdin.readline()
    if line == "done":
        break
    print("got", line, end="")
```

A. readline() returns None for the word done
B. The while True condition is checked only once
C. sys.stdin.readline() keeps the newline, so line is "done\n", which never equals "done"
D. break does not work inside while loops

585. Run in CPython, this code most likely prints an empty line. Why? Analysis

```
f = open("t.txt", "w")
f.write("hi")
print(open("t.txt").read())
```

A. "w" mode cannot write strings
B. A file cannot be opened twice
C. The written text is still in f's buffer, because f was never flushed or closed
D. read() always skips the first line

586. Which Java classes should a program use to copy a PNG image file? Analysis

A. FileReader and FileWriter, because they buffer the data
B. Scanner and PrintWriter, because they read tokens
C. BufferedReader.readLine() and println()
D. FileInputStream and FileOutputStream, because an image is raw bytes

587. This Python code copies photo.jpg. What is wrong with it? Analysis

```
with open("photo.jpg") as src, open("copy.jpg", "w") as dst:
    dst.write(src.read())
```

A. Nothing; text mode copies any file exactly
B. Text mode decodes the bytes as characters, so it may raise UnicodeDecodeError or corrupt the copy; use "rb" and "wb"
C. read() cannot read a whole file at once
D. write() needs a list of lines

588. The user types abc. This C++ loop then prints Try again for ever. What is the fix? Analysis

```
int x;
while (!(cin >> x)) {
    cout << "Try again\n";
}
```

A. Inside the loop, call cin.clear() and then cin.ignore(numeric_limits<streamsize>::max(), '\n')
B. Declare x as a double
C. Replace the loop with if (!(cin >> x))
D. Add cout.flush() inside the loop

589. The CSV line is Ali,"Lahore, PK",20. What does this Python code print, and what is the better approach? Analysis

```
line = 'Ali,"Lahore, PK",20'
print(len(line.split(",")))
```

A. 4; use the csv module, which treats the quoted comma as part of the field
B. 3; the code is already correct
C. 4; use split(", ") with a space instead
D. 2; use split(",", 1)

590. When the input is abc, this Java loop prints Enter a number for ever. Why, and what is the fix? Analysis

```
Scanner sc = new Scanner(System.in);
while (!sc.hasNextInt()) {
    System.out.println("Enter a number");
}
int n = sc.nextInt();
```

A. hasNextInt() does not consume the bad token; call sc.next() inside the loop to discard it
B. hasNextInt() blocks until a number is typed, so the loop never runs
C. Scanner cannot read letters; use BufferedReader
D. The loop needs sc.close() inside it

591. This script is run as python p.py > out.txt 2>&1. In what order do the lines usually appear in out.txt? Analysis

```
import sys
print("A")
print("B", file=sys.stderr)
```

A. A then B, because the statements run in that order
B. Only A, because stderr is not redirected
C. B then A, because stdout is block-buffered when redirected to a file while stderr is written at once
D. Only B, because stdout is discarded

592. A file was saved as UTF-8 with the text café. What does this Python code print, and why? Analysis

```
with open("c.txt", encoding="cp1252") as f:
    print(f.read())
```

A. café, because Python detects the encoding
B. UnicodeDecodeError, because cp1252 cannot read the file
C. cafÃ©, because the two UTF-8 bytes of é are decoded as two cp1252 characters
D. caf?, because unknown bytes become question marks

593. The file was written in binary as b"a\r\nb\r\n". Why does this Python code print 4 and 6? Analysis

```
with open("w.txt") as f:
    print(len(f.read()))
with open("w.txt", "rb") as f:
    print(len(f.read()))
```

A. Text mode drops the letters' encoding bytes
B. Text mode turns each \r\n into \n while reading; binary mode returns the raw bytes
C. Binary mode adds an extra \r to each line
D. Text mode stops reading at the first newline

Strings & Text Processing MCQs for NSCT

594. In Python, what happens when this code runs? Understanding

```
s = "cat"
s[0] = "b"
```

A. An IndexError is raised because index 0 is read-only
B. s becomes "bat"
C. A TypeError is raised because str objects do not support item assignment
D. A new string "bat" is created and s still holds "cat"

595. In Java, which of these classes gives you a mutable sequence of characters? Understanding

A. Integer
B. String
C. Character
D. StringBuilder

596. In C, how does a function such as strlen() know where a character-array string ends? Understanding

A. It uses the declared size of the array
B. It stops at the null character '\0'
C. It stops at the first space character
D. It reads a length field stored before the first character

597. In Java, which expression compares the characters of two String objects a and b rather than their references? Understanding

A. a == b
B. a.equals(b)
C. a.compareTo(b) == 1
D. a.hashCode() == b.hashCode()

598. What does Python's str.strip() remove when it is called with no arguments? Understanding

A. Whitespace at both the start and the end of the string
B. Only leading spaces
C. All whitespace, including spaces between words
D. Only trailing newline characters

599. Why is repeated s = s + piece inside a long Java loop considered inefficient? Understanding

A. The Java compiler rejects + on strings inside loops
B. String concatenation is not allowed with variables, only with literals
C. Each + forces an immediate garbage collection
D. Each iteration creates a new String and copies all existing characters into it

600. Which statement about std::string in C++ is correct? Understanding

A. It is immutable like Java's String
B. It is mutable: s[0] = 'b' changes the string in place
C. It cannot be compared with the == operator
D. It must always be terminated manually with '\0'

601. Which Python string literal is an f-string that inserts the value of the variable name? Understanding

A. f"Hello {name}"
B. b"Hello {name}"
C. r"Hello {name}"
D. "Hello {name}"

602. In Java, String a = "hi"; String b = "hi"; Why does a == b give true? Understanding

A. == always compares string contents in Java
B. The compiler converts == into equals() for strings
C. String literals with the same content are interned, so both variables refer to one pooled object
D. Both variables are primitive values, not objects

603. In Python, what is the difference between s.find(sub) and s.index(sub) when sub is not in s? Understanding

A. find raises ValueError, while index returns -1
B. find returns -1, while index raises ValueError
C. Both return None
D. Both return -1

604. Which Python method returns True for "2026" but False for "20.5"? Understanding

A. isdigit()
B. islower()
C. isspace()
D. isalpha()

605. What is the most efficient common way to build one large string from many small pieces in Python? Understanding

A. Append the pieces to a list and call "".join(list) once
B. Use += on a string inside the loop
C. Convert each piece to a tuple and add the tuples
D. Use str.replace() to insert each piece

606. What does the escape sequence \t represent inside a string literal in C++, Java and Python? Understanding

A. A literal backslash followed by t
B. A newline
C. The end of the string
D. A horizontal tab character

607. What does Python's ord('A') return? Understanding

A. 1
B. 65
C. 'A'
D. 97

608. Which Python method is designed for caseless matching, so that "Straße" and "STRASSE" compare equal after it is applied to both? Understanding

A. swapcase()
B. upper()
C. casefold()
D. lower()

609. In Python 3, what does len() count for a str object? Understanding

A. Bytes in the UTF-16 encoding
B. Visible words
C. Unicode code points
D. Bytes in the UTF-8 encoding

610. What does this Java code print? Application

```
String s = "nsct";
s.toUpperCase();
System.out.println(s);
```

A. Nsct
B. NSCT
C. nsct
D. Compilation error

611. What does this Python code print? Application

```
s = "caf\u00e9"
print(len(s), len(s.encode("utf-8")))
```

A. 5 5
B. 5 4
C. 4 5
D. 4 4

612. What does this Python code print? Application

```
s = "programming"
print(s[::-1])
```

A. gnimmargor
B. programming
C. g
D. gnimmargorp

613. What does this Java code print? Application

```
String s = "programming";
System.out.println(s.substring(3, 7));
```

A. gramm
B. ogram
C. gram
D. ogra

614. What does this Python code print? Application

```
print("{:.2f}".format(3.14159))
```

A. 3.14
B. 3.1
C. 3.142
D. 3.14159

615. What does this Python code print? (The vertical bars show where the output starts and ends.) Application

```
n, t = 7, "ab"
print(f"|{n:03d}|{t:>5}|")
```

A. |007|ab |
B. |007| ab|
C. |7 |ab |
D. | 7| ab|

616. What does this Java statement print? (The bar marks the end of the output.) Application

```
System.out.println(String.format("%5.1f|", 3.14159));
```

A. 3.14|
B. 3.1|
C. 3.14159|
D. 3.10|

617. What does this Python code print? Application

```
print("banana".count("an"), "aaaa".count("aa"))
```

A. 2 3
B. 2 1
C. 2 2
D. 1 2

618. What does this Python code print? Application

```
print("Hello World".replace("o", "0", 1))
```

A. Hell0 W0rld
B. Hello World
C. Hell0 World
D. Hell World

619. What does this Python code print? Application

```
print(len("a,b,,c".split(",")), len(" a  b ".split()))
```

A. 3 4
B. 4 4
C. 3 2
D. 4 2

620. What does this Python code print? Application

```
print(sorted(["b", "A", "a", "B"]))
```

A. ['a', 'A', 'b', 'B']
B. ['A', 'a', 'B', 'b']
C. ['a', 'b', 'A', 'B']
D. ['A', 'B', 'a', 'b']

621. What does this Python code print? Application

```
s = "abcdef"
print(s[1:5:2], s[-2:], s[5:1:-2])
```

A. bd ef fd
B. bd ef fdb
C. bc ef ec
D. bdf ef fd

622. What does this C code print, given that it includes <stdio.h> and <string.h>? Application

```
printf("%zu %zu\n", strlen("hello"), sizeof("hello"));
```

A. 6 5
B. 5 6
C. 6 6
D. 5 5

623. What does this Python code print? Application

```
import re
print(re.findall(r"\d+", "a12b3c456"))
```

A. ['12', '3', '456']
B. [12, 3, 456]
C. ['12', '456']
D. ['1', '2', '3', '4', '5', '6']

624. This Python function should treat "Race car" as a palindrome, but it prints False. Why? Analysis

```
def is_pal(s):
    s = s.lower()
    return s == s[::-1]

print(is_pal("Race car"))
```

A. == compares string identity, not content
B. s[::-1] returns a list, not a string
C. lower() does not change s because strings are immutable
D. The space is still in the string, so "race car" reversed is "rac ecar"

625. In Java, input holds "yes" read from a Scanner, yet this check prints "no". What is the correct fix? Analysis

```
if (input == "yes") {
    System.out.println("yes");
} else {
    System.out.println("no");
}
```

A. Use input.compareTo("yes") == 1
B. Use (input + "") == "yes"
C. Use input = "yes"
D. Use input.equals("yes")

626. This Python code is meant to count all vowels in "EducAtion", which has 5. What does it print, and why? Analysis

```
word = "EducAtion"
count = 0
for ch in word:
    if ch in "aeiou":
        count += 1
print(count)
```

A. 3, because the uppercase E and A are not in "aeiou"
B. 0, because ch is compared as a whole word
C. 4, because the loop skips the last character
D. 5, because in ignores case

627. Why does this C++ declaration fail to compile? Analysis

```
char name[5] = "Hello";
```

A. char arrays cannot be initialised from string literals in C++
B. The variable name must be declared as std::string
C. Array sizes must be a power of two
D. "Hello" needs 6 chars including the '\0' terminator, but the array holds only 5

628. A Java method builds a string of n characters with s = s + c inside a loop. What is the total running time, and what changes it? Analysis

A. O(n log n), because the JVM doubles the string's capacity
B. O(n^2), because each step copies the whole string; a StringBuilder makes it O(n)
C. O(1), because the compiler precomputes the result
D. O(n), because + appends in place

629. What does this Java code print? Analysis

```
String a = "hi";
String b = new String("hi");
String c = b.intern();
System.out.println((a == b) + " " + (a == c));
```

A. false true
B. true true
C. false false
D. true false

630. What does this Java code print? Analysis

```
System.out.println('a' + 1 + "b");
System.out.println("b" + 'a' + 1);
```

A. a1b and then b98
B. 98b and then ba1
C. 98b and then b98
D. ab1 and then ba1

631. A form checks quantities with if s.isdigit(): n = int(s), else it reports an error. Which input is rejected even though int() could convert it? Analysis

A. "-12"
B. "2026"
C. "12"
D. "007"

632. Which change stops this Java loop from throwing StringIndexOutOfBoundsException? Analysis

```
String s = "nsct";
for (int i = 0; i <= s.length(); i++) {
    System.out.print(s.charAt(i));
}
```

A. Replace charAt(i) with charAt(i - 1) and keep <=
B. Start the loop at i = 1
C. Declare s as final
D. Change i <= s.length() to i < s.length()

633. This Python line raises an error. Which fix prints Total: 5? Analysis

```
print("Total: " + 5)
```

A. print("Total: " - 5)
B. print("Total: ".join(5))
C. print("Total: " + str(5))
D. print("Total: " + 5.0)

634. In C, what does strcmp(a, b) return when the strings a and b contain exactly the same characters? Understanding

A. 0
B. 1
C. A non-zero value meaning true
D. The length of the strings

635. In C++, what value does std::string::find() return when the substring is not present? Understanding

A. -1 as a signed int
B. std::string::npos
C. 0
D. It throws std::out_of_range

636. In C++, what does s1 == s2 compare when s1 and s2 are std::string objects? Understanding

A. Their memory addresses, like == on Java String references
B. Only their lengths
C. Their characters, so equal text gives true
D. Nothing; == is not defined for std::string

637. In Java, which method joins the elements of a List<String> into one string with ", " between them? Understanding

A. list.concat(", ")
B. String.split(", ", list)
C. list.toString(", ")
D. String.join(", ", list)

638. In Python, which string method returns True only when every character is a letter (and the string is not empty)? Understanding

A. isalpha()
B. isalnum()
C. isdigit()
D. istitle()

639. How many bytes does a plain ASCII character such as 'A' take when the text is encoded in UTF-8? Understanding

A. 2
B. 1
C. 4
D. It depends on the operating system

640. In Python, what does the raw string literal r"\n" contain? Understanding

A. One newline character
B. The single letter n
C. Two characters: a backslash followed by n
D. An empty string

641. In Python 3, what type does "abc".encode("utf-8") return? Understanding

A. str
B. bytearray
C. list of int
D. bytes

642. In a regular expression, what does ^ mean when it appears at the start of the pattern (outside square brackets)? Understanding

A. Match at the start of the string (or line in multiline mode)
B. Match any character except a newline
C. Negate the whole pattern
D. Match the end of the string

643. In Python, what does str.join() require of the items in its iterable argument? Understanding

A. Items can be of any type; join converts them with str() automatically
B. Every item must already be a str, or a TypeError is raised
C. Items must be single characters
D. Items must be bytes objects

644. In C++, how does cin >> s differ from getline(cin, s) when reading into a std::string s? Understanding

A. getline stops at the first space; cin >> s reads the whole line
B. They behave identically for std::string
C. cin >> s stops at the first whitespace; getline reads the whole line up to the newline
D. cin >> s cannot read into a std::string, only into char arrays

645. In Java, how do StringBuilder and StringBuffer differ? Understanding

A. StringBuilder is immutable; StringBuffer is mutable
B. StringBuffer can only append; StringBuilder can also insert and delete
C. StringBuilder stores bytes; StringBuffer stores chars
D. StringBuffer's methods are synchronized (thread-safe); StringBuilder's are not, so it is faster in single-threaded code

646. Which is a real benefit of Java String objects being immutable? Understanding

A. They can be shared safely, for example in the string pool, and their hash code can be cached
B. Concatenation with + becomes O(1)
C. They use less memory than a char array of the same length
D. They can be changed in place without creating new objects

647. Why should Python code compare two strings with == rather than with is? Understanding

A. is is slower than == for strings
B. is tests whether both names refer to the same object, and equal strings built at run time can be different objects
C. == compares only the first character
D. is raises TypeError for strings

648. In Python, what does "user=ali=admin".partition("=") return? Understanding

A. ['user', 'ali', 'admin']
B. ('user', 'ali=admin')
C. ('user', '=', 'ali=admin')
D. ('user=ali', '=', 'admin')

649. In Python, what does s.startswith(("http://", "https://")) do? Understanding

A. Returns True only if s starts with both prefixes
B. Raises TypeError, because startswith accepts only one string
C. Returns the prefix that matched
D. Returns True if s starts with either prefix in the tuple

650. In Python, what does the conversion flag !r do in the f-string f"{name!r}"? Understanding

A. Formats the value with repr(), so a string appears with quotes
B. Right-aligns the value
C. Reverses the string
D. Inserts the value as a raw string with escapes ignored

651. Why was the C function gets() removed from the C11 standard? Understanding

A. It does not read spaces
B. It cannot be told the buffer size, so long input overflows the array
C. It leaves the newline in the buffer
D. It only works with files, not keyboard input

652. In C, what must be true of dest before calling strcat(dest, src)? Understanding

A. dest must be empty
B. dest and src must have the same length
C. dest must be a null-terminated string with room for both strings plus the null terminator
D. dest must be a string literal

653. In C++, what does s.c_str() return for a std::string s? Understanding

A. A new std::string that is a copy of s
B. The number of characters in s
C. A char* that the caller must free()
D. A const char* to a null-terminated copy of the text, for use with C functions

654. In C++, what is the type of the string literal "hi"? Understanding

A. const char[3]
B. std::string
C. char*
D. const char[2]

655. In Java, what does "\uD83D\uDE00".length() return for this string holding one emoji (U+1F600)? Understanding

A. 1
B. 2
C. 4
D. It throws an exception

656. In a regular expression, how does .* differ from .*? ? Understanding

A. .* matches zero or more characters; .*? matches exactly one character
B. .*? makes the whole pattern optional
C. .* is greedy and matches as much as possible; .*? is lazy and matches as little as possible
D. They are identical; ? is ignored after *

657. Why do many Java security guides store a password in a char[] rather than a String? Understanding

A. A String cannot hold special characters
B. char[] is encrypted automatically by the JVM
C. String has a maximum length of 16 characters
D. A char[] can be overwritten with zeros after use; an immutable String stays in memory until garbage collected

658. What does this Python code print? Application

```
print("-".join("abc"))
```

A. a-b-c
B. -abc-
C. abc
D. a-b-c-

659. What does this Python code print? Application

```
print("nsct exam".title())
```

A. NSCT EXAM
B. Nsct Exam
C. Nsct exam
D. nsct Exam

660. What does this JavaScript code print? Application

```
console.log("5" + 3, "5" - 3);
```

A. 8 2
B. 53 NaN
C. 53 2
D. 8 8

661. What does this Java statement print? Application

```
System.out.println("  NSCT  ".trim().length());
```

A. 8
B. 6
C. 2
D. 4

662. What does this Python code print? Application

```
print("a,b;c".replace(";", ",").split(","))
```

A. ['a', 'b', 'c']
B. ['a', 'b;c']
C. ['a,b,c']
D. ['a', ',', 'b', ',', 'c']

663. What does this C code print? Assume <stdio.h> and <string.h> are included and ASCII encoding. Application

```
char s[] = "hello";
s[0] = s[0] - 32;
printf("%s %d\n", s, (int)strlen(s));
```

A. hello 5
B. Hello 5
C. Hello 6
D. Compilation error, because strings are immutable

664. What does this Python code print? Application

```
print("42".zfill(5), "ab".center(6, "*"))
```

A. 42000 **ab**
B. 00042 ab****
C. 00042 **ab**
D. 42 **ab**

665. What does this Python code print? Application

```
print("a=b=c".split("=", 1))
```

A. ['a', 'b', 'c']
B. ['a=b', 'c']
C. ['a']
D. ['a', 'b=c']

666. What does this Python code print? Application

```
print(f"{1234567.891:,.2f}")
```

A. 1,234,567.89
B. 1234567.89
C. 1,234,567.891
D. 1.23e+06

667. What does this Python code print? Application

```
s = "Mississippi"
print(s.rfind("ss"), s.index("ss"))
```

A. 2 5
B. 5 2
C. 5 5
D. 6 3

668. What does this JavaScript code print? Application

```
const s = "JavaScript";
console.log(s.slice(-6), s.substring(4, 1));
```

A. Script Jav
B. Scrip ava
C. Script ava
D. Script (empty string)

669. What does this JavaScript code print? Application

```
console.log("7".padStart(3, "0") + "ab".repeat(2));
```

A. 700abab
B. 007ab2
C. 7ab
D. 007abab

670. What does this Java code print? Application

```
StringBuilder sb = new StringBuilder("abc");
sb.append("de").reverse();
sb.insert(0, 'X');
System.out.println(sb);
```

A. Xedcba
B. Xabcde
C. edcbaX
D. abcdeX

671. What does this Java statement print? Application

```
System.out.println("banana".indexOf('a') + " " + "banana".lastIndexOf("an"));
```

A. 1 4
B. 1 3
C. 0 3
D. 2 3

672. What does this C++ code print? Assume <iostream> and <string> are included and using namespace std. Application

```
string s = "programming";
cout << s.substr(7) << " " << s.find('m');
```

A. ming 7
B. mming 6
C. ming 6
D. gram 6

673. What does this Python code print? Application

```
print("".join(chr((ord(c) - 97 + 3) % 26 + 97) for c in "xyz"))
```

A. {|}
B. xyz
C. def
D. abc

674. What does this Python code print? Application

```
s = "level up"
print(s.swapcase(), s.find("up"))
```

A. LEVEL UP 6
B. LEVEL UP 7
C. Level Up 6
D. LEVEL UP -1

675. What does this Python code print? Application

```
import re
print(re.sub(r"\s+", " ", "a   b \t c"))
```

A. a b c
B. a b c
C. a b \t c
D. abc

676. What does this Python code print? (The bar marks the end of the output.) Application

```
print("{:^7}|".format("ab"))
```

A. ab |
B. ab |
C. ab |
D. ab|

677. What does this Java statement print? Application

```
System.out.println("Java".compareTo("Jav") + " " + "apple".compareTo("banana"));
```

A. 1 -5
B. true false
C. 4 -1
D. 1 -1

678. What does this C++ code print? Assume <iostream> and <string> are included and using namespace std. Application

```
string s = "abc";
s += 'd';
s.insert(1, "XY");
s.erase(0, 2);
cout << s << " " << s.size();
```

A. Ybcd 4
B. XYbcd 5
C. bcd 3
D. aXYbcd 6

679. This Java code is meant to split a dotted version number. What does it print, and why? Analysis

```
String[] parts = "1.2.3".split(".");
System.out.println(parts.length);
```

A. 3, because the string has three numbers
B. 0, because split takes a regex and "." matches every character, leaving only empty strings that are then removed
C. 2, because there are two dots
D. It throws PatternSyntaxException

680. This C code compiles but may crash. What is the problem? Analysis

```
char *p = "hello";
p[0] = 'H';
printf("%s\n", p);
```

A. printf needs %c, not %s, for a char pointer
B. p[0] is out of bounds because indexing starts at 1
C. p points to a string literal, and modifying a literal is undefined behaviour; declare char p[] = "hello" instead
D. 'H' must be written "H" because p is a string

681. What is wrong with this C code? Assume <stdio.h> and <string.h> are included. Analysis

```
char d[3];
strncpy(d, "hello", 3);
printf("%s\n", d);
```

A. strncpy copies the whole of "hello", overflowing d
B. It prints "hel" safely because strncpy always terminates the string
C. strncpy cannot be used with arrays, only with pointers
D. strncpy copies 3 characters but adds no null terminator, so printf reads past the end of d (undefined behaviour)

682. This Python code should count the words in a sentence, but it prints 3. Why? Analysis

```
text = "hello  world"
print(len(text.split(" ")))
```

A. split(" ") treats each single space as a separator, so the double space creates an empty string; use split() with no argument
B. split counts the spaces, not the words
C. len() counts characters, not list items
D. split(" ") keeps the separators in the list

683. This Python code tries to grab each HTML tag, but prints ['bold']. Which change gives ['', '']? Analysis

```
import re
print(re.findall(r"<.*>", "<b>bold</b>"))
```

A. Use re.search instead of re.findall
B. Use the lazy pattern r"<.*?>"
C. Use the pattern r"<.+>"
D. Escape the angle brackets as r"\<.*\>"

684. This JavaScript code sorts version numbers stored as strings. What does it print, and why? Analysis

```
console.log(["10", "9", "2"].sort().join(","));
```

A. 2,9,10, because sort converts numeric strings to numbers
B. 10,9,2, because sort keeps the original order of strings
C. 10,2,9, because strings are compared character by character and "1" comes before "2" and "9"
D. 9,2,10, because sort orders by string length

685. A Python program reads two numbers with a = input() and b = input(). The user types 3 and 4, and print(a + b) shows 34. What is the fix? Analysis

A. Use print(a, b)
B. Use print(str(a) + str(b))
C. Use a.join(b)
D. Convert first: print(int(a) + int(b))

686. What does this Java code print? (The compiler folds constant expressions.) Analysis

```
String a = "ab";
String b = "a" + "b";
String c = "a";
String d = c + "b";
System.out.println((a == b) + " " + (a == d));
```

A. true false
B. true true
C. false false
D. false true

687. This C++ code should print "found" only when s contains "x". Where does it go wrong? Assume <iostream> and <string> are included. Analysis

```
std::string s = "xyz";
if (s.find("x")) std::cout << "found";
else std::cout << "missing";
```

A. It prints "found" correctly for every input
B. It prints "missing": find returns 0 for a match at the start, and 0 is false; compare with std::string::npos instead
C. It fails to compile because find returns size_t, not bool
D. It prints "found" because find returns true when it matches

688. This Python check never finds the word in log text such as "Disk Error at 10:02". Why? Analysis

```
log = "Disk Error at 10:02"
if "Error" in log.lower():
    print("alert")
```

A. The in operator only works on lists
B. lower() changes log in place, so the original text is lost
C. log.lower() has no capital letters, so "Error" can never be found; search for "error" instead
D. in checks whole words only

689. Why does this Python code print False although both strings display as "café"? Analysis

```
a = "caf\u00e9"
b = "cafe\u0301"
print(a == b, len(a), len(b))
```

A. == compares memory addresses for non-ASCII strings
B. Python cannot store accented characters without encode()
C. a is bytes and b is str
D. b uses 'e' plus a combining accent, so the code points differ; normalise both with unicodedata.normalize("NFC", ...)

690. What does this Java code print, and why? Analysis

```
StringBuilder x = new StringBuilder("hi");
StringBuilder y = new StringBuilder("hi");
System.out.println(x.equals(y) + " " + x.toString().equals(y.toString()));
```

A. false true, because StringBuilder does not override equals(), so it compares references
B. true true, because both hold the same characters
C. false false, because toString() creates new objects
D. true false, because toString() loses the contents

691. This Python code should print a file's three-letter extension, but prints "tx". Which fix is correct? Analysis

```
name = "notes.txt"
print(name[-3:-1])
```

A. Use name[-4:-1]
B. Use name[-3:], because the stop index is exclusive
C. Use name[-3:0]
D. Use name[3:]

692. A script splits CSV lines with line.split(","). The line name,"Khan, Ali",21 gives 4 fields instead of 3. What is the best fix? Analysis

A. Use line.split(", ") with a space
B. Use line.strip().split(",")
C. Parse with the csv module, which respects quoted fields
D. Use line.rsplit(",", 1)

693. Two Python functions check whether two words are anagrams. f("Dormitory", "dirty room") returns False. Which version handles case and spaces? Analysis

```
def f(a, b):
    return sorted(a) == sorted(b)
```

A. Compare set(a) with set(b)
B. Compare len(a) with len(b)
C. Compare a.lower() with b.lower()[::-1]
D. Compare sorted(a.replace(" ", "").lower()) with sorted(b.replace(" ", "").lower())

Arrays & Collections MCQs for NSCT

694. In Java, what is the main difference between an int[] array and an ArrayList<Integer>? Understanding

A. An array is indexed from 1; an ArrayList is indexed from 0
B. An array is always sorted; an ArrayList keeps insertion order
C. An array's length is fixed when it is created; an ArrayList grows and shrinks as you add or remove items
D. An array can hold objects only; an ArrayList can hold primitives only

695. In Java, what value does each element of new int[5] hold before you assign anything? Understanding

A. An unpredictable garbage value
B. -1
C. 0
D. null

696. In Java, how do you get the number of elements in an array a? Understanding

A. a.length()
B. a.length
C. a.size()
D. a.count

697. In Python, which of these is immutable? Understanding

A. A dict
B. A set
C. A list
D. A tuple

698. An array holds n elements and uses zero-based indexing. What is the index of the last element? Understanding

A. n
B. n + 1
C. 1
D. n - 1

699. What is the average-case time to look up a key in a well-implemented hash map (Java HashMap, Python dict)? Understanding

A. O(n)
B. O(n log n)
C. O(1)
D. O(log n)

700. Which Java Map implementation keeps its keys in sorted order? Understanding

A. Hashtable
B. LinkedHashMap
C. HashMap
D. TreeMap

701. In Python 3.7 and later, what does the language guarantee about the order of keys in a dict? Understanding

A. Keys are kept in order of most recent access
B. Keys are kept in insertion order
C. Key order is random on every run
D. Keys are kept in sorted order

702. In C++, what is the difference between v[i] and v.at(i) for a std::vector v? Understanding

A. at() returns a copy; operator[] returns a reference
B. Both check the index and throw the same exception
C. at() checks the index and throws std::out_of_range; operator[] does not check
D. operator[] checks the index; at() does not

703. In C++, what happens when code writes to a[5] in an array declared as int a[5]? Understanding

A. An ArrayIndexOutOfBoundsException is thrown
B. The array grows automatically to six elements
C. It is undefined behaviour; C++ does not check array bounds
D. The compiler always reports an error

704. What is the time complexity of inserting an element at the front of a Python list or Java ArrayList with n elements? Understanding

A. O(1), because only the first slot changes
B. O(n), because every existing element must shift one place
C. O(log n), because the list is halved
D. O(n^2), because the list is copied twice

705. Which statement about this Java declaration is true? Understanding

```
int[][] m = new int[3][];
```

A. Every row must later be given exactly 3 columns
B. Each row can later be given a different length, making a jagged array
C. It is a compile error because the second size is missing
D. It creates a 3 by 3 array of zeros

706. Which collection type automatically rejects duplicate elements? Understanding

A. A list
B. A queue
C. A set
D. An array

707. Why does Python raise TypeError for d[[1, 2]] = 3? Understanding

A. A list is mutable and unhashable, so it cannot be a dict key
B. Dict values cannot be integers
C. The list has more than one element
D. Dict keys must be strings

708. In Python, what does a[-1] refer to for a non-empty list a? Understanding

A. The length of the list
B. The element before the first, which raises an error
C. The first element
D. The last element

709. In C++, how do std::map and std::unordered_map differ? Understanding

A. std::map stores values only; std::unordered_map stores key-value pairs
B. std::unordered_map keeps keys sorted; std::map does not
C. std::map keeps keys sorted (usually a balanced tree, O(log n)); std::unordered_map uses hashing (average O(1))
D. std::map allows duplicate keys; std::unordered_map does not

710. What does this Python code print? Application

```
d = {'a': 1, 'a': 2}
print(d)
```

A. {'a': 1}
B. {'a': 1, 'a': 2}
C. An error, because the key is duplicated
D. {'a': 2}

711. What does this Python code print? Application

```
print(len({1, 2, 2, 3, 3, 3}))
```

A. 6
B. 3
C. 1
D. 4

712. int a[4][5] is a C++ array stored in row-major order. It starts at address 1000 and each int takes 4 bytes. What is the address of a[2][3]? Application

A. 1052
B. 1034
C. 1068
D. 1046

713. What does this Python code print? Application

```
a = [1, 2]
b = a
b.append(3)
print(a)
```

A. [3]
B. [1, 2, 3]
C. [1, 2]
D. An error, because a cannot be changed through b

714. What does this Python code print? Application

```
a = [1, 2, 3]
b = a[:]
b.append(4)
print(len(a), len(b))
```

A. 3 4
B. 4 4
C. 4 3
D. 3 3

715. What does this Python code print? Application

```
a = [1, 2]
a.append([3, 4])
print(len(a))
```

A. 2
B. 3
C. 1
D. 4

716. What does this Python code print? Application

```
d = {'x': 1, 'y': 2}
print(d.get('z', 0))
```

A. KeyError
B. 'z'
C. None
D. 0

717. What does this Python code print? Application

```
words = "to be or not to be".split()
freq = {}
for w in words:
    freq[w] = freq.get(w, 0) + 1
print(freq)
```

A. {'to': 1, 'be': 1, 'or': 1, 'not': 1}
B. {'to': 2, 'be': 2, 'or': 1, 'not': 1}
C. {'be': 2, 'not': 1, 'or': 1, 'to': 2}
D. {'to': 2, 'be': 2}

718. What does this Python code print? Application

```
A = {1, 2, 3, 4}
B = {2, 4, 6}
print(sorted(A - B), sorted(A ^ B))
```

A. [1, 3] [1, 3, 6]
B. [2, 4] [1, 3, 6]
C. [1, 3, 6] [1, 3]
D. [1, 3] [2, 4]

719. What does this Python code print? Application

```
m = [[1, 2, 3],
     [4, 5, 6],
     [7, 8, 9]]
print(sum(m[i][i] for i in range(3)))
```

A. 45
B. 6
C. 12
D. 15

720. What does this Python code print? Application

```
m = [[1, 2, 3], [4, 5, 6]]
t = [[m[i][j] for i in range(2)] for j in range(3)]
print(t)
```

A. [[1, 4], [2, 5], [3, 6]]
B. [[1, 2, 3], [4, 5, 6]]
C. [[4, 1], [5, 2], [6, 3]]
D. [[1, 2], [3, 4], [5, 6]]

721. In C++ a 4-byte int is assumed. For int a[3][4];, what are the number of elements and sizeof(a)? Application

A. 7 elements and 28 bytes
B. 12 elements and 12 bytes
C. 12 elements and 48 bytes
D. 3 elements and 12 bytes

722. What does this Java code print? Application

```
int[] a = {1, 2, 3};
int[] b = a;
b[0] = 9;
System.out.println(a[0]);
```

A. Compilation error
B. 0
C. 1
D. 9

723. What does this Java code print? Application

```
Map<String, Integer> m = new HashMap<>();
m.put("a", 1);
int old = m.put("a", 2);
System.out.println(m.size() + " " + old);
```

A. 1 1
B. 1 2
C. 2 2
D. 2 1

724. What does this Python code print, and why? Analysis

```
grid = [[0] * 3] * 3
grid[0][0] = 1
print(grid)
```

A. [[1, 0, 0], [0, 0, 0], [0, 0, 0]], because only row 0 changed
B. An error, because * cannot be used on nested lists
C. [[1, 1, 1], [0, 0, 0], [0, 0, 0]], because * copies the value across the row
D. [[1, 0, 0], [1, 0, 0], [1, 0, 0]], because all three rows are the same list object

725. What does this Java code print? Analysis

```
List<Integer> l = new ArrayList<>(List.of(10, 20, 30));
l.remove(1);
System.out.println(l);
```

A. [20, 30]
B. [10, 30]
C. [10, 20]
D. [10, 20, 30]

726. In Java, a for-each loop over an ArrayList calls list.remove(x) when x is negative. The program throws an exception. Which one, and what is the fix? Analysis

A. ClassCastException; cast x to int before removing
B. ConcurrentModificationException; use an Iterator and call iterator.remove(), or list.removeIf(...)
C. ArrayIndexOutOfBoundsException; loop from the end instead of the start
D. NullPointerException; check x != null before removing

727. This Python code is meant to remove all even numbers. What does it print? Analysis

```
nums = [2, 4, 6, 7]
for n in nums:
    if n % 2 == 0:
        nums.remove(n)
print(nums)
```

A. [4, 7]
B. [2, 4, 6, 7]
C. [4, 6, 7]
D. [7]

728. After this Python code runs, nums is None. Why? Analysis

```
nums = [3, 1, 2]
nums = nums.sort()
print(nums)
```

A. list.sort() sorts in place and returns None, which is assigned to nums
B. sort() only works on lists of strings
C. sort() deletes the list after sorting
D. nums is a tuple, so sorting fails silently

729. A sign-up form must check quickly whether a username is already taken, among millions of names. Which collection fits best? Analysis

A. A linked list
B. A 2D array of characters
C. A hash set (HashSet / Python set)
D. An unsorted ArrayList searched with contains()

730. A Java program stores marks against student IDs and must print them in increasing ID order. Which type fits best? Analysis

A. TreeMap<Integer, Integer>
B. HashSet<Integer>
C. HashMap<Integer, Integer>
D. int[] marks of size 2

731. What does this Java code print? Analysis

```
int[] a = {1, 2};
int[] b = {1, 2};
System.out.println((a == b) + " " + a.equals(b) + " " + Arrays.equals(a, b));
```

A. false false true
B. false true true
C. true false true
D. true true true

732. A Java class Point has fields x and y but does not override equals() or hashCode(). What does this print? Analysis

```
Set<Point> s = new HashSet<>();
s.add(new Point(1, 2));
s.add(new Point(1, 2));
System.out.println(s.size());
```

A. 1
B. 0
C. 2
D. It throws an exception

733. In Java, this code compiles but fails at run time. What happens? Analysis

```
int[] a = new int[3];
a[3] = 5;
```

A. The value 5 is written to a[0] instead
B. A compile error, because 3 is a constant
C. The array grows to 4 elements and stores 5
D. ArrayIndexOutOfBoundsException, because valid indices are 0 to 2

734. In Python, how does a.append(x) differ from a.extend(x) when x is the list [4, 5]? Understanding

A. append adds the whole list as one element; extend adds 4 and 5 as two separate elements
B. append adds 4 and 5 separately; extend adds the list as one element
C. They do the same thing
D. append returns a new list; extend changes a in place

735. What must be true of an array before binary search can be used on it? Understanding

A. It must have an even number of elements
B. It must be sorted
C. It must contain no negative numbers
D. It must be stored in a linked list

736. In Java, why is List<int> a compile error? Understanding

A. List is an abstract class that cannot be used as a type
B. int values cannot be stored in any collection
C. Generic type arguments must be reference types, so you write List<Integer> and rely on autoboxing
D. List<int> is valid only inside a method

737. Which Java Map implementation keeps its entries in insertion order? Understanding

A. HashMap
B. TreeMap
C. Hashtable
D. LinkedHashMap

738. Which operation is O(1) for an array but O(n) for a singly linked list? Understanding

A. Reading the element at index i
B. Inserting at the front
C. Checking whether the structure is empty
D. Removing the first element

739. What does a C++ std::set<int> guarantee about the values it stores? Understanding

A. Values keep their insertion order
B. Each value appears once and iteration is in sorted order
C. Duplicates are allowed but kept sorted
D. Values are stored in a hash table with no order

740. In Python, what does collections.Counter("banana") store? Understanding

A. The total length of the string
B. The characters in sorted order
C. Each character mapped to how many times it appears
D. Each character mapped to its first index

741. In Java, what does m.getOrDefault(k, 0) return when key k is not in the map m? Understanding

A. null
B. It throws NoSuchElementException
C. It inserts k with value 0 and returns null
D. 0

742. What can you assume about the iteration order of a Java HashSet? Understanding

A. Nothing; the order is not guaranteed and may change as the set grows
B. Elements come out in insertion order
C. Elements come out in sorted order
D. Elements come out in reverse insertion order

743. Appending to a Python list or Java ArrayList is described as amortised O(1). Why? Understanding

A. Each append allocates exactly one new slot, which is always O(1)
B. The backing array grows by a constant factor when full, so the occasional O(n) copy is spread over many cheap appends
C. The list is a linked list, so appends never copy
D. Appends are O(n) but the constant is very small

744. Which Java class does the Java documentation recommend for a stack or a queue in place of the legacy Stack class? Understanding

A. Vector
B. Hashtable
C. ArrayDeque
D. TreeSet

745. Why is collections.deque preferred over a list for a FIFO queue in Python? Understanding

A. deque can hold mixed types and a list cannot
B. A list cannot remove from the front at all
C. deque is sorted automatically
D. deque.popleft() is O(1), while list.pop(0) is O(n) because every element shifts

746. For a C++ std::vector v, what is the difference between v.size() and v.capacity()? Understanding

A. size() is how many elements it holds; capacity() is how many it can hold before it must reallocate
B. They are always equal
C. capacity() is the size in bytes; size() is the number of elements
D. size() is the maximum the vector can ever hold

747. Why can a Python set hold tuples of numbers but not lists? Understanding

A. Sets can only hold numbers
B. Set elements must be hashable, and lists are mutable and unhashable
C. Lists are too large to store in a set
D. Tuples are converted to strings inside a set

748. In Python, what is a frozenset? Understanding

A. A set that is sorted and cannot hold duplicates
B. A set that is stored on disk
C. An immutable set, so it is hashable and can be a dict key or an element of another set
D. A set that raises an error when you test membership

749. In Python, d = {'a': 1}; k = d.keys(); d['b'] = 2. What does k contain afterwards? Understanding

A. Only 'a', because keys() made a copy
B. Nothing, because adding a key clears the view
C. It raises RuntimeError when k is used
D. Both 'a' and 'b', because keys() returns a live view of the dict

750. In column-major order (as used by Fortran and MATLAB), which elements of a 2D array are stored next to each other in memory? Understanding

A. Consecutive elements of the same column
B. Consecutive elements of the same row
C. Elements on the main diagonal
D. Elements are scattered with no fixed pattern

751. In Java, is int[] a = new int[0]; legal? Understanding

A. No; array sizes must be at least 1
B. Yes; it creates an empty array whose length is 0
C. Yes, but a is null
D. It compiles but throws NegativeArraySizeException

752. In Java, what does Arrays.asList("a", "b") return, and what happens if you call add("c") on it? Understanding

A. A normal ArrayList; add appends "c"
B. An immutable Set; add returns false
C. A fixed-size list backed by an array; add throws UnsupportedOperationException
D. A copy of the array; add silently does nothing

753. Which of these maps accepts a null key? Understanding

A. java.util.TreeMap using natural ordering
B. java.util.Hashtable
C. java.util.concurrent.ConcurrentHashMap
D. java.util.HashMap

754. In C++, why must a function parameter for a 2D array give the column count, as in void f(int a[][4], int rows)? Understanding

A. The compiler needs the row length to compute the address of a[i][j] in row-major order
B. It is only a convention; void f(int a[][], int rows) also compiles
C. The row count is needed instead, and the column count is optional
D. It stops the array from being copied

755. In Java, which sorting guarantee holds for Arrays.sort? Understanding

A. Both object and primitive sorts are unstable
B. Sorting an array of objects is stable; sorting an array of primitives is not guaranteed stable (it uses a quicksort)
C. Both use the same quicksort algorithm
D. Sorting objects is O(n^2) in the worst case

756. In Java, what happens at run time with Object[] o = new String[2]; o[0] = 1;? Understanding

A. It works; o[0] now holds the Integer 1
B. A compile error, because 1 is not a String
C. ArrayStoreException, because arrays are covariant and the JVM checks each store against the real element type
D. ClassCastException when o is created

757. In C, what does sizeof(arr) give inside void f(int arr[10]) on a 64-bit system? Understanding

A. 40, the size of ten ints
B. 10, the number of elements
C. A compile error, because the size in brackets is not allowed
D. The size of a pointer (typically 8), because an array parameter is adjusted to int*

758. What does this Python code print? Application

```
a = [1, 2, 3]
a.extend([4, 5])
a.append([6])
print(len(a))
```

A. 6
B. 5
C. 7
D. 3

759. What does this Python code print? Application

```
print([x * x for x in range(5) if x % 2])
```

A. [0, 4, 16]
B. [1, 9]
C. [1, 9, 25]
D. [0, 1, 4, 9, 16]

760. What does this Python code print? Application

```
s = {1, 2, 3}
s.add(2)
s.discard(5)
print(len(s))
```

A. 4
B. 2
C. 3
D. It raises KeyError

761. What does this Java code print? Application

```
int[] a = {4, 7, 1};
int s = 0;
for (int x : a) s += x;
System.out.println(s + " " + a.length);
```

A. 12 2
B. 3 12
C. 471 3
D. 12 3

762. A C array int a[10] starts at address 2000 and each int takes 4 bytes. What is the address of a[7]? Application

A. 2028
B. 2007
C. 2032
D. 2024

763. What does this C++ code print? Assume <iostream> and <set> are included and using namespace std. Application

```
set<int> s = {3, 1, 3, 2};
for (int x : s) cout << x;
```

A. 3132
B. 123
C. 312
D. 1233

764. What does this Python code print? Application

```
a = [5, 3, 8, 1]
print(a[1:3], a[::-1][0])
```

A. [3, 8, 1] 1
B. [5, 3] 5
C. [3, 8] 1
D. [3, 8] 5

765. What does this Python code print? Application

```
a = [1, 2, 3, 4, 5]
a[1:3] = [9]
print(a)
```

A. [1, 9, 3, 4, 5]
B. [1, [9], 4, 5]
C. [1, 9, 9, 4, 5]
D. [1, 9, 4, 5]

766. What does this Python code print? Application

```
d = {'a': 1}
d.setdefault('b', []).append(2)
d.setdefault('a', 5)
print(d)
```

A. {'a': 1, 'b': [2]}
B. {'a': 5, 'b': [2]}
C. {'a': 1, 'b': []}
D. {'a': 5, 'b': 2}

767. What does this Python code print? Application

```
from collections import Counter
print(Counter("banana").most_common(1))
```

A. [('b', 1)]
B. [('a', 3)]
C. ('a', 3)
D. {'a': 3}

768. What does this Python code print? Application

```
d = {'x': 3, 'y': 1, 'z': 2}
print(sorted(d, key=d.get))
```

A. ['x', 'y', 'z']
B. [1, 2, 3]
C. ['x', 'z', 'y']
D. ['y', 'z', 'x']

769. What does this Python code print? Application

```
print({1, 2, 3} & {2, 3, 4}, {1, 2} <= {1, 2, 3})
```

A. {2, 3} True
B. {1, 2, 3, 4} True
C. {2, 3} False
D. {1, 4} True

770. What does this JavaScript code print? Application

```
const a = [1, 2, 3];
a.push(4);
const b = a.map(x => x * 2).filter(x => x > 4);
console.log(b.join(","));
```

A. 2,4,6,8
B. 6,8
C. 3,4
D. 8

771. What does this JavaScript code print? Application

```
const a = [1, 2, 3];
console.log(a.length, a[5]);
```

A. 6 undefined
B. 3 null
C. 3 undefined
D. It throws RangeError

772. What does this Java code print? Application

```
List<String> l = new ArrayList<>();
l.add("a");
l.add("b");
l.add(0, "c");
l.set(2, "d");
System.out.println(l);
```

A. [c, d, b]
B. [a, b, c, d]
C. [d, a, b]
D. [c, a, d]

773. What does this Java code print? (java.util.* is imported.) Application

```
int[] a = {5, 2, 9, 1};
Arrays.sort(a);
System.out.println(Arrays.toString(a) + " " + Arrays.binarySearch(a, 5));
```

A. [1, 2, 5, 9] 2
B. [1, 2, 5, 9] 0
C. [5, 2, 9, 1] 0
D. [1, 2, 5, 9] 3

774. What does this Java code print? Application

```
Map<String, Integer> m = new TreeMap<>();
m.put("pear", 3);
m.put("apple", 5);
m.put("fig", 1);
System.out.println(m);
```

A. {pear=3, apple=5, fig=1}
B. {apple=5, fig=1, pear=3}
C. {fig=1, pear=3, apple=5}
D. {apple:5, fig:1, pear:3}

775. int a[3][4] is stored in column-major order starting at address 500, and each element takes 2 bytes. What is the address of a[1][2]? (Indexes start at 0.) Application

A. 512
B. 508
C. 514
D. 520

776. What does this Python code print? Application

```
m = [[1, 2], [3, 4]]
print([list(r) for r in zip(*m[::-1])])
```

A. [[1, 3], [2, 4]]
B. [[2, 4], [1, 3]]
C. [[4, 3], [2, 1]]
D. [[3, 1], [4, 2]]

777. What does this Java code print? (java.util.* is imported.) Application

```
Deque<Integer> d = new ArrayDeque<>();
d.push(1);
d.push(2);
d.offerLast(3);
System.out.println(d.pop() + " " + d.peekLast());
```

A. 2 3
B. 1 3
C. 3 1
D. 2 1

778. What does this Python code print, and why? Analysis

```
import copy
a = [[1, 2], [3]]
b = a.copy()
c = copy.deepcopy(a)
a[0].append(9)
print(b[0], c[0])
```

A. [1, 2] [1, 2], because both are copies
B. [1, 2, 9] [1, 2], because copy() is shallow and shares the inner lists
C. [1, 2, 9] [1, 2, 9], because both copies share the inner lists
D. [1, 2] [1, 2, 9], because deepcopy keeps references

779. What does this Python code print? Analysis

```
a = [1]
b = a
a += [2]
a = a + [3]
print(b)
```

A. [1]
B. [1, 2, 3]
C. [1, 2]
D. [1, 3]

780. This JavaScript code should sort numbers in ascending order. What does it print, and how do you fix it? Analysis

```
console.log([10, 1, 5].sort().join(","));
```

A. 1,5,10; the code is already correct
B. 10,5,1; sort is descending by default, so call reverse()
C. 10,1,5; sort does nothing without a comparator
D. 1,10,5; the default sort compares values as strings, so pass (a, b) => a - b

781. What does this C++ code print, and why? Assume <iostream>, <map> and <string> are included and using namespace std. Analysis

```
map<string, int> m;
m["b"] = 2;
m["a"] = 1;
if (m["c"] == 0) cout << m.size() << " " << m.begin()->first;
```

A. 3 a, because operator[] inserts a missing key with value 0, and the map is sorted
B. 2 b, because reading m["c"] does not change the map and order is insertion order
C. 2 a, because reading m["c"] does not change the map
D. Nothing, because m["c"] throws std::out_of_range

782. What does this Java code print, and why? (java.util.* is imported.) Analysis

```
int[] nums = {1, 2, 3};
List<int[]> l = Arrays.asList(nums);
System.out.println(l.size());
```

A. 3, because asList copies each int into the list
B. 1, because the int[] cannot be boxed element by element, so the whole array becomes one list element
C. A compile error, because asList needs Integer arguments
D. 0, because primitives are skipped

783. This Java code compares two values read from a list and prints false. Why? (java.util.* is imported.) Analysis

```
List<Integer> l = List.of(1000, 1000);
System.out.println(l.get(0) == l.get(1));
```

A. List.of stores copies as strings, so == fails
B. Integer cannot hold 1000
C. == compares Integer references, and values outside -128..127 are not cached, so the two boxes are different objects; use equals()
D. == on Integer always returns false

784. What is wrong with this Python function? Analysis

```
def add_item(item, basket=[]):
    basket.append(item)
    return basket

print(add_item(1), add_item(2))
```

A. It prints [1] [2], so nothing is wrong
B. append returns None, so it prints None None
C. A list cannot be a default argument, so it raises SyntaxError
D. The default list is created once and shared by every call, so it prints [1, 2] [1, 2]; use basket=None and create a new list inside

785. This Python code should add every number in the list, but prints 6 instead of 10. What is the bug? Analysis

```
a = [1, 2, 3, 4]
total = 0
for i in range(len(a) - 1):
    total += a[i]
print(total)
```

A. range(len(a) - 1) stops before the last index; use range(len(a)) or sum(a)
B. total should start at 1
C. a[i] should be a[i + 1]
D. range starts at 1, so the first element is skipped

786. A print shop handles jobs strictly in the order they arrive. Which collection fits best? Analysis

A. A stack
B. A queue, such as Java's ArrayDeque or Python's collections.deque
C. A sorted set
D. A hash map keyed by file name

787. A program counts how often each word appears in a 10-million-word file. Which approach is most efficient? Analysis

A. A list of words, calling list.count(w) for each word
B. A sorted array, re-sorted after every insert
C. A hash map from word to count (dict, HashMap or Counter), updated once per word
D. A set of words, since it stores each word once

788. This Python code checks whether 7 is in the list, but crashes. Which fix is correct? Analysis

```
a = [1, 3, 5]
if a.index(7) >= 0:
    print("found")
```

A. Use if a.index(7) != -1:
B. Use if a.find(7) >= 0:
C. Wrap the list as tuple(a) before calling index()
D. Use if 7 in a:, because index() raises ValueError when the value is missing

789. This Python code tries to remove keys with zero stock. What happens? Analysis

```
stock = {'pen': 0, 'ink': 4, 'pad': 0}
for k in stock:
    if stock[k] == 0:
        del stock[k]
```

A. RuntimeError, because the dict changes size during iteration; loop over list(stock) instead
B. It removes 'pen' and 'pad' correctly
C. It removes only 'pen' and skips 'pad' silently
D. KeyError, because del cannot remove dict keys

790. A Java program calls Arrays.binarySearch(a, 9) on int[] a = {5, 2, 9, 1} without sorting it first. What should you conclude about the result? Analysis

A. It always returns 2, the true position of 9
B. It is undefined by the API: it may or may not find 9, so sort a first
C. It always throws IllegalArgumentException for an unsorted array
D. It sorts the array automatically, then searches

791. What does this Java code print, and how should it be fixed? (java.util.* is imported.) Analysis

```
int[][] g = {{1, 2}, {3, 4}};
System.out.println(Arrays.toString(g));
```

A. [[1, 2], [3, 4]], so no fix is needed
B. [1, 2, 3, 4]; use Arrays.toString(g[0])
C. Something like [[I@1b6d3586, [I@4554617c]; use Arrays.deepToString(g)
D. A compile error, because toString needs a 1D array

792. Which list ends up with three independent rows in Python, so that changing grid[0][0] changes only row 0? Analysis

A. grid = [[0] * 3] * 3
B. row = [0] * 3; grid = [row, row, row]
C. grid = [[0, 0, 0]] * 3
D. grid = [[0] * 3 for _ in range(3)]

Object-Oriented Programming (OOP) MCQs for NSCT

793. What is the relationship between a class and an object? Understanding

A. A class and an object are two names for the same thing
B. A class holds data only; an object holds methods only
C. An object is a blueprint; a class is an instance created from it
D. A class is a blueprint; an object is an instance created from it

794. In Java and C++, which statement about a constructor is correct? Understanding

A. A class can have only one constructor
B. It must be declared static
C. It has the same name as the class and no return type, not even void
D. It must return the newly created object with a return statement

795. In Java, which keywords does a class use to inherit from a class and to implement an interface? Understanding

A. implements for a class, extends for an interface
B. extends for both
C. inherits for both
D. extends for a class, implements for an interface

796. Which statement about inheritance in Java is correct? Understanding

A. A class cannot both extend a class and implement an interface
B. A class can extend many classes and implement only one interface
C. A class can extend only one class but can implement many interfaces
D. A class can extend many classes and implement many interfaces

797. How does method overloading differ from method overriding? Understanding

A. Overloading uses the same name with different parameter lists and is resolved at compile time; overriding redefines an inherited method with the same signature and is resolved at run time
B. Overloading happens only between a parent and a child class; overriding happens only within one class
C. Overloading is resolved at run time; overriding is resolved at compile time
D. They are two names for the same feature

798. Which description best defines encapsulation? Understanding

A. Creating a new class from an existing class
B. Bundling data with the methods that use it and restricting direct access to the data
C. Running several methods at the same time
D. Letting one name refer to methods of different classes

799. Which description best defines abstraction in OOP? Understanding

A. Making every field public so other classes can use it
B. Storing objects in a database table
C. Copying the code of one class into another
D. Exposing what an object does through a simple interface while hiding how it does it

800. In Java, can two methods in the same class have the same name and the same parameter list but different return types? Understanding

A. Yes, but only if one of them is static
B. No; it is a compile error because the return type is not part of the signature used for overloading
C. Yes; the compiler picks a method by the return type the caller expects
D. Yes, but only if one of the return types is void

801. Which statement about a Java abstract class is correct? Understanding

A. It can be instantiated directly with new
B. All of its methods must be abstract
C. It cannot have constructors
D. It cannot be instantiated, but it can have constructors, fields and concrete methods

802. In Java, a field declared inside an interface, such as int MAX = 10;, is implicitly what? Understanding

A. private
B. an instance field of each implementing object
C. public static final
D. protected and mutable

803. C++ has no interface keyword. What makes a C++ class abstract? Understanding

A. It has no constructor
B. It declares at least one pure virtual function, written virtual void f() = 0;
C. All of its members are private
D. It is declared with the abstract keyword

804. In Java, which classes can access a member declared with no access modifier (package-private)? Understanding

A. Every class in the program
B. Only subclasses, in any package
C. Only classes in the same package
D. Only the class itself

805. Which statement about a protected member in Java is correct? Understanding

A. It is accessible only to subclasses, never to non-subclasses in the same package
B. It is accessible only inside its own class
C. It is accessible to every class in the same package and to subclasses in other packages
D. It means exactly the same as protected in C++

806. In C++, what is the default access level of members declared without a label? Understanding

A. private in both a class and a struct
B. public in both a class and a struct
C. protected in a class, public in a struct
D. private in a class, public in a struct

807. In C++, B and C both inherit from A, and D inherits from both B and C. How do you make D hold only one copy of A? Understanding

A. Make A an abstract class
B. Declare the inheritance as virtual: class B : virtual public A and class C : virtual public A
C. Declare D as a struct instead of a class
D. It is impossible; C++ does not allow D to inherit from both B and C

808. Which Java method can a subclass override? Understanding

A. A final method
B. A private method
C. A public, non-final instance method
D. A static method

809. What does this Java code print? Application

```
class Animal { String sound() { return "..."; } }
class Dog extends Animal { String sound() { return "Woof"; } }

// in main:
Animal a = new Dog();
System.out.println(a.sound());
```

A. Woof
B. Animal
C. ...
D. Compilation error

810. What does this Python code print? Application

```
class Counter:
    count = 0
    def __init__(self):
        Counter.count += 1

a = Counter(); b = Counter(); c = Counter()
print(Counter.count)
```

A. 1
B. An error, because count belongs to the class
C. 0
D. 3

811. What does this Python code print? Application

```
class Duck:
    def speak(self): return "Quack"

class Robot:
    def speak(self): return "Beep"

for x in (Duck(), Robot()):
    print(x.speak(), end=" ")
```

A. An error, because Robot does not inherit from Duck
B. Quack Beep
C. Beep Quack
D. Quack Quack

812. What does this Python code print? Application

```
class Animal: pass
class Dog(Animal): pass

print(isinstance(Dog(), Animal), isinstance(Animal(), Dog))
```

A. True False
B. False False
C. True True
D. False True

813. What does this Java code print? Application

```
class A { A() { System.out.print("A"); } }
class B extends A { B() { System.out.print("B"); } }
class C extends B { C() { System.out.print("C"); } }

// in main:
new C();
```

A. ABC
B. CBA
C. C
D. AC

814. What does this Java code print? Application

```
class A { String hi() { return "A"; } }
class B extends A { String hi() { return "B" + super.hi(); } }
class C extends B { String hi() { return "C" + super.hi(); } }

// in main:
System.out.println(new C().hi());
```

A. CBA
B. CB
C. ABC
D. C

815. What does this Python code print? Application

```
class A:
    def who(self): return "A"
class B(A):
    def who(self): return "B"
class C(A):
    def who(self): return "C"
class D(B, C):
    pass

print(D().who(), [k.__name__ for k in D.__mro__])
```

A. B ['D', 'B', 'C', 'A', 'object']
B. B ['D', 'B', 'A', 'C', 'object']
C. A ['D', 'B', 'A', 'C', 'object']
D. C ['D', 'C', 'B', 'A', 'object']

816. What happens when this Python code runs? Application

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

s = Shape()
print(s.area())
```

A. NameError, because area has no body
B. TypeError, because Shape has an abstract method and cannot be instantiated
C. It prints None
D. It prints 0

817. What does this Python code print? Application

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y
    def __str__(self):
        return f"Point({self.x}, {self.y})"

print(Point(2, 3))
```

A. <__main__.Point object>
B. 2 3
C. (2, 3)
D. Point(2, 3)

818. What does this Python code print? Application

```
class Account:
    def __init__(self):
        self.__balance = 100

a = Account()
print(a._Account__balance)
print(a.__balance)
```

A. AttributeError on the first print
B. 100 and then 100
C. 100, then an AttributeError is raised
D. None and then 100

819. What does this Java code print? Application

```
class P {
    static String id() { return "P"; }
    String name() { return "P"; }
}
class C extends P {
    static String id() { return "C"; }
    String name() { return "C"; }
}

// in main:
P p = new C();
System.out.println(p.id() + p.name());
```

A. PP
B. PC
C. CP
D. CC

820. What does this Java code print? Application

```
static void f(int x)    { System.out.print("int "); }
static void f(double x) { System.out.print("double "); }

// in main:
f(5); f(5L); f('A');
```

A. int long char
B. int double int
C. int double double
D. int int int

821. What does this Python code print? Application

```
class Shape:
    def __init__(self, name):
        self.name = name
    def area(self):
        return 0
    def describe(self):
        return f"{self.name}: {self.area()}"

class Square(Shape):
    def __init__(self, side):
        super().__init__("square")
        self.side = side
    def area(self):
        return self.side ** 2

print(Square(3).describe())
```

A. shape: 9
B. An AttributeError, because Shape has no side
C. square: 9
D. square: 0

822. What does this Java code print? Application

```
class Box {
    int w;
    Box(int w) { w = w; }
}

// in main:
System.out.println(new Box(5).w);
```

A. 5
B. null
C. Compilation error
D. 0

823. What does this C++ program print? Analysis

```
#include <iostream>
struct Base {
    void f() { std::cout << "Bf "; }
    virtual void g() { std::cout << "Bg "; }
};
struct Derived : Base {
    void f() { std::cout << "Df "; }
    void g() override { std::cout << "Dg "; }
};
int main() {
    Derived d;
    Base& r = d;
    r.f();
    r.g();
}
```

A. Bf Dg
B. Df Dg
C. Df Bg
D. Bf Bg

824. What does this Java code print? Analysis

```
class P { String n = "P"; String get() { return n; } }
class C extends P { String n = "C"; String get() { return n; } }

// in main:
P p = new C();
System.out.println(p.n + p.get());
```

A. CP
B. PP
C. CC
D. PC

825. A student writes two add methods in a Python class to "overload" it. What happens when this runs? Analysis

```
class Calc:
    def add(self, a, b):
        return a + b
    def add(self, a, b, c):
        return a + b + c

print(Calc().add(1, 2))
```

A. SyntaxError, because a class cannot define a name twice
B. TypeError, because the second add replaced the first and needs argument c
C. It prints 3, because Python picks the two-argument version
D. It prints 3, because c defaults to 0

826. Why does this Python code raise an AttributeError? Analysis

```
class Person:
    def __init__(self, name):
        self.name = name

class Student(Person):
    def __init__(self, name, roll):
        self.roll = roll

s = Student("Ayesha", 7)
print(s.name)
```

A. A subclass cannot add new attributes such as roll
B. name is private to Person
C. Student must be declared with the inherits keyword
D. Student.__init__ never calls super().__init__(name), so name is never set

827. What does this Python code print? Analysis

```
class Dog:
    tricks = []
    def add(self, t):
        self.tricks.append(t)

a = Dog(); b = Dog()
a.add("sit")
print(b.tricks)
```

A. ['sit']
B. An AttributeError
C. ['sit', 'sit']
D. []

828. In Java, Dog extends Animal. What happens with this code? Analysis

```
Animal a = new Animal();
Dog d = (Dog) a;
```

A. It compiles, but throws ClassCastException at run time because the object is not a Dog
B. It runs and d is set to null
C. It fails to compile because downcasting is not allowed
D. It runs and d behaves as a Dog with default values

829. Why does this Java code fail to compile? Analysis

```
class Base {
    public String describe() { return "base"; }
}
class Child extends Base {
    protected String describe() { return "child"; }
}
```

A. An overriding method cannot have weaker access than the method it overrides
B. toString() is final in Object
C. protected methods cannot be overridden
D. Subclasses must call super.describe()

830. In C++, Base has a non-virtual destructor, and a program runs Base* p = new Derived(); delete p;. What is the problem? Analysis

A. Base's destructor is skipped and only Derived's runs
B. The code does not compile because Base* cannot point to a Derived
C. The behaviour is undefined; in practice Derived's destructor usually does not run, leaking its resources
D. Nothing; C++ always calls destructors through dynamic dispatch

831. A Car needs an Engine. Which design models this best? Analysis

A. Inheritance: Engine extends Car
B. Inheritance: Car extends Engine
C. Composition: Car has an Engine field
D. Make Car and Engine the same class

832. In Java, several shape classes must share a colour field, a constructor that sets it and a common display() method, while each provides its own area(). Which is the best fit? Analysis

A. A final class Shape
B. An abstract class Shape with the field, constructor and display(), and an abstract area()
C. An interface Shape holding the colour field
D. A separate unrelated class for each shape with copied code

833. In Java, a class declares `static int total;`. How many copies of total exist when the program has created five objects of that class? Understanding

A. One copy, shared by all five objects and stored with the class
B. Five copies, one inside each object
C. Six copies: one for the class and one per object
D. None until a static method first assigns it

834. In Java, what does the keyword `this` refer to inside an instance method? Understanding

A. The object on which the method was called
B. The class in which the method is declared
C. The parent-class part of the object
D. The method that called the current method

835. A Java class declares only the constructor `Book(String title)`. What happens when another class runs `new Book()`? Understanding

A. It compiles, and title is set to null by the default constructor
B. Compile error, because Java adds a default no-argument constructor only when a class declares no constructor at all
C. It compiles, and title is set to an empty string
D. It compiles but throws an exception at run time

836. Which Java modifier on a class declaration stops any other class from extending it, as with java.lang.String? Understanding

A. static
B. abstract
C. final
D. private

837. In Python, what is the first parameter of an ordinary instance method conventionally named, and what does it receive? Understanding

A. this, which receives the instance the method was called on
B. self, which receives the class the method belongs to
C. cls, which receives the instance the method was called on
D. self, which receives the instance the method was called on

838. In Java, a class is declared as `class Car { }` with no extends clause. What is its direct superclass? Understanding

A. It has no superclass
B. java.lang.Object
C. java.lang.Class
D. java.lang.Record

839. A variable of type Shape refers first to a Circle and later to a Square, and `shape.area()` runs the Circle version and then the Square version. Which OOP concept does this show? Understanding

A. Encapsulation
B. Abstraction
C. Composition
D. Polymorphism

840. Which of these languages allow one class to inherit from two or more classes directly? Understanding

A. Java and Python, but not C++
B. C++ and Python, but not Java
C. C++ and Java, but not Python
D. Only C++

841. Which pairing of polymorphism type and binding time is correct in Java and C++? Understanding

A. Overloading is resolved at run time; overriding is resolved at compile time
B. Both overloading and overriding are resolved at run time
C. Overloading is resolved at compile time; overriding of virtual methods is resolved at run time
D. Both overloading and overriding are resolved at compile time

842. In a UML class diagram, which relationship means the part is owned by the whole and is destroyed with it, and how is it drawn? Understanding

A. Aggregation, drawn with a filled diamond at the part's end
B. Composition, drawn with a filled diamond at the whole's end
C. Aggregation, drawn with a hollow diamond at the whole's end
D. Generalisation, drawn with a hollow triangle at the whole's end

843. In a UML class diagram, what do the visibility markers +, - and # before a member mean? Understanding

A. public, protected, private
B. static, private, package
C. public, private, protected
D. public, private, static

844. In C++, what is a friend function of a class? Understanding

A. A member function that can be called without an object
B. A function inherited from a base class that a derived class can override
C. A function that two classes share through multiple inheritance
D. A non-member function that the class allows to access its private and protected members

845. Since Java 8, which of these can a Java interface contain? Understanding

A. Instance fields, with a separate copy in every implementing object
B. Constructors that run when an implementing class is instantiated
C. default methods with a body, which implementing classes inherit unless they override them
D. protected abstract methods visible only to implementing classes

846. In Python, how does a method decorated with @classmethod differ from one decorated with @staticmethod? Understanding

A. A classmethod receives the class as its first argument (cls); a staticmethod receives no automatic first argument
B. A classmethod receives the instance as its first argument; a staticmethod receives the class
C. A classmethod can only be called on the class; a staticmethod can only be called on an instance
D. They are identical; the names are interchangeable

847. In C++, `Point a(1, 2);` is followed by `Point b = a;`. Which special member function creates b? Understanding

A. The copy assignment operator
B. The default constructor
C. The copy constructor
D. The destructor

848. What does the Open/Closed Principle (the O in SOLID) state? Understanding

A. Classes should expose all fields publicly but keep methods private
B. A class should be open to one caller and closed to all others
C. Every class must be declared either open (non-final) or closed (final)
D. Software entities should be open for extension but closed for modification

849. Which combination of cohesion and coupling is the usual goal of good object-oriented design? Understanding

A. High cohesion within each class and low coupling between classes
B. Low cohesion within each class and high coupling between classes
C. High cohesion and high coupling
D. Low cohesion and low coupling

850. In Java, which pair of features is the standard way to implement the Singleton pattern? Understanding

A. A public constructor plus a final class
B. An abstract class plus a protected constructor
C. An interface plus a default method that creates the instance
D. A private constructor plus a static method or field that returns the single instance

851. In Java, a subclass constructor does not call super(...) explicitly. What happens? Understanding

A. The compiler inserts a call to the superclass's no-argument constructor, and compilation fails if that constructor does not exist
B. The superclass part of the object is left uninitialised
C. The superclass constructor with the most parameters is called with default values
D. The superclass constructor is called only after the subclass constructor finishes

852. What does the Liskov Substitution Principle require? Understanding

A. Every subclass must override every method of its base class
B. Objects of a subclass must be usable anywhere the base class is expected without breaking the program's correctness
C. A subclass may not add new methods that the base class lacks
D. Base classes must be abstract so they can never be substituted

853. In Java, Animal declares `Animal create()`. Which declaration in `class Dog extends Animal` is a legal override? Understanding

A. Dog create()
B. Object create()
C. void create()
D. Animal create() throws Exception

854. In C++, what is object slicing? Understanding

A. Splitting a large object across several memory pages
B. Copying a derived-class object into a base-class object by value, so only the base part is kept and derived data and overrides are lost
C. Deleting a derived object through a base pointer without a virtual destructor
D. Accessing only the public members of an object through a pointer

855. Which pair of modifiers on the same Java method always causes a compile error? Understanding

A. public and static
B. protected and final
C. private and static
D. abstract and final

856. In Java, a class overrides equals(Object). According to the Object contract, what else must it do? Understanding

A. Override toString() so that equal objects print the same text
B. Make the class final so equals cannot be overridden again
C. Override hashCode() so that unequal objects always return different hash codes
D. Override hashCode() so that any two objects that are equal also return the same hash code

857. What does this Python code print? Application

```
class Car:
    wheels = 4

c1 = Car()
c2 = Car()
c1.wheels = 3
print(c1.wheels, c2.wheels, Car.wheels)
```

A. 3 4 4
B. 3 3 3
C. 3 3 4
D. 3 4 3

858. What does this Python code print? Application

```
class Playlist:
    def __init__(self, songs):
        self.songs = songs
    def __len__(self):
        return len(self.songs)

p = Playlist(["a", "b", "c"])
print(len(p))
```

A. 1
B. 3
C. TypeError: object of type 'Playlist' has no len()
D. 0

859. What does this Java code print? Application

```
static int area(int s)        { return s * s; }
static int area(int l, int w) { return l * w; }

// in main:
System.out.println(area(3) + area(2, 5));
```

A. 9
B. 10
C. 19
D. 910

860. What does this C++ program print? Application

```
#include <iostream>
struct A {
    A()  { std::cout << "A"; }
    ~A() { std::cout << "a"; }
};
struct B : A {
    B()  { std::cout << "B"; }
    ~B() { std::cout << "b"; }
};
int main() {
    B obj;
}
```

A. ABba
B. ABab
C. BAba
D. BAab

861. What does this Python code print? Application

```
class Vec:
    def __init__(self, x, y):
        self.x, self.y = x, y
    def __add__(self, other):
        return Vec(self.x + other.x, self.y + other.y)
    def __repr__(self):
        return f"Vec({self.x}, {self.y})"

print(Vec(1, 2) + Vec(3, 4))
```

A. Vec(1, 2)Vec(3, 4)
B. TypeError: unsupported operand type(s) for +
C. Vec(3, 8)
D. Vec(4, 6)

862. What does this Python code print? Application

```
class Temp:
    def __init__(self, c):
        self._c = c
    @property
    def f(self):
        return self._c * 9 / 5 + 32

t = Temp(100)
t._c = 0
print(t.f)
```

A. 212.0
B. 32.0
C. 32
D. AttributeError, because f is a method and needs ()

863. What does this Python code print? Application

```
class A:
    def go(self): return "A"
class B(A):
    def go(self): return "B" + super().go()
class C(A):
    def go(self): return "C" + super().go()
class D(B, C):
    def go(self): return "D" + super().go()

print(D().go())
```

A. DBA
B. DBACA
C. DBCA
D. DCBA

864. What does this Java code print? Application

```
class Ticket {
    static int next = 1;
    int id;
    Ticket() { id = next++; }
}

// in main:
Ticket a = new Ticket();
Ticket b = new Ticket();
Ticket c = new Ticket();
System.out.println(a.id + " " + c.id + " " + Ticket.next);
```

A. 1 3 3
B. 3 3 4
C. 0 2 3
D. 1 3 4

865. What does this Java code print? Application

```
class Rect {
    int w, h;
    Rect()             { this(1);    System.out.print("A"); }
    Rect(int s)        { this(s, s); System.out.print("B"); }
    Rect(int w, int h) { this.w = w; this.h = h; System.out.print("C"); }
}

// in main:
Rect r = new Rect();
System.out.println(" " + r.w * r.h);
```

A. ABC 1
B. CBA 0
C. CBA 1
D. A 1

866. What does this C++ program print? Application

```
#include <iostream>
class Obj {
public:
    static int live;
    Obj()  { ++live; }
    ~Obj() { --live; }
};
int Obj::live = 0;
int main() {
    Obj a;
    {
        Obj b, c;
        std::cout << Obj::live << " ";
    }
    std::cout << Obj::live;
}
```

A. 3 1
B. 3 3
C. 2 0
D. 1 1

867. What does this Java code print? Application

```
interface Greeter {
    default String greet() { return "Hello"; }
}
class Polite implements Greeter {
    public String greet() { return "Good day"; }
}
class Plain implements Greeter { }

// in main:
Greeter g1 = new Polite(), g2 = new Plain();
System.out.println(g1.greet() + ", " + g2.greet());
```

A. Hello, Hello
B. Good day, Good day
C. Compile error, because Plain does not implement greet()
D. Good day, Hello

868. What does this Python code print? Application

```
class Date:
    def __init__(self, y, m, d):
        self.y, self.m, self.d = y, m, d
    @classmethod
    def from_str(cls, s):
        y, m, d = map(int, s.split("-"))
        return cls(y, m, d)

class MyDate(Date):
    pass

x = MyDate.from_str("2026-11-25")
print(type(x).__name__, x.m)
```

A. Date 11
B. MyDate 2026
C. Date 25
D. MyDate 11

869. What does this Python code print? Application

```
class P:
    def __init__(self, x):
        self.x = x

class Q(P):
    def __eq__(self, other):
        return self.x == other.x

print(P(1) == P(1), Q(1) == Q(1))
```

A. True True
B. False True
C. False False
D. True False

870. What does this Java code print? Application

```
class Base {
    Base() { show(); }
    void show() { System.out.print("Base "); }
}
class Child extends Base {
    int x = 5;
    void show() { System.out.print("Child " + x + " "); }
}

// in main:
new Child().show();
```

A. Child 0 Child 5
B. Base Child 5
C. Child 5 Child 5
D. Base Base

871. What does this C++ program print? Application

```
#include <iostream>
struct Animal {
    virtual const char* name() const { return "animal"; }
};
struct Cat : Animal {
    const char* name() const override { return "cat"; }
};
void byValue(Animal a)      { std::cout << a.name() << " "; }
void byRef(const Animal& a) { std::cout << a.name() << " "; }
int main() {
    Cat c;
    byValue(c);
    byRef(c);
}
```

A. cat cat
B. animal animal
C. cat animal
D. animal cat

872. What does this Java code print? Application

```
class Printer {
    void print(Object o) { System.out.print("Object "); }
    void print(String s) { System.out.print("String "); }
}

// in main:
Printer p = new Printer();
Object o = "hi";
p.print(o);
p.print("hi");
```

A. String String
B. Object String
C. Object Object
D. String Object

873. What does this Python code print? Application

```
class Sq:
    def __init__(self, s): self.s = s
    def area(self): return self.s * self.s

class Rect:
    def __init__(self, w, h): self.w, self.h = w, h
    def area(self): return self.w * self.h

shapes = [Sq(2), Rect(2, 3), Sq(1)]
print(sum(s.area() for s in shapes))
```

A. 11
B. 6
C. 9
D. AttributeError, because Sq and Rect share no base class

874. What does this Python code print? Application

```
class Employee:
    raise_pct = 10
    def __init__(self, pay):
        self.pay = pay
    def raised(self):
        return self.pay + self.pay * self.raise_pct // 100

class Manager(Employee):
    raise_pct = 20

print(Employee(1000).raised(), Manager(1000).raised())
```

A. 1100 1200
B. 1100 1100
C. 1200 1200
D. 1000 1200

875. What does this JavaScript code print? Application

```
class Animal {
  constructor(name) { this.name = name; }
  speak() { return `${this.name} makes a sound`; }
}
class Dog extends Animal {
  speak() { return `${super.speak()} and barks`; }
}
console.log(new Dog("Rex").speak());
```

A. Rex barks
B. undefined makes a sound and barks
C. ReferenceError, because Dog has no constructor that calls super()
D. Rex makes a sound and barks

876. What does this Java code print? Application

```
class Demo {
    static { System.out.print("S "); }
    { System.out.print("I "); }
    Demo() { System.out.print("C "); }
}

// in main:
new Demo(); new Demo();
```

A. S I C I C
B. S I C S I C
C. I C I C S
D. S C I C I

877. What does this Java code print? (B and A are in the same package.) Application

```
class A {
    private String id() { return "A"; }
    String show() { return id(); }
}
class B extends A {
    String id() { return "B"; }
}

// in main:
System.out.println(new B().show());
```

A. B
B. A
C. AB
D. Compile error, because B cannot redeclare id()

878. Bird has a fly() method. Penguin extends Bird and overrides fly() to throw UnsupportedOperationException, so code that loops over a List<Bird> and calls fly() crashes. Which principle does this design break most directly? Analysis

A. Single Responsibility Principle
B. Encapsulation
C. Dependency Inversion Principle
D. Liskov Substitution Principle

879. The age setter in this Python class rejects negative values, yet the code prints -5 instead of raising ValueError. Why? Analysis

```
class Person:
    def __init__(self, age):
        self._age = age
    @property
    def age(self):
        return self._age
    @age.setter
    def age(self, v):
        if v < 0:
            raise ValueError("negative")
        self._age = v

p = Person(-5)
print(p.age)
```

A. Properties are only checked when the object is printed
B. __init__ assigns self._age directly, which bypasses the setter; writing self.age = age would run the check
C. The setter runs, but ValueError is silently ignored inside __init__
D. @age.setter must be declared before @property for the check to work

880. What happens when the last line of this Python code runs, and why? Analysis

```
from abc import ABC, abstractmethod

class Payment(ABC):
    @abstractmethod
    def pay(self, amount): ...
    def receipt(self):
        return "ok"

class Card(Payment):
    def pay_amount(self, amount):
        return amount

c = Card()
```

A. A Card object is created, because Card defines a payment method
B. TypeError, because ABC subclasses cannot add new methods
C. TypeError, because Card never implements pay(), so it is still abstract
D. A Card object is created, and the error appears only when pay() is called

881. ReportService creates `new MySqlDatabase()` inside its own constructor, so its unit tests cannot use a fake database. Which change applies the Dependency Inversion Principle? Analysis

A. Make MySqlDatabase a subclass of ReportService
B. Make ReportService depend on a Database interface and pass the implementation into its constructor
C. Make the MySqlDatabase field public so tests can replace it
D. Copy the database code into ReportService so it has no dependency

882. What does this Python code print, and what is the design problem? Analysis

```
class A:
    def __init__(self): print("A", end=" ")
class B(A):
    def __init__(self): print("B", end=" "); A.__init__(self)
class C(A):
    def __init__(self): print("C", end=" "); A.__init__(self)
class D(B, C):
    def __init__(self):
        print("D", end=" "); B.__init__(self); C.__init__(self)

D()
```

A. D B C A; Python automatically runs each initialiser once
B. D B A; C's initialiser is skipped because B comes first in the MRO
C. D B A C A; A's initialiser runs twice because each class calls its parent by name instead of using super()
D. D C B A; the MRO runs the classes in reverse order

883. The author of this Java class wanted members to be read-only from outside. What does the code print, and what is the flaw? Analysis

```
import java.util.*;

class Team {
    private final List<String> members = new ArrayList<>();
    public List<String> getMembers() { return members; }
}

// in main:
Team t = new Team();
t.getMembers().add("Ali");
System.out.println(t.getMembers().size());
```

A. 0; the getter returns a copy, so adding to it does not affect the team
B. Compile error, because a final list cannot be added to
C. 1; the getter returns a reference to the private list itself, so callers can change it; return an unmodifiable view or a copy
D. It throws UnsupportedOperationException, because private lists are read-only outside the class

884. What does this Java code print, and why? Analysis

```
class Pt {
    int x;
    Pt(int x) { this.x = x; }
    public boolean equals(Pt o) { return o.x == x; }
}

// in main:
Object a = new Pt(1), b = new Pt(1);
System.out.println(a.equals(b) + " " + ((Pt) a).equals((Pt) b));
```

A. true true; both calls run the Pt version
B. false false; equals only compares references in Java
C. false true; equals(Pt) overloads equals(Object) instead of overriding it, so calls through Object use identity
D. true false; the cast forces Object's version to run

885. Every new shape forces someone to edit this Python function. Which refactor best applies polymorphism and the Open/Closed Principle? Analysis

```
def area(shape):
    if shape.kind == "circle":
        return 3.14 * shape.r ** 2
    elif shape.kind == "square":
        return shape.s ** 2
    elif shape.kind == "triangle":
        return 0.5 * shape.b * shape.h
```

A. Replace the if/elif chain with a match statement on shape.kind
B. Give each shape class its own area() method and call shape.area()
C. Move the function into a Utils class as a static method
D. Store the formulas as strings and evaluate them with eval()

886. What does this C++ program print, and why does it differ from the same design in Java? Analysis

```
#include <iostream>
struct Base {
    Base() { hello(); }
    virtual void hello() { std::cout << "Base "; }
};
struct Derived : Base {
    void hello() override { std::cout << "Derived "; }
};
int main() {
    Derived d;
    d.hello();
}
```

A. Derived Derived; virtual calls always dispatch to the most-derived override, as in Java
B. Base Derived; while Base's constructor runs the object is still only a Base, so the virtual call does not reach Derived
C. Base Base; virtual dispatch is disabled for the whole lifetime of objects built this way
D. Derived Base; the constructor call is dispatched late and the later call early

887. A student builds this Python stack by inheriting from list. A user then calls s.insert(0, 99), and pop() no longer returns the last item pushed. What is the best diagnosis? Analysis

```
class Stack(list):
    def push(self, x):
        self.append(x)

s = Stack()
s.push(1); s.push(2)
```

A. Stack must override insert() to call push(), and then it is correct
B. push should use insert(0, x) so the two methods agree
C. Python lists cannot be subclassed safely, so Stack should inherit from object and use a global list
D. Inheritance exposes every list method, so callers can break LIFO order; Stack should hold a private list (composition) and expose only push, pop and peek

888. Which scenario is an example of run-time (dynamic) polymorphism? Analysis

A. A class has two constructors, one with no parameters and one with a name
B. Math.max is called once with two ints and once with two doubles
C. A List<Employee> holds Manager and Engineer objects, and calling pay() on each element runs the subclass's own pay()
D. A private field is read through a public getter

889. Why does this Python code raise an error? Analysis

```
class Greeter:
    def hello(self):
        return "hi"

print(Greeter.hello())
```

A. Methods cannot return strings unless they are static
B. The class must define __init__ before any method can be called
C. hello is an instance method called on the class, so no object is passed for self and Python raises TypeError
D. Greeter must be imported before it can be used

890. Why does this Java code fail to compile? Analysis

```
class P {
    static void hi() { }
}
class C extends P {
    @Override
    static void hi() { }
}
```

A. Static methods are hidden, not overridden, so @Override on a static method is an error
B. A subclass may never declare a method with the same name as a static method in its superclass
C. @Override may only be used on methods declared in interfaces
D. Static methods must be public to be inherited

891. What happens when the last line of this Python code runs, and why? Analysis

```
class Circle:
    def __init__(self, r):
        self.r = r
    @property
    def area(self):
        return 3.14 * self.r ** 2

c = Circle(1)
c.area = 10
```

A. AttributeError, because area is a read-only property with no setter
B. c.area becomes 10, replacing the property for this object
C. r is updated so that the area becomes 10
D. TypeError, because a float cannot be assigned an int

892. A teammate says that naming a Python attribute self._balance makes it private, so code outside the class cannot read or change it. Is that right? Analysis

A. Yes. Python raises AttributeError for any access to an underscore-prefixed attribute from outside
B. Yes, but only when the class also inherits from object
C. No. A single underscore is only a convention; outside code can still access it. A double underscore adds name mangling but is still not true privacy
D. No. A single underscore makes the attribute read-only from outside the class

Memory Management Concepts MCQs for the NSCT

893. In C++, where is a non-static local variable such as `int count = 0;` declared inside a function normally stored? Understanding

A. On the call stack, in that function's stack frame
B. On the heap
C. In the text (code) segment
D. In the static/global data segment

894. In C++, which statement correctly releases memory obtained with `int* a = new int[50];`? Understanding

A. delete a;
B. free(a);
C. a = nullptr;
D. delete[] a;

895. In Java, where is the object created by `new StringBuilder()` stored? Understanding

A. On the stack of the method that created it
B. In the CPU registers until the method ends
C. In the class file on disk
D. On the heap; the variable holding it stores only a reference

896. What is a memory leak? Understanding

A. Reading memory beyond the end of an array
B. Heap memory that the program no longer needs or can reach but never releases
C. A variable that goes out of scope when a function returns
D. Memory that the operating system takes back when the program exits

897. In C++, what is a dangling pointer? Understanding

A. A pointer that has been set to nullptr
B. A pointer declared but never used
C. A pointer that still holds the address of memory that has been freed or has gone out of scope
D. A pointer to a function rather than to data

898. In Java, which error is thrown when a method calls itself recursively with no base case until the thread runs out of call-stack space? Understanding

A. OutOfMemoryError
B. NullPointerException
C. ArrayIndexOutOfBoundsException
D. StackOverflowError

899. How does CPython (the standard Python interpreter) mainly decide when to free an object? Understanding

A. The programmer must call del on every object to free it
B. Only a mark-and-sweep collector that runs every few seconds
C. Objects are freed when the function that created them returns, like stack variables
D. It keeps a reference count on every object and frees it when the count reaches zero, with a cycle-detecting collector for reference cycles

900. What does RAII (Resource Acquisition Is Initialisation) mean in C++? Understanding

A. A resource is acquired in an object's constructor and released in its destructor, so its lifetime is tied to the object's scope
B. All resources must be initialised to zero before use
C. Resources are allocated at compile time instead of at run time
D. The garbage collector releases resources when memory runs low

901. Which C++ smart pointer represents sole ownership and can be moved but not copied? Understanding

A. std::shared_ptr
B. std::unique_ptr
C. std::weak_ptr
D. A raw pointer declared const

902. Which C++ smart pointer refers to an object managed by std::shared_ptr without increasing its strong reference count? Understanding

A. std::unique_ptr
B. std::auto_ptr
C. A second std::shared_ptr
D. std::weak_ptr

903. In Java, when does an object become eligible for garbage collection? Understanding

A. As soon as one variable that referred to it is set to null
B. When the method that created it returns
C. When it can no longer be reached through any chain of references from a GC root such as a live thread's stack or a static field
D. Only after its finalize() method has been called by the programmer

904. In C++, which statement about references and pointers is correct? Understanding

A. A reference can be null, but a pointer cannot
B. A reference must be initialised when declared and cannot later be re-bound to another object; a pointer can be null and can be reassigned
C. Both references and pointers can be re-bound to different objects at any time
D. A reference stores a copy of the object, while a pointer stores its address

905. What is the main idea behind a generational garbage collector, as used in the JVM? Understanding

A. Most objects die young, so a small young generation is collected often and long-lived objects are promoted to an older generation collected less often
B. Objects are collected in the order they were created, oldest first
C. Each thread has its own private heap that is never shared
D. Objects are freed as soon as their reference count reaches zero

906. What does calling `System.gc()` in Java do? Understanding

A. It suggests that the JVM run garbage collection, but the JVM may ignore or delay it
B. It immediately frees every object in the heap
C. It forces collection of only the objects created by the calling method
D. It disables the garbage collector until the program ends

907. In a mark-and-sweep garbage collector, what happens in each phase? Understanding

A. Mark: flag every unreachable object. Sweep: move reachable objects to a new region
B. Mark: count references to each object. Sweep: free objects whose count is zero
C. Mark: starting from the roots, flag every reachable object. Sweep: reclaim every object that was not flagged
D. Mark: record each object's allocation time. Sweep: free the oldest objects

908. A heap has plenty of free memory in total, but a request for one large block fails because the free memory is split into many small, non-adjacent pieces. What is this called, and which GC technique addresses it? Understanding

A. Internal fragmentation; reference counting frees objects sooner
B. A memory leak; calling the destructor manually fixes it
C. External fragmentation; a compacting or copying collector moves live objects together
D. Stack overflow; increasing the recursion limit fixes it

909. What does this Python code print? Application

```
a = [1, 2, 3]
b = a
b.append(4)
print(len(a))
```

A. 3
B. 7
C. An error, because b is a copy
D. 4

910. What does this Python code print? Application

```
a = [1, 2]
b = a[:]
print(a == b, a is b)
```

A. True True
B. True False
C. False False
D. False True

911. What does this C++ program print? Application

```
#include <iostream>
int main() {
    int x = 10;
    int* p = &x;
    *p = 20;
    std::cout << x;
}
```

A. 20
B. 10
C. The address of x
D. 0

912. What does this C++ program print? Application

```
#include <iostream>
int main() {
    int a = 5;
    int& r = a;
    r = r + 3;
    int b = r;
    b = 100;
    std::cout << a << " " << r;
}
```

A. 5 8
B. 8 8
C. 8 100
D. 100 100

913. What does this C++ program print? Application

```
#include <iostream>
int main() {
    int arr[4] = {10, 20, 30, 40};
    int* p = arr;
    p += 2;
    std::cout << *p << " " << *(p - 1);
}
```

A. 30 20
B. 30 40
C. 12 11
D. 20 10

914. This C++ program calls f three times. How many 100-int arrays are leaked when main ends? Application

```
void f(bool early) {
    int* p = new int[100];
    if (early) return;
    delete[] p;
}
int main() {
    f(true);
    f(false);
    f(true);
}
```

A. 1
B. 2
C. 3
D. 0

915. What does this C++ program print? Application

```
#include <iostream>
#include <memory>
int main() {
    auto p = std::make_unique<int>(7);
    auto q = std::move(p);
    std::cout << (p == nullptr) << " " << *q;
}
```

A. 0 7
B. 1 0
C. 1 7
D. It does not compile, because unique_ptr cannot be moved

916. What does this C++ program print? Application

```
#include <iostream>
#include <memory>
int main() {
    auto a = std::make_shared<int>(1);
    auto b = a;
    {
        auto c = a;
        std::cout << a.use_count() << " ";
    }
    std::cout << a.use_count();
}
```

A. 3 3
B. 2 2
C. 3 2
D. 1 1

917. In CPython, what does this code print? Application

```
import sys
x = []
before = sys.getrefcount(x)
y = x
z = [x, x]
print(sys.getrefcount(x) - before)
```

A. 2
B. 3
C. 1
D. 4

918. What does this Python code print? Application

```
a = [1, 2]
b = a
del a
print(b)
```

A. An error, because the list was deleted
B. None
C. [1, 2]
D. []

919. In CPython, in what order are the lines printed? Application

```
class R:
    def __init__(self, n):
        self.n = n
    def __del__(self):
        print("free", self.n)

r1 = R(1)
r2 = r1
del r1
print("after del")
r2 = None
print("end")
```

A. after del, free 1, end
B. free 1, after del, end
C. after del, end, free 1
D. free 1, end, after del

920. When f(3) is called, what is the largest number of f frames on the Python call stack at the same time? Application

```
def f(n):
    if n == 0:
        return 0
    return n + f(n - 1)

print(f(3))
```

A. 4
B. 3
C. 1
D. 5

921. After line 4 of this Java code runs, how many of the two Box objects are eligible for garbage collection? Application

```
Box a = new Box();   // object 1
Box b = new Box();   // object 2
Box c = a;           // line 3
a = null; b = c;     // line 4
```

A. 0
B. 2
C. 1
D. It cannot be known until System.gc() is called

922. In this Java code, how many Node objects are eligible for garbage collection after the last line? Application

```
class Node { Node next; }

Node x = new Node();
Node y = new Node();
x.next = y;
y.next = x;
x = null;
y = null;
```

A. 0
B. 1
C. 0 now, but 2 once each node's next field is set to null
D. 2

923. A C++ function declares `int x = 42;` and then returns `&x`. The caller stores the result in `int* p` and later prints `*p`. What is wrong? Analysis

A. Nothing; x keeps its value because its address was returned
B. x lives in the function's stack frame, which is released on return, so p dangles and reading *p is undefined behaviour
C. It leaks memory, because x was never deleted
D. It does not compile, because a function cannot return a pointer

924. A C++ class allocates an array with new[] in its constructor and frees it with delete[] in its destructor, but declares no copy constructor or copy assignment operator. What happens with `Buffer b2 = b1;` when both objects go out of scope? Analysis

A. The default copy copies only the pointer, so both destructors delete[] the same array: a double free and undefined behaviour
B. Each object gets its own deep copy of the array, so it is safe
C. The code does not compile until a copy constructor is written
D. Only b1's destructor runs, so the array is freed exactly once

925. This C++ program prints nothing, although each Node has a destructor that prints. Why are the destructors never called? Analysis

```
#include <iostream>
#include <memory>
struct Node {
    std::shared_ptr<Node> next;
    ~Node() { std::cout << "freed "; }
};
int main() {
    auto a = std::make_shared<Node>();
    auto b = std::make_shared<Node>();
    a->next = b;
    b->next = a;
}
```

A. Destructors never run for objects created with make_shared
B. a and b own each other through shared_ptr, so each count stays at 1 after main's variables are destroyed and neither object is freed
C. std::cout is closed before destructors run at the end of main
D. shared_ptr frees objects only when the program calls delete on them

926. This Java server calls handle() once per request and eventually throws OutOfMemoryError. Why can't the garbage collector reclaim the arrays? Analysis

```
import java.util.*;

class Cache {
    static List<byte[]> items = new ArrayList<>();
    static void handle() {
        items.add(new byte[1024 * 1024]);
    }
}
```

A. Arrays of bytes are never garbage collected in Java
B. The collector only runs when System.gc() is called
C. Each array is stored on the stack, which the collector does not manage
D. Every array is still reachable through the static list, so none of them is garbage

927. A Java program crashes with StackOverflowError while its heap usage is low. Which cause is most likely? Analysis

A. A static list that keeps growing
B. Too many objects created inside a loop
C. A recursive method with a missing or unreachable base case
D. Calling System.gc() too often

928. If load() throws, what goes wrong in process(), and what is the best fix in modern C++? Analysis

```
#include <stdexcept>
void load(int* d, int n) {
    if (n > 1000) throw std::runtime_error("too big");
    for (int i = 0; i < n; ++i) d[i] = i;
}
void process(int n) {
    int* data = new int[n];
    load(data, n);
    delete[] data;
}
```

A. Nothing goes wrong, because the array is freed automatically when the exception propagates
B. The array is freed twice; remove the delete[] line
C. delete[] is skipped and the array leaks; replace the raw array with std::vector<int> (RAII) so it is freed on every exit path
D. The program cannot compile; load() must be declared noexcept

929. In CPython with automatic garbage collection disabled, what does this code print, in order? Analysis

```
import gc
gc.disable()

class Node:
    def __del__(self):
        print("freed")

n = Node()
n.self_ref = n
del n
print("after del")
gc.collect()
print("end")
```

A. after del, freed, end
B. freed, after del, end
C. after del, end
D. after del, end, freed

930. After these four lines, the list of 1,000 zeros is still in memory. In CPython, what is the minimum that must also happen for it to be freed immediately? Analysis

```
a = [0] * 1000
b = {"k": a}
c = [a]
a = None
```

A. Only `del b["k"]`
B. Only `c.clear()`
C. Nothing; it was already freed when a was set to None
D. Both `del b["k"]` and `c.clear()`

931. Why does the second call print [1, 2] rather than [2]? Analysis

```
def add(item, bucket=[]):
    bucket.append(item)
    return bucket

add(1)
print(add(2))
```

A. Python caches the return value of the first call
B. The default list is created once, when def runs, and the same object is reused and mutated by every call that omits bucket
C. append copies the list, and the copy is kept in global memory
D. The list is stored on the stack and is not cleared between calls

932. A C++ program allocates `int* p = new int[50];` and later frees it with `delete p;` (without brackets). What is the correct assessment? Analysis

A. It is fine, because delete and delete[] are interchangeable for built-in types
B. It frees only the first element and leaks the other 49, with no other effect
C. It does not compile, because the compiler checks that new[] is matched with delete[]
D. It is undefined behaviour; memory from new[] must be released with delete[]

933. In C and C++, where are global variables and static local variables stored? Understanding

A. On the stack, in main's frame
B. On the heap, allocated when first used
C. In CPU registers, because they are used often
D. In the static data area (the data and BSS segments), which lasts for the whole run of the program

934. In C++, what does `new` do that C's `malloc` does not? Understanding

A. It allocates memory on the stack instead of the heap
B. It frees the memory automatically when the pointer goes out of scope
C. It always zero-fills the allocated memory
D. It runs the object's constructor and returns a correctly typed pointer

935. In C, how does `calloc(n, size)` differ from `malloc(n * size)`? Understanding

A. calloc allocates on the stack; malloc allocates on the heap
B. calloc memory is freed automatically; malloc memory needs free()
C. calloc sets every byte of the block to zero; malloc leaves the contents indeterminate
D. calloc can only allocate arrays of int

936. In C++, what is a double free? Understanding

A. Allocating two blocks with one call to new
B. Freeing a block and then allocating it again
C. Freeing two different pointers in the same statement
D. Releasing the same heap block twice, for example calling delete twice on one pointer, which is undefined behaviour

937. Which C++11 keyword gives a null pointer value with its own type (std::nullptr_t), replacing NULL and 0? Understanding

A. nullptr
B. null
C. void
D. None

938. In a typical process memory layout, which region holds the program's compiled machine instructions? Understanding

A. The heap
B. The stack
C. The text (code) segment
D. The BSS segment

939. In C, what does `realloc(p, bigger)` do when it cannot enlarge the block in place? Understanding

A. Allocates a new block, copies the old contents, frees the old block and returns the new address
B. Returns NULL and frees the old block
C. Extends the block into the stack
D. Returns the same address and silently truncates future writes

940. In C++, what is a wild pointer? Understanding

A. A pointer to memory that has already been freed
B. A pointer that was never initialised, so it holds an indeterminate address
C. A pointer set to nullptr
D. A pointer that points to a global variable

941. What does the C++ Rule of Three state? Understanding

A. Every class must have exactly three constructors
B. A class may own at most three heap allocations
C. If a class needs a user-defined destructor, copy constructor or copy assignment operator, it almost certainly needs all three
D. Every new must be matched by three deletes in different scopes

942. In Java, what is the string pool? Understanding

A. A memory area where string literals with the same content share one interned String object
B. A cache that stores every String created with new
C. The part of the stack where String variables live
D. A buffer that StringBuilder uses to resize itself

943. Since Java 8, where does the HotSpot JVM store class metadata that older versions kept in PermGen? Understanding

A. The young generation of the heap
B. Metaspace, which is allocated from native memory
C. Each thread's stack
D. The string pool

944. Which pair of HotSpot JVM options sets the maximum heap size and the per-thread stack size? Understanding

A. -Xmx and -Xss
B. -Xms and -Xmx
C. -Xss and -Xms
D. -Xmn and -Xmx

945. What is a stop-the-world pause in garbage collection? Understanding

A. A period when all application threads are halted so the collector can safely run part of its work
B. The moment the JVM shuts down after an OutOfMemoryError
C. A pause in allocation while the operating system adds swap space
D. The time a single thread waits for a lock held by the collector

946. How does a copying (semispace) garbage collector work? Understanding

A. It counts references to each object and copies an object when its count reaches zero
B. It copies the live objects from the in-use half of the heap to the empty half and then treats the old half as free, which also compacts memory
C. It copies every object to disk before freeing the heap
D. It marks dead objects and leaves live objects where they are

947. What is internal fragmentation? Understanding

A. Free memory split into many small gaps between allocated blocks
B. Memory that the program can no longer reach
C. Wasted space inside an allocated block, because the allocator gave out more bytes than were requested
D. A stack frame that is larger than the function needs

948. In Python, what does a weak reference created with the weakref module do? Understanding

A. It makes a copy of the object that is freed sooner
B. It keeps the object alive only on the stack
C. It refers to an object without increasing its reference count, so the object can still be freed
D. It prevents the garbage collector from moving the object

949. Why is `std::make_shared<T>(...)` usually preferred over `std::shared_ptr<T>(new T(...))` in C++? Understanding

A. It makes the object live on the stack instead of the heap
B. It produces a shared_ptr that can never form reference cycles
C. It lets the object be copied without calling the copy constructor
D. It allocates the object and the reference-count control block in one allocation and avoids a leak if an exception is thrown in between

950. What does Java's try-with-resources statement guarantee? Understanding

A. Every resource declared in the try header is closed automatically when the block ends, even if an exception is thrown
B. The objects created in the block are garbage collected immediately when it ends
C. No exception can escape from the block
D. The resources are allocated on the stack instead of the heap

951. In Java, how does a SoftReference differ from a WeakReference? Understanding

A. A softly reachable object is normally kept until the JVM is short of memory; a weakly reachable object can be cleared at the next collection
B. A SoftReference keeps the object forever; a WeakReference frees it immediately
C. A WeakReference keeps the object until memory is low; a SoftReference frees it at the next collection
D. They are identical; the names are kept only for compatibility

952. Why should modern Java code not rely on overriding Object.finalize() to release resources? Understanding

A. finalize runs immediately when a reference is set to null, which is too early
B. finalize can only free stack memory
C. finalize is called once for every reference to the object
D. finalize is deprecated, and the JVM does not guarantee when, or even whether, it will run

953. Which set of items is typically stored in a function's stack frame? Understanding

A. Every object the function creates with new
B. Its parameters, its local variables and the return address
C. The program's global variables
D. The function's compiled machine code

954. In C++, what does a typical move constructor for a class that owns a raw heap pointer do? Understanding

A. It allocates a new block and copies every element from the source
B. It deletes the source's memory and allocates a fresh block
C. It takes the source's pointer and sets the source's pointer to nullptr, so only the new object frees the memory
D. It copies the pointer and leaves the source unchanged, so both objects own the block

955. In CPython, what is the default recursion limit returned by `sys.getrecursionlimit()`, and what is raised when it is exceeded? Understanding

A. 1000, and StackOverflowError is raised
B. There is no limit, so the process crashes
C. 1000, and RecursionError is raised
D. 100, and MemoryError is raised

956. On a typical compiler where int is 4 bytes with 4-byte alignment, why is sizeof(S) usually 8 for `struct S { char c; int i; };` rather than 5? Understanding

A. The compiler inserts 3 padding bytes after c so that i starts on a 4-byte boundary
B. char always takes 4 bytes inside a struct
C. Every struct has a 3-byte header that stores its size
D. The compiler stores i twice for faster access

957. What does this Python code print? Application

```
import copy

a = [[1, 2], [3]]
b = copy.copy(a)
b[0].append(9)
b.append([4])
print(a)
```

A. [[1, 2], [3]]
B. [[1, 2, 9], [3], [4]]
C. [[1, 2], [3], [4]]
D. [[1, 2, 9], [3]]

958. What does this Python code print? Application

```
import copy

a = {"k": [1]}
b = copy.deepcopy(a)
b["k"].append(2)
print(a["k"], b["k"])
```

A. [1, 2] [1, 2]
B. [1] [1]
C. [1] [1, 2]
D. [1, 2] [1]

959. What does this Python code print? Application

```
def f(lst):
    lst.append(1)
    lst = [99]
    lst.append(2)

x = [0]
f(x)
print(x)
```

A. [0]
B. [99, 2]
C. [0, 1]
D. [0, 1, 2]

960. What does this Java code print? Application

```
static void swap(StringBuilder a, StringBuilder b) {
    StringBuilder t = a; a = b; b = t;
    a.append("!");
}

// in main:
StringBuilder x = new StringBuilder("x");
StringBuilder y = new StringBuilder("y");
swap(x, y);
System.out.println(x + " " + y);
```

A. y! x
B. x! y
C. y x!
D. x y!

961. What does this Java code print? Application

```
String a = "nsct";
String b = "nsct";
String c = new String("nsct");
System.out.println((a == b) + " " + (a == c) + " " + a.equals(c));
```

A. true true true
B. true false true
C. false false true
D. true false false

962. On a standard JVM with default settings, what does this Java code print? Application

```
Integer a = 100, b = 100;
Integer c = 1000, d = 1000;
System.out.println((a == b) + " " + (c == d));
```

A. true true
B. true false
C. false false
D. false true

963. What does this C++ program print? Application

```
#include <iostream>
#include <memory>
struct R {
    int id;
    R(int i) : id(i) { std::cout << "+" << id; }
    ~R() { std::cout << "-" << id; }
};
int main() {
    auto a = std::make_unique<R>(1);
    {
        auto b = std::make_unique<R>(2);
    }
    auto c = std::make_unique<R>(3);
}
```

A. +1+2-2+3-3-1
B. +1+2+3-1-2-3
C. +1+2-2+3-1-3
D. +1+2+3-3-2-1

964. What does this C++ program print? (bool values print as 1 and 0 by default.) Application

```
#include <iostream>
#include <memory>
int main() {
    auto s = std::make_shared<int>(5);
    std::weak_ptr<int> w = s;
    std::cout << w.use_count() << w.expired() << " ";
    s.reset();
    std::cout << w.use_count() << w.expired();
}
```

A. 20 11
B. 10 00
C. 11 01
D. 10 01

965. What does this C++ program print? Application

```
#include <iostream>
void grow(int* p) { p = new int(50); }
void set(int* p)  { *p = 50; }
int main() {
    int* q = new int(1);
    grow(q);
    std::cout << *q << " ";
    set(q);
    std::cout << *q;
    delete q;
}
```

A. 1 50
B. 50 50
C. 1 1
D. 50 1

966. What does this C++ program print? Application

```
#include <iostream>
int next() {
    static int s = 0;
    int t = 0;
    ++s; ++t;
    return s * 10 + t;
}
int main() {
    next(); next();
    std::cout << next();
}
```

A. 11
B. 33
C. 31
D. 13

967. In CPython, what does this code print? Application

```
import weakref

class Big:
    pass

b = Big()
r = weakref.ref(b)
print(r() is b, end=" ")
del b
print(r())
```

A. True <__main__.Big object>
B. True None
C. False None
D. TypeError, because weak references cannot be called

968. In CPython, in what order are the lines printed? Application

```
class R:
    def __init__(self, n): self.n = n
    def __del__(self): print("free", self.n)

def make():
    r = R(1)
    tmp = R(2)
    return lambda: r.n

f = make()
print("made")
f = None
print("end")
```

A. free 2, made, free 1, end
B. made, free 2, free 1, end
C. free 2, free 1, made, end
D. made, end, free 1, free 2

969. What does this Python code print? Application

```
a = [1, 2]
b = a
a += [3]
c = a
a = a + [4]
print(b, c is b, a is b)
```

A. [1, 2, 3] True False
B. [1, 2] False False
C. [1, 2, 3, 4] True True
D. [1, 2, 3] False False

970. After work() returns, how many of the three Box objects are eligible for garbage collection? Application

```
class Demo {
    static Box keep;
    static void work() {
        Box a = new Box();   // object 1
        Box b = new Box();   // object 2
        Box c = new Box();   // object 3
        keep = b;
        a = c;
    }
}
```

A. 1
B. 2
C. 3
D. 0

971. What does this JavaScript code print, and which collection keeps the object alive? Application

```
let cache = new Map();
let wcache = new WeakMap();
let key = { id: 1 };
cache.set(key, "data");
wcache.set(key, "data");
key = null;
console.log(cache.size);
```

A. 0; setting key to null removes it from both collections
B. 1; both the Map and the WeakMap keep the object alive forever
C. 0; the garbage collector empties the Map immediately
D. 1; the Map holds a strong reference to the key object, so it stays reachable, while a WeakMap entry alone would not keep it alive

972. What does this C++ program print? Application

```
#include <iostream>
class Buf {
public:
    int* d;
    Buf(int v) : d(new int(v)) {}
    Buf(const Buf& o) : d(new int(*o.d)) {}
    ~Buf() { delete d; }
};
int main() {
    Buf a(1);
    Buf b = a;
    *b.d = 9;
    std::cout << *a.d << *b.d;
}
```

A. 99
B. 11
C. 91
D. 19

973. What does this C++ program print? Application

```
#include <iostream>
#include <memory>
void show(std::shared_ptr<int> p)        { std::cout << p.use_count() << " "; }
void peek(const std::shared_ptr<int>& p) { std::cout << p.use_count() << " "; }
int main() {
    auto a = std::make_shared<int>(0);
    show(a);
    peek(a);
    std::cout << a.use_count();
}
```

A. 2 1 1
B. 1 1 1
C. 2 2 1
D. 2 2 2

974. What does this Python code print? Application

```
a = [10, 20, 30]
b = a
del a[0]
print(b)
```

A. [10, 20, 30]
B. [20, 30]
C. NameError: name 'b' is not defined
D. [10, 30]

975. What does this C++ program print? Application

```
#include <iostream>
struct T {
    T()  { std::cout << "c"; }
    ~T() { std::cout << "d"; }
};
int main() {
    T* arr = new T[3];
    delete[] arr;
}
```

A. cdcdcd
B. cccd
C. cccddd
D. ccc

976. What does this Python code print? Application

```
grid = [[0] * 2] * 3
grid[0][0] = 1
print(grid)
```

A. [[1, 0], [0, 0], [0, 0]]
B. [[1, 1], [0, 0], [0, 0]]
C. [[1, 0, 0, 0, 0, 0]]
D. [[1, 0], [1, 0], [1, 0]]

977. A C++ loop runs 500 times and each pass executes `double* p = new double[1000];` with no matching delete[]. With 8-byte doubles and ignoring allocator overhead, how much memory has leaked when the loop ends? Application

A. 8,000 bytes
B. 4,000,000 bytes (about 4 MB)
C. 500,000 bytes
D. 8,000,000 bytes

978. In C++, why do many programmers write `p = nullptr;` immediately after `delete p;`? Analysis

A. Because delete does not free the memory until p is set to nullptr
B. So p no longer dangles: a later delete p is a harmless no-op and an accidental dereference is easier to detect
C. Because setting p to nullptr returns the memory to the operating system
D. Because the compiler requires it after every delete

979. Memory keeps rising while this Python program creates and replaces Screen objects in a loop. Why are old Screens never freed? Analysis

```
class Bus:
    listeners = []
    @classmethod
    def subscribe(cls, fn):
        cls.listeners.append(fn)

class Screen:
    def __init__(self):
        self.pixels = bytearray(10_000_000)
        Bus.subscribe(self.on_event)
    def on_event(self, e):
        pass

for _ in range(100):
    s = Screen()
```

A. bytearray memory is never freed in Python
B. The loop variable s keeps all 100 Screens alive
C. Class methods leak memory whenever they are called
D. Each bound method self.on_event stored in the class-level list holds a reference to its Screen, so every Screen stays reachable

980. This Java code prints 1, and the map keeps an entry that can no longer be found or removed. Why? Analysis

```
class Key {
    int v;
    Key(int v) { this.v = v; }
    public int hashCode() { return v; }
    public boolean equals(Object o) {
        return o instanceof Key && ((Key) o).v == v;
    }
}

// in main:
Map<Key, String> m = new HashMap<>();
Key k = new Key(1);
m.put(k, "a");
k.v = 2;
m.remove(k);
System.out.println(m.size());
```

A. HashMap.remove only works with String keys
B. remove compares keys with ==, and k is a different object
C. The key's hash code changed after insertion, so remove looks in the wrong bucket; the stale entry stays reachable from the map and leaks
D. The garbage collector already removed the key, so remove has nothing to do

981. A Python function builds a 1 GB list `data`, computes `total = sum(data)`, then runs a long loop that never uses data again. Peak memory stays high during the loop. What is the simplest fix in CPython? Analysis

A. Run `del data` (or move the build-and-sum into a helper function) before the loop, so the list's reference count drops to zero and it is freed at once
B. Call gc.collect() inside the loop on every pass
C. Use `data = data[:]` before the loop
D. Declare data as global so it is freed sooner

982. A Java method recursively walks a singly linked list of 1,000,000 nodes and throws StackOverflowError, even though the base case is correct. What is the best fix? Analysis

    A. Increase the heap with -Xmx, because the nodes do not fit in memory
    B. Rewrite the walk as a loop, because each node adds a stack frame and the default thread stack cannot hold a million frames
    C. Call System.gc() before the walk to free stack space
    D. Mark the method static so it uses no stack frames

983. A C++ function needs a 4-byte counter used only during the call, and a 200 MB buffer whose size is known only at run time. Where should each live? Analysis

    A. Both on the stack, because stack memory is faster
    B. The counter as a local on the stack; the buffer on the heap, for example in a std::vector
    C. Both on the heap with raw new and delete
    D. The counter on the heap and the buffer on the stack

984. Which C++ factory signature best expresses that the caller becomes the sole owner of the new Widget and makes leaks hard? Analysis

    A. Widget* makeWidget();
    B. std::unique_ptr<Widget> makeWidget();
    C. Widget& makeWidget();
    D. std::weak_ptr<Widget> makeWidget();

985. A Task from start() is placed in a long-lived thread-pool queue after its Activity is closed. Why is the 50 MB array never collected? Analysis

```
class Activity {
    byte[] screen = new byte[50_000_000];
    class Task implements Runnable {
        public void run() { /* long-running work */ }
    }
    Runnable start() { return new Task(); }
}
```

    A. byte arrays larger than 10 MB are allocated outside the garbage-collected heap
    B. Runnable objects are never garbage collected
    C. A non-static inner class object holds a hidden reference to its enclosing Activity, so the queued Task keeps the Activity and its array reachable
    D. The screen field is not private, so the collector cannot see it

986. Which C++ fragment causes a memory leak rather than a dangling pointer? Analysis

    A. int* p = new int(5); int* q = p; delete p;
    B. int* f() { int x = 5; return &x; }
    C. int* p = new int(5); delete p; delete p;
    D. int* p = new int(5); p = new int(6); delete p;

987. A developer adds `list = null;` as the last line of every Java method, believing it frees memory sooner. What is the correct assessment? Analysis

    A. It is required, otherwise local objects are never collected
    B. It frees the list immediately, without waiting for the garbage collector
    C. It does almost nothing: local variables stop being roots once the method returns, so nulling them is only worth it for long-lived fields or long-running methods
    D. It causes a NullPointerException when the method returns

988. A system must close each file handle the instant the last reference to its wrapper object disappears. Which memory-management approach gives that guarantee, and what is its main cost? Analysis

    A. Tracing garbage collection; it frees immediately but cannot handle cycles
    B. Mark-and-sweep; it frees immediately but needs twice the memory
    C. Generational collection; it frees immediately but only works for short-lived objects
    D. Reference counting; it frees deterministically but must update a count on every reference change and needs extra help for cycles

989. An allocator rounds every request up to a multiple of 32 bytes. A program requests 1,000 blocks of 40 bytes each. How much memory is lost to internal fragmentation? Analysis

    A. 8,000 bytes
    B. 24,000 bytes
    C. 32,000 bytes
    D. 64,000 bytes

990. A reviewer says the commented line must not be added after this C++ code. Why? Analysis

```
std::vector<int> v = {1, 2, 3};
int* p = &v[0];
v.push_back(4);
// next: std::cout << *p;
```

    A. &v[0] is not allowed on a vector
    B. push_back always invalidates the pointer and sets it to nullptr
    C. Vectors store their elements on the stack, which push_back clears
    D. push_back may reallocate the vector's storage when size exceeds capacity, freeing the old array, so p could be dangling

991. A Java program reads a whole 4 GB log file into one byte[] and fails with OutOfMemoryError: Java heap space when run with -Xmx2g. What is the best fix? Analysis

    A. Process the file as a stream in fixed-size chunks or line by line, so only a small buffer is in memory at once
    B. Call System.gc() before reading the file
    C. Increase the thread stack size with -Xss4g
    D. Store the byte[] in a static field so it is not collected

992. In CPython, why is "closed" not printed before "done" in this code? Analysis

```
class Conn:
    def __del__(self):
        print("closed")

pool = []
c = Conn()
pool.append(c)
del c
print("done")
```

    A. __del__ runs only when gc.collect() is called
    B. del c raises an error that is silently ignored
    C. The list pool still refers to the Conn object, so del c removes only one name and the object stays alive
    D. Python never calls __del__ for objects stored in lists

Exception & Error Handling MCQs for the NSCT

993. In Java, which block runs whether or not the try block throws an exception (unless the JVM exits, for example through System.exit)? Understanding
    A. catch
    B. throws
    C. else
    D. finally

994. Which Python keyword is used to signal an exception explicitly? Understanding
    A. throw
    B. throws
    C. raise
    D. except

995. In Java, which of these is a checked exception? Understanding
    A. NullPointerException
    B. IOException
    C. ArithmeticException
    D. ArrayIndexOutOfBoundsException

996. In Java, what is the difference between throw and throws? Understanding
    A. throw raises an exception object at a point in the code; throws in a method signature declares exceptions the method may pass to its caller
    B. throw declares exceptions in a signature; throws raises an exception object
    C. They are interchangeable spellings of the same keyword
    D. throw is used for checked exceptions and throws for unchecked exceptions

997. In Python's try/except/else/finally, when does the else block run? Understanding
    A. Only when an exception was raised and caught
    B. Always, just before finally
    C. Only when no except clause matched the exception
    D. Only when the try block finished without raising an exception

998. In C++, which handler catches an exception of any type? Understanding
    A. catch (...)
    B. catch (std::exception& e)
    C. catch (void)
    D. catch (Object e)

999. In Java, which class is the direct superclass of both Exception and Error? Understanding
    A. RuntimeException
    B. Object
    C. Throwable
    D. BaseException

1000. What does it mean for an exception to be checked in Java? Understanding
    A. The JVM checks for it before the program starts
    B. It is caught automatically by the JVM and logged
    C. The compiler requires every method that can throw it either to catch it or to declare it with throws
    D. It can only be thrown by code in the java.lang package

1001. In C++, why should you usually catch exceptions by reference, as in `catch (const std::exception& e)`? Understanding

A. Catching by value is a compile error in C++
B. References make the exception object be freed earlier
C. It lets the handler catch exceptions of any type, including int
D. It avoids copying the exception and prevents slicing, so overridden members such as what() behave polymorphically

1002. Which interface must a class implement to be used in a Java try-with-resources statement? Understanding

A. AutoCloseable (Closeable extends it)
B. Serializable
C. Runnable
D. Iterable

1003. In Python, which class should a custom application exception normally inherit from? Understanding

A. BaseException
B. Exception
C. object
D. Error

1004. When an exception is not caught in the function where it is thrown, what happens next? Understanding

A. The program always terminates at once
B. The exception is ignored and the next statement runs
C. The function restarts from its first line
D. The function's frame is exited and the exception propagates to its caller, continuing up the call stack until a matching handler is found

1005. In Java, which statement about Error is correct? Understanding

A. Error is a checked exception that must be declared with throws
B. Error represents serious problems such as OutOfMemoryError that applications normally should not try to catch, and it is unchecked
C. Error is a subclass of RuntimeException
D. Error can only be thrown by user code, never by the JVM

1006. In C++, what happens when a thrown exception is never caught by any handler? Understanding

A. std::terminate() is called, which by default aborts the program
B. The exception is silently ignored
C. main returns 0
D. The compiler reports an error before the program runs

1007. In C++11 and later, what happens if an exception escapes a function declared noexcept? Understanding

A. The exception propagates normally to the caller
B. The exception is converted to std::bad_exception and rethrown
C. The program fails to compile
D. std::terminate() is called

1008. In Java, which multi-catch clause causes a compile error? Understanding

A. catch (FileNotFoundException | IOException e)
B. catch (IOException | SQLException e)
C. catch (IllegalArgumentException | IllegalStateException e)
D. catch (NumberFormatException | ArithmeticException e)

1009. What does this Python code print (one item per line)? Application

```
try:
    print("A")
    x = 1 / 0
    print("B")
except ZeroDivisionError:
    print("C")
finally:
    print("D")
```

A. A B C D
B. A C D
C. A D
D. A C

1010. What does this Python code print? Application

```
try:
    n = int("abc")
except Exception as e:
    print(type(e).__name__)
```

A. TypeError
B. SyntaxError
C. NameError
D. ValueError

1011. What does this JavaScript code print? Application

```
try {
  JSON.parse("{bad");
} catch (e) {
  console.log(e.name);
}
```

A. TypeError
B. Error
C. undefined
D. SyntaxError

1012. What does this Python code print (one item per line)? Application

```
def f():
    try:
        r = 10
    except Exception:
        print("E")
    else:
        print("L")
    finally:
        print("F")
    return r

print(f())
```

A. L F 10
B. F L 10
C. L 10 F
D. E F 10

1013. What does this Java code print? Application

```
static int f() {
    try {
        return 1;
    } finally {
        return 2;
    }
}
// in main:
System.out.println(f());
```

A. 1
B. 1 then 2
C. It does not compile
D. 2

1014. What does this Java code print? Application

```
static int f() {
    int x = 10;
    try {
        return x;
    } finally {
        x = 20;
    }
}
// in main:
System.out.println(f());
```

A. 20
B. 10
C. 30
D. It does not compile

1015. What does this Python code print? Application

```
try:
    [1, 2][5]
except LookupError:
    print("lookup")
except IndexError:
    print("index")
```

A. lookup
B. index
C. lookup then index
D. Nothing; the IndexError is uncaught

1016. What does this JavaScript code print (one item per line)? Application

```
function f() {
  try {
    throw new Error("x");
  } catch (e) {
    console.log("catch " + e.message);
    return "c";
  } finally {
    console.log("finally");
  }
}
console.log(f());
```

A. catch x, c, finally
B. catch x, finally, c
C. finally, catch x, c
D. catch x, c

1017. What does this Python code print (one item per line)? Application

```
def inner():
    raise KeyError("k")

def outer():
    try:
        inner()
    except ValueError:
        print("V")
    finally:
        print("F")

try:
    outer()
except KeyError:
    print("K")
```

A. V F K
B. K F
C. F K
D. F

1018. What does this Python code print? Application

```
class LowBalance(Exception):
    pass

def withdraw(bal, amt):
    if amt > bal:
        raise LowBalance(f"need {amt - bal} more")
    return bal - amt

try:
    withdraw(50, 80)
except LowBalance as e:
    print(type(e).__name__, e)
```

A. Exception need 30 more
B. LowBalance
C. LowBalance need 30 more
D. need 30 more

1019. What happens when this Java method is compiled? Application

```
import java.io.*;

public void read() {
    FileReader r = new FileReader("a.txt");
}
```

A. It compiles and throws FileNotFoundException at run time if the file is missing
B. Compile error: the checked FileNotFoundException must be caught or declared with throws
C. It compiles, because FileNotFoundException is unchecked
D. Compile error: FileReader must be closed in a finally block

1020. What does this C++ program print? Application

```
#include <iostream>
int main() {
    try {
        throw 5;
    } catch (double d) {
        std::cout << "double";
    } catch (int i) {
        std::cout << "int " << i;
    } catch (...) {
        std::cout << "other";
    }
}
```

A. int 5
B. double
C. other
D. double then int 5

1021. What does this Python code print? Application

```
count = 0
for i in range(3):
    try:
        if i == 1:
            continue
    finally:
        count += 1
print(count)
```

A. 3
B. 2
C. 1
D. 0

1022. What does this Python code print? Application

```
try:
    try:
        int("x")
    except ValueError as e:
        raise RuntimeError("bad input") from e
except RuntimeError as err:
    print(type(err.__cause__).__name__)
```

A. RuntimeError
B. None
C. ValueError
D. Exception

1023. What happens when this Python script runs? Analysis

```
import sys

try:
    sys.exit(1)
except Exception:
    print("caught")
print("still running")
```

A. caught, then still running
B. Nothing is printed and the program exits with status 1
C. Only still running
D. Only caught

1024. Why does this Java code fail to compile? Analysis

```
import java.io.IOException;

class Base {
    void load() throws IOException { }
}
class Child extends Base {
    @Override
    void load() throws Exception { }
}
```

A. An overriding method cannot declare a broader checked exception than the method it overrides
B. @Override cannot be used on methods that declare throws
C. Child.load() must declare exactly the same throws clause as Base.load()
D. Exception is unchecked, so it cannot appear in a throws clause

1025. parse_age("abc") and parse_age(None) both return 0. What is the main problem with this design? Analysis

```
def parse_age(s):
    try:
        return int(s)
    except Exception:
        return 0
```

A. It is slower than checking with an if statement first
B. The broad except hides different failures, including a TypeError caused by a caller's bug, behind a valid-looking age of 0
C. except Exception also catches SyntaxError in the caller's file
D. int() never raises, so the except clause is dead code

1026. What does this C++ program print? Analysis

```
#include <iostream>
#include <stdexcept>
#include <string>
struct Guard {
    std::string n;
    Guard(std::string s) : n(s) { std::cout << "+" << n << " "; }
    ~Guard() { std::cout << "-" << n << " "; }
};
void f() {
    Guard a("a");
    Guard b("b");
    throw std::runtime_error("x");
}
int main() {
    try { f(); }
    catch (const std::exception& e) { std::cout << "caught"; }
}
```

A. +a +b caught -a -b
B. +a +b caught
C. +a +b -b -a caught
D. +a +b -a -b caught

1027. What does this Java code print? Analysis

```
class R implements AutoCloseable {
    String n;
    R(String n) { this.n = n; }
    public void close() { System.out.print("close" + n + " "); }
}
// in main:
try (R a = new R("A"); R b = new R("B")) {
    System.out.print("body ");
    throw new RuntimeException();
} catch (RuntimeException e) {
    System.out.print("catch");
}
```

A. body catch closeB closeA
B. body closeA closeB catch
C. body closeB closeA catch
D. body catch

1028. What does this Python code print? Analysis

```
try:
    try:
        raise ValueError("first")
    finally:
        raise TypeError("second")
except Exception as e:
    print(type(e).__name__, type(e.__context__).__name__)
```

A. ValueError NoneType
B. TypeError NoneType
C. ValueError TypeError
D. TypeError ValueError

1029. A Java library method receives a negative quantity because the calling code has a bug. Which exception should it throw? Analysis

A. IOException, so the caller is forced to catch it
B. A custom checked exception that every caller must declare
C. IllegalArgumentException, an unchecked exception for invalid arguments
D. Error, because invalid input is unrecoverable

1030. This Python code checks that a file exists before opening it, yet it occasionally crashes with FileNotFoundError on a busy server. What is the best explanation and fix? Analysis

```
import os

if os.path.exists(path):
    with open(path) as f:
        data = f.read()
```

A. os.path.exists() is unreliable on Linux and should be called twice
B. open() needs mode "r+" to avoid the error
C. Another process can delete the file between the check and the open; use try/except FileNotFoundError around open() (EAFP)
D. The with statement raises FileNotFoundError when the file is closed

1031. What does this JavaScript code print (one item per line)? Analysis

```
function run() {
  try {
    throw new Error("A");
  } catch (e) {
    throw new Error("B");
  } finally {
    console.log("F");
  }
}
try {
  run();
} catch (e) {
  console.log(e.message);
}
```

A. B F
B. F B
C. F A
D. A F

1032. Why is it better to put use(data) in the else block, as shown, rather than inside the try block after load()? Analysis

```
try:
    data = load(path)
except OSError:
    print("could not load")
else:
    use(data)
```

A. Because code in else runs faster than code in try
B. Because else runs even when load() fails
C. Because exceptions raised in else are ignored
D. So that an OSError raised by use() is not mistaken for a loading failure and silently handled by the except clause

1033. In Python, which class is the root of the whole built-in exception hierarchy? Understanding

A. BaseException
B. Exception
C. Error
D. Throwable

1034. In Python, why does an `except Exception:` clause not catch the KeyboardInterrupt raised when the user presses Ctrl+C? Understanding

A. KeyboardInterrupt is an operating-system signal, so no except clause can ever catch it
B. KeyboardInterrupt inherits directly from BaseException, not from Exception
C. except Exception only matches exceptions raised with an explicit raise statement
D. KeyboardInterrupt is a checked exception that must be declared before it can be caught

1035. In Java, which class should a custom exception extend if callers must NOT be forced by the compiler to catch or declare it? Understanding

A. Exception
B. IOException
C. RuntimeException
D. Throwable

1036. In Java, a try block is followed by `catch (Exception e)` and then `catch (IOException e)`. What happens when the code is compiled? Understanding

A. It compiles, and the JVM picks the more specific IOException handler at run time
B. It compiles, and an IOException is handled by both blocks in order
C. It compiles with only a warning, and the second handler is ignored
D. Compile error: the IOException handler is unreachable because the Exception handler already catches it

1037. In Python, what is the effect of writing `raise ConfigError("bad file") from None` inside an except block? Understanding

A. The traceback no longer shows the original exception as context; only ConfigError is reported
B. ConfigError is raised with no message
C. The original exception is re-raised and ConfigError is discarded
D. The exception is raised but cannot be caught by any except clause

1038. In Python, what does a bare `raise` statement (with no expression after it) do inside an except block? Understanding

A. Raises a new generic Exception with no message
B. Re-raises the exception currently being handled, keeping its original traceback
C. Ends the except block silently, like pass
D. Raises RuntimeError because raise needs an argument

1039. In C++, which member function of std::exception returns a text description of the error? Understanding

A. getMessage()
B. message()
C. what()
D. toString()

1040. In C++, an exception passes up through several functions before it is caught. What happens to the fully constructed local objects in those functions? Understanding

A. They stay in memory until the program ends
B. They are destroyed only if the exception is caught by catch(...)
C. Their destructors are skipped, so RAII cannot release resources during exceptions
D. Their destructors run as the stack unwinds, in reverse order of construction

1041. In JavaScript, which values may appear after the throw keyword? Understanding

A. Any value, such as an Error object, a string or a number
B. Only instances of Error or its subclasses
C. Only strings and Error objects
D. Only objects created with the new keyword

1042. In Java, what does calling `e.printStackTrace()` on a caught exception do? Understanding

A. Re-throws the exception to the calling method
B. Prints the exception's type, message and the chain of method calls to the standard error stream
C. Returns the stack trace as a String without printing anything
D. Stops the program after printing the message

1043. In Python (run normally, without the -O flag), what happens when `assert x > 0, "x must be positive"` executes with x = -1? Understanding

A. A warning is printed and execution continues
B. ValueError is raised with the message "x must be positive"
C. AssertionError is raised with the message "x must be positive"
D. Nothing happens, because assert only runs inside test files

1044. In Python, an exception is raised inside a `with open("log.txt") as f:` block. What happens to the file? Understanding

A. It stays open until the garbage collector runs
B. It stays open, and the exception is suppressed
C. It is deleted, because the with block did not finish
D. It is closed by the file object's __exit__ method before the exception continues to propagate

1045. Which error in a Python source file cannot be handled by a try/except written in that same file? Understanding

A. A SyntaxError in the file, because the whole file is parsed before any of it runs
B. A ZeroDivisionError raised at run time
C. A FileNotFoundError from open()
D. A ValueError raised by int("abc")

1046. In C++, what does the statement `throw;` (with no operand) do inside a catch block? Understanding

A. Throws a default-constructed std::exception
B. Re-throws the exception object currently being handled, keeping its original dynamic type
C. Ends the catch block and resumes after the try statement
D. Throws an int with the value 0

1047. In JavaScript, which statement creates and throws an error object with the message "bad input"? Understanding

A. raise Error("bad input");
B. throws new Error("bad input");
C. throw new Error("bad input");
D. Error.throw("bad input");

1048. In Python, which built-in class is the direct parent of ZeroDivisionError? Understanding

A. ValueError
B. RuntimeError
C. LookupError
D. ArithmeticError

1049. In Java, unbounded recursion ends with StackOverflowError. How is this class classified? Understanding

A. As an Error (through VirtualMachineError), signalling a serious JVM problem that applications normally should not catch
B. As a checked exception that must be declared with throws
C. As a RuntimeException caused by a programming bug
D. As an IOException, because the stack is stored on disk

1050. Compared with returning error codes, what is a main advantage of reporting failures with exceptions? Understanding

A. Exceptions always run faster than if statements that check return codes
B. A failure cannot be silently ignored by forgetting to check a return value, and handling code is kept apart from the normal path
C. Exceptions make it impossible for a program to crash
D. Exceptions remove the need to validate input

1051. In C++, what does a plain `new` expression do when it cannot allocate the requested memory? Understanding

A. Returns nullptr
B. Throws std::out_of_range
C. Throws std::bad_alloc
D. Calls exit(1)

1052. In Java, which of these is a legal try statement? Understanding

A. try { ... } on its own, with no catch, finally or resources
B. catch (Exception e) { ... } with no try before it
C. finally { ... } followed by try { ... }
D. try { ... } finally { ... } with no catch block

1053. Python 3.11 added the `except*` syntax. What is it for? Understanding

A. Handling exceptions inside an ExceptionGroup, where each except* clause takes the matching members and the rest keep propagating
B. Catching every exception type, like a bare except
C. Catching an exception and automatically retrying the try block
D. Catching only exceptions raised from C extension modules

1054. In Java, a data-access layer catches an SQLException and wants to throw its own ServiceException without losing the original. What is the standard way? Understanding

A. Copy e.getMessage() into the new exception and discard e
B. Pass the SQLException as the cause, e.g. new ServiceException("load failed", e), so getCause() returns it
C. Call e.printStackTrace() and then throw ServiceException with no cause
D. Declare the method with throws SQLException, ServiceException

1055. In C++11 and later, what happens if a destructor that has no explicit noexcept(false) throws an exception that leaves it? Understanding

A. The exception propagates normally to the caller
B. The exception is silently discarded
C. std::terminate is called, because destructors are implicitly noexcept
D. The object is destroyed a second time

1056. In Java try-with-resources, the try body throws exception A, and then the resource's close() throws exception B. What does the catch block receive? Understanding

A. B, and A is lost
B. A, and B is lost without trace
C. A new exception whose cause is B
D. A, with B attached as a suppressed exception available through getSuppressed()

1057. What does this Python code print? Application

```
try:
    n = int(None)
except (ValueError, TypeError):
    print("bad input")
else:
    print(n)
```

A. bad input
B. None
C. 0
D. An uncaught TypeError traceback

1058. What does this Python code print? Application

```
try:
    raise ValueError("x", 2)
except ValueError as e:
    print(e.args)
```

A. x
B. ('x', 2)
C. x 2
D. ValueError('x', 2)

1059. What does this Python code print (one item per line)? Application

```
def f():
    for i in range(3):
        try:
            return i
        finally:
            print("fin", i)

print(f())
```

A. 0, fin 0
B. fin 0, fin 1, fin 2, 0
C. fin 0, 0
D. fin 0, fin 1, fin 2, 2

1060. What happens when this Python code runs? Application

```
try:
    int("a")
except ValueError:
    print("V")
    [][0]
except IndexError:
    print("I")
```

A. V and then I are printed
B. Only I is printed
C. Only V is printed and the IndexError is ignored
D. V is printed, then the program stops with an uncaught IndexError

1061. What does this Python code print? Application

```
class AppError(Exception):
    pass

class DbError(AppError):
    pass

try:
    raise DbError("db down")
except AppError as e:
    print("app:", e)
```

A. app: db down
B. app: DbError
C. app: AppError
D. An uncaught DbError traceback

1062. What does this Python code print? Application

```
try:
    {}["k"]
except KeyError as e:
    print(str(e))
```

A. k
B. 'k'
C. KeyError: 'k'
D. KeyError

1063. What does this Python code print? Application

```
try:
    value = {"a": 1}["b"]
except KeyError:
    value = 0
print(value + 5)
```

A. 6
B. 1
C. 5
D. An uncaught KeyError traceback

1064. What does this Python code print (one item per line)? Application

```
class Quiet:
    def __enter__(self):
        return self
    def __exit__(self, exc_type, exc, tb):
        print("exit", exc_type.__name__)
        return True

with Quiet():
    raise ValueError("x")
print("after")
```

A. exit ValueError
B. after
C. exit ValueError, then an uncaught ValueError traceback
D. exit ValueError, after

1065. What does this JavaScript code print? Application

```
try {
  null.x;
} catch (e) {
  console.log(e instanceof TypeError);
}
```

A. true
B. false
C. undefined
D. Nothing; reading null.x returns undefined

1066. What does this JavaScript code print? Application

```
try {
  throw "oops";
} catch (e) {
  console.log(typeof e);
}
```

A. object
B. string
C. Error
D. undefined

1067. What does this JavaScript code print (one item per line)? Application

```
async function f() {
  throw new Error("a");
}
f().catch(e => console.log("caught " + e.message));
console.log("after");
```

A. caught a, after
B. after
C. after, caught a
D. An uncaught Error: a

1068. What does this Java code print? Application

```
try {
    int[] a = new int[2];
    a[3] = 1;
    System.out.print("S");
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.print("A");
} catch (RuntimeException e) {
    System.out.print("R");
} finally {
    System.out.print("F");
}
```

A. ARF
B. SAF
C. RF
D. AF

1069. What does this Java code print? Application

```
try {
    int n = Integer.parseInt("12a");
    System.out.print("P" + n);
} catch (IllegalArgumentException e) {
    System.out.print("I");
} catch (RuntimeException e) {
    System.out.print("R");
}
```

A. I
B. R
C. P12
D. It does not compile, because NumberFormatException is not caught

1070. What does this Java code print? Application

```
String s = null;
try {
    System.out.println(s.length());
} catch (Exception e) {
    System.out.println(e.getClass().getSimpleName());
}
```

A. 0
B. NullPointerException
C. null
D. IllegalStateException

1071. What does this C++ program print? Application

```
#include <iostream>
#include <stdexcept>
#include <vector>
int main() {
    std::vector<int> v(3);
    try {
        v.at(5) = 1;
        std::cout << "set";
    } catch (const std::out_of_range&) {
        std::cout << "range";
    }
}
```

A. set
B. Nothing is guaranteed; the behaviour is undefined
C. range
D. It does not compile

1072. What does this C++ program print? Application

```
#include <iostream>
#include <stdexcept>
int main() {
    try {
        throw std::runtime_error("x");
    } catch (const std::exception&) {
        std::cout << "base";
    } catch (const std::runtime_error&) {
        std::cout << "derived";
    }
}
```

A. derived
B. basederived
C. It does not compile, because the handlers are in the wrong order
D. base

1073. What does this C++ program print? Application

```
#include <iostream>
void f() {
    try {
        throw 1;
    } catch (int) {
        std::cout << "inner ";
        throw;
    }
}
int main() {
    try {
        f();
    } catch (int x) {
        std::cout << "outer " << x;
    }
}
```

A. inner outer 1
B. inner
C. outer 1
D. inner outer 0

1074. What does this C++ program print? Application

```
#include <iostream>
struct T {
    char c;
    ~T() { std::cout << "~" << c << " "; }
};
void g() {
    T a{'a'};
    T b{'b'};
    throw 0;
}
int main() {
    try { g(); }
    catch (int) { std::cout << "caught"; }
}
```

A. ~a ~b caught
B. ~b ~a caught
C. caught ~b ~a
D. caught

1075. What does this Python code print (one item per line)? Application

```
def b():
    raise ValueError("v")

def a():
    try:
        b()
    finally:
        print("a-fin")

try:
    a()
except ValueError:
    print("main")
```

A. main, a-fin
B. main
C. a-fin, main
D. a-fin, then an uncaught ValueError traceback

1076. What happens when this Python 3 code runs? Application

```
e = "start"
try:
    1 / 0
except ZeroDivisionError as e:
    pass
print(e)
```

A. It prints start
B. It prints division by zero
C. It prints ZeroDivisionError
D. NameError, because the name e is deleted at the end of the except clause

1077. What does this Python code print? Application

```
total = 0
for s in ["1", "x", "3"]:
    try:
        n = int(s)
    except ValueError:
        continue
    else:
        total += n
print(total)
```

A. 4
B. 1
C. 13
D. An uncaught ValueError traceback

1078. What does this Python code print, and what is wrong with it? Analysis

```
try:
    int("x")
except Exception:
    print("general")
except ValueError:
    print("value")
```

A. It prints value, because Python chooses the most specific matching handler
B. It prints general; the ValueError handler can never run because the broader Exception handler is checked first
C. It prints general and then value
D. It raises SyntaxError, because a broader handler may not come before a narrower one

1079. A Python worker loop is written as shown. During testing the operator cannot stop it with Ctrl+C. Why? Analysis

```
while True:
    try:
        process_next_job()
    except:
        pass
```

A. Ctrl+C never works inside while True loops in Python
B. pass re-raises the exception, which restarts the loop
C. A bare except catches every exception, including KeyboardInterrupt and SystemExit, and pass discards it
D. process_next_job() blocks the keyboard until it returns

1080. What does this Python code print, and why? Analysis

```
def f():
    try:
        raise ValueError("lost")
    finally:
        return "ok"

print(f())
```

A. An uncaught ValueError traceback, because finally cannot stop an exception
B. ok, and then the ValueError traceback
C. None, because the function exits through an exception
D. ok, because a return in finally discards the exception that was propagating

1081. This pre-Java-7 code is meant to read a file safely. If readAll() throws an IOException and then close() also throws one, what does the caller see? Analysis

```
Reader r = new FileReader("a.txt");
try {
    readAll(r);
} finally {
    r.close();
}
```

A. Only the exception from close(); the original exception from readAll() is lost
B. Only the exception from readAll(); the close() failure is ignored
C. Both exceptions, with close()'s attached as suppressed
D. No exception, because finally handles them

1082. A Python function `isqrt_checked(n)` receives n = -4, an int. Which built-in exception should it raise? Analysis

A. TypeError, because negative numbers are the wrong type
B. ValueError, because the argument has the right type but an unacceptable value
C. KeyError, because -4 is not a valid key
D. RuntimeError, because the problem is detected at run time

1083. In this Python code, process() sometimes raises an exception. What is the problem, and what is the best fix? Analysis

```
f = open("data.txt")
data = f.read()
process(data)
f.close()
```

A. open() must be called inside process() instead
B. f.close() should be moved before f.read()
C. If process() raises, f.close() never runs; open the file with a with statement so it is always closed
D. There is no problem, because Python closes files immediately when an exception occurs

1084. What happens when this C++ program runs? Analysis

```
#include <iostream>
struct D {
    ~D() noexcept(false) { throw 2; }
};
int main() {
    try {
        D d;
        throw 1;
    } catch (int x) {
        std::cout << x;
    }
}
```

A. It prints 1
B. It prints 2
C. It prints 12
D. std::terminate is called, because a second exception escapes a destructor during stack unwinding

1085. What does this Python code print? Analysis

```
attempts = 0

def fetch():
    global attempts
    attempts += 1
    if attempts < 3:
        raise ConnectionError("down")
    return "data"

for i in range(5):
    try:
        result = fetch()
    except ConnectionError:
        continue
    break
print(attempts, result)
```

A. 3 data
B. 5 data
C. 2 data
D. An uncaught NameError, because result is never assigned

1086. What happens when this JavaScript code runs in Node.js? Analysis

```
try {
  setTimeout(() => {
    throw new Error("late");
  }, 0);
  console.log("scheduled");
} catch (e) {
  console.log("caught");
}
```

A. scheduled and then caught are printed
B. scheduled is printed, then the process crashes with an uncaught Error: late
C. Only caught is printed
D. scheduled is printed and the error is silently ignored

1087. In this Python code, write() fails with an OSError. What is the flaw, and what is the best fix? Analysis

```
def save(data):
    try:
        write(data)
    except OSError:
        rollback()

save(order)
print("Order saved")
```

A. rollback() should be moved into a finally block so it always runs
B. except OSError should be changed to except Exception
C. The error is swallowed, so the caller reports success; add a bare raise after rollback() so the failure still reaches the caller
D. There is no flaw, because rollback() has already undone the changes

1088. A Java server wraps its request loop in `catch (Throwable t) { log(t); continue; }`. What is the main risk? Analysis

A. Throwable cannot be caught, so the code does not compile
B. It catches only checked exceptions, so runtime bugs still crash the server
C. Logging a Throwable always throws a second exception
D. It also catches Errors such as OutOfMemoryError, after which the JVM may be in an unreliable state, yet the server keeps running

1089. In this C++ code, the developer placed the catch-all handler first. What happens when it is compiled? Analysis

```
try {
    risky();
} catch (...) {
    std::cout << "any";
} catch (const std::exception& e) {
    std::cout << e.what();
}
```

A. Compile error: a catch(...) handler must be the last handler of its try block
B. It compiles, and std::exception objects reach the second handler
C. It compiles, and both handlers run for a std::exception
D. It compiles, but catch(...) only catches non-class types such as int

1090. A student wants to catch both ValueError and TypeError. What happens when this Python code runs? Analysis

```
try:
    int(None)
except ValueError or TypeError:
    print("caught")
```

A. It prints caught, because or lists alternative exception types
B. The TypeError is not caught, because `ValueError or TypeError` evaluates to just ValueError
C. SyntaxError, because or is not allowed in an except clause
D. It prints caught twice, once for each type

1091. What does this Java program print before it ends? Analysis

```
public static void main(String[] args) {
    try {
        throw new IllegalStateException("x");
    } catch (IllegalStateException e) {
        System.out.print("catch ");
        throw new RuntimeException("y");
    } finally {
        System.out.print("finally ");
    }
}
```

A. catch, and then main ends with RuntimeException: y without running finally
B. catch finally, and the program ends normally
C. catch finally, and then main ends with an uncaught RuntimeException: y
D. finally catch, and then main ends with IllegalStateException: x

1092. Deep inside a Python app, load_settings() finds that config.json is missing. It cannot fix this, but the top-level main() can show the user a clear message. Where should the exception be handled? Analysis

A. Catch it in load_settings() and return an empty dictionary so the app keeps running
B. Catch it in load_settings() and call sys.exit() there
C. Wrap every function in the program in its own try/except Exception
D. Let it propagate (or raise a specific error) from load_settings() and catch it in main(), where a useful response is possible

Modules, Packages & Libraries MCQs for the NSCT

1093. In Python, what is a module? Understanding

A. A folder of .py files that must contain setup.py
B. A single .py file whose functions, classes and variables can be imported by other code
C. A compiled binary that only the interpreter can load
D. Any function defined with the def keyword

1094. Which Java statement lets you refer to java.util.ArrayList simply as ArrayList? Understanding

A. import java.util.ArrayList;
B. #include <java/util/ArrayList>
C. using java.util.ArrayList;
D. package java.util.ArrayList;

1095. In C++, what does `#include <vector>` do? Understanding

A. It downloads the vector library from the internet
B. The preprocessor copies the contents of the vector header into the source file before compilation
C. It links the compiled vector library at run time
D. It imports only the names used later in the file, like Python's from-import

1096. Which tool installs third-party Python packages from the Python Package Index (PyPI)? Understanding

A. npm
B. Maven
C. pip
D. javac

1097. Which Java package is imported automatically into every source file? Understanding

A. java.util
B. java.io
C. java.base
D. java.lang

1098. What is the main purpose of a namespace in C++? Understanding

A. To group names under a named scope so identifiers from different libraries do not collide
B. To allocate memory for global variables
C. To make a class's members private
D. To speed up compilation by caching headers

1099. What does the `if __name__ == "__main__":` check do in a Python file? Understanding

A. Runs the guarded code only when the file is imported
B. Runs the guarded code only when the file is executed directly, not when it is imported as a module
C. Marks the file as the package's entry point for pip
D. Prevents the module from being imported twice

1100. In Python, which file turns a directory into a regular (non-namespace) package and runs when the package is first imported? Understanding

A. __main__.py
B. setup.py
C. __init__.py
D. requirements.txt

1101. What is the difference between a static library and a dynamic (shared) library in C/C++? Understanding

A. A static library is loaded at run time; a dynamic library is copied in at compile time
B. Static libraries can only contain variables, dynamic libraries only functions
C. A static library's code is copied into the executable at link time; a dynamic library (.dll/.so) is loaded at run time and can be shared by programs
D. There is no difference except the file extension

1102. In a Java Maven project, what is pom.xml used for? Understanding

A. Storing the compiled .class files
B. Listing the Python packages the project needs
C. Configuring the operating system's environment variables
D. Declaring the project's dependencies (by groupId, artifactId and version) and its build configuration

1103. Why do Python developers create a virtual environment (for example with `python -m venv .venv`) for each project? Understanding

A. To run Python code faster
B. To give each project its own isolated set of installed packages and versions
C. To compile Python into a native executable
D. To share one global set of packages across all projects

1104. Under semantic versioning (MAJOR.MINOR.PATCH), which part should increase when a release makes a backward-incompatible API change? Understanding

A. MINOR
B. PATCH
C. MAJOR
D. Any part, as long as the number increases

1105. In Java, what access does a class member have when it is declared with no access modifier? Understanding

A. Package-private: accessible only from classes in the same package
B. Public: accessible from everywhere
C. Private: accessible only inside the class
D. Protected: accessible only from subclasses

1106. In npm's package.json, which versions does the range "^1.4.2" allow? Understanding

A. Any version from 1.4.2 up to, but not including, 2.0.0
B. Only 1.4.2 exactly
C. Any version from 1.4.2 up to, but not including, 1.5.0
D. Any version 1.4.2 or newer, including 2.x and 3.x

1107. What problem do include guards (`#ifndef X_H / #define X_H / ... / #endif`) or `#pragma once` solve in a C++ header? Understanding

A. They stop the header being included by more than one .cpp file in the program
B. They make the header's functions inline automatically
C. They prevent linker errors when a function is defined in two .cpp files
D. They stop the same header's contents being processed twice in one translation unit, which would cause redefinition errors

1108. In a Python module, what does the `__all__` list control? Understanding

A. Which names other modules may access with `import module`
B. Which functions run when the module is imported
C. Which public names `from module import *` imports
D. Which third-party packages pip installs with the module

1109. What does this Python code print? Application

```
import math as m
print(m.sqrt(16))
```

A. 4
B. 16
C. An error, because m is not a module name
D. 4.0

1110. What does this Python code print? Application

```
from math import pi, floor
print(floor(pi))
```

A. 3.14
B. 3
C. 4
D. An error, because math was not imported

1111. The project has two files. What does running `python main.py` print (one item per line)? Application

```
# counter.py
print("loading counter")
count = 0

# main.py
import counter
import counter
counter.count += 1
import counter
print(counter.count)
```

A. loading counter, loading counter, loading counter, 1
B. loading counter, 3
C. loading counter, 1
D. 1

1112. Running `python main.py` with these two files prints what? Application

```
# tools.py
print(__name__)

# main.py
import tools
```

A. __main__
B. main
C. tools.py
D. tools

1113. What does this Python code print? Application

```
def sqrt(x):
    return "mine"

from math import *
print(sqrt(16))
```

A. mine
B. 16
C. 4.0
D. An error, because sqrt is defined twice

1114. Running `python main.py` with these files produces what? Application

```
# shapes.py
__all__ = ["area"]
def area(r): return 3 * r * r
def perimeter(r): return 6 * r

# main.py
from shapes import *
print(area(2))
print(perimeter(2))
```

A. 12 and 12
B. 12, then a NameError for perimeter
C. A NameError for area
D. 12 only, with no error

1115. What does this C++ program print? Application

```
#include <iostream>
namespace A {
    int v = 1;
    namespace B { int v = 2; }
}
int v = 3;
int main() {
    std::cout << A::v << A::B::v << ::v;
}
```

A. 123
B. 111
C. 321
D. It does not compile, because v is defined three times

1116. Running `python main.py` from the project folder prints what? Application

```
# shop/__init__.py   (empty)

# shop/pricing.py
def tax(x):
    return x // 10

# shop/cart.py
from .pricing import tax
def total(x):
    return x + tax(x)

# main.py
from shop.cart import total
print(total(200))
```

A. 220
B. 200
C. 20
D. An ImportError, because relative imports are not allowed

1117. In Node.js (CommonJS modules), what does `node main.js` print? Application

```
// counter.js
let n = 0;
module.exports = { inc: () => ++n };

// main.js
const a = require("./counter");
const b = require("./counter");
a.inc();
a.inc();
console.log(b.inc());
```

A. 3
B. 1
C. 2
D. An error, because counter.js is required twice

1118. A teammate shares a requirements.txt file. Which command installs every package listed in it? Application

A. pip install requirements.txt
B. pip install -r requirements.txt
C. pip freeze requirements.txt
D. npm install requirements.txt

1119. A package.json lists the dependency "~2.3.1". Which of these versions can npm install for it? Application

A. 2.4.0
B. 2.3.9
C. 3.0.0
D. 2.3.0

1120. Running `python main.py` with these files prints what? Application

```
# config.py
level = 1
def bump():
    global level
    level += 1

# main.py
from config import level, bump
import config
bump()
print(level, config.level)
```

A. 2 2
B. 1 1
C. 2 1
D. 1 2

1121. What happens when this Java class is compiled? Application

```
import java.util.*;
import java.sql.*;

public class T {
    public static void main(String[] args) {
        Date d = null;
    }
}
```

A. It compiles and uses java.util.Date, because it was imported first
B. It compiles and uses java.sql.Date, because it was imported last
C. It compiles, because unused variables are removed
D. Compile error: the reference to Date is ambiguous, because both java.util and java.sql contain a Date class

1122. Which Java import lets you call `sqrt(2.0)` and use `PI` without writing `Math.` in front? Application

A. import java.lang.Math.*;
B. import java.lang.Math;
C. Nothing; java.lang is imported automatically, so it already works
D. import static java.lang.Math.*;

1123. A student saves this script as random.py and runs `python random.py`. What happens? Analysis

```
# random.py
import random
print(random.randint(1, 6))
```

A. An AttributeError, because `import random` finds the student's own random.py instead of the standard library module
B. It prints a random number between 1 and 6
C. A SyntaxError, because random is a reserved word
D. It loops forever, importing itself again and again

1124. Running `python main.py` with these three files fails. What is the cause? Analysis

```
# a.py
import b
x = 1

# b.py
from a import x

# main.py
import a
```

A. x must be declared global in a.py before it can be imported
B. A module cannot be imported by two different files
C. from-import cannot be used with modules that contain variables
D. A circular import: b.py asks for x from a.py while a.py is still only partly executed, so x does not exist yet

1125. Package shop/ contains an empty __init__.py, pricing.py (which defines tax) and the cart.py shown. From the project folder, a student runs `python shop/cart.py` directly. What happens? Analysis

```
# shop/cart.py
from .pricing import tax
def total(x):
    return x + tax(x)
```

A. It works exactly as when cart is imported from main.py
B. SyntaxError, because the dot before pricing is invalid
C. ModuleNotFoundError for pricing.py, because files must be imported with their .py extension
D. ImportError: a relative import has no known parent package when the file runs as a top-level script

1126. Why does Main.java fail to compile? Analysis

```
// com/shop/Item.java
package com.shop;
public class Item {
    int price = 5;
}

// com/app/Main.java
package com.app;
import com.shop.Item;
public class Main {
    public static void main(String[] args) {
        System.out.println(new Item().price);
    }
}
```

A. Item must be imported with a wildcard, as in `import com.shop.*;`
B. Two classes in different packages cannot be compiled together
C. price has default (package-private) access, so code in com.app cannot read a field declared in com.shop
D. Fields cannot be read through an object created inline with new

1127. A header util.h contains include guards and the full definition `int sq(int x) { return x * x; }`. Both a.cpp and b.cpp include it. Compilation succeeds but linking fails. Why? Analysis

A. The include guards stop b.cpp from seeing sq, so it is undefined there
B. Headers may only contain classes, never functions
C. Each .cpp gets its own copy of the non-inline definition, so the linker finds sq defined twice (an ODR violation); declare it inline or move the body to one .cpp
D. The linker needs sq to be declared static in main.cpp

1128. Project A needs requests 2.31 and Project B needs requests 2.20, and both use the same global Python installation. Installing for one breaks the other. What is the best fix? Analysis

A. Install both versions globally; Python picks the right one automatically
B. Uninstall requests and copy its source code into both projects
C. Give each project its own virtual environment with its own pinned requirements file
D. Run the projects with different file extensions

1129. A company wants security fixes in a shared compression library to reach all its installed C++ applications without rebuilding each application. Which linking choice fits? Analysis

A. Link the library dynamically (.dll/.so), so replacing the shared file updates every program that loads it
B. Link the library statically, so its code is copied into each executable
C. Copy the library's source into each project
D. Use header-only templates for the whole library

1130. Running `python main.py` with these files prints what? Analysis

```
# settings.py
items = []

# main.py
from settings import items
import settings
items.append(1)
settings.items = [9]
print(items, settings.items)
```

A. [1] [9]
B. [9] [9]
C. [1] [1]
D. [] [9]

1131. A code review flags `using namespace std;` placed in a shared header file. What is the strongest reason? Analysis

A. It makes the compiled program slower at run time
B. Every file that includes the header has all std names dumped into its global scope, which can cause ambiguous or surprising name clashes
C. It is a syntax error inside header files
D. It prevents the header from using std::vector

1132. Two developers run `npm install` on the same package.json, which lists "^4.17.0" for a library, a month apart, and get different library versions. What prevents this? Analysis

A. Deleting node_modules before every install
B. Committing package-lock.json and installing with `npm ci`, which uses the exact locked versions
C. Changing "^" to "~" in package.json
D. Running npm install twice

1133. In Python, when a script runs `import utils`, where does the interpreter look for utils? Understanding

A. In the directories listed in sys.path, which starts with the script's own folder, then PYTHONPATH entries, the standard library and site-packages
B. Only in the current working directory
C. Only in the folder where Python itself is installed
D. On PyPI, downloading the module automatically if it is not installed

1134. During an interactive Python session you edit mymod.py, but running `import mymod` again changes nothing. Which call re-executes the module's code? Understanding

A. import mymod, force=True
B. importlib.reload(mymod)
C. del mymod followed by import mymod
D. sys.refresh(mymod)

1135. Which of these Python modules is part of the standard library and needs no pip install? Understanding

A. requests
B. numpy
C. json
D. flask

1136. In a Python project, what does the command `pip freeze > requirements.txt` produce? Understanding

A. A frozen, read-only copy of the project's source code
B. A file of packages that are out of date
C. A compiled executable of the project
D. A file listing every installed package in the current environment with its exact version, e.g. requests==2.31.0

1137. By Java convention, a company that owns the domain example.com names the package for its billing code how? Understanding

A. com.example.billing
B. example.com.billing
C. Billing.Example.Com
D. billing@example.com

1138. In Java, a class declared with `package com.shop;` is named Item. Where should its source file normally be placed relative to the source root? Understanding

A. Item/com/shop.java
B. com/shop/Item.java
C. com.shop.Item.java in the source root
D. shop/com/Item.java

1139. In Java, what is a JAR file? Understanding

A. A Java source file that contains more than one class
B. A native executable produced by the C++ compiler
C. A ZIP-format archive that bundles compiled .class files and resources, with an optional manifest
D. A configuration file used only by Maven

1140. What is the role of the classpath in Java? Understanding

A. It sets the order in which classes are initialised
B. It defines which methods of a class are public
C. It is the folder where compiled classes are deleted after running
D. It lists the directories and JAR files where javac and the JVM look for classes and packages

1141. In the Java Platform Module System (Java 9+), what do the `requires` and `exports` directives in module-info.java do? Understanding

A. requires declares a module this module depends on; exports makes one of its packages accessible to other modules
B. requires lists packages to download; exports uploads the module to a repository
C. requires imports a class; exports marks a method as public
D. Both are aliases of import and have no effect at run time

1142. Which statement about Java's import declaration is correct? Understanding

A. It copies the imported class's bytecode into the compiled file
B. It only lets the source file use a short class name; it does not copy, load or link any code
C. It loads the class into memory as soon as the file is compiled
D. Without it, classes from other packages cannot be used at all

1143. A C++ program compiles every file without errors, but the build then fails with "undefined reference to `parse(int)'". Which stage reported this? Understanding

A. The preprocessor, which could not find a header
B. The compiler, which found a syntax error
C. The linker, which could not find a definition for a function that was declared and called
D. The program at run time, when parse was called

1144. In a typical C++ project, what usually goes into util.h and what into util.cpp? Understanding

A. util.h holds the definitions; util.cpp holds only #include lines
B. util.h holds main(); util.cpp holds everything else
C. Both files must contain identical code
D. util.h holds declarations (function prototypes, class definitions); util.cpp holds the function definitions

1145. In C++, what is the usual difference between `#include <file>` and `#include "file"`? Understanding

A. The quoted form searches the including file's directory first, then the system paths; the angle-bracket form searches only the configured system/include paths
B. Angle brackets are for C files and quotes are for C++ files
C. The quoted form includes the file once; angle brackets include it every time
D. There is no difference; the two are interchangeable in every compiler

1146. On Linux, which file extension does a shared (dynamic) library normally have? Understanding

A. .dll
B. .so
C. .a
D. .jar

1147. In C++, what is the effect of declaring a helper inside an unnamed namespace, e.g. `namespace { int helper() { return 1; } }`, in a .cpp file? Understanding

A. helper becomes global and can be called from every file without a declaration
B. helper can only be called from inside another namespace
C. helper gets internal linkage, so it is visible only in that translation unit and cannot clash with a helper in another file
D. It is a syntax error, because every namespace needs a name

1148. In C++, what is the difference between `using std::cout;` and `using namespace std;`? Understanding

A. The first imports the whole std namespace; the second imports only cout
B. The first links the iostream library; the second does not
C. They are identical
D. The first brings only the name cout into scope; the second makes every name in std visible without qualification

1149. In C++, which standard header must be included to use std::sort? Understanding

A. <algorithm>
B. <vector>
C. <sort>
D. <iostream>

1150. In JavaScript ES modules, geo.js contains `export default function area(r) {...}` and `export const PI = 3.14;`. Which import statement is correct? Understanding

A. import { area, PI } from "./geo.js";
B. import area, { PI } from "./geo.js";
C. import area, PI from "./geo.js";
D. import default area, PI from "./geo.js";

1151. In a Node.js project, what does running `npm install` with no package name do? Understanding

A. Installs the latest version of npm itself
B. Creates a new package.json file
C. Installs all dependencies listed in package.json (using package-lock.json versions if present) into node_modules
D. Publishes the project to the npm registry

1152. After `npm install` in a project, where are the downloaded packages placed by default? Understanding

A. In package.json
B. In package-lock.json
C. In the operating system's Program Files folder
D. In a node_modules folder inside the project

1153. What is the usual distinction between a library and a framework? Understanding

A. Your code calls a library; a framework calls your code (inversion of control)
B. A library is always larger than a framework
C. A framework is a single module; a library is a group of frameworks
D. Libraries are compiled and frameworks are interpreted

1154. In Python 3.3+, a directory named plugins contains .py files but no __init__.py, and it sits on sys.path. What happens on `import plugins.alpha`? Understanding

A. ImportError, because a package must have an __init__.py
B. It works: plugins is treated as an implicit namespace package (PEP 420)
C. Python creates an __init__.py file automatically
D. It works only if the files are renamed to .pyc

1155. Why does a __pycache__ folder with .pyc files appear next to modules after a Python program imports them? Understanding

A. It stores a backup copy of the source code
B. It holds the packages installed by pip
C. Python caches each imported module's compiled bytecode so later imports can skip recompiling unchanged source
D. It contains log files of every run

1156. On PyPI, many packages are published as wheel (.whl) files. What is a wheel? Understanding

A. A source-code tarball that must be compiled on install
B. A Python virtual environment saved to a file
C. A lockfile listing exact dependency versions
D. A built distribution archive that pip can install by unpacking it, without running a build step

1157. What happens when this Python code runs? Application

```
from math import sqrt
print(math.sqrt(4))
```

A. NameError: name 'math' is not defined
B. 2.0
C. 2
D. ImportError: cannot import name 'sqrt'

1158. What does this Python code print? Application

```
import random
random.seed(1)
a = random.random()
random.seed(1)
b = random.random()
print(a == b)
```

A. False
B. True
C. It varies from run to run
D. An error, because seed can only be set once

1159. Running `python main.py` with these two files prints what? Application

```
# greet.py
name = "Ali"
def hello():
    return "Hi " + name

# main.py
import greet
greet.name = "Sara"
print(greet.hello())
```

A. Hi Ali
B. An AttributeError, because module variables are read-only
C. Hi Sara
D. Hi greet.name

1160. Running `python main.py` from the project folder prints what? Application

```
# shop/__init__.py   (empty)

# shop/pricing.py
print(__name__)

# main.py
import shop.pricing
```

A. pricing
B. __main__
C. shop
D. shop.pricing

1161. Running `python main.py` from the project folder prints what? Application

```
# shop/__init__.py
from .pricing import tax

# shop/pricing.py
def tax(x):
    return x // 10

# main.py
import shop
print(shop.tax(50))
```

A. 5
B. AttributeError: module 'shop' has no attribute 'tax'
C. ImportError, because relative imports are not allowed in __init__.py
D. 5.0

1162. Running `python main.py` from the project folder produces what? Application

```
# shop/__init__.py   (empty)

# shop/pricing.py
def tax(x):
    return x // 10

# main.py
import shop
print(shop.pricing.tax(50))
```

A. 5
B. AttributeError: module 'shop' has no attribute 'pricing'
C. ImportError: No module named 'shop'
D. NameError: name 'pricing' is not defined

1163. What does this Python code print? Application

```
import json
print(json.dumps({"a": 1}))
```

A. {'a': 1}
B. {"a":1}
C. {"a": 1}
D. ["a", 1]

1164. What does this Python code print? Application

```
import importlib
m = importlib.import_module("math")
print(m.factorial(4))
```

A. math
B. 16
C. An error, because modules can only be imported with the import statement
D. 24

1165. What does this Python code print? Application

```
import math as m
print(m.__name__)
```

A. math
B. m
C. __main__
D. math as m

1166. In Node.js (CommonJS modules), what does `node main.js` print (one item per line)? Application

```
// log.js
console.log("load");
module.exports = 1;

// main.js
const a = require("./log");
const b = require("./log");
console.log(a + b);
```

A. load, load, 2
B. load, 2
C. 2
D. load, 11

1167. In Node.js (CommonJS modules), what does `node main.js` print? Application

```
// m.js
exports = { a: 1 };

// main.js
console.log(require("./m").a);
```

A. 1
B. An error: exports is read-only
C. undefined
D. { a: 1 }

1168. In Node.js (CommonJS modules), what does `node main.js` print? Application

```
// sq.js
module.exports = x => x * x;

// main.js
const sq = require("./sq");
console.log(sq(4));
```

A. undefined
B. An error: sq is not a function
C. [Function]
D. 16

1169. What does this C++ program print? Application

```
#include <iostream>
namespace a { int f() { return 1; } }
namespace b { int f() { return 2; } }
using b::f;
int main() {
    std::cout << f() + a::f();
}
```

A. 3
B. 2
C. 4
D. It does not compile, because f is ambiguous

1170. What happens when this C++ program is compiled? Application

```
#include <iostream>
int main() {
    cout << "hi";
}
```

A. It prints hi
B. Compile error: cout is not declared in this scope, because it lives in namespace std
C. Link error: the iostream library is missing
D. It compiles but prints nothing

1171. What does this C++ program print? Application

```
#include <iostream>
namespace N { int a = 1; }
namespace N { int b = 2; }
int main() {
    std::cout << N::a + N::b;
}
```

A. It does not compile, because N is declared twice
B. 2
C. 3
D. 12

1172. What happens when this Java code runs (with no import statements in the file)? Application

```
java.util.List<Integer> xs = java.util.List.of(1, 2);
System.out.println(xs.size());
```

A. Compile error: List must be imported first
B. It prints [1, 2]
C. Compile error: java.util is not accessible without import
D. It compiles and prints 2

1173. What happens when this Java file is compiled? Application

```
import java.util.*;

public class T {
    public static void main(String[] a) {
        Function<Integer, Integer> sq = x -> x * x;
        System.out.println(sq.apply(3));
    }
}
```

A. Compile error: cannot find symbol Function, because java.util.* does not import the subpackage java.util.function
B. It prints 9
C. It prints 6
D. Compile error: lambdas cannot be assigned to variables

1174. Which pip command displays the installed version, location and dependencies of the requests package? Application

A. pip info requests
B. pip show requests
C. pip version requests
D. pip get requests

1175. A Node.js project uses jest only to run its tests. Which command adds it to the devDependencies section of package.json? Application

    A. npm install jest
    B. npm install -g jest
    C. npm install --save-dev jest
    D. npm init jest

1176. A library is at version 3.4.1 under semantic versioning. The next release adds a new, backward-compatible feature and no breaking changes. What should its version be? Application

    A. 4.0.0
    B. 3.4.2
    C. 3.5.1
    D. 3.5.0

1177. On Linux, main.c calls sqrt() from the C maths library. Which flag added to `gcc main.c -o app` links that library (libm)? Application

    A. -lm
    B. -l m
    C. -L m
    D. -o m

1178. Running `python main.py` fails with ImportError (cannot import name 'helper' ... partially initialized module). Which single change fixes it? Analysis

```
# a.py
from b import helper
def main_value():
    return 10

# b.py
from a import main_value
def helper():
    return main_value() + 1

# main.py
import a
print(a.helper())
```

    A. In a.py, change the first line to `from b import *`
    B. In b.py, replace the from-import with `import a` and call a.main_value() inside helper()
    C. Add an empty __init__.py next to the three files
    D. Run the program with `python -O main.py`

1179. Running `python main.py` with these files produces what? Analysis

```
# mod.py
def calc(x):
    return _helper(x) * 2
def _helper(x):
    return x + 1

# main.py
from mod import *
print(calc(2))
print(_helper(2))
```

A. 6, then 3
B. NameError on the first print, because calc cannot see _helper
C. 6, then NameError: name '_helper' is not defined
D. ImportError, because mod has no __all__

1180. Running `python main.py` with these files prints what (one item per line)? Analysis

```
# a.py
print("a start")
import b
print("a end")

# b.py
print("b")

# main.py
import a
import b
print("main")
```

A. a start, b, a end, b, main
B. b, a start, a end, main
C. a start, a end, b, main
D. a start, b, a end, main

1181. stats.py ends with code that downloads a 2 GB dataset and trains a model. Another script only wants its mean() function, but `from stats import mean` takes 20 minutes. What is the best fix? Analysis

A. Move the download-and-train code into a function called under `if __name__ == "__main__":` in stats.py
B. Use `import stats` instead of `from stats import mean`
C. Delete stats.py's __pycache__ folder
D. Rename mean to _mean so it is imported faster

1182. A C++ tool must be copied as one file to air-gapped machines where its compression library is not installed and cannot be added. How should the library be linked? Analysis

A. Dynamically, so the executable loads the library at run time
B. Statically, so the library code is copied into the executable
C. It does not matter; executables never need external libraries
D. By putting the library's header file next to the executable

1183. config.h contains `int counter = 0;` (with include guards). Both a.cpp and b.cpp include it, and each file compiles. Why does linking fail, and what is the standard fix? Analysis
    A. Include guards are missing; adding #pragma once fixes it
    B. counter must be made const in both .cpp files
    C. Each .cpp file defines its own counter, giving a multiple-definition link error; declare `extern int counter;` in the header and define it in one .cpp file
    D. The files must be compiled in a different order

1184. main.cpp includes util.h and calls area(). util.cpp defines area(). The build command is `g++ main.cpp -o app`, and it fails with "undefined reference to area". What is the cause? Analysis
    A. util.h has no include guards
    B. area() must be declared static
    C. The header must be included with angle brackets
    D. util.cpp was not compiled and linked; use `g++ main.cpp util.cpp -o app`

1185. A Java app's classpath contains old-lib.jar and then new-lib.jar. Both contain a different version of com.acme.Util. At run time, which Util does the app use? Analysis
    A. The one in old-lib.jar, because the class loader takes the first match in classpath order
    B. The one in new-lib.jar, because the newest JAR always wins
    C. The JVM merges both classes into one
    D. The JVM refuses to start because of the duplicate class

1186. A Node.js app works on the developer's laptop but crashes in production with "Cannot find module 'express'". The server installs with `npm install --omit=dev`. What is the most likely cause? Analysis
    A. express must be installed globally with -g on servers
    B. express is listed under devDependencies instead of dependencies
    C. package-lock.json must be deleted before production installs
    D. require() cannot load packages in production mode

1187. A student runs `pip install requests`, which succeeds, but `python app.py` then fails with ModuleNotFoundError: No module named 'requests'. What is the most likely explanation? Analysis
    A. requests must also be imported with its version number
    B. The package needs a restart of the computer before it can be used
    C. The pip command belongs to a different Python installation or environment than the python that runs app.py; use `python -m pip install requests`
    D. requests is part of the standard library, so pip installed a conflicting copy

1188. package.json lists "^2.9.3" for a library. Version 3.0.0 of the library is released. What happens on a fresh `npm install` with no lockfile, and why? Analysis
    A. 3.0.0 is installed, because npm always picks the latest version
    B. Installation fails, because 3.0.0 is newer than allowed
    C. 2.9.3 exactly is installed, because the caret pins the version
    D. The newest 2.x version is installed, because a caret range does not cross a MAJOR version

1189. With ES modules in Node.js, what does `node main.mjs` print? Analysis

```
// counter.mjs
export let count = 0;
export function inc() { count++; }

// main.mjs
import { count, inc } from "./counter.mjs";
inc();
inc();
console.log(count);
```

A. 2
B. 0
C. An error, because imported bindings cannot change
D. undefined

1190. Running `python main.py` with these files prints what? Analysis

```
# a.py
def f():
    return "a"

# b.py
def f():
    return "b"

# main.py
from a import f
from b import f
print(f())
```

A. a
B. b
C. ab
D. ImportError, because the name f is imported twice

1191. Why does this C++ program fail to compile? Analysis

```
#include <algorithm>
#include <iostream>
using namespace std;
int count = 0;
int main() {
    count++;
    cout << count;
}
```

A. Global variables cannot be named count
B. <algorithm> must be included after using namespace std
C. The name count is ambiguous between the global variable and std::count, which the using-directive made visible
D. count++ is not allowed on a global variable

1192. A student plans to split CSV lines with line.split(",") instead of using Python's csv module. Which input breaks that approach but is handled by csv.reader? Analysis

A. A line with only three fields
B. A line containing numbers
C. A file with a header row
D. A quoted field that contains a comma, such as "Lahore, PK"

Advanced Programming Concepts MCQs for the NSCT

1193. What is the main benefit of using generics, such as List<String> in Java, instead of a raw List of Object? Understanding

A. Elements are stored more compactly in memory
B. The compiler checks element types, so casts are not needed when reading
C. The list becomes thread-safe automatically
D. A class can inherit from several parent classes

1194. Which of these is a valid Python lambda that doubles its argument? Understanding

A. lambda x: x * 2
B. lambda x: return x * 2
C. lambda(x) => x * 2
D. def lambda(x): x * 2

1195. In Java, what happens to the type argument of a generic such as List<String> at run time? Understanding

A. A separate class is generated for each type argument, as with C++ templates
B. It is kept and checked on every add() call
C. It is converted to the wrapper class Object[] and copied
D. It is erased, so List<String> and List<Integer> share the same runtime class

1196. When is the code for a C++ function template such as template <typename T> T maxOf(T a, T b) actually generated? Understanding

A. At run time, when the function is first called
B. Once, using a single version that works on void pointers
C. Only when the program is linked, never by the compiler
D. At compile time, once for each distinct type the template is used with

1197. Which pair of special methods makes a Python object an iterator? Understanding

A. __getitem__ and __len__
B. hasNext and next
C. __iter__ and __next__
D. __enter__ and __exit__

1198. Which two methods must every Java Iterator<E> provide (apart from the optional remove())? Understanding

A. hasNext() and next()
B. begin() and end()
C. __iter__() and __next__()
D. first() and last()

1199. How does a Python for loop know that an iterator has no more items? Understanding

A. __next__ returns None
B. __next__ raises IndexError
C. __next__ raises StopIteration
D. The loop checks len() before each step

1200. What is a higher-order function? Understanding

A. A function that takes a function as an argument or returns a function
B. A function declared inside a class
C. A function with more than three parameters
D. A recursive function that calls itself

1201. What is a closure? Understanding

A. A function that closes all open files when it ends
B. A function that cannot be overridden in a subclass
C. A block of code that runs when the program exits
D. An inner function that keeps access to variables of its enclosing function after that function has returned

1202. What happens when you call a Python function whose body contains yield? Understanding

A. It runs the whole body at once and returns a list
B. It raises a SyntaxError unless the call is inside a for loop
C. It runs the body until the first yield and returns that value
D. It returns a generator object; the body runs only as values are requested

1203. In Java, what does the bounded type parameter in <T extends Comparable<T>> T largest(List<T> xs) allow? Understanding

A. Passing a list of any object type at all
B. Calling compareTo on values of type T inside the method
C. Creating new T objects with new T()
D. Storing primitive int values in the list without boxing

1204. A Java method parameter is declared List<? extends Number> nums. Which statement is correct? Understanding

A. You can add any Number, but cannot read elements
B. You can both read and add Integer and Double values freely
C. You can read elements as Number, but the compiler rejects nums.add(5)
D. The declaration is illegal because wildcards need super

1205. What must a Java interface have so that a lambda expression can implement it? Understanding

A. At least two default methods
B. No methods at all
C. Exactly one abstract method
D. A public constructor

1206. In a C++ lambda, what does the capture list [&] mean? Understanding

A. Every outside variable used in the body is captured by reference
B. Every outside variable used in the body is copied when the lambda is created
C. The lambda captures nothing and can only use its parameters
D. Only the variable named & is captured

1207. Why is new T() inside a generic Java class a compile-time error? Understanding

A. Generic classes cannot contain constructors
B. T is erased, so the run-time type needed to create the object is unknown
C. new can only be used inside static methods
D. T might be a primitive type such as int

1208. In Python, a decorator @timer is written above def work(): .... What is this equivalent to? Understanding

A. work = work(timer)
B. timer = work(timer)
C. work = timer(work)
D. work.timer = True

1209. What does this Python code print? Application

```
print(list(map(lambda x: x * x, [1, 2, 3])))
```

A. [1, 2, 3]
B. [2, 4, 6]
C. [1, 4, 9]
D. <map object>

1210. What does this Python code print? Application

```
words = ['pear', 'fig', 'banana']
print(sorted(words, key=len))
```

A. ['banana', 'fig', 'pear']
B. ['banana', 'pear', 'fig']
C. ['fig', 'pear', 'banana']
D. ['pear', 'fig', 'banana']

1211. What does this Python code print? Application

```
def gen():
    for i in range(3):
        yield i * 10

g = gen()
print(next(g), next(g))
```

A. 0 0
B. 0 10
C. 10 20
D. [0, 10, 20]

1212. What does this Python code print? Application

```
print(list(filter(lambda n: n % 3 == 0, range(1, 10))))
```

A. [3, 6, 9]
B. [1, 2, 4, 5, 7, 8]
C. [0, 3, 6, 9]
D. [0, 3, 6]

1213. What does this Python code print? Application

```
def make_counter():
    count = 0
    def inc():
        nonlocal count
        count += 1
        return count
    return inc

c1 = make_counter()
c2 = make_counter()
c1(); c1()
print(c1(), c2())
```

A. 3 3
B. 3 1
C. 1 1
D. 2 1

1214. What does this Python code print? Application

```
from functools import reduce
print(reduce(lambda a, b: a * 10 + b, [4, 0, 7]))
```

A. 11
B. 47
C. 4070
D. 407

1215. What does this Python code print? Application

```
def shout(fn):
    def wrapper(name):
        return fn(name).upper() + "!"
    return wrapper

@shout
def greet(name):
    return "hi " + name

print(greet("ali"))
```

A. hi ali!
B. HI ALI!
C. HI ALI
D. hi ali

1216. What does this Python code print? Application

```
it = iter([1, 2, 3])
print(sum(it), sum(it))
```

A. 6 6
B. 6 12
C. 0 0
D. 6 0

1217. What does this JavaScript (Node.js) code print? Application

```
const add = (a, b) => a + b;
console.log([1, 2, 3].map(x => add(x, 1)));
```

A. [1, 2, 3]
B. [2, 3, 4]
C. [2, 4, 6]
D. 9

1218. What does this JavaScript code print? Application

```
console.log([1, 2, 3, 4].reduce((acc, x) => acc + x * x, 0));
```

A. 30
B. 10
C. 100
D. 0

1219. What does this Python code print? Application

```
nums = [1, 2, 3]
g = (x * 2 for x in nums)
nums.append(4)
print(list(g))
```

A. [2, 4, 6]
B. []
C. TypeError: list changed during iteration
D. [2, 4, 6, 8]

1220. What does this C++ code print? Application

```
#include <iostream>
using namespace std;

template <typename T>
T maxOf(T a, T b) { return a > b ? a : b; }

int main() {
    cout << maxOf(3, 7) << " " << maxOf(2.5, 1.5);
}
```

A. 7 2
B. 3 1.5
C. 7.0 2.5
D. 7 2.5

1221. What does this Java code print? Application

```
List<String> names = new ArrayList<>(List.of("Sara", "Ali", "Bilal"));
names.sort((a, b) -> b.compareTo(a));
System.out.println(names);
```

A. [Ali, Bilal, Sara]
B. [Sara, Ali, Bilal]
C. [Sara, Bilal, Ali]
D. [Bilal, Ali, Sara]

1222. What does this Python code print? Application

```
pairs = [(2, 'b'), (1, 'z'), (2, 'a')]
print(sorted(pairs, key=lambda t: (-t[0], t[1])))
```

A. [(1, 'z'), (2, 'a'), (2, 'b')]
B. [(2, 'b'), (2, 'a'), (1, 'z')]
C. [(1, 'z'), (2, 'b'), (2, 'a')]
D. [(2, 'a'), (2, 'b'), (1, 'z')]

1223. A student expects this Python code to print [0, 1, 2]. What does it print, and why? Analysis

```
fs = [lambda: i for i in range(3)]
print([f() for f in fs])
```

A. [0, 1, 2], because each lambda copies i when it is created
B. [2, 2, 2], because each lambda looks up i when called, after the loop has ended
C. [0, 0, 0], because i starts at 0
D. A NameError, because i no longer exists after the comprehension

1224. Using template <typename T> T maxOf(T a, T b), a student writes maxOf(3, 2.5) in C++. What happens? Analysis

A. Compile error: T is deduced as int from 3 and double from 2.5, which conflict
B. It returns 3, because the first argument fixes T as int
C. It returns 3.0, because int is promoted to double automatically
D. Run-time error when the comparison is evaluated

1225. Why does the second line of this Java code fail to compile, when String is a subclass of Object? Analysis

```
List<String> names = new ArrayList<>();
List<Object> objs = names;   // line 2
```

A. Generics are invariant: List<String> is not a subtype of List<Object>
B. ArrayList cannot be assigned to a List variable
C. The diamond operator <> is required on the right-hand side
D. Object lists must be created with new Object[]

1226. What is the output of this Python code? Analysis

```
squares = (x * x for x in range(4))
print(sum(squares))
print(max(squares))
```

A. 14 and then 9
B. 14, then a ValueError because the generator is already exhausted
C. 14 and then 0
D. A TypeError on the first line, because sum cannot take a generator

1227. What happens when this Java code runs? Analysis

```
List<Integer> xs = new ArrayList<>(List.of(1, 2, 3, 4));
for (Integer x : xs) {
    if (x % 2 == 0) xs.remove(x);
}
```

A. It throws ConcurrentModificationException on the iteration after 2 is removed
B. It prints nothing and xs becomes [1, 3]
C. It removes the element at index 2 and index 4
D. It loops forever because the size keeps changing

1228. What does this Python code print, and why? Analysis

```
def log(fn):
    def wrapper(*args):
        return fn(*args)
    return wrapper

@log
def area(r):
    return 3.14 * r * r

print(area.__name__)
```

A. area, because decorators keep the original name
B. wrapper, because the decorator returned a different function object
C. log, because the decorator's name is used
D. None, because wrapper has no name

1229. What does this Python code print? Analysis

```
class Countdown:
    def __init__(self, n):
        self.n = n
    def __iter__(self):
        return self
    def __next__(self):
        if self.n <= 0:
            raise StopIteration
        self.n -= 1
        return self.n + 1

c = Countdown(3)
print(list(c), list(c))
```

A. [3, 2, 1] [3, 2, 1]
B. [2, 1, 0] []
C. [3, 2, 1] []
D. [3, 2, 1, 0] []

1230. What does this JavaScript code print? Analysis

```
const fs = [];
for (var i = 0; i < 3; i++) fs.push(() => i);
console.log(fs.map(f => f()).join(" "));
```

A. 0 1 2
B. 0 0 0
C. undefined undefined undefined
D. 3 3 3

1231. This Python code is meant to sort words ignoring case. What happens? Analysis

```
words = ['Banana', 'apple', 'cherry']
print(sorted(words, key=lambda w: w.lower))
```

A. It sorts correctly: ['apple', 'Banana', 'cherry']
B. It sorts by the original case: ['Banana', 'apple', 'cherry']
C. TypeError, because the key returns bound methods, which cannot be compared with <
D. AttributeError, because str has no lower attribute

1232. A C++ program defines struct Point { int x, y; }; with no operators, and the template maxOf from earlier. It compiles until a line calls maxOf(p1, p2). Why does the error appear only then? Analysis

A. A template is type-checked fully only when instantiated, and Point has no operator>
B. Structs cannot be passed to templates
C. maxOf must be declared virtual to accept user types
D. Templates only accept built-in numeric types

1233. In Java, which java.util.function interface fits a lambda that takes one argument of type T and returns a boolean, such as s -> s.isEmpty()? Understanding

A. Supplier<T>
B. Consumer<T>
C. Runnable
D. Predicate<T>

1234. In Java, which functional interface represents an operation that takes one argument and returns no result, such as x -> System.out.println(x)? Understanding

A. Consumer<T>
B. Supplier<T>
C. Predicate<T>
D. UnaryOperator<T>

1235. In Java, the method reference String::length is used where a Function<String, Integer> is expected. Which lambda is it equivalent to? Understanding

A. () -> String.length()
B. (s, n) -> s.length(n)
C. s -> String.valueOf(s)
D. s -> s.length()

1236. In Java, a lambda body reads a local variable declared in the enclosing method. What rule must that variable satisfy? Understanding

A. It must be declared static
B. It must be declared volatile
C. It must be final or effectively final (never reassigned after initialisation)
D. It must be a field of the functional interface

1237. In Java, a method should accept a List<Integer>, a List<Number> or a List<Object> and add Integer values to it. Which parameter type allows this? Understanding

A. List<? extends Integer>
B. List<? super Integer>
C. List<Integer>
D. List<?>

1238. Why can a Java class not declare both void print(List<String> x) and void print(List<Integer> x)? Understanding

A. Java does not allow overloading methods that take collections
B. Both erase to print(List), so the two methods have the same signature after type erasure
C. List<Integer> is a subtype of List<String>, so the calls are ambiguous
D. Generic types cannot be used as method parameters

1239. A C++ program declares template <typename T> class Stack { ... }; and uses both Stack<int> and Stack<double>. What does the compiler produce? Understanding

A. One shared class that stores every element as void*
B. One class whose type is decided at run time
C. Two separate class types, one generated for int and one for double
D. Nothing until the program is linked

1240. In C++, given template <typename T> class Box { ... }; a programmer also writes template <> class Box<bool> { ... }. What is the second declaration? Understanding

A. An explicit (full) specialisation that gives Box a separate implementation when T is bool
B. A second primary template that replaces the first one for every type
C. A syntax error, because template <> must list at least one parameter
D. A partial specialisation that applies to every pointer type

1241. In C++, template <int N> struct Buffer { int data[N]; }; is used as Buffer<16>. Which statement about N is correct? Understanding

A. N is a non-type template parameter and must be a compile-time constant
B. N can be any int variable whose value is read from the user at run time
C. N must be a type such as int or double
D. N is ignored, because arrays inside templates are always resized dynamically

1242. Why does sum(x * x for x in range(10_000_000)) in Python use far less memory than sum([x * x for x in range(10_000_000)])? Understanding

A. The generator expression produces one value at a time instead of building the whole list first
B. Generator expressions store their values in compressed form
C. Python skips the multiplication for generator expressions
D. The list comprehension is evaluated twice

1243. Inside a Python generator, what does the statement yield from sub() do? Understanding

A. It yields the generator object sub() itself as a single value
B. It calls sub() to completion and yields only its last value
C. It imports the function sub from another module
D. It yields every value that the sub-generator sub() produces, one by one, and evaluates to sub()'s return value

1244. In C++, what does v.end() return for a std::vector<int> v? Understanding

A. An iterator to the last element
B. The value of the last element
C. The number of elements in v
D. An iterator one position past the last element

1245. In C++, a program stores auto it = v.begin(); for a std::vector<int> v and then calls v.push_back(9), which makes the vector reallocate. Which statement is correct? Understanding

A. it is updated automatically to point into the new storage
B. it is invalidated, because the elements moved to a new block of memory, so using it is undefined behaviour
C. it now points to the new element 9
D. push_back never moves elements, so it stays valid

1246. What does placing @functools.lru_cache(maxsize=None) above a Python function fib(n) do? Understanding

A. It wraps fib so results are stored by argument, and repeated calls with the same n return the stored result
B. It runs fib in a background thread
C. It converts fib into a generator
D. It limits fib to being called once

1247. In Python 3, which of these objects is an iterator (it has a __next__ method)? Understanding

A. The list [1, 2]
B. The string 'ab'
C. The tuple (1, 2)
D. The object returned by map(str, [1, 2])

1248. How does a JavaScript arrow function decide what this means inside its body? Understanding

A. this is always the object the arrow function was called on
B. this is always the global object
C. It has no this of its own and uses the this of the surrounding (enclosing) scope
D. this is always the arrow function itself

1249. In functional programming, what is a pure function? Understanding

A. A function that has no parameters
B. A function written as a lambda
C. A function that is called only once
D. A function whose result depends only on its arguments and which has no side effects

1250. A function add(a, b) is rewritten so that it is called as add(2)(3). What is this transformation called? Understanding

A. Recursion
B. Overloading
C. Memoisation
D. Currying

1251. Why does List<int> fail to compile in Java, while List<Integer> compiles? Understanding

A. Java type arguments must be reference types, so the primitive int must be replaced by its wrapper Integer
B. int is a reserved word that cannot appear inside angle brackets
C. List can only hold String values
D. Generics only work with user-defined classes

1252. A Java Comparator<T> is passed to list.sort(). Its compare(a, b) returns a negative number. What does the sort take this to mean? Understanding

A. a should be placed before b
B. b should be placed before a
C. a and b are equal
D. The comparison failed and the sort stops

1253. In Java, nums.stream().filter(n -> n > 2).map(n -> n * 10) is written with no further call. What work is done? Understanding

A. Every element is filtered and mapped immediately and the results are stored
B. None: filter and map are lazy intermediate operations that run only when a terminal operation such as collect() or sum() is called
C. Only the filter runs; map waits for a terminal operation
D. The stream throws an exception because it was not closed

1254. Why does the Python line f = lambda x: y = x + 1 raise a SyntaxError? Understanding

A. Lambdas cannot take parameters
B. The lambda is missing a return keyword
C. Lambdas must be defined inside a class
D. A lambda body must be a single expression, and an assignment is a statement

1255. In Python, two decorators are stacked: @a on the first line, @b on the line below, then def f(): .... What is this equivalent to? Understanding

A. f = b(a(f))
B. f = a(f); f = b(f) is applied to the original f only
C. f = a(b(f))
D. f = a(f) + b(f)

1256. A Python generator g has been started with next(g). What does g.send(10) do? Understanding

A. It appends 10 to the generator's output list
B. It resumes g, makes the paused yield expression evaluate to 10, and returns the next value g yields
C. It restarts g from the beginning with 10 as its argument
D. It raises an exception, because only next() can resume a generator

1257. What does this Python code print? Application

```
print(list(zip([1, 2, 3], "ab")))
```

A. [(1, 'a'), (2, 'b')]
B. [(1, 'a'), (2, 'b'), (3, None)]
C. [(1, 'a'), (2, 'b'), (3, '')]
D. A ValueError, because the lengths differ

1258. What does this Python code print? Application

```
print(list(enumerate("hi", start=1)))
```

A. [(0, 'h'), (1, 'i')]
B. [(1, 'h'), (2, 'i')]
C. [('h', 1), ('i', 2)]
D. [1, 2]

1259. What does this Python code print? Application

```
from functools import partial

def power(b, e):
    return b ** e

sq = partial(power, e=2)
print(sq(5), sq(3))
```

A. 32 8
B. 10 6
C. 25 9
D. A TypeError, because power needs two arguments

1260. What does this Python code print? Application

```
fs = [lambda x, k=k: x * k for k in (1, 2, 3)]
print([f(10) for f in fs])
```

A. [30, 30, 30]
B. [1, 2, 3]
C. [10, 20, 30]
D. A TypeError, because each lambda takes two arguments

1261. What does this Python code print? Application

```
def inner():
    yield 1
    yield 2
    return "done"

def outer():
    r = yield from inner()
    yield r

print(list(outer()))
```

A. [1, 2]
B. [1, 2, None]
C. [1, 2, 'done']
D. ['done']

1262. What does this Python code print? Application

```
def acc():
    total = 0
    while True:
        x = yield total
        total += x

g = acc()
next(g)
g.send(5)
print(g.send(10))
```

A. 10
B. 5
C. None
D. 15

1263. What does this Python code print? Application

```
def a(f):
    return lambda: "a(" + f() + ")"

def b(f):
    return lambda: "b(" + f() + ")"

@a
@b
def g():
    return "g"

print(g())
```

A. b(a(g))
B. a(g)
C. b(g)
D. a(b(g))

1264. What does this Python code print? Application

```
import itertools

def nat():
    n = 1
    while True:
        yield n
        n += 1

print(list(itertools.islice(nat(), 2, 5)))
```

A. [3, 4, 5]
B. [2, 3, 4]
C. [3, 4, 5, 6]
D. It never finishes, because nat() is infinite

1265. What does this Python code print? Application

```
print(sorted([-5, 3, -1, 4], key=abs))
```

A. [-5, -1, 3, 4]
B. [1, 3, 4, 5]
C. [-1, 3, 4, -5]
D. [4, 3, -1, -5]

1266. What does this Python code print? Application

```
def repeat(n):
    def deco(f):
        def wrapper():
            return [f() for _ in range(n)]
        return wrapper
    return deco

@repeat(3)
def hi():
    return "hi"

print(hi())
```

A. ['hi', 'hi', 'hi']
B. hihihi
C. ['hi']
D. A TypeError, because repeat(3) is not a function

1267. What does this Python code print? Application

```
class Bag:
    def __init__(self, items):
        self.items = items
    def __iter__(self):
        for x in self.items:
            yield x * 2

b = Bag([1, 2])
print(list(b), list(b))
```

A. [2, 4] []
B. [2, 4] [2, 4]
C. [1, 2] [1, 2]
D. A TypeError, because Bag has no __next__ method

1268. What does this JavaScript (Node.js) code print? Application

```
console.log([3, 1, 2].map((x, i) => x * i));
```

A. [3, 1, 2]
B. [3, 2, 6]
C. [0, 1, 4]
D. [0, 0, 0]

1269. What does this JavaScript (Node.js) code print? Application

```
function makeCounter() {
  let c = 0;
  return () => ++c;
}
const a = makeCounter();
const b = makeCounter();
a();
a();
console.log(a(), b());
```

A. 3 3
B. 1 1
C. 3 1
D. 3 4

1270. What does this JavaScript (Node.js) code print? Application

```
console.log([1, 2, 3, 4, 5].filter(n => n % 2).map(n => n * 10).join(","));
```

A. 20,40
B. 10,30,50
C. 10,20,30,40,50
D. 1,3,5

1271. What does this JavaScript (Node.js) generator code print? Application

```
function* g() {
  const x = yield 1;
  yield x * 2;
}
const it = g();
console.log(it.next().value, it.next(5).value, it.next().done);
```

A. 1 10 true
B. 1 2 false
C. 1 NaN true
D. 1 10 false

1272. What does this Java code print? Application

```
List<Integer> xs = List.of(4, 7, 10, 3);
int s = xs.stream()
          .filter(x -> x > 3)
          .mapToInt(x -> x * 2)
          .sum();
System.out.println(s);
```

A. 48
B. 42
C. 24
D. 21

1273. What does this C++ code print? Application

```
#include <iostream>
using namespace std;

int main() {
    int x = 1;
    auto byVal = [x]() { return x; };
    auto byRef = [&x]() { return x; };
    x = 5;
    cout << byVal() << " " << byRef();
}
```

A. 5 5
B. 1 1
C. 1 5
D. 5 1

1274. What does this C++ code print? Application

```
#include <iostream>

template <int N>
int fact() { return N * fact<N - 1>(); }

template <>
int fact<0>() { return 1; }

int main() {
    std::cout << fact<4>();
}
```

A. 0
B. 24
C. 4
D. It does not compile, because the recursion never ends

1275. What does this Python code print? Application

```
from functools import reduce
print(reduce(lambda acc, s: acc + len(s), ["ab", "cde", ""], 10))
```

A. 15
B. 5
C. 10
D. A TypeError, because acc starts as an int

1276. What does this Python code print? Application

```
print(list(map(lambda a, b: a + b, [1, 2, 3], [10, 20])))
```

A. [11, 22, 3]
B. [11, 22]
C. [11, 22, 33]
D. A ValueError, because the lists have different lengths

1277. What does this Python code print? Application

```
def compose(f, g):
    return lambda x: f(g(x))

inc = lambda x: x + 1
dbl = lambda x: x * 2
print(compose(inc, dbl)(5), compose(dbl, inc)(5))
```

A. 12 11
B. 11 11
C. 11 12
D. 12 12

1278. A student expects this Python code to print 1. What happens? Analysis

```
def counter():
    c = 0
    def inc():
        c += 1
        return c
    return inc

f = counter()
print(f())
```

A. It prints 1
B. UnboundLocalError, because the assignment c += 1 makes c a local variable of inc
C. It prints 0, because inc changes a copy of c
D. NameError, because c is destroyed when counter() returns

1279. What does this Python program print? Analysis

```
def risky():
    print("start")
    yield 1 / 0

g = risky()
print("made")
```

A. start, then made
B. A ZeroDivisionError
C. start, then made, then a ZeroDivisionError
D. made

1280. What does this Python code print? Analysis

```
it = iter([1, 2, 3, 4, 5])
pairs = list(zip(it, range(2)))
print(pairs, next(it))
```

A. [(1, 0), (2, 1)] 4
B. [(1, 0), (2, 1)] 3
C. [(1, 0), (2, 1), (3, None)] 4
D. A StopIteration error from next(it)

1281. What happens when this Java code is compiled and run? Analysis

```
List<String> names = new ArrayList<>();
List raw = names;          // raw type
raw.add(42);
String s = names.get(0);
```

A. It fails to compile at raw.add(42)
B. It runs and prints nothing, because 42 is converted to "42"
C. It throws ClassCastException at raw.add(42)
D. It compiles with an unchecked warning, then throws ClassCastException at the get line

1282. Why does this Java code fail to compile? Analysis

```
int count = 0;
List<String> xs = List.of("a", "b");
xs.forEach(x -> count++);
```

A. forEach cannot take a lambda argument
B. count is modified, so it is not effectively final and the lambda cannot use it
C. x is never used inside the lambda
D. List.of returns an immutable list, so forEach is not allowed

1283. What is wrong with this C++ code? Analysis

```
#include <functional>

std::function<int()> makeGetter() {
    int n = 5;
    return [&n]() { return n; };
}

int main() {
    auto g = makeGetter();
    return g();
}
```

A. Nothing; the lambda keeps n alive for as long as the lambda exists
B. It does not compile, because a lambda cannot be stored in a std::function
C. The lambda captures n by reference, but n is destroyed when makeGetter returns, so calling g() is undefined behaviour
D. It always returns 0, because local variables are reset when a function returns

1284. This file is run with Node.js as a CommonJS script. What does it print, and why? Analysis

```
const obj = {
  n: 3,
  regular() { return this.n; },
  arrow: () => this.n,
};
console.log(obj.regular(), obj.arrow());
```

A. 3 3, because both functions are called on obj
B. 3 undefined, because the arrow function takes this from the module scope, not from obj
C. undefined undefined, because object methods never receive this
D. A TypeError, because arrow functions cannot be object properties

1285. What happens when this Python code runs? Analysis

```
d = {"a": 1, "b": 2}
for k in d:
    d[k + "x"] = 0
```

A. It adds 'ax' and 'bx' and finishes normally
B. It loops forever, adding 'ax', 'axx', 'axxx' and so on
C. KeyError, because 'ax' does not exist yet
D. RuntimeError, because the dictionary changes size while it is being iterated

1286. What does this Python code print? Analysis

```
n = 2
g = (x for x in range(5) if x % n == 0)
n = 3
print(list(g))
```

A. [0, 2, 4]
B. [0, 1, 2, 3, 4]
C. [0, 2, 3, 4]
D. [0, 3]

1287. A student expects this Python code to print all four pairs (1, 1), (1, 2), (2, 1), (2, 2). What does it print? Analysis

```
class Two:
    def __init__(self):
        self.i = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.i >= 2:
            raise StopIteration
        self.i += 1
        return self.i

t = Two()
print([(a, b) for a in t for b in t])
```

A. [(1, 1), (1, 2), (2, 1), (2, 2)]
B. [(1, 1), (2, 2)]
C. [(1, 2)]
D. []

1288. A student expects this Python code to print call and then 5. What does it print? Analysis

```
def log(fn):
    def wrapper(*a):
        print("call")
        fn(*a)
    return wrapper

@log
def add(a, b):
    return a + b

print(add(2, 3))
```

A. call, then None
B. call, then 5
C. Only 5
D. A TypeError, because wrapper takes *a

1289. This Java code is meant to sort in ascending order. What does it print, and why? Analysis

```
List<Integer> xs = new ArrayList<>(List.of(5, Integer.MIN_VALUE));
xs.sort((a, b) -> a - b);
System.out.println(xs);
```

A. [-2147483648, 5], because the comparator sorts in ascending order
B. [5, -2147483648], because a - b overflows and gives the wrong sign
C. An ArithmeticException, because a - b overflows
D. A compile error, because a lambda cannot be a Comparator

1290. A student expects [1, 2, 3]. What does this JavaScript (Node.js) code print, and why? Analysis

```
console.log(["1", "2", "3"].map(parseInt));
```

A. [1, 2, 3], because parseInt converts each string
B. ['1', '2', '3'], because map returns the original strings
C. [1, NaN, NaN], because map passes the index as parseInt's radix argument
D. A TypeError, because parseInt is not a lambda

1291. What does this Python code print? Analysis

```
def g():
    yield 1
    return 5

print(list(g()))
```

A. [1, 5]
B. [5]
C. A StopIteration error
D. [1]

1292. This C++ code should remove the even numbers from v. What is wrong with it? Analysis

```
std::vector<int> v = {1, 2, 4, 5};
for (auto it = v.begin(); it != v.end(); ++it)
    if (*it % 2 == 0) v.erase(it);
```

A. erase invalidates it, so the ++it that follows is undefined behaviour; use it = v.erase(it) and only increment when nothing is erased
B. Nothing; erase shifts the iterator to the next element automatically
C. It does not compile, because erase needs an index, not an iterator
D. It removes the odd numbers instead, because % 2 == 0 tests for odd values

Concurrency & Parallelism MCQs for the NSCT

1293. Which statement correctly separates concurrency from parallelism? Understanding

A. Parallelism is possible on a single core; concurrency needs more than one core
B. They are two names for the same thing
C. Concurrency applies only to processes and parallelism only to threads
D. Concurrency is possible on a single core; parallelism needs more than one processing unit

1294. Which resource is private to each thread rather than shared by all threads of the same process? Understanding

A. The heap
B. Global variables
C. Its stack (and register values)
D. Open files

1295. What is a race condition? Understanding

A. A thread that runs faster than the others
B. A bug where the result depends on the unpredictable timing or interleaving of threads
C. A deadlock involving more than two threads
D. A compiler optimisation that reorders loops

1296. What is a critical section? Understanding

A. The first function that runs when a thread starts
B. A piece of code that accesses shared data and must not be executed by two threads at once
C. Code that handles fatal exceptions
D. The part of a program with the highest CPU usage

1297. How does a counting semaphore initialised to 3 differ from a mutex? Understanding

A. Up to three threads can hold it at once, while a mutex allows only one owner
B. It allows exactly three threads in total over the whole program run
C. It can be released only by the thread that acquired it, unlike a mutex
D. It is used only for processes, while a mutex is only for threads

1298. Which of these is NOT one of the four necessary conditions for deadlock (the Coffman conditions)? Understanding

A. Mutual exclusion
B. Hold and wait
C. Circular wait
D. Bounded waiting

1299. Why do CPU-bound Python threads usually give little speed-up in standard CPython? Understanding

A. Python threads are always run on a single core by the operating system
B. Python does not support threads, only coroutines
C. The Global Interpreter Lock lets only one thread run Python bytecode at a time
D. Each thread copies the whole program, which wastes the time saved

1300. In Java, what does marking an instance method synchronized do? Understanding

A. The calling thread must acquire the object's intrinsic lock before running the method
B. The method runs in a new thread each time it is called
C. The method's local variables become shared between threads
D. The method is guaranteed to run before any other method of the class

1301. What does calling t.join() on a thread t do? Understanding

A. The calling thread waits until t has finished
B. It starts t
C. It merges t into the main thread so they share a stack
D. It stops t immediately

1302. A Java programmer calls t.run() instead of t.start() on a Thread object. What happens? Understanding

A. A new thread is created, exactly as with start()
B. run() executes in the current thread, so no new thread is created
C. A compile-time error occurs
D. An IllegalThreadStateException is thrown

1303. A Java field is declared volatile int count. Which guarantee does volatile give? Understanding

A. count++ becomes an atomic operation
B. Only one thread may read count at a time
C. Writes by one thread become visible to other threads, but count++ is still not atomic
D. count is stored on each thread's stack

1304. Two threads keep reacting to each other: each releases its lock and retries at the same moment, so both stay busy but neither makes progress. What is this called? Understanding

A. Livelock
B. Deadlock
C. Starvation
D. Busy waiting

1305. What does it mean for an operation to be atomic? Understanding

A. It completes as one indivisible step that other threads can never see half-done
B. It runs on the fastest core
C. It uses the least memory
D. It can be interrupted and resumed safely at any point

1306. In the classic bounded-buffer producer-consumer solution (as in Silberschatz), which synchronisation objects are used? Understanding

A. A mutex plus two counting semaphores, empty (initially n) and full (initially 0)
B. A single mutex only
C. Two mutexes, one for the producer and one for the consumer
D. One counting semaphore initialised to n, with no mutex

1307. What is the main benefit of a thread pool such as Java's ExecutorService? Understanding

A. It removes the need for locks on shared data
B. It reuses a fixed set of threads, avoiding the cost of creating one per task
C. It guarantees tasks finish in the order they were submitted
D. It makes every task run in parallel on its own core

1308. According to Amdahl's law, what is the maximum possible speed-up of a program whose serial fraction is s, however many processors are added? Understanding

A. 1 / (1 − s)
B. s × the number of processors
C. Unlimited, as processors keep increasing
D. 1 / s

1309. 80% of a program can be parallelised. Using Amdahl's law, what is the speed-up on 4 processors? Application

A. 3.2
B. 4
C. 2.5
D. 2

1310. A program is 90% parallelisable. What is the upper limit on its speed-up with unlimited processors? Application

A. 9
B. 10
C. 90
D. Unlimited

1311. 75% of a task can run in parallel. What speed-up does Amdahl's law predict on 3 cores? Application

A. 3
B. 2
C. 2.25
D. 1.75

1312. What does this Python program print? Application

```
import threading

count = 0
lock = threading.Lock()

def work():
    global count
    for _ in range(1000):
        with lock:
            count += 1

ts = [threading.Thread(target=work) for _ in range(4)]
for t in ts: t.start()
for t in ts: t.join()
print(count)
```

A. 1000
B. A different value each run, below 4000
C. 0
D. 4000

1313. What does this Python program print? Application

```
import threading

def task():
    print("child")

t = threading.Thread(target=task)
t.start()
t.join()
print("main")
```

A. main, then child
B. Only main
C. child, then main
D. Either order, depending on scheduling

1314. A semaphore is initialised to 3. Five threads each call acquire() (wait) and none has released yet. What is the state? Application

A. All five proceed
B. One proceeds and four are blocked
C. Three proceed and two receive an error
D. Three threads proceed and two are blocked

1315. A counting semaphore S starts at 7. Then 20 wait (P) operations and 15 signal (V) operations complete. What is the final value of S? Application

A. 12
B. −13
C. 2
D. 22

1316. Thread 1 locks A and then B. Thread 2 locks B and then A. Which change removes the possibility of deadlock? Application

A. Make the locks reentrant
B. Give Thread 1 a higher priority
C. Add sleep(1) between the two lock calls
D. Make both threads lock A before B

1317. Shared count starts at 5. Two threads each run count = count + 1 as three separate steps (read, add, write) with no lock. Which final values are possible? Application

A. Only 7
B. 5, 6 or 7
C. Only 6
D. 6 or 7

1318. What does this Python program print? Application

```
from concurrent.futures import ThreadPoolExecutor

with ThreadPoolExecutor(max_workers=3) as ex:
    print(list(ex.map(lambda x: x * x, [3, 1, 2])))
```

A. [9, 1, 4]
B. [1, 4, 9]
C. The squares in whichever order the threads finish
D. [3, 1, 2]

1319. What does this Node.js code print (one value per line)? Application

```
console.log("A");
setTimeout(() => console.log("B"), 0);
Promise.resolve().then(() => console.log("C"));
console.log("D");
```

A. A B C D
B. A D B C
C. A D C B
D. A C D B

1320. A job takes 120 s on one core and 40 s on 4 cores. What is the parallel efficiency? Application

A. 300%
B. 33%
C. 25%
D. 75%

1321. In the dining philosophers problem with 5 philosophers and 5 forks (each needs two forks to eat), what is the most philosophers that can eat at the same time? Application

A. 5
B. 1
C. 3
D. 2

1322. What does this Python asyncio program print? Application

```
import asyncio

async def job(name, delay):
    await asyncio.sleep(delay)
    print(name)
    return name

async def main():
    r = await asyncio.gather(job("x", 0.2), job("y", 0.1))
    print(r)

asyncio.run(main())
```

A. x, then y, then ['x', 'y']
B. y, then x, then ['x', 'y']
C. y, then x, then ['y', 'x']
D. x, then y, then ['y', 'x']

1323. Two Java threads each run count++ 1,000 times on a shared int with no synchronisation. The printed total is often below 2,000. Why? Analysis

A. count++ is a read, an add and a write, so concurrent increments can overwrite each other
B. Java threads skip loop iterations when the CPU is busy
C. int overflows after 1,000 increments
D. The JVM runs only one of the two threads

1324. Thread 1 runs transfer(a, b) and Thread 2 runs transfer(b, a) at the same time. Sometimes the program hangs. What is the cause? Analysis

```
def transfer(src, dst, amount):
    with src.lock:
        with dst.lock:
            src.balance -= amount
            dst.balance += amount
```

A. Python locks are reentrant, so a thread can lock the same account twice
B. The GIL stops either thread from acquiring a lock
C. Each thread can hold its first lock while waiting for the other's, creating circular wait
D. with blocks release locks too early

1325. Two threads call withdraw(80) at the same time on an account with balance 100. Sometimes the balance ends at −60. Why? Analysis

```
def withdraw(amount):
    global balance
    if balance >= amount:
        balance -= amount
```

A. Floating-point rounding makes the comparison wrong
B. Both threads can pass the check before either subtracts (a check-then-act race)
C. The second call subtracts 80 twice
D. The if statement is evaluated after the subtraction

1326. A CPU-bound Python function is split across 4 threads in standard CPython, but it runs no faster than with 1 thread. What is the best explanation and fix? Analysis

A. Four threads are too few; use 100 threads
B. The GIL serialises bytecode execution; use multiprocessing (separate processes) instead
C. The threads need join() calls to start running
D. Python threads only run on I/O; replace the loops with time.sleep()

1327. A program is 60% parallelisable. By roughly how much does its Amdahl speed-up improve when the cores are doubled from 4 to 8? Analysis

A. From about 1.82 to about 2.11
B. From 2.4 to 4.8
C. From 4 to 8
D. It does not change

1328. What does this Python program print? Analysis

```
import threading

lock = threading.Lock()
rlock = threading.RLock()
lock.acquire()
rlock.acquire()
print(lock.acquire(timeout=0.1), rlock.acquire(timeout=0.1))
```

A. True True
B. False True
C. True False
D. It hangs forever

1329. A developer makes a shared Java counter volatile int hits and has several threads run hits++. Counts are still lost. What is the correct diagnosis? Analysis

A. volatile only works on long and double fields
B. volatile makes each thread keep a private copy of hits
C. volatile gives visibility but not atomicity; use AtomicInteger or synchronized
D. The lost counts come from integer overflow

1330. A readers-writers lock always lets new readers in while any reader is active. Under heavy read traffic a writer waits for minutes. What problem is this? Analysis

A. Deadlock
B. Livelock
C. Writer starvation
D. A race condition on the data

1331. A program runs two threads on a machine with a single CPU core. Which description is accurate? Analysis

A. The threads run concurrently by interleaving, but not in parallel
B. The threads run in parallel because each has its own stack
C. The second thread cannot start until the first one ends
D. The threads cannot share any variables

1332. Two threads each need locks A and B. Each takes one lock, tries the other with tryLock(), and on failure releases what it holds and retries at once. Both threads are busy but neither finishes. What is happening and what is a common fix? Analysis

A. Deadlock; make the locks reentrant
B. Starvation; raise one thread's priority
C. A race condition; make the variables volatile
D. Livelock; add a random back-off before retrying

1333. One CPU instruction adds four pairs of numbers at once using wide vector registers. In Flynn's taxonomy, which category is this? Understanding

A. SIMD (single instruction, multiple data)
B. SISD (single instruction, single data)
C. MISD (multiple instruction, single data)
D. MIMD (multiple instruction, multiple data)

1334. An array of one million numbers is split into four equal chunks, and each of four cores sums one chunk before the partial sums are added. What kind of parallelism is this? Understanding

A. Data parallelism
B. Task parallelism
C. Pipeline parallelism
D. Instruction-level parallelism

1335. How does Gustafson's law differ from Amdahl's law? Understanding

A. Gustafson ignores the serial part of a program, while Amdahl includes it
B. Gustafson assumes the problem size grows with the number of processors, while Amdahl assumes a fixed problem size
C. Gustafson applies only to GPUs, while Amdahl applies only to CPUs
D. Gustafson measures energy use, while Amdahl measures time

1336. A thread holds a mutex and calls wait() on a condition variable associated with it. What happens? Understanding

A. It releases the mutex and sleeps until signalled, then re-acquires the mutex before wait() returns
B. It keeps the mutex and sleeps, so no other thread can enter
C. It releases the mutex permanently and returns immediately
D. It busy-waits in a loop, checking the condition

1337. Why should a thread check its condition in a while loop around wait(), rather than with a single if? Understanding

A. A while loop makes wait() return faster
B. An if statement cannot contain a call to wait()
C. After waking, the condition may be false again because of spurious wake-ups or another thread acting first, so it must be re-checked
D. The while loop keeps the lock held during the wait

1338. In Java, a thread calls obj.wait() without holding obj's monitor (it is not inside synchronized (obj)). What happens? Understanding

A. An IllegalMonitorStateException is thrown
B. The thread waits normally
C. The call is ignored
D. The JVM automatically acquires the monitor first

1339. In Java, what is the key difference between Thread.sleep(1000) and obj.wait(1000) when called inside synchronized (obj)? Understanding

A. sleep releases the monitor, but wait keeps it
B. Both release the monitor
C. sleep keeps the monitor locked while pausing, but wait releases it so other threads can enter
D. Neither pauses the thread; both only yield the CPU once

1340. In Java, what does notifyAll() do that notify() does not? Understanding

A. It wakes the waiting threads in priority order
B. It wakes every thread waiting on the object's monitor, not just one
C. It releases the monitor immediately
D. It interrupts threads that are sleeping

1341. Why is Python's queue.Queue a common choice for passing work from producer threads to consumer threads? Understanding

A. It runs each item in a separate process
B. It removes the Global Interpreter Lock
C. It sorts items by priority automatically
D. It does its own locking, so put() and get() are thread-safe, and get() blocks while the queue is empty

1342. In Python, a thread is created with daemon=True. What does that mean? Understanding

A. The thread runs with a higher priority than other threads
B. The thread runs in a separate process
C. The thread restarts automatically if it crashes
D. The program can exit while the thread is still running; the thread is stopped abruptly at exit

1343. What is thread-local storage? Understanding

A. Memory shared by all threads of a process
B. A variable for which each thread has its own separate copy
C. The stack of the main thread
D. A file each thread writes its log to

1344. What does an atomic compare-and-swap (CAS) instruction do? Understanding

A. It swaps the contents of two registers without using memory
B. It compares two threads' priorities and swaps which one runs
C. If a memory location still holds an expected value, it replaces it with a new value, as one indivisible step, and reports whether it succeeded
D. It locks the whole memory bus until the thread finishes its critical section

1345. When is a spinlock (a lock that busy-waits in a loop) a reasonable choice? Understanding

A. When the lock is held only very briefly on a multiprocessor, so spinning costs less than a context switch
B. When the lock may be held for several seconds
C. On a single-core machine with long critical sections
D. Whenever the critical section performs disk I/O

1346. In Java, thread T2 tries to enter a synchronized block whose monitor is held by thread T1. Which Thread.State does T2 have while it waits? Understanding

A. BLOCKED
B. WAITING
C. TIMED_WAITING
D. RUNNABLE

1347. Why are immutable objects, such as Java String or a Python tuple of numbers, safe to share between threads without locks? Understanding

A. They are copied automatically into each thread
B. They are stored in thread-local storage
C. Their state cannot change after creation, so no thread can see a half-finished update
D. The runtime locks them every time they are read

1348. Under the many-to-one threading model in Silberschatz, many user threads map to one kernel thread. Which limitation follows? Understanding

A. Each user thread needs its own kernel thread, which is expensive
B. Threads cannot share global variables
C. If one thread makes a blocking system call the whole process blocks, and the threads cannot run in parallel on multiple cores
D. The operating system schedules each user thread separately

1349. According to Silberschatz, which three requirements must a correct solution to the critical-section problem satisfy? Understanding

A. Mutual exclusion, hold and wait, and circular wait
B. Atomicity, consistency and isolation
C. Preemption, priority and fairness
D. Mutual exclusion, progress and bounded waiting

1350. Node.js runs JavaScript on a single thread. How can one Node.js server handle many network requests concurrently? Understanding

A. It creates a new JavaScript thread for every request
B. It handles one request completely before accepting the next
C. It uses non-blocking I/O and an event loop that runs callbacks when each I/O operation completes
D. It runs each request on a separate CPU core automatically

1351. In JavaScript, what does a function declared with async always return? Understanding

A. A Promise
B. The value written after return
C. undefined
D. A generator object

1352. Why does a global variable changed inside a Python multiprocessing.Process not change in the parent process? Understanding

A. Each process has its own separate memory, so the child changes its own copy
B. The Global Interpreter Lock blocks writes to globals
C. Globals are read-only inside functions
D. The parent process restores its globals after join()

1353. Two threads on different cores repeatedly update two different variables that happen to sit in the same cache line. The program runs far slower than expected. What is this effect called? Understanding

A. Deadlock
B. Priority inversion
C. False sharing
D. Starvation

1354. Why does Java's ConcurrentHashMap usually perform better under heavy multithreaded use than a map wrapped with Collections.synchronizedMap()? Understanding

A. It turns off thread safety for reads
B. It copies the whole map for each thread
C. It does not lock the whole map for every operation, so many reads and writes to different parts can proceed at once
D. It stores keys in sorted order

1355. What does a barrier, such as Python's threading.Barrier(4), do? Understanding

A. It allows at most 4 threads into a critical section
B. It stops 4 threads from ever running at the same time
C. It cancels any thread that arrives late
D. It makes each thread wait until all 4 threads have reached the barrier, then releases them together

1356. In Java, a task is submitted to an ExecutorService and returns a Future<Integer> f. What does f.get() do? Understanding

A. It starts the task
B. It cancels the task
C. It returns null immediately if the task is not finished
D. It waits, if necessary, until the task finishes and then returns its result

1357. Half of a program can be parallelised. Using Amdahl's law, what is the speed-up on 8 processors? Application

A. 4
B. 8
C. About 1.14
D. About 1.78

1358. Benchmarks show a program can never exceed a speed-up of 5, however many cores are added. According to Amdahl's law, what fraction of it is parallelisable? Application

A. 80%
B. 20%
C. 50%
D. 95%

1359. Using Gustafson's law, S = N - s(N - 1), what is the scaled speed-up on 16 processors when the serial fraction s is 0.1? Application

A. 6.4
B. 16
C. 1.6
D. 14.5

1360. A job takes 200 s on one core and 25 s on 10 cores. What are the speed-up and the parallel efficiency? Application

A. Speed-up 10, efficiency 100%
B. Speed-up 8, efficiency 8%
C. Speed-up 175, efficiency 17.5%
D. Speed-up 8, efficiency 80%

1361. 90% of a program can run in parallel. Using Amdahl's law, what speed-up does it get on 10 cores? Application

A. About 5.26
B. 9
C. 10
D. About 1.82

1362. What does this Python program print? Application

```
import threading

ready = threading.Event()
out = []

def worker():
    ready.wait()
    out.append("worker")

t = threading.Thread(target=worker)
t.start()
out.append("main")
ready.set()
t.join()
print(out)
```

A. ['main', 'worker']
B. ['worker', 'main']
C. ['worker']
D. It changes from run to run

1363. What does this Python producer-consumer program print? Application

```
import threading, queue

q = queue.Queue()
results = []

def producer():
    for i in range(3):
        q.put(i * i)
    q.put(None)

def consumer():
    while (item := q.get()) is not None:
        results.append(item)

c = threading.Thread(target=consumer)
p = threading.Thread(target=producer)
c.start(); p.start()
p.join(); c.join()
print(results)
```

A. [0, 1, 4, None]
B. [4, 1, 0]
C. It changes from run to run
D. [0, 1, 4]

1364. What does this Python asyncio program print (one pair per line)? Application

```
import asyncio

async def task(name, n):
    for i in range(n):
        print(name, i)
        await asyncio.sleep(0)

async def main():
    await asyncio.gather(task("A", 2), task("B", 2))

asyncio.run(main())
```

A. A 0, A 1, B 0, B 1
B. B 0, B 1, A 0, A 1
C. B 0, A 0, B 1, A 1
D. A 0, B 0, A 1, B 1

1365. What does this Python asyncio program print (the elapsed time in whole seconds)? Application

```
import asyncio, time

async def main():
    start = time.perf_counter()
    await asyncio.gather(asyncio.sleep(1), asyncio.sleep(2))
    print(round(time.perf_counter() - start))

asyncio.run(main())
```

A. 3
B. 2
C. 1
D. 0

1366. What does this Node.js code print (one value per line)? Application

```
async function f() {
  console.log("f1");
  await null;
  console.log("f2");
}
console.log("start");
f();
console.log("end");
```

A. start, f1, end, f2
B. start, f1, f2, end
C. start, end, f1, f2
D. f1, start, end, f2

1367. What does this Node.js code print (one value per line)? Application

```
const wait = (ms, v) => new Promise(r => setTimeout(() => r(v), ms));
const ps = () => [wait(30, "a"), wait(10, "b"), wait(20, "c")];
Promise.all(ps()).then(v => console.log(v.join("")));
Promise.race(ps()).then(v => console.log(v));
```

A. abc, then b
B. b, then abc
C. b, then bca
D. a, then abc

1368. Semaphore S starts at 0. Thread X and thread Y run the pseudocode below at the same time. In which order are the lines printed? Application

```
Thread X:            Thread Y:
  wait(S)              print("first")
  print("second")      signal(S)
```

A. second, then first
B. Either order, depending on scheduling
C. Only first; X waits forever
D. first, then second

1369. What does this Java code print? Application

```
AtomicInteger c = new AtomicInteger(5);
int a = c.getAndIncrement();
int b = c.incrementAndGet();
System.out.println(a + " " + b + " " + c.get());
```

A. 6 7 7
B. 5 6 7
C. 6 6 6
D. 5 7 7

1370. Each resource has one instance. T1 holds R1 and requests R2. T2 holds R2 and requests R3. T3 holds R3 and requests nothing more. Which statement is true? Application

A. T1, T2 and T3 are deadlocked
B. Only T1 and T2 are deadlocked
C. There is no deadlock: T3 can finish and release R3, then T2, then T1
D. There is no deadlock, and T1 finishes first

1371. What does this Python program print? Application

```
from multiprocessing import Process

count = 0

def work():
    global count
    count += 100

if __name__ == "__main__":
    p = Process(target=work)
    p.start()
    p.join()
    print(count)
```

A. 100
B. 200
C. 0
D. It changes from run to run

1372. What does this Python program print? Application

```
from concurrent.futures import ThreadPoolExecutor

def sq(x):
    return x * x

with ThreadPoolExecutor(max_workers=3) as ex:
    futs = [ex.submit(sq, n) for n in (3, 1, 2)]
    print([f.result() for f in futs])
```

A. [1, 4, 9]
B. [3, 1, 2]
C. [9, 1, 4]
D. It changes from run to run

1373. What does this Python program print? Application

```
import threading

data = threading.local()
data.x = "main"

def work():
    data.x = "worker"

t = threading.Thread(target=work)
t.start()
t.join()
print(data.x)
```

A. worker
B. main
C. None
D. An AttributeError

1374. What does this Python program print? Application

```
import threading

s = threading.Semaphore(2)
print(s.acquire(blocking=False), s.acquire(blocking=False), s.acquire(blocking=False))
s.release()
print(s.acquire(blocking=False))
```

A. True True True, then True
B. True False False, then False
C. True True False, then True
D. True True False, then False

1375. A thread pool has 4 worker threads. 10 independent tasks are submitted, each taking exactly 1 s, and the machine has enough cores. Ignoring overhead, what is the shortest total time? Application

A. 2.5 s
B. 3 s
C. 10 s
D. 4 s

1376. 16 numbers are summed in parallel by a tree reduction: in each step, pairs of partial sums are added at the same time. With as many processors as needed, how many addition steps are required? Application

A. 15
B. 4
C. 8
D. 16

1377. On one core a program spends 40 s in a serial part and 60 s in a part that divides perfectly across cores. How long does it take on 4 cores? Application

A. 25 s
B. 55 s
C. 40 s
D. 100 s

1378. This Python code should create exactly one shared Config object. With two threads calling get_config() at once, two objects are sometimes created. What is the cause and fix? Analysis

```
_instance = None

def get_config():
    global _instance
    if _instance is None:
        _instance = Config()
    return _instance
```

A. Config() is too slow; cache its result in a local variable
B. global is not thread-safe; use nonlocal instead
C. Both threads can see _instance is None before either assigns it; hold one lock around both the check and the assignment
D. The GIL prevents this, so the second object must come from another module

1379. This Java double-checked locking code sometimes lets a thread see a Helper whose fields are not yet initialised. What is the standard fix? Analysis

```
private static Helper helper;

static Helper getHelper() {
    if (helper == null) {
        synchronized (Helper.class) {
            if (helper == null) helper = new Helper();
        }
    }
    return helper;
}
```

A. Declare the field as private static volatile Helper helper, so the write publishing the object happens-before other threads' reads
B. Remove the outer if, because it can never be true
C. Make the getHelper method private
D. Replace synchronized with Thread.sleep(1)

1380. Two consumer threads use this Python code with one producer. Occasionally a consumer crashes with IndexError: pop from empty list. Why? Analysis

```
def consume():
    with cond:
        if not items:
            cond.wait()
        return items.pop()
```

A. notify() wakes both consumers, which is not allowed
B. list.pop() is never thread-safe, even under a lock
C. wait() does not release the lock, so the producer never adds items
D. The if checks the condition only once; a woken consumer can find the item already taken, so the check must be a while loop

1381. What does this Python program print? Analysis

```
import threading, time

def bg():
    time.sleep(1)
    print("bg done")

threading.Thread(target=bg, daemon=True).start()
print("main done")
```

A. main done, then bg done
B. bg done, then main done
C. Nothing, because the program exits immediately
D. main done

1382. In Java, the synchronized method a() calls the synchronized method b() on the same object. Will a thread calling a() deadlock? Analysis

```
class Account {
    synchronized void a() { b(); }
    synchronized void b() { /* ... */ }
}
```

A. Yes: the thread waits forever for the lock it already holds
B. No: Java intrinsic locks are reentrant, so the thread that holds the monitor can acquire it again
C. Yes, but only if another thread is also calling b()
D. It does not compile, because one synchronized method cannot call another

1383. Thread 1 updates a shared counter inside with lock_a:, and Thread 2 updates the same counter inside with lock_b:. Updates are still lost. Why? Analysis

A. Two locks always cause a deadlock
B. with blocks release the lock before the update
C. The threads use different locks, so they never exclude each other
D. Counters cannot be protected by locks

1384. A program achieves a speed-up of 9.5 on 8 cores, which is more than 8. What is the most likely explanation? Analysis

A. Amdahl's law was broken, so the measurement must be wrong
B. Each core's share of the data now fits in its cache, so memory accesses got faster (a superlinear speed-up)
C. The serial fraction is negative
D. The operating system secretly used 9.5 cores

1385. A Python web server handles each request in its own thread, but every handler wraps a 2 s database call inside the same global lock. Throughput is no better than one thread. What is the best diagnosis? Analysis

A. The GIL prevents threads from waiting on a database
B. There are too few threads; add more
C. The critical section is too large: the lock serialises all requests, so only the shared-state update should be inside it
D. The database call must be made with the lock held to be safe

1386. A high-priority task H waits for a mutex held by low-priority task L. A medium-priority task M, which needs no lock, keeps pre-empting L, so H misses its deadline. What is this and the standard fix? Analysis

A. Deadlock; fix it by ordering the locks
B. Priority inversion; fix it with priority inheritance, so L temporarily runs at H's priority
C. Livelock; fix it with random back-off
D. Starvation of M; fix it with ageing

1387. This Python asyncio program prints the elapsed time in whole seconds. A student expects 1. What does it print, and why? Analysis

```
import asyncio, time

async def job():
    time.sleep(1)

async def main():
    start = time.perf_counter()
    await asyncio.gather(job(), job())
    print(round(time.perf_counter() - start))

asyncio.run(main())
```

A. 2, because time.sleep blocks the whole event loop, so the two coroutines run one after the other
B. 1, because gather runs the coroutines concurrently
C. 0, because time.sleep is skipped inside coroutines
D. It raises an error, because time.sleep cannot be used in a coroutine

1388. A Python script downloads 20 web pages one after another. A student claims that using 20 threads cannot help, because of the GIL. Which statement is correct? Analysis

A. The student is right; threads never speed up Python code
B. Threads help here: the work is I/O-bound, and a thread releases the GIL while it waits on the network
C. Only multiprocessing can make network calls in parallel
D. Threads help only if the machine has 20 cores

1389. To prevent deadlock, a program makes each thread request all the locks it needs at once, and if any is unavailable it takes none and tries again later. Which Coffman condition does this remove? Analysis

A. Hold and wait
B. Mutual exclusion
C. No preemption
D. Circular wait

1390. Two threads each create their own Counter object and call increment() 1,000 times. The total, read from Counter.total, is often below 2,000. Why? Analysis

```
class Counter {
    static int total = 0;
    synchronized void increment() { total++; }
}
```

A. synchronized methods cannot modify static fields
B. A synchronized instance method locks this, and the threads use different objects, so nothing protects the shared static field
C. 1,000 calls are too many for one synchronized method
D. total++ is atomic, so the counts must be lost elsewhere

1391. A Node.js server computes a large prime in a tight loop inside its request handler. While one request is being served, every other request stalls. Why, and what is the usual fix? Analysis

    A. Node runs only one request per CPU core; add more cores
    B. The loop blocks the single event-loop thread; move the computation to a worker thread (worker_threads) or a separate process
    C. The handler should be marked async, which runs it on another thread
    D. Requests stall because the loop uses too much memory

1392. In this Java code the worker thread sometimes never stops after main sets running = false. What is the cause and fix? Analysis

```
class Worker extends Thread {
    boolean running = true;
    public void run() {
        while (running) { }
    }
}
// main: worker.start(); ...; worker.running = false;
```

    A. The while loop needs a Thread.yield() to read memory
    B. Booleans cannot be shared between threads; use an int
    C. main must call worker.run() to stop the loop
    D. Without volatile, the worker may never see the new value of running; declare the field volatile

Debugging, Testing & Optimization MCQs for the...

1393. What is a breakpoint in a debugger? Understanding

    A. A line that the compiler skips
    B. A marked line where execution pauses so you can inspect the program's state
    C. The point where a program crashes
    D. A comment telling other developers where a bug is

1394. Execution is paused on a line that calls calculate(x). Which debugger command moves into the body of calculate? Understanding

    A. Step over
    B. Step out
    C. Step into
    D. Continue

1395. A loop runs 10,000 times, and a bug appears only when i == 7342. Which debugger feature is most useful? Understanding

    A. An ordinary breakpoint on the loop line
    B. Step over through every iteration
    C. A conditional breakpoint with the condition i == 7342
    D. Restarting the program in release mode

1396. A program compiles and runs without crashing, but it prints the wrong total. What kind of error is this? Understanding

    A. Logic error
    B. Syntax error
    C. Compile-time type error
    D. Runtime exception

1397. What is a unit test? Understanding

A. A test of the whole system by end users
B. A test that measures how fast the program runs
C. A manual check of the user interface
D. An automated test that checks one small piece of code, such as a function, in isolation

1398. What is the order of the test-driven development (TDD) cycle? Understanding

A. Write a failing test, write code to pass it, then refactor
B. Write the code, then write tests, then refactor
C. Refactor, write a test, then write the code
D. Write all tests for the project first, then all the code

1399. In unit testing, how does a mock differ from a stub (in the sense used by Meszaros and Fowler)? Understanding

A. A stub verifies calls; a mock only returns canned answers
B. A mock also verifies how it was called; a stub only returns canned answers
C. A mock is a real database; a stub is an in-memory copy
D. They are identical; the terms are interchangeable in every textbook

1400. After a bug fix, the team re-runs the existing test suite to make sure nothing that used to work has broken. What is this called? Understanding

A. Acceptance testing
B. Regression testing
C. Smoke testing
D. Load testing

1401. What distinguishes white-box testing from black-box testing? Understanding

A. White-box tests are designed from the code's internal structure; black-box tests only from the specification
B. White-box testing is done by users; black-box testing by developers
C. White-box tests are automated; black-box tests are always manual
D. White-box testing checks the user interface; black-box testing checks the database

1402. What is a profiler used for? Understanding

A. Finding syntax errors
B. Formatting code to a style guide
C. Generating unit tests automatically
D. Measuring where a program spends its time or memory

1403. Which statement about code coverage is correct? Understanding

A. 100% branch coverage implies 100% statement coverage, but not the other way round
B. 100% statement coverage implies 100% branch coverage
C. 100% coverage proves the code has no bugs
D. Branch coverage and statement coverage always give the same percentage

1404. In the Arrange-Act-Assert pattern for unit tests, what happens in the Act step? Understanding

A. Test data and objects are set up
B. The result is compared with the expected value
C. Test files are deleted
D. The code under test is called

1405. Which testing level checks that separately tested modules work correctly together? Understanding

A. Unit testing
B. Acceptance testing
C. Static analysis
D. Integration testing

1406. Donald Knuth warned that 'premature optimization is the root of all evil'. What does this mean in practice? Understanding

A. Never optimise code under any circumstances
B. Write clear, correct code first, then measure and optimise the parts that are actually slow
C. Always optimise every function as you write it
D. Optimisation should be left entirely to the compiler

1407. Memoisation stores a function's results and returns the stored value for repeated arguments. For which kind of function is it safe? Understanding

A. A function that reads the current time
B. A function that writes to a file each time it runs
C. A pure function, whose result depends only on its arguments and which has no side effects
D. A function that returns a random number

1408. A program crashes deep inside nested function calls. Which debugger view shows the chain of calls that led to the current line? Understanding

A. The watch window
B. The call stack
C. The breakpoints list
D. The console output

1409. A field accepts ages from 18 to 60 inclusive. Using boundary value analysis, which set of test inputs is most appropriate? Application

A. 18, 39, 60
B. 0, 100
C. 20, 30, 40, 50
D. 17, 18, 60, 61

1410. A function accepts an integer from 1 to 100. Using equivalence partitioning, what is the minimum number of test cases to cover every partition once? Application

A. 2
B. 3
C. 100
D. 4

1411. A function contains two if statements and one while loop, with no other decisions. What is its cyclomatic complexity? Application

A. 3
B. 2
C. 4
D. 5

1412. What is the minimum number of test cases needed for 100% branch coverage of this function? Application

```
def f(a, b):
    if a > 0:
        print("A")
    if b > 0:
        print("B")
```

A. 4
B. 1
C. 3
D. 2

1413. Which assert statement fails when this Python code runs? Application

```
def avg(xs):
    return sum(xs) / len(xs)

assert avg([2, 4]) == 3
assert avg([1, 2]) == 1
```

A. The second one: avg([1, 2]) is 1.5, not 1
B. The first one: avg([2, 4]) is 3.0, not 3
C. Neither assertion fails
D. Both fail, because / always returns a float

1414. Without memoisation, how many calls to fib (including the first) does fib(5) make? Application

```
def fib(n):
    if n < 2:
        return n
    return fib(n - 1) + fib(n - 2)
```

A. 5
B. 15
C. 6
D. 32

1415. A cProfile report shows: main (1 call, tottime 0.1 s, cumtime 10.0 s); parse (1 call, tottime 0.5 s, cumtime 8.0 s); lookup (1,000,000 calls, tottime 7.4 s, cumtime 7.5 s). Which function should you optimise first? Application

A. lookup, because it has by far the largest own time (tottime)
B. main, because its cumulative time is highest
C. parse, because it calls the slow code
D. None; 10 s is too short to matter

1416. Inside a loop over list a (n items), the code checks if x in b, where b is a list of m items. Converting b to a set first changes the average complexity to what? Application

A. From O(n + m) to O(n·m)
B. From O(n·m) to O(n log m)
C. From O(n·m) to O(n + m)
D. It stays O(n·m)

1417. What does this Python code print? Application

```
def find_max(xs):
    best = 0
    for x in xs:
        if x > best:
            best = x
    return best

print(find_max([-5, -2, -9]))
```

A. 0
B. -2
C. -5
D. -9

1418. What does this Python code print? Application

```
def total(xs):
    s = 0
    for i in range(1, len(xs)):
        s += xs[i]
    return s

print(total([4, 5, 6]))
```

A. 15
B. 11
C. 9
D. 10

1419. A loop is written as for i in range(n): if i < math.sqrt(limit): ..., and limit never changes inside the loop. Which optimisation applies? Application

A. Replace the for loop with recursion
B. Unroll the loop by hand
C. Use multiple threads for the square root
D. Compute math.sqrt(limit) once before the loop (hoist the loop-invariant computation)

1420. What does this Python code print? Application

```
import math
print(0.1 + 0.2 == 0.3, math.isclose(0.1 + 0.2, 0.3))
```

A. True True
B. False False
C. False True
D. True False

1421. A bug appeared somewhere in the last 1,000 commits. Using binary search (as git bisect does), what is the maximum number of commits you need to test? Application

A. 500
B. 1,000
C. 10
D. 100

1422. A function is_leap(y) returns y % 4 == 0. Which single test input exposes the bug? Application

A. 2024
B. 2023
C. 2000
D. 1900

1423. A bug disappears whenever a developer adds print statements or runs the program under a debugger. What kind of cause is most likely? Analysis

A. A timing-dependent bug, such as a race condition between threads
B. A syntax error that the debugger fixes automatically
C. A misspelt variable name
D. A wrong formula in a pure function

1424. The add function returns a - b by mistake, yet this test passes. Why? Analysis

```
def test_add():
    result = add(2, 2)
```

A. The test has no assertion, so it can only fail if an exception is raised
B. Test frameworks ignore return values of integers
C. 2 - 2 equals 2 + 2
D. pytest only runs tests whose names start with check_

1425. A test passes when run on its own but fails when the full suite runs. What is the most likely cause? Analysis

A. The test file is too long
B. The code under test has a syntax error
C. Unit tests must always be run one at a time
D. Tests share mutable state, such as a global or a database row, that another test changes

1426. Profiling shows 90% of run time is in function X. A developer makes X twice as fast. What is the overall speed-up? Analysis

A. 2
B. About 1.82
C. 1.9
D. 1.45

1427. A developer adds @lru_cache to speed up this function. What happens when it is called? Analysis

```
from functools import lru_cache

@lru_cache
def total(xs):
    return sum(xs)

print(total([1, 2, 3]))
```

A. It prints 6 and caches the result
B. It prints 6, but the cache is ignored
C. TypeError, because a list argument is unhashable and cannot be a cache key
D. It returns None because lru_cache needs a size

1428. A Java method builds a report with String s = ""; for (String line : lines) s += line; and becomes very slow for 100,000 lines. Why? Analysis

    A. The for-each loop is slower than a while loop
    B. Java limits strings to 65,535 characters
    C. The garbage collector stops the loop after every iteration
    D. Strings are immutable, so each += copies the whole string so far, giving $O(n^2)$ work

1429. This Python code crashes with TypeError: 'NoneType' object is not subscriptable. What is the root cause? Analysis

```
users = {1: {"name": "Ayesha"}}

def get_user(uid):
    if uid in users:
        return users[uid]

name = get_user(7)["name"]
```

    A. Dictionaries cannot use integer keys
    B. The variable name is reserved
    C. get_user has no return for a missing id, so it returns None, and None["name"] fails
    D. users must be a list, not a dict

1430. A sort function that does nothing at all passes the test assert my_sort([1, 2, 3]) == [1, 2, 3]. What is wrong with the test? Analysis

    A. Sorting can never be unit-tested
    B. The expected list should be in descending order
    C. Its input is already sorted, so it cannot tell a working sort from one that does nothing
    D. assert cannot compare lists

1431. This binary search hangs for a = [1, 3] and t = 3. Why? Analysis

```
def bsearch(a, t):
    lo, hi = 0, len(a) - 1
    while lo < hi:
        mid = (lo + hi) // 2
        if a[mid] < t:
            lo = mid
        else:
            hi = mid
    return lo
```

    A. The list must have an odd length
    B. When a[mid] < t, lo = mid can leave lo unchanged, so the range stops shrinking
    C. // performs floating-point division
    D. while lo < hi should be while lo > hi

1432. To check whether any two numbers in a list sum to a target, a student uses two nested loops. Which change improves the time complexity, and to what? Analysis

    A. Store seen numbers in a set and check target − x for each x: O(n) on average
    B. Swap the order of the two loops: still $O(n^2)$ but faster
    C. Use a while loop instead of for: O(log n)
    D. Sort the list after the nested loops: O(n log n)

1433. Execution is paused on a line that calls save_record(r). You trust save_record and want it to run completely, then pause on the next line of the current function. Which debugger command do you use? Understanding

A. Step into
B. Step out
C. Step over
D. Restart

1434. You are paused inside a helper function and have already seen what you need there. Which debugger command runs the rest of the helper and pauses again in the caller, just after the call? Understanding

A. Step over
B. Step out
C. Step into
D. Continue

1435. Which debugger feature keeps displaying the current value of an expression, such as total / count, every time execution pauses? Understanding

A. A conditional breakpoint
B. The call stack view
C. A watch (watch expression)
D. Step over

1436. A program reads a divisor from the user and crashes with ZeroDivisionError when the user enters 0. What kind of error is this? Understanding

A. Syntax error
B. Logic error
C. Compile-time type error
D. Runtime error

1437. Python prints tracebacks under the heading 'Traceback (most recent call last)'. Where is the line that actually raised the exception? Understanding

A. In the first frame listed, just below the heading
B. In the middle frame, since Python sorts frames by line number
C. Nowhere: Python only shows function names, not lines
D. In the last frame listed, just above the exception message

1438. A developer explains her code line by line to a toy duck on her desk and spots the bug halfway through. What is this technique called? Understanding

A. Pair programming
B. Delta debugging
C. Rubber duck debugging
D. Code coverage analysis

1439. Which kind of tool reports problems such as unused variables, unreachable code and possible null dereferences without running the program? Understanding

A. A profiler
B. A debugger
C. A unit test runner
D. A static analyser or linter, such as Pylint or SpotBugs

1440. According to the test pyramid, which kind of test should a project have the most of? Understanding

A. End-to-end UI tests
B. Unit tests
C. Manual exploratory tests
D. Integration tests

1441. A new build is deployed to the test server. Before deeper testing, the team runs a short set of checks that the app starts, logs in and opens its main screen. What is this called? Understanding

A. Regression testing
B. Stress testing
C. Smoke testing
D. Acceptance testing

1442. How does beta testing differ from alpha testing? Understanding

A. Beta testing is white-box testing by developers; alpha testing is black-box testing by users
B. Beta testing is done by real users in their own environment; alpha testing is done in-house, usually at the developer's site
C. Alpha testing happens after public release; beta testing happens before any code is written
D. Beta testing checks only performance; alpha testing checks only functionality

1443. In Python's unittest framework, what is the setUp method of a TestCase class for? Understanding

A. It runs once after all tests to delete temporary files
B. It registers which methods count as tests
C. It runs before each test method to prepare a fresh fixture, such as objects the tests need
D. It stores the expected output for every test

1444. A mutation-testing tool changes if x > 0 to if x >= 0 in the code, re-runs the test suite, and every test still passes. What does this result show? Understanding

A. The mutant survived: no test checks x = 0, so the suite is too weak to catch this change
B. The original code has a bug at x = 0
C. The suite has full branch coverage, so it is strong enough
D. Nothing about the tests, because mutation testing only measures speed

1445. Which technique feeds a program large numbers of random or malformed inputs to find crashes and security holes? Understanding

A. Boundary value analysis
B. Smoke testing
C. Code review
D. Fuzz testing (fuzzing)

1446. In Meszaros's test-double vocabulary, which double has a real working implementation that takes a shortcut, such as an in-memory database used in place of PostgreSQL? Understanding

A. Fake
B. Stub
C. Mock
D. Dummy

1447. In software testing, what is a test oracle? Understanding

A. A tool that generates random test inputs
B. The source used to decide the expected result of a test, such as a specification, a reference implementation or a hand calculation
C. The person who writes the test cases
D. A database of bugs found in earlier releases

1448. Which coverage criterion is the strongest, in the sense that fully satisfying it also satisfies the others listed? Understanding

A. Path coverage
B. Statement coverage
C. Branch coverage
D. Function coverage

1449. Which standard Python logging level is meant for detailed diagnostic messages that are usually switched off in production? Understanding

A. INFO
B. DEBUG
C. WARNING
D. CRITICAL

1450. Why should Python code not rely on assert statements to validate user input? Understanding

A. assert only works inside unit test files
B. Running Python with the -O flag removes assert statements, so the check silently disappears
C. assert raises SyntaxError instead of a catchable exception
D. assert is evaluated only once per program run

1451. How does a sampling profiler differ from an instrumenting (deterministic) profiler such as Python's cProfile? Understanding

A. It records the call stack at regular intervals, giving statistical results with low overhead
B. It counts every function call exactly, at the cost of higher overhead
C. It measures only memory use, never time
D. It requires the program to be rewritten with timing statements

1452. In C++, what is a memory leak? Understanding

A. Heap memory allocated with new that is never released with delete after it is no longer needed
B. Reading an array element past its end
C. Freeing the same block of memory twice
D. Using a local variable before giving it a value

1453. A C program sometimes reads past the end of a malloc'd buffer. Which tool is designed to detect such invalid memory accesses while the program runs? Understanding

A. gprof
B. Valgrind (Memcheck) or AddressSanitizer
C. clang-format
D. git bisect

1454. In bottom-up integration testing, a low-level module is tested before the modules that call it have been written. What temporary code calls it during the test? Understanding

A. A stub
B. A fake database
C. A teardown method
D. A driver

1455. Why does sum(x * x for x in range(10**7)) use far less memory than sum([x * x for x in range(10**7)])? Understanding

A. Generator expressions are compiled to machine code
B. The list version computes every square twice
C. sum skips every other element of a generator
D. The generator expression produces one value at a time instead of building a 10-million-item list first

1456. How many log lines does this Python code print? Application

```
import logging
logging.basicConfig(level=logging.WARNING, format="%(levelname)s: %(message)s")
logging.debug("a")
logging.info("b")
logging.warning("c")
logging.error("d")
```

A. 4
B. 1
C. 3
D. 2

1457. What does this Python code print? Application

```
def add_item(item, basket=[]):
    basket.append(item)
    return basket

print(add_item("pen"), add_item("ink"))
```

A. ['pen'] ['ink']
B. ['pen'] ['pen', 'ink']
C. ['pen', 'ink'] ['pen', 'ink']
D. TypeError: list is not a valid default

1458. What does this Python code print? Application

```
funcs = [lambda: i * 10 for i in range(3)]
print([f() for f in funcs])
```

A. [0, 10, 20]
B. [0, 0, 0]
C. [20, 20, 20]
D. NameError: name 'i' is not defined

1459. A student wants to remove every 2 from the list. What does this code print? Application

```
nums = [1, 2, 2, 3]
for n in nums:
    if n == 2:
        nums.remove(n)
print(nums)
```

A. [1, 3]
B. [1, 2, 3]
C. [1, 2, 2, 3]
D. RuntimeError: list changed size during iteration

1460. What does this JavaScript code print? Application

```
for (var i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 0);
}
```

A. 0, 1 and 2
B. 0, three times
C. ReferenceError: i is not defined
D. 3, three times

1461. What does this JavaScript code print? Application

```
console.log(0 == "0", 0 === "0");
```

A. false true
B. true false
C. true true
D. false false

1462. What does this Python code print? Application

```
grid = [[0] * 3] * 2
grid[0][0] = 1
print(grid)
```

A. [[1, 0, 0], [0, 0, 0]]
B. [[1, 0, 0], [1, 0, 0]]
C. [[1, 1, 1], [0, 0, 0]]
D. [[1, 0, 0]]

1463. What does this Java code print? Application

```
int total = 7, count = 2;
double avg = total / count;
System.out.println(avg);
```

A. 3.0
B. 3.5
C. 3
D. 4.0

1464. Marks are valid from 0 to 100. Marks 0 to 49 mean 'fail' and 50 to 100 mean 'pass'. Using equivalence partitioning, how many partitions should the test cases cover, counting invalid ones? Application

A. 4
B. 2
C. 3
D. 6

1465. What is the minimum number of test cases needed for 100% statement coverage of this function? Application

```
def label(m):
    result = "fail"
    if m >= 50:
        result = "pass"
    return result
```

A. 2
B. 3
C. 1
D. 0

1466. A program has two functions. Their control-flow graphs together have 14 edges, 13 nodes and 2 connected components. Using McCabe's formula V(G) = E - N + 2P, what is the total cyclomatic complexity? Application

A. 3
B. 1
C. 7
D. 5

1467. Profiling shows that one function takes 40% of a program's run time. A developer makes that function 4 times faster and changes nothing else. What is the overall speed-up? Application

A. About 1.43
B. 4
C. 1.6
D. About 1.3

1468. An O(n^2) sorting routine takes 2 seconds for n = 1,000. Roughly how long should it take for n = 4,000? Application

A. 32 seconds
B. 8 seconds
C. 16 seconds
D. 64 seconds

1469. What does this Python code print? Application

```
from functools import lru_cache
calls = 0

@lru_cache(maxsize=None)
def fib(n):
    global calls
    calls += 1
    return n if n < 2 else fib(n - 1) + fib(n - 2)

fib(30)
print(calls)
```

A. 30
B. 2692537
C. 60
D. 31

1470. What happens when this Python code runs? Application

```
scores = {"ali": 80, "sara": 91}
print(scores["Ali"])
```

A. It raises KeyError: 'Ali'
B. It prints 80
C. It prints None
D. It raises NameError: name 'Ali' is not defined

1471. A fence is 20 m long, with a post every 2 m including one at each end. A student's program prints 10 posts. What is the correct count, and what is this kind of bug called? Application

A. 10, so there is no bug
B. 9, an overflow error
C. 11, an off-by-one (fencepost) error
D. 12, a rounding error

1472. Which unittest code correctly checks that int("abc") raises ValueError? Application

A. self.assertRaises(ValueError, int("abc"))
B. self.assertEqual(int("abc"), ValueError)
C. with self.assertRaises(ValueError): int("abc")
D. self.assertTrue(int("abc") == ValueError)

1473. What does this Python code print? Application

```
from unittest.mock import Mock
send = Mock(return_value=True)
send("a@x.pk")
send("b@x.pk")
print(send.call_count, send.call_args)
```

A. 2 call('a@x.pk')
B. 1 call('b@x.pk')
C. 2 call('b@x.pk')
D. True True

1474. What does this JavaScript code print? Application

```
const marks = [100, 25, 9];
marks.sort();
console.log(marks);
```

A. [9, 25, 100]
B. [100, 9, 25]
C. [100, 25, 9]
D. [25, 9, 100]

1475. A test expects round(2.5) to be 3. What does this Python 3 code print? Application

```
print(round(0.5), round(1.5), round(2.5))
```

A. 1 2 3
B. 0 1 2
C. 0 2 2
D. 1 2 2

1476. This function is too slow for many queries. It is optimised by building prefix sums p with p[0] = 0 and p[k] = a[0] + ... + a[k - 1]. Which expression then gives each query's answer in O(1)? Application

```
def range_sums(a, queries):
    return [sum(a[l:r + 1]) for l, r in queries]
```

A. p[r] - p[l]
B. p[r] - p[l - 1]
C. p[r + 1] - p[l]
D. p[r + 1] - p[l + 1]

1477. This code later crashes with NameError: name 'total' is not defined at the print line. Why is the real bug hard to find? Analysis

```
try:
    total = compute_total(order)
except Exception:
    pass
print(total)
```

A. The except block silently swallows the original exception, so its message and traceback are lost
B. print cannot display variables assigned inside a try block
C. compute_total must be called outside any try block
D. NameError always hides the line number of the bug

1478. For a 100,000-character string s, this C loop is extremely slow. Why? Analysis

```
int count = 0;
for (int i = 0; i < strlen(s); i++) {
    if (s[i] == ' ') count++;
}
```

A. strlen walks the whole string and is re-evaluated in the loop condition on every iteration, making the loop O(n^2)
B. Indexing s[i] is O(n) in C, because arrays are linked lists
C. count++ forces the string to be copied on each iteration
D. Comparing a char with ' ' converts the whole string to an int

1479. This Java code prints a negative total. Why? Analysis

```
int[] sales = {1_000_000_000, 1_000_000_000, 1_000_000_000};
int total = 0;
for (int s : sales) total += s;
System.out.println(total);
```

A. One of the array elements is stored as a negative number
B. The for-each loop skips the first element
C. println displays numbers above one billion in two's complement
D. The true sum, 3,000,000,000, exceeds the int maximum of 2,147,483,647, so it overflows and wraps around; use long

1480. This test passes when the team runs it at 10 am but fails in the nightly CI run at 11 pm. What is the root cause, and the best fix? Analysis

```
from datetime import datetime

def greeting():
    return "Good morning" if datetime.now().hour < 12 else "Good evening"

def test_greeting():
    assert greeting() == "Good morning"
```

A. The CI server is slower, so the test should be retried until it passes
B. The result depends on the real clock; pass the time in as a parameter (or mock it) so the test controls it
C. assert must be replaced by assertEqual for strings
D. datetime.now() raises an error on CI machines

1481. A developer spends a week making parse_config() ten times faster, but the program's total run time barely changes. Profiling shows parse_config() took 2% of the run time. What is the lesson? Analysis

    A. The speed-up was measured wrongly, since a 10x faster function must make the program 10x faster
    B. Compiler flags would have made parse_config() 100x faster
    C. No optimisation can help because all programs are I/O-bound
    D. Profile first and optimise what dominates: speeding up a 2% part can save at most 2%

1482. This test passes and gives 100% statement and branch coverage of ratio, yet users report crashes. What is the best explanation? Analysis

```
def ratio(a, b):
    return a / b

def test_ratio():
    assert ratio(6, 3) == 2
```

    A. ratio(6, 3) returns 2.0, so the assertion actually fails
    B. Coverage tools cannot measure one-line functions
    C. Division must use // for the test to be valid
    D. Coverage only shows that each line ran, not that all inputs were tested; no test checks b = 0

1483. billing.compute_total has a bug, but this test always passes. Why? Analysis

```
from unittest.mock import patch
import billing

def test_total():
    with patch("billing.compute_total", return_value=150):
        assert billing.compute_total([100, 50]) == 150
```

    A. The test patches the very function it should test, so it only checks the mock's return value
    B. patch works only with pytest, so the assertion is skipped
    C. 100 + 50 is 150, so compute_total must be correct
    D. return_value must be a list for patch to take effect

1484. A long-running Python web server uses this function. Memory use grows for days until the process is killed. Python has a garbage collector, so why does memory keep growing? Analysis

```
_cache = {}

def render(page_id, user_id):
    key = (page_id, user_id)
    if key not in _cache:
        _cache[key] = build_html(page_id, user_id)
    return _cache[key]
```

    A. The garbage collector does not run on dictionaries
    B. Tuples used as keys are never freed by Python
    C. build_html must call del on its result before returning
    D. The module-level dict keeps a reference to every page ever rendered, so the garbage collector can never free them; bound or evict the cache

1485. Why does this loop never end? Analysis

```
x = 0.0
while x != 1.0:
    x += 0.1
print("done")
```

A. Python does not allow != between floats
B. 0.1 cannot be stored exactly in binary, so after ten additions x is 0.9999999999999999 and never equals 1.0
C. x += 0.1 creates a new local variable each time, so x stays 0.0
D. A while loop needs an explicit break to stop

1486. is_sorted([1, 2, 3]) raises IndexError. Why? Analysis

```
def is_sorted(a):
    for i in range(len(a)):
        if a[i] > a[i + 1]:
            return False
    return True
```

A. range(len(a)) starts at 1, skipping a[0]
B. Lists cannot be compared with >
C. return True must be inside the loop
D. On the last iteration i = len(a) - 1, so a[i + 1] is past the end; loop over range(len(a) - 1)

1487. Converted prices are right in the morning but wrong by the afternoon, until the server is restarted. What is the cause? Analysis

```
from functools import lru_cache

@lru_cache(maxsize=None)
def get_rate(currency):
    return fetch_rate_from_bank(currency)
```

A. The rate changes over time, but the cache returns the first value fetched for each currency forever
B. lru_cache cannot cache string arguments
C. maxsize=None turns caching off
D. fetch_rate_from_bank is too slow to finish before noon

1488. With 200,000 items in data, this loop takes a very long time. Why, and what is the fix? Analysis

```
result = []
for x in data:
    result.insert(0, x)
```

A. Python lists have a maximum size near 200,000
B. insert(0, x) shifts every existing element, so the loop is O(n^2); append and reverse once, or use collections.deque.appendleft
C. insert copies x every time, so memory runs out
D. Iterating over data with for is itself O(n^2)

1489. The user types yes, but Saved never prints. Why? Analysis

```
Scanner in = new Scanner(System.in);
String answer = in.nextLine();
if (answer == "yes") System.out.println("Saved");
```

A. nextLine keeps the trailing newline, so answer is "yes\n"
B. == compares object references, and the string read from input is a different object from the literal; use "yes".equals(answer)
C. Strings in Java can only be compared with compareTo
D. "yes" is a char array, not a String

1490. What happens when this code runs, and why? Analysis

```
def fact(n):
    if n == 0:
        return 1
    return n * fact(n - 1)

print(fact(-1))
```

A. It prints 1, because the base case is reached immediately
B. RecursionError, because n goes -1, -2, -3 and never reaches the base case 0
C. It prints -1
D. It prints 0, because negative factorials are zero

1491. DEBUG messages are switched off, yet profiling shows expensive_dump() taking 40% of the run time. Why? Analysis

```
import logging
logging.basicConfig(level=logging.INFO)

for order in orders:
    logging.debug(f"order dump: {expensive_dump(order)}")
```

A. The f-string is built before logging.debug is called, so expensive_dump runs every time even though the message is discarded; guard it with logging.getLogger().isEnabledFor(logging.DEBUG)
B. basicConfig ignores the level argument, so DEBUG messages are still written
C. Discarded log messages are saved to a hidden file
D. The profiler counts calls that were never executed

Software Development Practices MCQs for the NSCT

1492. Which Git command moves changes from the working directory into the staging area (the index)? Understanding

A. git commit
B. git push
C. git add
D. git init

1493. What is the main purpose of a version control system such as Git? Understanding

A. To record the history of changes so you can compare, restore and merge versions of code
B. To compile code faster
C. To find bugs automatically
D. To host websites

1494. By default, what does git pull do? Understanding

A. Fetches new commits from the remote and merges them into the current branch
B. Uploads local commits to the remote
C. Only downloads commits without changing any local branch
D. Deletes local changes and copies the remote branch

1495. How does git revert differ from git reset when undoing a commit? Understanding

A. revert adds a new commit that undoes the change; reset moves the branch pointer back, removing commits from the branch
B. revert deletes the commit permanently; reset keeps it
C. They are identical commands with different names
D. revert works only on unpushed commits; reset works only on pushed ones

1496. What does the DRY principle stand for? Understanding

A. Do Repeat Yourself
B. Debug, Refactor, Yield
C. Document Rarely, Yearly
D. Don't Repeat Yourself

1497. What does the YAGNI principle advise? Understanding

A. Always add extension points for possible future needs
B. Write all documentation before any code
C. Never delete unused code
D. Don't build features until they are actually needed

1498. According to PEP 8, how should a Python function that calculates total marks be named? Understanding

A. CalculateTotalMarks
B. calculate_total_marks
C. calculateTotalMarks
D. CALCULATE_TOTAL_MARKS

1499. Which name follows standard Java naming conventions for a constant? Understanding

A. maxStudents
B. MaxStudents
C. max_students
D. MAX_STUDENTS

1500. In clean-code terms, what is a 'magic number'? Understanding

A. A random number produced by a secure generator
B. An unexplained literal value in code, such as 86400, whose meaning is not obvious
C. A number that the compiler optimises away
D. A constant declared with a descriptive name

1501. What is a Python docstring? Understanding

A. Any comment that starts with #
B. A separate .txt file stored next to the module
C. A string literal written as the first statement of a module, class or function, available through __doc__ and help()
D. A string that the interpreter executes as code

1502. According to common clean-code advice, what should a good code comment mainly explain? Understanding

A. What every line does, restating the code in English
B. Why the code does something, such as a business rule or a non-obvious decision
C. The author's name and the date of every change
D. Code that has been commented out for later

1503. What is the main purpose of a code review? Understanding

A. To measure which developer writes the most lines
B. To replace automated testing
C. To find defects and design problems early and to share knowledge across the team
D. To let the most senior developer rewrite the code

1504. A library at version 2.4.1 (semantic versioning) makes a change that breaks backward compatibility. What should the next version be? Understanding

A. 2.5.0
B. 2.4.2
C. 3.0.0
D. 3.4.1

1505. Why is it usually advised not to rebase a branch that others have already pulled? Understanding

A. Rebase deletes all files in the branch
B. Rebased branches cannot be merged ever again
C. Rebase only works on the main branch
D. Rebasing rewrites commits with new hashes, so other people's copies no longer match the shared history

1506. In Git, what does HEAD normally refer to? Understanding

A. The first commit ever made in the repository
B. The commit currently checked out, usually through the current branch
C. The latest commit on the remote server
D. The file at the top of the project folder

1507. What is continuous integration (CI)? Understanding

A. Developers merge to the shared mainline often, and every merge is automatically built and tested
B. Developers integrate their work only once, at the end of the project
C. Code is deployed to customers without any tests
D. Two developers write code on one computer at the same time

1508. Which single command creates a new branch called feature and switches to it? Application

A. git switch -c feature
B. git branch feature
C. git merge feature
D. git commit -b feature

1509. You run these commands in a new repository. What content of a.txt is stored in the commit? Application

```
echo v1 > a.txt
git add a.txt
echo v2 > a.txt
git commit -m "first"
```

A. v2, because commit always uses the latest file on disk
B. Both v1 and v2 as separate commits
C. Nothing, because the file changed after git add
D. v1, because commit records the staged snapshot, not the later edit

1510. In a repository with one commit, you create a new file notes.txt and run git status --short. What does it show for that file? Application

A. A notes.txt
B. M notes.txt
C. D notes.txt
D. ?? notes.txt

1511. Starting with commit A on main, you create branch feature and commit B on it, switch back to main and commit C, then run git merge feature (which creates a merge commit). How many commits does git log --oneline show on main? Application

A. 3
B. 4
C. 2
D. 5

1512. main is at commit A. Branch feature starts at A and adds commits B and C. Nothing new is on main. What happens when you run git merge feature on main? Application

A. A merge commit is always created
B. A conflict, because both branches contain A
C. B and C are copied as new commits with new hashes
D. A fast-forward: main simply moves to C and no merge commit is created

1513. main changes the line in c.txt to color = green, and feature changes the same line to color = blue. After git merge feature on main reports a conflict, what does c.txt contain? Application

A. Only color = green, because the current branch wins
B. Only color = blue, because the merged branch wins
C. The original line, because Git cancels the merge
D. The current branch's line between <<<<<<< HEAD and =======, then the feature line before >>>>>>> feature

1514. A branch has three commits. You run git revert HEAD. How many commits are now in git log? Application

A. 2
B. 3
C. 1
D. 4

1515. You have two commits and run git reset --soft HEAD~1. What is the result? Application

A. The last commit is removed from the branch, but its changes stay staged
B. The last commit and its changes are deleted completely
C. The changes stay in the files but are unstaged
D. A new commit is created that undoes the last one

1516. A variable holds the number of days since a book was borrowed. Which name is clearest? Application

A. d
B. days_since_borrowed
C. data1
D. x_days_tmp

1517. The same 3-line tax calculation is copied into five functions. Which refactoring best applies? Application

A. Add a comment above each copy explaining the formula
B. Move all five functions into a single file
C. Extract the calculation into one function and call it from all five places
D. Rename the variables in each copy so they look different

1518. A class Report has methods calculate_totals(), print_pdf() and save_to_database(). Which principle does it break, and what is the usual fix? Application

A. Single Responsibility Principle; split it into classes for calculation, formatting and storage
B. DRY; copy the methods into each caller
C. YAGNI; delete print_pdf()
D. Open/Closed Principle; make every method private

1519. What does this Python code print? Application

```
def area(r):
    """Return the area of a circle of radius r."""
    return 3.14159 * r * r

print(area.__doc__)
```

A. Return the area of a circle of radius r.
B. area
C. None
D. 3.14159 * r * r

1520. A library at version 1.9.3 adds a new feature without breaking existing code. What is the next semantic version? Application

A. 2.0.0
B. 1.9.4
C. 1.10.0
D. 1.10.3

1521. You have an uncommitted edit to a tracked file and need a clean working tree to switch branches. After running git stash, what does git status --short show? Application

A. M file, staged
B. Nothing: the working tree is clean
C. ?? file
D. The edit plus a stash warning

1522. A 300-line function reads a file, validates each record, calculates totals and writes a report. What is the main problem, and the standard refactoring? Analysis

A. Magic numbers; replace them with constants
B. Long Method smell; extract each step into its own well-named function
C. Dead code; delete the function
D. Too many comments; remove them

1523. A developer rebases the shared develop branch and force-pushes it. Teammates who pull next see duplicate commits and confusing conflicts. What went wrong? Analysis

    A. Force-push deletes the remote repository
    B. Rebase cannot be used with remote branches at all
    C. The rebase replaced published commits with new ones, so teammates' local history diverged from the remote
    D. The teammates forgot to run git init

1524. A developer commits an API key in config.py, notices, and deletes it in the next commit before pushing both. Why is the key still exposed? Analysis

    A. Deleting a line never takes effect until the next release
    B. Git encrypts deleted lines but keeps them readable
    C. The earlier commit still contains the key in history; rotate the key and rewrite history before pushing
    D. It is not exposed, because the latest commit no longer has it

1525. What is the problem in this code? Analysis

```
# apply the 10% student discount
price = price * 0.85
```

    A. The comment says 10% but the code applies 15%; one of them is wrong and must be corrected
    B. Comments are not allowed above assignments
    C. 0.85 should be written as 85
    D. There is no problem; comments are ignored

1526. In a code review you see the call process(data, True, False, True). What is the most useful review comment? Analysis

    A. The boolean arguments are unreadable; use named arguments or separate, clearly named functions
    B. Rename data to d to save space
    C. Add more boolean parameters for future options
    D. Approve it; style issues do not matter in reviews

1527. On main you edit file b; on feature you edit file a. Both branches started from the same commit. Why does git merge feature complete without any conflict? Analysis

    A. Git always prefers the current branch, so conflicts never happen
    B. A fast-forward merge was performed
    C. The branches changed different files, so Git can combine both sets of changes automatically
    D. Git skipped feature's changes

1528. You check out an old commit by its hash, make a new commit there, and then run git switch main. What happens to the new commit? Analysis

    A. It is automatically added to main
    B. It belongs to no branch; it can be recovered through git reflog until Git garbage-collects it
    C. Git refuses to make commits in that state
    D. It is pushed to the remote automatically

1529. The status value 3 (meaning 'shipped') is hard-coded in five if statements. A new rule renumbers statuses. What is the main risk, and the fix? Analysis

A. The program will run faster with literals
B. One of the five copies may be missed; define a named constant or enum and use it everywhere
C. There is no risk, because the compiler updates literals automatically
D. if statements cannot compare integers

1530. A bug fix commit on feature is cherry-picked onto main, which has moved on since feature branched off. What is true afterwards? Analysis

A. main and feature now share the identical commit object
B. The commit is moved: it disappears from feature
C. All of feature's commits are merged into main
D. main gets a new commit with the same changes but a different hash

1531. config.env was committed, then added to .gitignore. After you edit config.env, git status still shows it as modified. Why? Analysis

A. .gitignore needs a restart of Git to take effect
B. The pattern in .gitignore must start with a slash
C. .gitignore only affects untracked files; run git rm --cached config.env to stop tracking it
D. Git ignores .gitignore when the file name contains a dot

1532. What does running git init in a folder do? Understanding

A. Creates a new, empty Git repository there by adding a hidden .git directory
B. Downloads a copy of a remote repository into the folder
C. Commits every file in the folder
D. Connects the folder to a GitHub account

1533. Git is called a distributed version control system. What does that mean? Understanding

A. The files of one project are split across several servers
B. Only one developer can edit a file at a time, using locks
C. The history is stored only on a central server
D. Every clone holds the full history, so you can commit, branch and view the log without contacting a server

1534. What does git fetch do? Understanding

A. Downloads new commits and updates remote-tracking branches such as origin/main, without changing your current branch or working files
B. Downloads new commits and merges them into your current branch
C. Uploads your local commits to the remote
D. Discards your local commits so your branch matches the remote

1535. With no arguments, what does git diff compare? Understanding

A. The staging area against the last commit
B. The current branch against main
C. The working directory against the staging area, showing changes not yet staged
D. Your local branch against the remote branch

1536. What does git commit --amend do? Understanding

A. Adds a new commit after the last one
B. Edits any older commit in place while keeping its hash
C. Replaces the last commit with a new one (new hash) that includes anything now staged and, optionally, a new message
D. Undoes the last commit and throws its changes away

1537. After git clone, what does the name origin refer to? Understanding

A. The first commit in the repository
B. The main branch
C. The default name for the remote repository you cloned from
D. The local working directory

1538. On GitHub or GitLab, what is a pull request (merge request)? Understanding

A. A command that downloads changes from the remote
B. A request to merge changes from one branch into another, where teammates review and discuss the change and CI runs before merging
C. A bug report filed by a user
D. A request to be given write access to a repository

1539. What does the KISS principle advise? Understanding

A. Keep it simple: prefer the simplest design that solves the problem
B. Keep interfaces small and split by client
C. Keep every line documented with a comment
D. Keep inheritance hierarchies deep to maximise reuse

1540. What does the Open/Closed Principle state? Understanding

A. Software entities should be open for extension but closed for modification, so new behaviour is added without editing tested code
B. Every class should be declared public
C. Files should be closed as soon as they have been read
D. Methods should accept any type of input

1541. What is refactoring? Understanding

A. Adding new features to existing code
B. Changing the internal structure of code to make it easier to understand and change, without changing its external behaviour
C. Rewriting code to fix a bug
D. Translating code into another programming language

1542. Which Javadoc tag documents an exception that a method may throw? Understanding

A. @throws
B. @return
C. @param
D. @see

1543. What is the most essential content of a project's README file? Understanding

A. The full source code of the main module
B. What the project does and how to install, run and use it
C. The complete commit history
D. Each developer's personal to-do notes

1544. Which combination do good modular designs aim for? Understanding

A. Low cohesion and high coupling
B. High cohesion and high coupling
C. High cohesion and low coupling
D. Low cohesion and low coupling

1545. How does continuous delivery differ from continuous deployment? Understanding

A. In continuous delivery every passing change is ready to release but a person decides when it goes to production; in continuous deployment it goes to production automatically
B. In continuous deployment a person approves each release; in continuous delivery releases are automatic
C. Continuous delivery only compiles the code; continuous deployment also runs tests
D. Continuous deployment means merging branches once a week

1546. In pair programming, what are the two roles? Understanding

A. The tester writes tests; the developer writes code on another machine
B. The author writes code; the approver merges it a day later
C. One person writes the front end and the other the back end
D. The driver types the code; the navigator reviews each line and thinks about direction and problems ahead

1547. What is the difference between a formatter, such as Black or Prettier, and a linter, such as Pylint or ESLint? Understanding

A. A linter rewrites code layout; a formatter reports bugs
B. Both compile the code, but a linter also runs it
C. A formatter runs the unit tests; a linter measures coverage
D. A formatter rewrites code layout automatically to one consistent style; a linter analyses code and reports likely bugs and rule violations

1548. What does the term 'technical debt' mean? Understanding

A. Money a company owes for software licences
B. The number of open bug reports
C. The future extra work caused by choosing a quick, easy solution now instead of a better one
D. The monthly cost of the servers that run the software

1549. What is meant by an 'atomic' commit? Understanding

A. A commit containing one complete logical change, so it can be reviewed, reverted or cherry-picked on its own
B. A commit that can never be reverted
C. A commit that contains all of a day's work
D. A commit that changes exactly one file

1550. HEAD is a merge commit created by running git merge feature while on main. What do HEAD~1 and HEAD^2 refer to? Understanding

A. Both refer to the same commit
B. HEAD~1 is feature's tip; HEAD^2 is main's previous tip
C. HEAD~1 is the first parent (main's previous tip); HEAD^2 is the second parent (feature's tip)
D. HEAD~1 is the parent; HEAD^2 is the grandparent along main

1551. Under Semantic Versioning 2.0.0, what does a version such as 0.7.3 signal? Understanding

A. A pre-release of version 7.3
B. A version that may only receive patch fixes
C. Initial development: anything may change at any time, and the public API should not be considered stable
D. A version guaranteed to be compatible with 1.0.0

1552. In package.json, a dependency is declared as "^1.4.2". Which versions may npm install? Understanding

A. Exactly 1.4.2 and nothing else
B. From 1.4.2 up to, but not including, 1.5.0
C. Any version from 1.4.2 up to, but not including, 2.0.0
D. Any version from 1.4.2 upward, including 2.x

1553. What is the main purpose of a lock file such as package-lock.json or poetry.lock? Understanding

A. To stop other developers from editing package.json
B. To encrypt the downloaded dependencies
C. To lock the repository while a merge is in progress
D. To record the exact version of every dependency, including indirect ones, so every install reproduces the same set

1554. How does an annotated Git tag (git tag -a v1.1 -m "release") differ from a lightweight tag (git tag v1.0)? Understanding

A. A lightweight tag cannot be pushed to a remote
B. An annotated tag moves forward with new commits, like a branch
C. Annotated tags can only point at merge commits
D. An annotated tag is a full Git object that stores the tagger, date and message; a lightweight tag is just a name pointing at a commit

1555. A repository has commits A and B, where B modified a.txt. You run git reset HEAD~1 (the default --mixed mode). What does git status --short show? Application

A. M a.txt (modified, not staged)
B. M a.txt (modified and staged)
C. Nothing: the working tree is clean
D. ?? a.txt

1556. In a repository whose only tracked file is a.txt, you edit a.txt, create a new file new.txt, and run git commit -am "update". What does git status --short show afterwards? Application

A. Nothing: both files were committed
B. M a.txt and ?? new.txt
C. A new.txt
D. ?? new.txt

1557. You edit the tracked file a.txt, run git add a.txt, then edit a.txt again. What does git status --short show? Application

A. M a.txt
B. M a.txt
C. AM a.txt
D. MM a.txt

1558. On main, you run git branch -d feature, but feature has a commit that was never merged into main. What happens? Application

A. Git refuses with 'not fully merged'; git branch -D would force the deletion
B. The branch and its commit are deleted at once
C. Git merges feature into main and then deletes it
D. The branch is deleted and its commit is moved onto main

1559. main has commits A then C. Branch feature started from A and has commits B then D. On feature you run git rebase main, and there are no conflicts. How many commits does git log on feature show, and how many of them are merge commits? Application

A. 5 commits, 1 merge commit
B. 4 commits, 0 merge commits
C. 3 commits, 0 merge commits
D. 4 commits, 1 merge commit

1560. You run git stash twice, saving two different sets of changes. Then you run git stash apply. How many entries does git stash list show now? Application

A. 1
B. 2
C. 0
D. 3

1561. A branch has two commits. You stage a forgotten file and run git commit --amend --no-edit. How many commits does git log --oneline show? Application

A. 3
B. 1
C. 4
D. 2

1562. main is at commit A. Branch feature adds F1, F2 and F3. On main you run git merge --squash feature, then git commit -m "S". Which statement is true? Application

A. main has 5 commits, including F1, F2 and F3
B. main has 2 commits (A and S) and no merge commit, and git branch -d feature refuses because feature is not recorded as merged
C. main has a merge commit S with two parents
D. main fast-forwards to F3 and S is empty

1563. A new repository's .gitignore has two lines: *.log and !important.log. The folder also holds a.log, important.log and b.txt. Apart from .gitignore itself, which files does git status --short list as untracked? Application

A. Only b.txt
B. important.log and b.txt
C. a.log, important.log and b.txt
D. a.log and b.txt

1564. What does git clean -n do? Application

A. Deletes all untracked files immediately
B. Lists the untracked files that git clean would delete, without deleting anything
C. Deletes only the files matched by .gitignore
D. Discards uncommitted changes to tracked files

1565. A tracked file contains one committed line. You append the line b and run git add on the file, then append the line c and do not stage it. Which command's output shows only the added line c? Application

A. git diff
B. git diff --staged
C. git diff HEAD
D. git show

1566. A library at version 2.0.9 releases a backward-compatible bug fix. What is the next version under semantic versioning? Application

A. 2.1.0
B. 2.0.10
C. 2.0.9.1
D. 3.0.0

1567. Which list puts these versions in increasing order under Semantic Versioning: 1.10.0, 1.2.0, 1.9.1 and 1.10.0-beta? Application

A. 1.10.0, 1.10.0-beta, 1.2.0, 1.9.1
B. 1.2.0, 1.9.1, 1.10.0-beta, 1.10.0
C. 1.2.0, 1.9.1, 1.10.0, 1.10.0-beta
D. 1.10.0-beta, 1.10.0, 1.2.0, 1.9.1

1568. Following standard Java naming conventions, how should a class that stores one student's exam record be named? Application

A. studentExamRecord
B. StudentExamRecord
C. student_exam_record
D. STUDENT_EXAM_RECORD

1569. What does this Python code print? Application

```
def volume(r):
    pi = 3.14159
    """Return the volume of a sphere of radius r."""
    return 4 / 3 * pi * r ** 3

print(volume.__doc__)
```

A. Return the volume of a sphere of radius r.
B. pi = 3.14159
C. None
D. AttributeError: function has no attribute __doc__

1570. What does this Python code print? Application

```
def double(x: int) -> int:
    return x * 2

print(double("ab"))
```

A. TypeError: expected int
B. abab
C. 0
D. ab2

1571. After a failure report, what is the last line this Python code prints? Application

```
def square(n):
    """Return n squared.

    >>> square(3)
    9
    >>> square(-2)
    -4
    """
    return n * n

import doctest
print(doctest.testmod())
```

A. TestResults(failed=0, attempted=2)
B. TestResults(failed=1, attempted=2)
C. TestResults(failed=2, attempted=2)
D. TestResults(failed=1, attempted=1)

1572. Which guard-clause version returns the same result as this function for every input? Application

```
def fee(age, member):
    if age >= 18:
        if member:
            return 500
        else:
            return 800
    else:
        return 300
```

A. if member: return 500; if age < 18: return 300; return 800
B. if age <= 18: return 300; if member: return 500; return 800
C. if age < 18: return 300; if not member: return 500; return 800
D. if age < 18: return 300; if member: return 500; return 800

1573. A branch has commits A, B, C and D. You run git reset --hard HEAD~3 by mistake, then run git reset --hard HEAD@{1}. How many commits does git log show now? Application

A. 1
B. 3
C. 0
D. 4

1574. You edit the tracked file a.txt, run git add a.txt, then run git restore --staged a.txt. What does git status --short show? Application

A. Nothing: the change is discarded
B. M a.txt
C. M a.txt
D. ?? a.txt

1575. git merge feature stops with a conflict in c.txt. What is the correct way to finish the merge? Application

A. Run git commit straight away; Git keeps both versions automatically
B. Edit c.txt to the wanted content and remove the conflict markers, run git add c.txt, then git commit
C. Run git push first, so the remote resolves the conflict
D. Delete c.txt and run git merge feature again

1576. You commit on main and run git push, but it is rejected with '! [rejected] main -> main (fetch first)'. A teammate pushed first. What is the right next step? Analysis
    A. git push --force, so your version replaces the remote
    B. Delete the local repository and clone it again
    C. git reset --hard origin/main, then push
    D. git pull (fetch, then merge or rebase your commit onto theirs), resolve any conflicts, then push again

1577. A pull request leaves 40 lines of old code commented out 'in case we need it later'. What is the best review comment? Analysis
    A. Keep it, but add today's date above it
    B. Move it into a separate file called old_code.py
    C. Turn it into a docstring so it is documented
    D. Delete it: Git history already keeps the old version if it is ever needed

1578. A function has five nested if levels, each checking a precondition (user logged in, cart not empty, address present and so on), with the real work at the innermost level. What should a reviewer suggest? Analysis
    A. Use guard clauses: return or raise early when each precondition fails, so the main logic is not deeply indented
    B. Combine all the conditions into one long expression joined with and
    C. Add a comment at each level explaining the nesting
    D. Replace every if with a try/except block

1579. A developer opens one pull request with 3,000 changed lines across 40 files, mixing a refactor, a new feature and formatting changes. Reviewers approve it quickly, but it later causes several bugs. What is the main problem? Analysis
    A. The branch should have been rebased instead of merged
    B. CI should be turned off for large pull requests
    C. The change is too large and mixed for an effective review; it should be split into small, focused pull requests
    D. Large pull requests need an approval from the manager

1580. requirements.txt lists requests and pandas with no versions. A fresh install on the CI server months later fails with errors that never happen on the developer's laptop. What is the most likely cause? Analysis
    A. The dependencies are unpinned, so CI installed newer releases with breaking changes; pin versions or use a lock file
    B. The CI server needs more RAM
    C. pip cannot install two packages from one file
    D. requirements.txt must be renamed setup.py

1581. A reviewer flags this line in an OrderPrinter class: order.getCustomer().getAddress().getCity(). Which principle does it break, and why does that matter? Analysis
    A. DRY, because getCity() is called once
    B. YAGNI, because the city may never be printed
    C. The Law of Demeter: OrderPrinter depends on the internal structure of Customer and Address, so changes to them ripple into unrelated code
    D. The Liskov Substitution Principle, because getters cannot be chained

1582. A team commits its compiled build/ folder and .class files. Almost every merge now conflicts in these files, even when the source changes do not overlap. What is the best fix? Analysis

A. Always resolve the conflicts by taking 'ours'
B. Commit more often
C. Stop tracking generated files: add them to .gitignore and remove them from the index with git rm -r --cached, and let the build recreate them
D. Use squash merges for every branch

1583. A teammate's commit log is full of messages such as 'fixed stuff', 'update' and 'asdf'. Why is this a problem? Analysis

A. The messages should record what changed and why; vague ones make git log, git blame and bisect results hard to understand
B. Git rejects messages shorter than ten characters
C. Short messages make the repository much larger
D. Vague messages cause merge conflicts

1584. Your app depends on a library with the range ^1.4.0. The library releases 1.5.0, which removes a public function your app uses, and your build breaks. Under Semantic Versioning, who made the mistake? Analysis

A. You, because ^ allows any newer version
B. Nobody, because minor versions may remove features
C. npm, because ^ should never allow minor updates
D. The library: removing a public function is a breaking change and needed a new major version, 2.0.0

1585. Each developer works on a private branch for a month and merges at the end. Every merge day brings days of conflicts and broken builds. Which practice addresses the root cause? Analysis

A. Keep feature branches open even longer, so features are complete before merging
B. Let one senior developer do all the merging
C. Continuous integration: merge small changes into the mainline at least daily, with an automated build and tests on every merge
D. Freeze all code for a week before each merge

1586. A reviewer sees create_student(name, father_name, cnic, phone, email, street, city, province, postal_code). What is the usual refactoring? Analysis

A. Introduce a parameter object: group related values, such as an Address with street, city, province and postal code
B. Make all the values global variables
C. Split it into nine one-parameter functions
D. Give every parameter a default value

1587. In a file with lines l1 to l4, main changes line 2 and feature changes line 3. git merge feature reports a conflict. Why, when different lines were changed? Analysis

A. Git merges by hunks; edits on adjacent lines with no unchanged line between them overlap, so Git cannot decide automatically
B. Git always reports a conflict when both branches change the same file
C. The two branches must have different line endings
D. feature had to be rebased onto main before merging

1588. Which code review comment best follows good review practice? Analysis

A. "This is stupid code."
B. "This loop queries the database once per student, which will be slow for big classes. Could we fetch them in one query?"
C. "I would never write it like this."
D. "LGTM" on a change the reviewer has not read

1589. After git fetch, git status -sb shows '## main...origin/main [ahead 1, behind 1]'. What does this mean? Analysis

A. You are one commit ahead, so git push will succeed now
B. fetch merged one commit, and you still need to push the other
C. One file is staged and one is unstaged
D. Your main has one commit that origin/main lacks and origin/main has one that yours lacks, so the branches have diverged and must be merged or rebased before a normal push

1590. Adding a new payment method, such as JazzCash, requires edits in nine files: validation, pricing, receipts, logging and more. Which code smell is this, and what is the usual remedy? Analysis

A. Long method; split each function into smaller ones
B. Magic numbers; replace literals with named constants
C. Feature envy; move each method into the class whose data it uses most
D. Shotgun surgery; gather the scattered logic, for example one class per payment method behind a common interface

OOP Code Questions for the NSCT

1591. Does ColourPrinter override Printer's print method? Understanding

```
class Printer {
    void print() { System.out.println("plain"); }
}
class ColourPrinter extends Printer {
    void print(String colour) { System.out.println(colour); }
}
```

A. No; print(String) has a different parameter list, so it overloads print() and both remain available
B. Yes; any method with the same name in a subclass overrides the parent's
C. No; the code fails to compile because the names clash
D. Yes, but only when the call is made through a Printer reference

1592. What happens when this code is compiled? Understanding

```
class Animal { void eat() { System.out.println("eats"); } }
class Dog extends Animal { void fetch() { System.out.println("fetches"); } }

Animal a = new Dog();
a.eat();     // line 1
a.fetch();   // line 2
```

A. Both lines compile, and line 2 prints fetches
B. Line 1 fails, because a refers to a Dog
C. It compiles, but line 2 throws ClassCastException at run time
D. Line 2 fails to compile, because the declared type Animal has no fetch method

1593. What does this C++ program print? Understanding

```
#include <iostream>
using namespace std;
struct Base {
    void show() { cout << "Base "; }
    virtual void hello() { cout << "Hi-Base "; }
};
struct Derived : Base {
    void show() { cout << "Derived "; }
    void hello() override { cout << "Hi-Derived "; }
};
int main() {
    Derived d;
    Base& r = d;
    r.show();
    r.hello();
}
```

A. Base Hi-Derived
B. Derived Hi-Derived
C. Base Hi-Base
D. Derived Hi-Base

1594. What does this Python code print? Understanding

```
class A:
    def hello(self):
        return "A"

class B:
    def hello(self):
        return "B"

class C(A, B):
    pass

print(C().hello(), [k.__name__ for k in C.__mro__])
```

A. B ['C', 'B', 'A', 'object']
B. A ['C', 'A', 'B', 'object']
C. A ['C', 'B', 'A', 'object']
D. TypeError: ambiguous method hello

1595. What does this Java code print? Understanding

```
class A {
    static String who() { return "A"; }
    String me() { return "A"; }
}
class B extends A {
    static String who() { return "B"; }
    String me() { return "B"; }
}
A obj = new B();
System.out.println(obj.who() + obj.me());
```

A. BB
B. AA
C. BA
D. AB

1596. Which line fails to compile, and why? Understanding

```
class Parent {
    private int secret = 1;
    protected int shared = 2;
}
class Child extends Parent {
    int read1() { return shared; }   // line 1
    int read2() { return secret; }   // line 2
}
```

A. Line 1, because protected members are only visible in the same package
B. Line 2, because a private member is not accessible in a subclass
C. Both lines, because subclasses cannot read inherited fields
D. Neither line; both fields are inherited

1597. What does new SportsCar() print? Application

```
class Vehicle {
    Vehicle() { System.out.print("Vehicle "); }
}
class Car extends Vehicle {
    int wheels = init();
    int init() { System.out.print("wheels "); return 4; }
    Car() { System.out.print("Car "); }
}
class SportsCar extends Car {
    SportsCar() { System.out.print("SportsCar"); }
}
// in main:
new SportsCar();
```

A. SportsCar Car wheels Vehicle
B. wheels Vehicle Car SportsCar
C. Vehicle wheels Car SportsCar
D. Vehicle Car wheels SportsCar

1598. What does this Java code print? Application

```
class Shape  { double area() { return 0; } }
class Circle extends Shape {
    double r; Circle(double r) { this.r = r; }
    double area() { return 3 * r * r; }   // pi taken as 3
}
class Square extends Shape {
    double a; Square(double a) { this.a = a; }
    double area() { return a * a; }
}
// in main:
Shape[] s = { new Circle(2), new Square(3), new Shape() };
double total = 0;
for (Shape x : s) total += x.area();
System.out.println(total);
```

A. 21.0
B. 0.0
C. 12.0
D. 21

1599. Which overloads are chosen for Calc.f(5) and Calc.f(5.0f)? Application

```
class Calc {
    static void f(long x)    { System.out.print("long "); }
    static void f(Integer x) { System.out.print("Integer "); }
    static void f(double x)  { System.out.print("double "); }
}
// in main:
Calc.f(5);
Calc.f(5.0f);
```

A. Integer double
B. long long
C. double double
D. long double

1600. What does this Python code print? Application

```
class Counter:
    count = 0
    def hit(self):
        self.count += 1

a, b = Counter(), Counter()
a.hit(); a.hit(); b.hit()
Counter.count = 10
print(a.count, b.count, Counter().count)
```

A. 2 1 10
B. 10 10 10
C. 3 3 10
D. 12 11 10

1601. What is the last line printed by this Python code? Application

```
class Base:
    def greet(self):
        return "Base"

class Mid(Base):
    def greet(self):
        return "Mid>" + super().greet()

class Leaf(Mid):
    def greet(self):
        return "Leaf>" + super().greet()

for obj in (Base(), Mid(), Leaf()):
    print(obj.greet())
```

A. Leaf
B. Leaf>Base
C. Leaf>Mid>Base
D. Base>Mid>Leaf

1602. What does new Derived() print? Application

```
class Base {
    Base() { show(); }
    void show() { System.out.print("base "); }
}
class Derived extends Base {
    int x = 7;
    Derived() { show(); }
    void show() { System.out.print(x + " "); }
}
// in main:
new Derived();
```

A. base 7
B. 7 7
C. 0 7
D. base 0

1603. What does this C++ program print? Application

```
#include <iostream>
using namespace std;
struct Emp {
    virtual int pay() { return 100; }
    int bonus() { return 10; }
};
struct Manager : Emp {
    int pay() override { return 300; }
    int bonus() { return 50; }
};
int main() {
    Manager m;
    Emp e;
    Emp* staff[] = { &e, &m };
    int total = 0;
    for (Emp* p : staff) total += p->pay() + p->bonus();
    cout << total;
}
```

A. 460
B. 420
C. 220
D. 400

1604. What does this Python code print? Application

```
class A:
    def f(self):
        return ["A"]

class B(A):
    def f(self):
        return ["B"] + super().f()

class C(A):
    def f(self):
        return ["C"] + super().f()

class D(B, C):
    def f(self):
        return ["D"] + super().f()

print(D().f())
```

A. ['D', 'B', 'A', 'C', 'A']
B. ['D', 'B', 'A']
C. ['D', 'B', 'C', 'A']
D. ['D', 'C', 'B', 'A']

1605. What does this Java code print, and why? Analysis

```
class A {
    int x = 1;
    int getX() { return x; }
}
class B extends A {
    int x = 2;
    int getX() { return x; }
}
// in main:
A ref = new B();
System.out.println(ref.x + " " + ref.getX());
```

A. 2 2, because both lookups use the object's real type
B. 1 1, because ref is declared as A
C. 2 1, because fields are dynamic and methods are static
D. 1 2, because fields are chosen by the reference's declared type, while methods are chosen by the object's type

1606. What are the two lines of output? Analysis

```
class Log {
    static void write(Object o) { System.out.println("Object"); }
    static void write(String s) { System.out.println("String"); }
}
// in main:
Log.write(null);
Object o = "hello";
Log.write(o);
```

A. String, then Object
B. Object, then String
C. String, then String
D. A compile error: write(null) is ambiguous

1607. speak is virtual. What does the program print, and why? Analysis

```
#include <iostream>
using namespace std;
struct Animal {
    virtual void speak() const { cout << "..." << endl; }
};
struct Cat : Animal {
    void speak() const override { cout << "Meow" << endl; }
};
void byValue(Animal a)      { a.speak(); }
void byRef(const Animal& a) { a.speak(); }
int main() {
    Cat c;
    byValue(c);
    byRef(c);
}
```

A. Meow then Meow, because speak is virtual
B. ... then ..., because both parameters have type Animal
C. ... then Meow, because byValue copies only the Animal part of c (slicing)
D. Meow then ..., because references disable virtual calls

1608. What does this Python code print, and what is the bug? Analysis

```
class Team:
    members = []
    def __init__(self, name):
        self.name = name
    def add(self, person):
        self.members.append(person)

red, blue = Team("red"), Team("blue")
red.add("Ali")
blue.add("Sara")
print(len(red.members), red.members is blue.members)
```

A. 1 False; each Team gets its own list
B. 2 True; members is one list on the class, shared by every Team
C. 1 True; the lists are equal but separate
D. 2 False; append copies the list

1609. What does this Java code print? Analysis

```
class Point {
    int x, y;
    Point(int x, int y) { this.x = x; this.y = y; }
    public boolean equals(Point p) { return x == p.x && y == p.y; }
}
// in main:
Point a = new Point(1, 2), b = new Point(1, 2);
List<Point> list = new ArrayList<>();
list.add(a);
System.out.println(a.equals(b) + " " + list.contains(b));
```

A. true true
B. false false
C. false true
D. true false

1610. Only __str__ is overridden. What do the two print calls show? Analysis

```
class Item:
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return f"Item({self.name})"

x = Item("pen")
print(x)
print([x])
```

A. Item(pen), then [Item(pen)]
B. Item(pen), then [<__main__.Item object at 0x...>]
C. <__main__.Item object at 0x...> twice
D. Item(pen), then ['Item(pen)']

Pointers and Memory Practice for the NSCT

1611. Which expression gives the value 25 through the pointer p? Understanding

```
int x = 25;
int *p = &x;
```

A. &p
B. p
C. &x
D. *p

1612. On a system where int is 4 bytes, p is an int* holding address 2000. What address does p + 1 hold? Understanding

A. 2001
B. 2008
C. 2000
D. 2004

1613. For an array arr and an int i, which expression is always equivalent to arr[i] in C and C++? Understanding

A. *arr + i
B. *(arr + i)
C. &arr + i
D. *(&arr[i] + 1)

1614. Which statement about p and q is correct? Understanding

```
int a = 1, b = 2;
const int *p = &a;   // declaration P
int *const q = &a;   // declaration Q
```

A. Both forbid changing a through them
B. p cannot be re-pointed, and q cannot change the value
C. *p = 5 is an error but p = &b is allowed; *q = 5 is allowed but q = &b is an error
D. Both can be re-pointed, but neither can change the value

1615. What does this Python code print? Understanding

```
a = [1, 2, 3]
b = a
b.append(4)
print(a, a is b)
```

A. [1, 2, 3] False
B. [1, 2, 3, 4] False
C. [1, 2, 3, 4] True
D. [1, 2, 3] True

1616. On a 64-bit system with 4-byte int and 8-byte pointers, what does f print when given a 10-element int array? Understanding

```
void f(int a[10]) {
    printf("%zu\n", sizeof(a));
}
```

A. 40, the size of 10 ints
B. 8, because the parameter a is really an int*
C. 10, the number of elements
D. 4, the size of one int

1617. What does this C program print? Application

```
#include <stdio.h>
int main(void) {
    int arr[] = {7, 14, 21, 28, 35};
    int *p = arr + 1;
    printf("%d %d %d\n", *p, *(p + 2), p[-1]);
    return 0;
}
```

A. 7 21 0
B. 14 16 13
C. 14 21 7
D. 14 28 7

1618. double d[6] starts at address 0x3000, with an 8-byte double. What are the addresses &d[4] and (char*)d + 4? Application

A. 0x3020 and 0x3004
B. 0x3004 and 0x3004
C. 0x3020 and 0x3020
D. 0x3010 and 0x3004

1619. With 4-byte ints, what does the program print? Application

```
int arr[8] = {0};
int *p = &arr[1];
int *q = &arr[6];
printf("%td\n", q - p);
```

A. 20
B. 6
C. 24
D. 5

1620. What does this C++ program print? Application

```
#include <iostream>
using namespace std;
void update(int *a, int b) {
    *a += b;
    b *= 2;
}
int main() {
    int x = 4, y = 3;
    update(&x, y);
    update(&y, x);
    cout << x << " " << y;
}
```

A. 7 6
B. 7 10
C. 4 3
D. 14 10

1621. What does this C++ program print? Application

```
#include <iostream>
using namespace std;
int main() {
    int x = 5;
    int *p = &x;
    int **pp = &p;
    **pp = 9;
    int y = 2;
    *pp = &y;
    *p += 10;
    cout << x << " " << y << " " << **pp;
}
```

A. 9 12 12
B. 15 2 15
C. 19 2 19
D. 9 2 9

1622. What does this C program print? Application

```
#include <stdio.h>
int main(void) {
    int m[3][4];
    for (int r = 0; r < 3; r++)
        for (int c = 0; c < 4; c++)
            m[r][c] = r * 10 + c;
    printf("%d %d\n", *(*(m + 2) + 1), *(&m[0][0] + 6));
    return 0;
}
```

A. 21 6
B. 12 21
C. 21 12
D. 21 60

1623. What does this Python code print? Application

```
import copy
grid = [[1, 2], [3, 4]]
shallow = copy.copy(grid)
deep = copy.deepcopy(grid)
grid[0][0] = 99
grid.append([5, 6])
print(shallow[0][0], deep[0][0], len(shallow))
```

A. 1 1 2
B. 99 1 2
C. 99 99 3
D. 99 1 3

1624. What does this C program print? Application

```
#include <stdio.h>
int main(void) {
    const char *s = "NSCT2026";
    s += 4;
    printf("%s %c %d\n", s, *(s + 1), (int)(s[3] - '0'));
    return 0;
}
```

A. NSCT2026 S 6
B. 2026 0 6
C. 2026 2 6
D. T2026 0 2

1625. What does the program print, and how should make be fixed? Analysis

```
void make(int *p) {
    p = new int(42);
}
int main() {
    int *ptr = nullptr;
    make(ptr);
    if (ptr == nullptr) std::cout << "still null";
    else std::cout << *ptr;
}
```

A. still null; make receives a copy of the pointer, so take int** (or int*&) and assign *p = new int(42)
B. 42; new always updates the caller's pointer
C. still null; new failed and returned nullptr
D. It crashes, because ptr is dereferenced while null

1626. What does this C program print? Analysis

```
#include <stdio.h>
int main(void) {
    int a[] = {5, 10, 15};
    int *p = a;
    int x = *p++;
    int y = (*p)++;
    int z = ++*p;
    printf("%d %d %d %d %d\n", x, y, z, a[1], (int)(p - a));
    return 0;
}
```

A. 6 10 11 11 1
B. 5 15 16 10 2
C. 5 10 11 12 1
D. 5 10 12 12 1

1627. What does this C++ program print? Analysis

```
#include <iostream>
struct Box { int *data; };
int main() {
    int v = 7;
    Box a{&v};
    Box b = a;        // default copy
    *b.data = 42;
    int w = 1;
    b.data = &w;
    *b.data = 99;
    std::cout << v << " " << *a.data << " " << w;
}
```

A. 7 7 99
B. 42 99 99
C. 42 42 99
D. 7 42 1

1628. What does this Python code print? Analysis

```
rows = [[0] * 3] * 2
rows[0][1] = 5
cols = [[0] * 3 for _ in range(2)]
cols[0][1] = 5
print(rows)
print(cols)
```

A. [[0, 5, 0], [0, 5, 0]] then [[0, 5, 0], [0, 0, 0]]
B. [[0, 5, 0], [0, 0, 0]] twice
C. [[0, 5, 0], [0, 5, 0]] twice
D. [[5, 0, 0], [0, 0, 0]] then [[0, 5, 0], [0, 0, 0]]

1629. What does this C++ program print? Analysis

```
#include <iostream>
int main() {
    int x = 1, y = 2;
    int &r = x;
    r = y;
    y = 5;
    r += 10;
    std::cout << x << " " << y;
}
```

A. 1 15
B. 15 5
C. 12 5
D. 12 15

1630. On a 64-bit system with 4-byte int and 8-byte pointers, what does the program print? Analysis

```
#include <stdio.h>
int main(void) {
    int arr[10];
    int *p = arr;
    printf("%zu %zu\n", sizeof(arr) / sizeof(arr[0]),
                        sizeof(p) / sizeof(p[0]));
    return 0;
}
```

A. 10 2
B. 10 10
C. 40 8
D. 2 10

Answer key and explanations

1. **C** — A compiler translates the whole source program ahead of time into object or machine code; the program is then run as a separate step. An interpreter translates and executes statements one at a time while the program runs.
2. **D** — An interpreter reads a statement, translates it and executes it before moving on, so no standalone executable is produced. This is why an interpreted program can run its first lines before reaching a later error that is only found at run time.
3. **D** — javac compiles Java source into platform-independent bytecode stored in .class files. The Java Virtual Machine (JVM) then interprets that bytecode and, in modern JVMs, also JIT-compiles hot code to native instructions.
4. **A** — The linker resolves references between object files and libraries (for example, a call to a function defined in another file) and writes the final executable. The loader later places that executable in memory to run it.
5. **C** — Directives starting with # are handled by the preprocessor. #include pastes the named header into the source text, and #define performs macro substitution, all before the compiler translates the code.
6. **B** — Python single-line comments start with #. The // and /* */ forms are comments in C++, Java and JavaScript, and <!-- --> is an HTML comment.
7. **C** — A syntax error breaks the rules for how statements must be written, so the compiler or interpreter cannot parse the code. It is reported before the faulty code can run, unlike runtime errors (raised during execution) and logic errors (code runs but gives the wrong result).
8. **C** — A logic error is a mistake in the program's reasoning, such as a wrong formula or loop bound. The code is valid and runs, so neither the compiler nor the runtime reports anything; only testing reveals it.
9. **A** — The JVM starts a Java application by calling public static void main(String[] args) in the class it is given. It is static so it can run without creating an object, and void because a Java main returns nothing.
10. **C** — CPython first compiles the whole source file to bytecode (cached as .pyc files for imported modules), then its virtual machine interprets that bytecode. So calling Python 'purely interpreted' is a simplification; the compile step is why a syntax error anywhere in the file stops the file before any line runs.
11. **B** — A JIT compiler, as used in the JVM and JavaScript engines such as V8, translates hot (frequently executed) code to native instructions at run time. This combines the portability of bytecode with speed close to ahead-of-time compiled code.
12. **D** — Identifiers may contain letters, digits and underscores but cannot start with a digit. total-marks contains a minus sign, which is an operator, and class is a reserved keyword in all three languages.
13. **B** — An assembler converts low-level assembly instructions such as MOV and ADD into binary machine code, roughly one machine instruction per mnemonic. Turning machine code back into source is disassembly or decompilation.
14. **A** — The C++ standard makes main a special case: flowing off the end of main is equivalent to return 0;. For any other function with a non-void return type, flowing off the end is undefined behaviour, which is why students often wrongly assume main must have a return.
15. **B** — A diamond holds a yes/no condition and has two outgoing branches. A rectangle is a process step, a parallelogram is input or output, and an oval (terminator) marks start or end.
16. **A** — Java is statically typed, so int n = "ten"; is rejected by the compiler as incompatible types. Integer division by zero throws ArithmeticException and a bad index throws ArrayIndexOutOfBoundsException only when those lines run, and a wrong loop operator is a logic error nobody reports.
17. **D** — A def statement only defines the function; its body runs when it is called. Execution goes top to bottom: print('Start') runs first, then greet() is called and prints Hi.
18. **A** — In Python, def is an executable statement that creates the function when that line is reached. Line 1 calls square before the def has executed, so the name is undefined and a NameError is raised at run time.
19. **C** — x is 2, so the condition x > 3 is false and the indented block (print('A')) is skipped. print('B') is not indented, so it is outside the if and always runs.
20. **B** — In Java (and C++) every simple statement must end with a semicolon. int count = 10 has none, so the compiler reports a syntax error such as "';' expected".

21. **D** — C++ identifiers are case-sensitive, so Total and total are different names. total was never declared, so the compiler stops with an error such as "'total' was not declared in this scope". Java and Python are case-sensitive too.
22. **B** — JavaScript hoists function declarations: the whole function is created before any code in its scope runs, so sq can be called above its definition. The same pattern in Python raises NameError, a common cross-language contrast.
23. **C** — 10 / 0 is grammatically valid, so the script compiles and starts running. print('one') executes, then the division raises ZeroDivisionError at run time, which ends the program before print('two').
24. **A** — Python normally ends a statement at the end of the line, but a semicolon may separate several simple statements on one line. a and b are set to 3 and 4, and a * b prints 12.
25. **B** — A Java source file may contain at most one public top-level class, and the file must be named after it. Because the public class is Test, javac rejects Main.java; renaming the file to Test.java fixes it.
26. **C** — Line 1 is entirely a comment. On line 2 everything after # is a comment, so only B is printed. Line 3 is a string literal expression that is evaluated and discarded; it is not code, so D is never printed.
27. **D** — Java is statically typed: x is declared int and keeps that type, so assigning a String is rejected at compile time. In Python the same idea (x = 5 then x = 'five') is legal because names are not bound to a type.
28. **B** — Inside a, the name b is looked up only when a() is called. By the time print(a()) runs, both def statements have executed, so b exists and returns 10, and a returns 10 + 1 = 11.
29. **B** — Arguments are evaluated before a function is called. To evaluate g(f()), Python first calls f, which prints f and returns 1. Then g(1) prints g and returns 2, and finally print shows 2.
30. **D** — std::cout is declared in the <iostream> header. Without #include <iostream>, the preprocessor never pastes in that declaration, so the compiler reports that cout is not a member of std (or not declared).
31. **A** — CPython compiles the whole file to bytecode before executing anything, and a missing colon is a syntax error. Compilation fails, so even print('Ready') on line 1 never runs. Contrast this with a runtime error such as division by zero, where earlier lines do run.
32. **C** — When a file is imported, its top-level code runs but __name__ is set to 'shapes', not '__main__'. So the if test is false and main() is not called, while the last print still runs. Run directly, the same file would print run then end.
33. **C** — Java's compiler performs reachability analysis and treats code that can never execute as an error: "unreachable statement". A bare return; is legal in a void method, so the problem is only the line after it. Python, by contrast, accepts dead code after return silently.
34. **D** — return ends the function immediately, so print('x') is never reached. Python allows unreachable code without any error or warning, and f() returns 1, which is printed. Java would reject the same structure at compile time.
35. **A** — The code is valid and runs, so it is a logic error. Division has higher precedence than subtraction, so the return computes (170 / 2) - 1 = 84.0 instead of 85.0. Removing - 1 fixes it.
36. **A** — With an empty list len(marks) is 0, so total / 0 raises ZeroDivisionError. Runtime errors like this depend on the data and appear only when that input occurs, which is why edge cases such as empty input belong in tests.
37. **D** — Ahead-of-time compilation (for example, C++ to a native executable) gives a standalone binary that starts quickly, needs no interpreter or VM, and does not expose source code. The trade-off is building a separate executable for each operating system and CPU.
38. **B** — javac targets the JVM, a virtual machine that exists for each platform, so one bytecode file runs anywhere a JVM is installed. A C++ compiler emits native instructions in an OS-specific executable format, so it must be recompiled for each target.
39. **D** — The prototype int square(int n); lets the compiler type-check the call, so compilation succeeds. But no function body exists in any object file, so the linker reports an error such as "undefined reference to square(int)" and cannot build the executable.
40. **A** — A misspelled name is not a syntax error; Python only looks the name up when that line executes. x > 10 is false, so line 3 never runs and the bug stays hidden. This is why untested branches in interpreted languages can hide errors a compiler would catch.

41. **A** — The loader is the operating-system component that reads an executable from disk into main memory, sets up its address space and jumps to its entry point. The compiler, assembler and preprocessor all finish their work before the program is ever run.
42. **D** — Source code is the text written by the programmer, for example a .cpp, .java or .py file. Machine code, bytecode and executables are all outputs that translators produce from that source.
43. **D** — Assembly language maps almost one-to-one onto a processor's machine instructions and uses registers and memory addresses directly, so it is classed as low level. Python, Java and JavaScript are high-level languages that hide the hardware.
44. **A** — The source is first split into tokens (lexical analysis), the tokens are checked against the grammar to build a parse tree (syntax analysis), types and declarations are checked (semantic analysis), and only then is target code generated.
45. **C** — The lexer (scanner) reads characters and groups them into tokens, for example int, total, =, 5 and ;. The parse tree comes from the next phase, syntax analysis; type errors come from semantic analysis; object code comes last.
46. **A** — A parallelogram marks input and output. A rectangle is a processing step, a diamond is a decision and an oval (terminal) marks the start or end of the flowchart.
47. **A** — Ovals are terminal symbols: every flowchart begins at a Start oval and finishes at an End (or Stop) oval. Decisions use diamonds, calculations use rectangles and input uses parallelograms.
48. **C** — An algorithm needs finiteness, definiteness, input, output and effectiveness. It can be expressed in plain language, pseudocode or a flowchart; a programming language is only one way to write it down.
49. **C** — Python uses indentation to mark blocks, so inconsistent indentation breaks the grammar. It is reported as IndentationError (a subclass of SyntaxError) before any line of the file runs. The other three are runtime errors.
50. **D** — CPython compiles imported modules to bytecode and saves it as .pyc files in __pycache__, so the next import can skip compilation if the source has not changed. .class files belong to Java, .o files to C/C++ builds, and CPython never produces a native executable.
51. **A** — A transpiler's output is still high-level source code, for example TypeScript compiled to JavaScript. Converting assembly to machine code is the assembler's job, joining object files is the linker's job, and running bytecode is a virtual machine's job.
52. **C** — A cross-compiler is used when the target cannot build code itself, for example compiling on a Windows PC to produce code for an ARM microcontroller. Turning machine code back into source is decompilation, not cross-compilation.
53. **B** — Every token is legal and the statement has the correct shape (type, name, =, expression, ;), so the lexer and parser accept it. The semantic phase checks meaning, including whether a String can be assigned to an int, and reports the incompatible types.
54. **A** — Documentation comments start with /** and end with */, and javadoc turns them into HTML docs. // and /* */ are ordinary comments that javadoc ignores, and # is not a comment in Java at all.
55. **A** — With dynamic linking the executable keeps references to shared libraries that the loader resolves at load or run time, so many programs can share one copy. Copying the library into the executable is static linking.
56. **B** — An exit status of 0 conventionally means success; a non-zero value signals some kind of failure. Scripts and build tools check this value to decide whether the next step should run.
57. **D** — pass is a placeholder statement that has no effect, so def todo(): pass is valid. continue and break only make sense inside loops, and skip is not a Python keyword.
58. **D** — A /* comment ends at the first */ that follows it, so /* a /* b */ c */ leaves c */ as code and causes an error. They can span many lines, and comments are discarded early in translation, long before linking.
59. **D** — #include is a preprocessor directive that pastes the header's contents into the translation unit. A Java import is just a naming shortcut: the class is still loaded separately at run time and nothing is copied into your source or .class file.
60. **D** — static is a Java keyword; Python has no such keyword (it uses the @staticmethod decorator instead). if, return and class are keywords in both languages, so none of them can be used as variable names in either.
61. **B** — The Boolean constants in Python are True and False, spelled with a capital first letter. true and TRUE are ordinary names (and undefined by default), and Python uses None, not null.

62. **D** — The compiler and assembler turn each .cpp file into an object file of machine code that may still contain unresolved references. The linker then combines object files and libraries into an executable. The word 'object' here has nothing to do with object-oriented programming.
63. **B** — Pseudocode uses structured English with keywords such as IF, WHILE and RETURN to plan an algorithm. It is not compiled or run, so it does not have to follow any one language's exact syntax.
64. **D** — sep is placed between the arguments, giving a-b, and end replaces the usual newline after the last argument, so the line is a-b!. end is not separated from the last argument by sep.
65. **D** — Python joins adjacent string literals at compile time, so "Hello" "World" is one literal, "HelloWorld". No space is added, and because it is a single argument print adds no separator either.
66. **C** — def is an executable statement that binds a name to a function. The second def rebinds show to the new function, so the call runs the later definition only. Python does not treat a repeated def as an error.
67. **A** — The inner call runs first and prints hi. print returns None, so the outer call receives None and prints it on the next line.
68. **B** — A # inside a string literal is just a character, so the string "# total" is printed in full. The second # is outside the string, so everything after it is a comment and is ignored.
69. **B** — sys.argv is a list of the command-line words starting with the script name, so it holds ['app.py', 'red', 'blue'] and its length is 3. Only sys.argv[1:] are the user's arguments.
70. **B** — print writes text without a newline, while println writes its text and then a newline. So A and B share the first line, the newline comes after B, and C starts the second line.
71. **D** — + is evaluated left to right. In line 1, "1" + 2 is already a String ("12"), so + 3 appends to give 123. In line 2, 1 + 2 is integer addition (3) before "3" is appended, giving 33.
72. **A** — The compiler only checks types, and int / int is well typed. The value of b is only known when the program runs, and integer division by zero then throws java.lang.ArithmeticException: / by zero. Infinity only appears in floating-point division.
73. **C** — The << operations run left to right. std::endl writes a newline (and flushes the stream) between A and B, so B starts on a new line.
74. **B** — var declarations are hoisted to the top of their scope, but the assignment is not. When line 1 runs, x exists but has not been given 3 yet, so its value is undefined.
75. **C** — Only the var greet declaration is hoisted, with the value undefined. The function expression is assigned when its line runs, which is after the call, so calling undefined throws a TypeError. A function declaration (function greet() {...}) would have been hoisted whole.
76. **A** — let variables are in a temporal dead zone from the start of the block until their declaration runs. Reading y on line 1 throws ReferenceError: Cannot access 'y' before initialization. Only var gives undefined here.
77. **D** — A string literal placed as the first statement of a function is its docstring. Python stores it in the function's __doc__ attribute, so printing area.__doc__ shows the text. Unlike a # comment, it is kept at run time.
78. **D** — A backslash at the end of a line joins it to the next line, so the three lines form one statement, total = 1 + 2 + 3, which is 6.
79. **B** — The JVM puts only the words after the class name into args, so args is {"hello", "world"} and args.length is 2. The class name itself is not included.
80. **A** — The preprocessor substitutes text, not values. SQ(2 + 3) becomes 2 + 3 * 2 + 3, and because * binds tighter than +, that is 2 + 6 + 3 = 11. Writing the macro as ((x) * (x)) would give 25.
81. **C** — end=" " makes the first print finish with a space instead of a newline, so the second print continues on the same line: 10 20. The second print uses the default end, a newline.
82. **C** — Inside a string, \\ is an escape for one backslash, so line 1 prints C:\new. In line 2, \n is the newline escape, so the output breaks after C: and new appears on the following line.
83. **C** — Top-level code runs from top to bottom, so top prints first. When a file is run directly, __name__ is "__main__", so the if test is true and main() then prints main.
84. **C** — C++ allows unreachable code; a compiler may warn but must still build it. return 0 ends main, so the second output statement never runs and only A is printed. The 'unreachable statement' compile error is a Java rule.

85. **A** — JavaScript's automatic semicolon insertion ends a return statement at a line break. The function becomes return; followed by an unreachable 5 + 5;, so it returns undefined. Keep the returned expression on the same line as return.
86. **C** — CPython compiles the whole file to bytecode before running any of it. The incomplete expression 1 + is a grammar error, so compilation fails and not even the first print runs. Whether the function is called does not matter for syntax errors.
87. **B** — javac checks that every path through a non-void method ends in a return or a throw. When n is 0 neither condition is true and control falls off the end. Adding return 0; or a final else fixes it.
88. **B** — Python's sys.argv is ['app.py', 'a', 'b'], so the user's arguments start at index 1. Java passes {"a", "b"}, so they start at index 0. Mixing these conventions is a common off-by-one bug when porting code.
89. **C** — Each file is a separate translation unit, so each compiles happily with its own definition. The linker then finds two definitions of the same function, which breaks C++'s one-definition rule and is reported as a multiple-definition (duplicate symbol) error.
90. **B** — 5.0 / 0 is floating-point division: the 0 is promoted to 0.0 and IEEE 754 gives Infinity. 5 / 0 is integer division, which has no infinity, so it throws ArithmeticException: / by zero at run time. javac at most warns about it.
91. **C** — The program runs without any message and gives a wrong value, which is a logic error. range(1, n) produces 1 to 4, so the sum is 10. range(1, n + 1) fixes it.
92. **A** — #include pastes text, so point.h's contents appear twice in main.cpp after preprocessing. Redefining the same struct in one translation unit is a compile error. Include guards make the second inclusion expand to nothing.
93. **A** — Java resolves every name at compile time, so an undeclared identifier stops compilation even in code that would rarely run. Python looks names up only when a line executes, which is why the error waited for that branch.
94. **D** — def is an ordinary statement executed at run time. Because debug is False, the body of the if is skipped, the name log is never bound, and the call raises NameError: name 'log' is not defined.
95. **C** — A missing semicolon breaks the grammar, so javac stops. An out-of-range index compiles but throws ArrayIndexOutOfBoundsException when it runs. The wrong operator compiles and runs but gives a wrong result, which is a logic error.
96. **B** — In C++, // begins a comment, so the compiler sees std::cout << 7 with no semicolon, followed by return 0;. That is a syntax error. Integer division in C++ is written 7 / 2.
97. **D** — CPython compiles the source once to bytecode, but each bytecode instruction is then dispatched by the interpreter, which also checks the object types involved. A C++ compiler resolves types ahead of time and emits machine instructions the CPU runs directly.
98. **B** — The compiler reads the file top to bottom, so at the call it has never seen twice and reports 'twice was not declared in this scope'. A forward declaration (prototype) before main fixes it. JavaScript hoists function declarations, so it has no such rule.
99. **B** — The function uses > where >= is needed, which only matters at the boundary. For 50 it returns False although 50 should pass. Testing boundary values is the standard way to expose this kind of logic error.
100. **A** — The Java Language Specification fixes int at 32 bits (two's complement) on every platform, giving a range of -2,147,483,648 to 2,147,483,647. This fixed size is a key difference from C++, where int size is implementation-defined.
101. **C** — A Java char is a 16-bit unsigned UTF-16 code unit, range 0 to 65,535. C and C++ char is 1 byte (usually 8 bits), which is a common cross-language confusion.
102. **D** — C++ defines sizeof(char) as exactly 1 byte. The standard only sets minimum ranges for other types: int must be at least 16 bits and long at least 32, so their sizes vary between compilers and platforms. Most modern desktop compilers use a 32-bit int.
103. **A** — Python has no const or final keyword for variables. PEP 8 recommends UPPER_CASE names such as MAX_MARKS to signal 'do not change', but the interpreter will still allow reassignment.
104. **C** — final prevents reassignment of a variable once it has a value, for example final double PI = 3.14159;. const is a reserved word in Java but is not used, and static means the field belongs to the class, not that it is constant.

105. **C** — A const variable cannot be modified after initialisation, so the compiler rejects x = 6; with an error such as "assignment of read-only variable 'x'". The check happens at compile time, not run time.
106. **A** — byte is an 8-bit signed two's-complement integer. With 8 bits the range is -2^7 to 2^7 - 1, which is -128 to 127. 0 to 255 would be an unsigned byte, which Java does not have.
107. **C** — Java has eight primitives: byte, short, int, long, float, double, char and boolean. String is a class in java.lang, so a String variable holds a reference to an object.
108. **D** — Python 3 int is arbitrary precision, so 2 ** 100 is computed exactly without overflow. The other values are the limits of a 32-bit signed int, a 64-bit signed long and the largest double.
109. **D** — In Python, x = 5 then x = 'five' is legal because the name is not tied to a type; the type is a property of each object and is checked when operations run. Python is still strongly typed: "5" + 3 raises TypeError rather than converting silently.
110. **B** — int to double is a widening conversion, so Java performs it automatically and d holds 10.0. 3.5 and 2.5 are double literals; assigning them to int, float or long is a narrowing conversion that needs a cast (or, for float, the literal 2.5f).
111. **A** — Java (like C++) uses block scope: a local variable exists from its declaration to the end of the enclosing { } block. Using it after the closing brace is a compile error such as "cannot find symbol".
112. **A** — Java gives fields default values (0, 0.0, false, null). Local variables get none, and the compiler's definite-assignment check rejects reading one before it is assigned ("variable might not have been initialized"). In C++, an uninitialised local int holds an indeterminate value.
113. **C** — A floating-point literal without a suffix is a double in Java, C++ and C#. To write a float literal you add f or F, as in 3.14f; this is why float f = 3.14; fails to compile in Java.
114. **D** — Python resolves names with the LEGB rule: the current function's local scope, then any enclosing function scopes, then module globals, and finally built-ins such as len and print. The first match wins.
115. **B** — Java keeps boolean separate from numeric types: if (1) and (int) true are both compile errors. In C++ and Python, by contrast, true/True converts to 1, which is a frequent cross-language trap.
116. **B** — In Python 3 the / operator always performs true division and returns a float, even for two ints. 7 / 2 is 3.5, and type(x).__name__ is 'float'. Use // for floor division.
117. **C** — int('42') converts the string to the integer 42 (explicit conversion), and 42 + 8 is 50. Without int(), '42' + 8 would raise TypeError.
118. **A** — int() on a float truncates toward zero, it does not round or floor. So int(-3.99) is -3. math.floor(-3.99) would give -4, which is the usual distractor.
119. **A** — Python is strongly typed: it does not convert implicitly between str and int, so '5' + 3 raises TypeError. Use int('5') + 3 for 8 or '5' + str(3) for '53'. JavaScript, by contrast, would give '53'.
120. **C** — The explicit cast (int) from double to int drops the fractional part, truncating toward zero, so 9.99 becomes 9. It does not round.
121. **B** — 130 does not fit in a byte (-128 to 127). The narrowing cast keeps only the low 8 bits: 130 is 1000 0010 in binary, which as a signed 8-bit value is 130 - 256 = -126.
122. **B** — Java int arithmetic wraps around silently on overflow; no exception is thrown. Adding 1 to Integer.MAX_VALUE (2,147,483,647) wraps to Integer.MIN_VALUE, -2,147,483,648.
123. **D** — In 'A' + 2 the char is promoted to int (binary numeric promotion), so the result is the int 65 + 2 = 67, and println(int) prints 67. Casting the sum, (char) ('A' + 2), would print C.
124. **B** — In Python bool is a subclass of int, with True equal to 1 and False equal to 0. So True + True + 1 is 1 + 1 + 1 = 3.
125. **B** — JavaScript coerces types implicitly. The * operator only works on numbers, so '5' * 2 converts '5' to 5 and gives 10. The + operator concatenates if either side is a string, so '5' + 2 gives '52'.
126. **A** — The assignment x = 20 inside f creates a new local variable, which shadows the global x only inside the function. f returns 20 and the global x is still 10.
127. **D** — let is block-scoped. The inner let x = 2 creates a separate variable that exists only inside the braces, so the outer x is still 1 when it is logged.
128. **C** — var is function-scoped, not block-scoped, so y declared inside the if block is visible after it and holds 5. With let or const instead of var, the console.log would throw a ReferenceError.

129. **B** — global n tells Python that n inside bump refers to the module-level variable, so each call changes it. Two calls take n from 1 to 3.
130. **D** — Because count is assigned inside inc, Python treats it as local for the whole function. count += 1 must first read that local, which has no value yet, so UnboundLocalError is raised. Adding global count fixes it.
131. **C** — Both operands of a / 2 are int, so C++ performs integer division and gets 3. Only then is 3 converted to 3.0 for b, and cout prints 3. Writing a / 2.0 or static_cast<double>(a) / 2 gives 3.5.
132. **B** — For a reference variable, final means a cannot be made to point to a different array. The array object itself is still mutable, so a[0] = 9 is legal and println prints 9. a = new int[3]; would fail.
133. **C** — Python has no enforced constants for ordinary variables. PI = 3 simply rebinds the name, so print shows 3. Tools such as linters, or typing.Final with a type checker, can flag it, but the interpreter does not.
134. **D** — int() accepts a string only if it contains an integer literal. '3.5' is not one, so int raises ValueError. To truncate it you must convert in two steps: int(float('3.5')) gives 3.
135. **D** — In c = c + 2 the char is promoted to int, and Java will not narrow int to char without a cast ("possible lossy conversion"). Compound assignment is defined as c = (char)(c + 2), so c += 2 compiles and makes c equal to C.
136. **A** — The multiplication is done entirely in int because every literal is an int; only the final result is widened to long. 86,400,000,000 overflows 32 bits and wraps to 500,654,080 before the assignment. Writing 24L * 60 * 60 * 1000 * 1000 fixes it.
137. **B** — x holds the string '10', not a number. For strings, * with an int means repetition, so '10' * 2 is '1010'. Reading input with input() gives strings, which is how this bug usually appears; convert with int() first.
138. **D** — Unsigned arithmetic in C++ is defined modulo 2^n, so 0 - 1 wraps to 2^32 - 1 = 4,294,967,295. Overflow of signed integers is undefined behaviour, but unsigned wrap-around is well-defined.
139. **A** — list is a built-in name, not a keyword, so it can be rebound. After list = [1, 2], the name finds the global variable before the built-in (LEGB), and calling a list object raises TypeError: 'list' object is not callable.
140. **A** — Java's int is 32-bit with a maximum of 2,147,483,647, so 5 billion does not fit. long is 64-bit (up to about 9.2 x 10^18). short and char are only 16-bit.
141. **A** — None is the single instance of the built-in class NoneType, so type(None) shows <class 'NoneType'>. Python has no null or void type; those names come from other languages.
142. **C** — float is a 32-bit IEEE 754 type with a 24-bit significand (about 7 decimal digits). double is 64-bit with a 53-bit significand (about 15 to 16 digits). The 4 and 8 figures are their sizes in bytes, not digits.
143. **D** — Each Java primitive has a wrapper class in java.lang: int has Integer, char has Character, and the rest use the type name with a capital letter (Double, Boolean and so on). Number is the abstract superclass of the numeric wrappers.
144. **C** — Collections such as ArrayList hold objects, not primitives, so list.add(5) silently becomes list.add(Integer.valueOf(5)). The reverse, Integer to int, is called unboxing.
145. **D** — The JLS lists long to float as a widening primitive conversion because float's range is larger, even though some precision can be lost. double to float, int to short and int to char are narrowing and need an explicit cast (except for compile-time constants that fit).
146. **D** — var asks the compiler to infer the type from the initialiser, so count is an int and Java remains statically typed. Assigning a String to it later is a compile error. var only works for local variables with an initialiser.
147. **C** — auto deduces the type from the initialiser. The f suffix makes 5.0f a float literal, so x is a float. Without the suffix, 5.0 would be a double.
148. **D** — const creates a block-scoped binding that must be initialised and cannot be reassigned. let is block-scoped but can be reassigned, and var is function-scoped. static is only used inside classes.
149. **C** — Scope is where a name can be used; lifetime is how long the storage exists. A static local can only be named inside its function, but it is initialised once and survives until the program ends, so it can count calls.
150. **A** — A static field belongs to the class itself, so all objects see the same value. Each object gets its own copy of an instance field, and locals and parameters exist only while their method runs.
151. **A** — Without a declaration, assigning to a name in a nested function creates a new local. nonlocal tells Python the name refers to the nearest enclosing function's variable. For module-level names you use global instead.
152. **B** — #define is plain text substitution done before compilation: it has no type and ignores scopes. A const variable is a real, typed object that the compiler checks. It cannot be changed after initialisation.

153. **A** — A tuple cannot be changed after it is created: you cannot assign to an element or append to it. Lists, dictionaries and sets are mutable and can be changed in place.
154. **B** — All ordinary JavaScript numbers share one type, number, stored as a double-precision float. Integers are exact only up to 2^53 - 1 (Number.MAX_SAFE_INTEGER). BigInt is a separate type written with an n suffix.
155. **B** — These decimals have repeating expansions in binary, like 1/3 in decimal. The nearest doubles are used, and their sum is 0.30000000000000004. Compare floats with a tolerance, for example math.isclose(a, b).
156. **C** — Scientific notation always produces a float in Python, so 1e3 is 1000.0. 1000 and 0x3E8 (hexadecimal) are ints with the same value, and "1e3" is a string.
157. **D** — static gives one shared copy, final stops reassignment, and constant names use UPPER_SNAKE_CASE. const is reserved in Java but unused, var cannot be used for fields, and #define is C/C++ preprocessor syntax.
158. **B** — Annotations are kept in area.__annotations__ but ignored when the function runs, so area("5") still executes. Static checkers like mypy read them to report mistakes before the program runs.
159. **C** — Primitive variables hold their values directly. b = a copies the number 5 into b, and changing b afterwards has no effect on a. Only reference types share an object between variables.
160. **B** — With 8 bits and no sign bit, there are 2^8 = 256 values from 0 to 255. The signed 8-bit range is -128 to 127.
161. **A** — CPython's float wraps a C double, so it behaves like Java's double: about 15 to 16 significant digits and a maximum near 1.8 x 10^308. Only Python's int has unlimited precision; exact decimals need the decimal module.
162. **C** — char is a 16-bit unsigned type holding UTF-16 code units from 0 to 65,535. All of Java's integer types byte, short, int and long are signed two's-complement types.
163. **C** — Inside the constructor the parameter shadows the field, so a plain x means the parameter. this.x names the field of the current object, so this.x = x; copies the argument into the field.
164. **D** — A string is falsy only when it is empty. "" gives False, but "False" is a non-empty five-character string, so it gives True. bool does not read the text.
165. **D** — The second argument to int is the base. "1010" read in base 2 is 8 + 2 = 10.
166. **D** — The right-hand side b, a is evaluated first into the tuple (9, 5), and then unpacked into a and b. That swaps the values without a temporary variable.
167. **C** — Python strings cannot be changed in place, so item assignment raises TypeError: 'str' object does not support item assignment. To get Lava, build a new string, such as "L" + s[1:].
168. **B** — Python has no block scope, so names bound inside a loop remain in the enclosing scope after it ends. The final pass has i = 2 and last = 20. In Java or C++, a variable declared in the loop header would not exist after the loop.
169. **A** — 0x1F is hexadecimal for 31. A leading 0 makes 010 an octal literal, which is 8. The sum is 39. Reading 010 as decimal 10 gives the trap answer 41.
170. **D** — c++ increments the char's code (97 to 98) and keeps the type char, because ++ includes an implicit cast back to char. println(char) prints the character, b.
171. **B** — Casting the char 'A' to int gives its code, 65. Casting the int 66 to char gives the character B. The String in the middle turns the rest into concatenation.
172. **B** — Java's Math.round returns the closest long, with ties rounded towards positive infinity: it behaves like Math.floor(x + 0.5). So 2.5 becomes 3 and -2.5 becomes -2.
173. **C** — Python 3's round uses round-half-to-even (banker's rounding): an exact .5 goes to the nearest even integer. 2.5 goes to 2 and 3.5 goes to 4. This differs from Java's Math.round.
174. **A** — typeof null returns "object", a long-standing quirk kept for compatibility. NaN (Not-a-Number) is still a value of the number type, so typeof NaN is "number".
175. **C** — parseInt reads digits from the start and stops at the first character that is not part of the number, giving 12. Number converts the whole string and returns NaN if any part is invalid.
176. **B** — Copy-initialising an int from a double is an implicit narrowing conversion that truncates towards zero, so x is 3. The char holds code 66, and cout prints a char as a character, B.
177. **B** — For the comparison f is widened to double, but the float nearest to 0.1 (0.100000001490116...) differs from the double nearest to 0.1 (0.1000000000000000055...). The values are not equal, so it prints false.

178. **B** — In Python, bool is a subclass of int, so True is an int instance with the value 1. Both expressions are therefore True. Java, by contrast, keeps boolean completely separate from int.
179. **C** — nonlocal n makes inc rebind outer's n rather than create a new local, so each call adds 1 to the same variable. Two calls give 2. Without nonlocal, n += 1 would raise UnboundLocalError.
180. **B** — A const binding cannot be reassigned. The assignment on line 2 throws a TypeError at run time, so console.log never runs.
181. **C** — A static local is initialised only once and keeps its value between calls. The three calls raise n to 1, 2 and 3, and the last value is printed. A non-static local would restart at 0 and print 1.
182. **C** — 1L << 40 is 2^40, a 1 followed by 40 zero bits. Casting a long to int keeps only the low 32 bits, which are all zero, so the result is 0. Narrowing casts do not clamp to Integer.MAX_VALUE.
183. **D** — 9007199254740993 is 2^53 + 1, beyond Number.MAX_SAFE_INTEGER. A double cannot represent it, so the literal rounds to 9007199254740992 and both sides are equal. BigInt literals (with an n suffix) keep them distinct.
184. **D** — == on Integer objects compares references. Autoboxing uses Integer.valueOf, and the JLS requires values from -128 to 127 to come from a shared cache, so a and b are the same object. 128 is outside the guaranteed cache, so c and d are normally separate objects. Use equals() to compare values.
185. **A** — Chained assignment binds both names to the same list object. Appending through a changes that one list, so b shows [1]. Use a, b = [], [] for two separate lists.
186. **C** — [[0] * 2] * 2 repeats a reference to the same inner list, so both rows are one object. Changing grid[0][0] is visible through grid[1][0] too. Build rows with [[0] * 2 for _ in range(2)] to get independent lists.
187. **A** — A variable declared in a for statement's header is scoped to that statement. After the loop, i is unknown and javac reports 'cannot find symbol'. Python would keep the variable, but Java's block scope does not.
188. **A** — Inside the inner block, the innermost x (3) shadows the others, while ::x always names the global x (1). After the block ends, its x is gone and the local x of main (2) is visible again. The output is 3, 1, 2.
189. **B** — var is function-scoped, so all three arrow functions close over the same i. They run after the loop has finished, when i is 3. Declaring the loop variable with let creates a new binding per iteration and prints 0,1,2.
190. **A** — 0.1 has no exact binary representation, so every addition carries a tiny rounding error and the sum ends at 0.9999999999999999. Compare floats with math.isclose(total, 1.0) or use the decimal module for exact decimal arithmetic.
191. **A** — Copy initialisation in line 1 allows the narrowing and truncates to 3 (compilers may warn). C++11 list initialisation with braces makes narrowing ill-formed, so line 2 is an error. This is a key reason to prefer brace initialisation.
192. **A** — int to float is a widening conversion, so no cast is needed, but float has only a 24-bit significand. 2^24 + 1 cannot be stored and rounds to 16777216.0, so the comparison is true. Widening does not guarantee no loss of precision.
193. **A** — input() always returns a string, here "20". Python is strongly typed, so "20" + 1 raises TypeError rather than converting either side. Writing int(input()) gives 21.
194. **B** — == applies type coercion, converting "5" to the number 5 before comparing, so it is true. === compares type and value without conversion, and a string is not a number, so it is false.
195. **D** — A blank final local may be declared without a value and assigned exactly once later, so lines 1 and 2 are fine. Line 3 is a second assignment, and javac reports that limit might already have been assigned.
196. **D** — int(price) converts the string "250" to the integer 250 before the multiplication, so the result is 250 * 2 = 500. Without int(), "250" * 2 would repeat the string as 250250.
197. **D** — inner has no local x, so Python follows the LEGB rule: Local, then Enclosing (outer's x), then Global, then Built-in. It finds outer's x first and returns "enclosing".
198. **B** — In a comparison of int with unsigned int, the usual arithmetic conversions turn the int into unsigned, so -1 becomes 4294967295. That is not less than 1, so the result is false and cout prints 0. Compilers usually warn about signed/unsigned comparisons.
199. **D** — Multiplicative operators (*, /, %) bind tighter than additive ones (+, -), which bind tighter than comparisons (==), and assignment has very low precedence. So a = b + c * d == e groups as a = ((b + (c * d)) == e).
200. **A** — // is floor division, so 7 // 2 is 3 and -7 // 2 is -4 (rounded down, toward minus infinity). In C++ and Java, // starts a comment, and their int division truncates toward zero instead.

201. **A** — % is the modulus (remainder) operator: 17 = 3 × 5 + 2, so 17 % 5 is 2. All three languages agree for positive operands; they differ only when an operand is negative.
202. **D** — The assignment operator is right-associative, so the expression groups as a = (b = (c = 7)). Each assignment expression produces the assigned value, so all three variables end up as 7.
203. **C** — Exponentiation is right-associative, following the maths convention, so 2 ** 3 ** 2 is 2 ** 9 = 512, not 8 ** 2 = 64. Most other binary operators, such as - and /, are left-associative.
204. **C** — Java (and C++) use && for logical AND, while Python uses the keyword and. Python has no && operator, and & is the bitwise AND in all three.
205. **C** — For a && b, if a is false the result must be false, so b is skipped; for a || b, if a is true, b is skipped. This lets code like p != null && p.size() > 0 avoid errors. Python's and and or behave the same way.
206. **A** — Both add 1 to x. The postfix form x++ gives the expression the value x had before the increment, and the prefix form ++x gives the updated value. On a statement of its own, x++; and ++x; have the same effect.
207. **C** — ^ is bitwise XOR in all three languages: each result bit is 1 when the two input bits differ. So 5 ^ 3 is 6, not 125. Powers use ** in Python and pow() / Math.pow() in C++ and Java.
208. **D** — A left shift by n moves every bit n places left and fills with zeros, which multiplies by 2^n. 2^3 is 8, so 5 << 3 is 40. A right shift by n divides by 2^n (rounding down for non-negative values).
209. **A** — Python does not have increment or decrement operators. x++ on its own line is a syntax error; write x += 1 instead. ++x is legal but is just two unary plus signs, so it does nothing to x.
210. **B** — With boolean operands, Java's & is a logical AND that is not short-circuit: both sides are always evaluated, and the result is a boolean. && evaluates b only if a is true. The difference matters when b has a side effect or could throw.
211. **A** — && sits below the relational and equality operators, so a < b && b < c groups as (a < b) && (b < c) with no brackets needed. *, + and unary ! all bind tighter than <.
212. **C** — Python chains comparisons, so a < b < c means a < b and b < c. In C++ and Java, 1 < x < 5 does not mean this: C++ compares the bool result with 5 (always true), and Java rejects it as a compile error.
213. **D** — >>> is the unsigned right shift: it fills with 0 bits, so a negative int becomes a large positive one, and -8 >>> 1 is 2147483644. >> is the arithmetic (signed) right shift that copies the sign bit, so -8 >> 1 is -4. C++ has no >>> operator.
214. **D** — a == b calls the equality method and asks whether the values are equal. a is b asks whether a and b are the very same object in memory. Two separate lists [1, 2] are == but not is. Use is mainly for None checks, as in x is None.
215. **B** — * has higher precedence than +, so 3 * 4 is computed first and gives 12, then 2 + 12 = 14. (2 + 3) * 4 would give 20.
216. **D** — 10 // 3 is floor division, giving 3, and 10 % 3 is the remainder, 1. The identity (a // b) * b + a % b == a always holds: 3 × 3 + 1 = 10.
217. **A** — Python's // rounds down toward minus infinity, so -7 // 2 is -4 (since -3.5 rounds down to -4). The remainder takes the sign of the divisor: -7 - (-4 × 2) = 1.
218. **A** — Java int division truncates toward zero, so -7 / 2 is -3, and % takes the sign of the dividend: -7 - (-3 × 2) = -1. The same rules apply in C++ since C++11. Python gives -4 and 1 for the same numbers.
219. **B** — y = x++ assigns the old value of x (5) to y, then increments x to 6. So x is 6 and y is 5.
220. **B** — ++x increments x to 6 first and yields 6, then 6 * 2 = 12 is stored in y. x is modified only once in the statement, so the behaviour is well defined.
221. **C** — In binary, 6 is 110 and 3 is 011. AND keeps bits set in both: 010 = 2. OR keeps bits set in either: 111 = 7. XOR keeps bits set in exactly one: 101 = 5.
222. **D** — 5 << 2 shifts 101 left two places to 10100, which is 5 × 4 = 20. 20 >> 3 shifts 10100 right three places to 10, which is 20 // 8 = 2; shifts on integers always give an integer.
223. **B** — ~ is bitwise NOT. For integers in two's complement, ~x equals -x - 1, so ~5 is -6. It is not logical negation and does not just flip the sign.
224. **B** — + is left-associative. In 1 + 2 + '3', 1 + 2 is numeric addition (3), then 3 + '3' concatenates to '33'. In '1' + 2 + 3 the first + already involves a string, giving '12', and then '123'.

225. **B** — Python's conditional expression is A if condition else B. 7 % 2 is 1, which is truthy, so the expression gives 'odd'.
226. **D** — In a compound assignment the whole right-hand side is evaluated first. 3 * 2 is 6, so x -= 6 makes x 10 - 6 = 4.
227. **A** — x != 0 is False, so and short-circuits and check() is never called; nothing inside the if runs. Only end is printed. If the operands were swapped, check() would run and print checked.
228. **B** — ** has higher precedence than unary minus in Python, so -2 ** 2 is -(2 ** 2) = -4. Write (-2) ** 2 to get 4.
229. **D** — a++ yields 5 and sets a to 6. Then ++a sets a to 7 and yields 7. So b = 5 + 7 = 12 and a is 7. Java defines this order; in C++ modifying a twice in one expression like this is undefined behaviour, so never write it there.
230. **A** — not has lower precedence than comparison operators in Python, so the expression is not (1 == 2), which is not False, which is True. Misreading it as (not 1) == 2 would give False == 2, which is False.
231. **D** — + is evaluated left to right. 1 + 2 is int addition, giving 3. 3 + "3" is string concatenation, "33". From then on, the left operand is a String, so 4 and 5 are appended as text: "3345".
232. **B** — In C++ (and Java, C and JavaScript) the equality operators have higher precedence than bitwise &. So the condition is x & (1 == 0), which is 4 & 0 = 0, false, and the else branch prints odd. Write (x & 1) == 0. Python is different: its comparisons bind looser than &, so the same expression works there.
233. **C** — and evaluates left to right and short-circuits. In the first version d != 0 is False, so n / d is never evaluated. In the second, n / d runs first and raises ZeroDivisionError. Ordering guards before risky operations is a standard idiom.
234. **C** — Python's or and and return one of their operands, not just True or False. 0 or 'x': 0 is falsy, so or returns its right operand, 'x'. 3 and 0: 3 is truthy, so and returns its right operand, 0.
235. **B** — == uses loose equality with type coercion, so 0 == '0' converts '0' to 0 and is true. === is strict and checks the type too, so 0 === '0' is false. null == undefined is a special case defined as true for loose equality.
236. **C** — Python has no -- operator. --x parses as -(-x), two unary minus signs, which equals 5 and leaves x unchanged. -(-x) is the same thing written explicitly, so both print 5.
237. **C** — Arithmetic operators bind tighter than shifts, so 1 << 2 + 1 is 1 << (2 + 1) = 1 << 3 = 8. Reading it left to right as (1 << 2) + 1 gives 5. C++ and Java use the same precedence here.
238. **A** — Java evaluates i++ first: it yields the old value 5 and increments i to 6. Then the assignment stores that 5 back into i, overwriting the 6. So i is 5. In Java this is well defined; in C++ before C++17, i = i++ was undefined behaviour.
239. **A** — The conditional operator cond ? a : b is the only ternary operator in C++ and Java: it takes a condition and two result expressions. && and += are binary operators, and [] takes just an array and an index.
240. **B** — In Python 3 the / operator always performs true division and returns a float, so 7 / 2 is 3.5. Floor division needs //, which would give 3. The answer 3 comes from Python 2 or C-style integer division.
241. **C** — >= compares two values and yields true or false, so it is a relational operator. = and += are assignment operators, and & is bitwise AND.
242. **D** — n % 2 is the remainder after dividing by 2, and it is 0 exactly when n is even. For a negative odd n, C++ and Java give -1 while Python gives 1, but an even n gives 0 in all three, so the test works everywhere.
243. **A** — Python overloads * for sequences: a string times an int repeats it, so "ab" * 3 is "ababab". Python never converts the string to a number here. A TypeError only appears for str * str or str * float.
244. **B** — in is the membership operator. It returns True if x equals some element of a list, tuple or set, is a key of a dict, or is a substring of a string. Type checks use isinstance, and identity uses is.
245. **C** — Binary numeric promotion converts the int 5 to double because the other operand is a double literal, so the result is the double 7.0. A literal like 2.0 is a double, not a float, and + only concatenates when one operand is a String.
246. **D** — x += 1 is Python's augmented assignment for incrementing. x++ is a syntax error, ++x parses as +(+x) and changes nothing, and x =+ 1 simply assigns +1 to x.
247. **A** — Precedence and associativity fix the structure of an expression: f() + g() * h() groups as f() + (g() * h()). They do not say whether f() or g() is called first. Java and Python evaluate operands left to right, while C++ leaves most operand order unspecified, which is why side effects inside one C++ expression are risky.

248. **B** — / is left-associative, so operators of equal precedence group from the left: (100 / 10) / 5 = 10 / 5 = 2. Grouping from the right would give 100 / 2 = 50, which is why associativity matters for / and -. Operands are never reordered.

249. **C** — An assignment expression such as if (n := len(data)) > 10: stores len(data) in n and yields the same value for the comparison. A plain = is a statement, so it cannot appear inside an if condition.

250. **D** — obj instanceof Type yields true if obj is non-null and its class is Type or a subclass or implementer of it, and false otherwise, including for null. It does not cast; casting is the separate (Type) operator, and getClass().getName() gives the name.

251. **A** — sizeof is a built-in unary operator whose value the compiler works out from the type; sizeof(char) is always 1. It counts bytes, not elements (you divide by the size of one element to count an array), and strlen, not sizeof, measures a C string.

252. **B** — Python gives not the highest of the three, then and, then or, mirroring !, && and || in C++ and Java. So not a or b and c means (not a) or (b and c).

253. **C** — ?? is the nullish coalescing operator: it falls back to b only for null or undefined. That is its key difference from ||, which also replaces 0, "", false and NaN. So 0 ?? 5 is 0, but 0 || 5 is 5.

254. **D** — Java defines int arithmetic as 32-bit two's complement with silent wrap-around, so 2147483647 + 1 becomes -2147483648, which is Integer.MIN_VALUE. No exception is thrown (Math.addExact would throw one). In C++ signed overflow is undefined behaviour, so this guarantee is specific to Java.

255. **A** — A compound assignment evaluates the whole right-hand side first and then combines it with x, so x *= y + 1 means x = x * (y + 1), not (x * y) + 1. With x = 3 and y = 4 the result is 15, not 13. y is not changed.

256. **B** — && binds more tightly than ||, just as * binds more tightly than +. So a || b && c means a || (b && c). Left-to-right grouping only applies among operators of the same precedence.

257. **C** — For reference types, == tests identity: whether both variables point at the same object. Two strings built separately, for example with new String or by run-time concatenation, are different objects, so == is false while s1.equals(s2) is true. Only literals are interned and can share one object.

258. **D** — The built-in comma operator evaluates its left operand, discards it, then evaluates the right operand, which becomes the result. The left side is fully finished before the right starts. So int x = (3, 7); sets x to 7. It is mostly seen in for headers such as i++, j--.

259. **A** — Java promotes byte operands to int, so b + 5 has type int, and assigning an int expression to a byte needs an explicit (byte) cast. The language specification defines E1 op= E2 as E1 = (T)((E1) op (E2)) with the cast built in, so b += 5 compiles. 15 fits in a byte, so overflow is not the reason.

260. **B** — ~x flips every bit, which in two's complement equals -x - 1. Adding 1 gives -x, which is how hardware negates a number. ~x - 1 gives -x - 2, x ^ 1 only flips the lowest bit, and x >> 1 roughly halves the value.

261. **C** — 1 << k has only bit k set, and ~ turns it into a mask with every bit set except bit k. ANDing x with that mask forces bit k to 0 and keeps the rest. | sets bit k, ^ toggles it, and x & (1 << k) keeps only bit k instead of clearing it.

262. **D** — & works bit by bit: 110 & 011 = 010, which is 2. && treats each operand as a truth value, and any non-zero int converts to true, so 6 && 3 is true (printed as 1). Java would reject 6 && 3, but C++ allows it through the int-to-bool conversion.

263. **A** — - is left-associative, so 8 - 3 - 2 is (8 - 3) - 2 = 3. Grouping from the right, 8 - (3 - 2), would wrongly give 7.

264. **B** — * binds tighter than +, so "ab" * 3 runs first and gives "abababab"; then "c" is concatenated to give "abababc". Repeating "abc" would need parentheses: ("ab" + "c") * 3.

265. **C** — x **= 2 makes x 16. x //= 3 is floor division, so 16 // 3 = 5, and because both operands are ints the result stays an int.

266. **D** — // floors the quotient: 7.5 / 2 = 3.75, which floors to 3. When either operand is a float the result is a float, so Python prints 3.0, not 3.

267. **A** — x %= 4 sets x to 10 % 4 = 2. x <<= 1 shifts left by one bit, doubling 2 to 4.

268. **B** — ** is exponentiation, so 2 ** 10 is 1024. ^ is bitwise XOR, not power: 0010 ^ 1010 = 1000, which is 8.

269. **C** — * and / share precedence and group left to right, so 7 / 2 runs first. Both are ints, so it truncates to 3; only then is 3 promoted for 3 * 2.0 = 6.0. The result is a double, so Java prints 6.0, not 6.

270. **D** — In c + 1 the char is promoted to int (its code is 97), so the sum is the int 98 and println prints 98. To get the letter b you need a cast: (char)(c + 1).

271. **A** — c += 2 includes an implicit cast back to char, so c becomes the character with code 99, which is 'c', and println(char) prints the character. Writing c = c + 2 without a cast would be the compile error.

272. **B** — Python's >> is an arithmetic shift: x >> 1 equals x // 2, which floors. So -8 >> 1 is -4, 8 >> 1 is 4, and -9 >> 1 is floor(-4.5) = -5, not the -4 that truncation would give.

273. **C** — bool is a subclass of int, with True equal to 1. * binds first, so True * 3 is 3, then True + 3 is 4. Arithmetic on bools returns an int, so the output is 4, not True. 6 would need (True + True) * 3.

274. **D** — a / b divides two ints, so it truncates to 2 before the assignment happens. Only then is 2 converted to the double 2.0. To get 2.5, convert an operand first: static_cast<double>(a) / b.

275. **A** — ?: is right-associative, so the expression reads x > 5 ? (x > 10 ? "big" : "mid") : "small". 7 > 5 is true, so the inner test runs; 7 > 10 is false, giving "mid". Nesting without parentheses is legal, though parentheses help readability.

276. **B** — The whole right side b, a + b is evaluated first using the old values, giving (5, 8); only then are a and b assigned. Updating a first and then computing a + b would give 5 10.

277. **D** — x += x * 2 means x = x + (x * 2). The right-hand side uses the old x: 5 * 2 = 10, then 5 + 10 = 15. Reading it as (x + x) * 2 gives the wrong 20.

278. **A** — In two's complement, -x equals ~x + 1, which keeps the lowest set bit of x and flips every bit above it. ANDing x with -x therefore isolates the lowest set bit. 0b10110 is 22, and its lowest set bit is 0b10, which is 2.

279. **B** — ?? replaces only null and undefined, so null ?? "d" is "d" but 0 ?? "d" keeps 0. || replaces any falsy value, and 0 is falsy, so 0 || "d" is "d".

280. **C** — Java first saves x's current value, 3. Then x++ * 2 yields 3 * 2 = 6 and bumps x to 4. Finally 3 + 6 = 9 is assigned to x, overwriting the 4. Using the bumped value 4 would give 10. Avoid writing this in C++, where such side effects are not safe to rely on.

281. **D** — Conditional expressions group from the right: 5 if 0 else (3 if None else 1). 0 is falsy, so the else part runs. None is also falsy, so the inner expression gives 1.

282. **A** — a++ yields 10 and makes a 11, so 10 / 3 = 3 (integer division). Then --a makes a 10 again and yields 10, so 10 % 3 = 1. The sum is 3 + 1 = 4.

283. **B** — Division has higher precedence than addition, so a + b / 2 is a + (b / 2) = 4 + 4.0 = 8.0. The fix is (a + b) / 2, which gives 6.0.

284. **C** — Python rejects assignment with = inside an if condition at compile time, so no line runs. This prevents the C-style bug of writing = for ==. Use == to compare, or := if an assignment expression is really wanted.

285. **D** — total / 2 divides two ints and truncates to 3 before the value reaches avg. Making one operand a double, as in total / 2.0, forces floating-point division and gives 3.5.

286. **A** — JavaScript numbers are IEEE 754 doubles. 0.1 and 0.2 are stored as close approximations, so their sum is 0.30000000000000004, which differs from the double nearest 0.3. Compare floats with a tolerance, such as Math.abs(a - b) < 1e-9.

287. **B** — For a list, y += [3] calls the in-place extend, so the single list that both x and y refer to gains a 3. By contrast y = y + [3] would build a new list and leave x as [1, 2].

288. **C** — < is left-associative, so a < b < 2 is (a < b) < 2. a < b is true, which converts to 1, and 1 < 2 is true, printed as 1. C++ never chains comparisons the way Python does; write a < b && b < 2 for that.

289. **D** — / always gives a float in Python 3, so 10 / 5 is 2.0. == compares numeric values across int and float, so 2.0 == 2 is True. The type, however, is float.

290. **A** — With boolean operands, & is a logical AND that does not short-circuit. c > 0 is false, but ++c > 0 still runs and sets c to 1. With && the right side would be skipped and c would stay 0.

291. **B** — Unary + uses the Number conversion: the empty string becomes 0, "42" becomes 42, and "3a" is not a valid numeric string, so it becomes NaN. parseInt("3a") would give 3, but unary + does not parse a prefix.

292. **C** — i -= i - 3 means i = i - (i - 3). The right side is 10 - 3 = 7, so i becomes 10 - 7 = 3. Expanding it as i - i - 3 without the parentheses gives the wrong -3.

293. **D** — and binds tighter than or, so the expression is f(0) or (f(2) and f(0)). f(0) prints 0 and is falsy, so or evaluates its right side. f(2) prints 2 and is truthy, so and evaluates f(0), which prints 0 and becomes the result. The final print adds that 0.

294. **A** — In Python, & binds tighter than ==, so the line is (x & 1) == 0, which is 0 == 0, True. In C++, == binds tighter than &, so it is x & (1 == 0), which is 4 & 0 = 0. The same text means two different things.

295. **B** — lo + hi is 4,100,000,000, which exceeds the int maximum of 2,147,483,647, so it wraps to -194,967,296. Dividing by 2 gives -97,483,648. The safe form is lo + (hi - lo) / 2, a classic binary-search bug.
296. **C** — When the second and third operands of ?: are both numeric, Java applies binary numeric promotion to the whole expression. One branch is a double, so the type is double and the chosen value 1 becomes 1.0.
297. **D** — x++ yields 5 and sets x to 6. 5 * 2 is 10. The right operand of + then reads x, which is now 6, giving 16. Finally 16 is assigned to x, which replaces the 6 set by x++.
298. **A** — A do-while loop is exit-controlled: the body runs first and the condition is tested afterwards, so the body always executes at least once. while, for and for-each test before the body, so they can run zero times.
299. **B** — Without break, control does not leave the switch; it continues into the next case's statements regardless of that case's label. This is called fall-through. It is legal in both languages (Java and C++17 compilers may only warn).
300. **D** — continue abandons the remaining statements of the current pass only. The loop then tests its condition (while) or takes the next item (for). break, not continue, ends the loop.
301. **D** — pass is a null statement, for example in an empty if body or a placeholder loop. continue and break change loop flow, and skip is not a Python keyword at all.
302. **B** — A loop's else clause runs when the loop ends because its items (or condition) ran out. If break ends the loop, the else is skipped. It also runs when the loop body runs zero times, because no break happened.
303. **A** — The C++ switch condition must be of integral or enumeration type (or a class type with a single conversion to one). std::string and double are not allowed; you need an if/else chain for those. char and enums are integral-like and are allowed.
304. **C** — Java added String to switch in Java 7, alongside int, char, short, byte, their wrappers and enums. Java switch still does not accept long, float, double or boolean. C++ switch does not accept std::string.
305. **D** — for and while both test the condition before each pass, so their bodies can run zero times. do-while is exit-controlled: it tests after the body.
306. **D** — This is the dangling-else rule: an else binds to the closest unmatched if. Indentation does not matter to the compiler, so use braces to make the intended pairing explicit.
307. **A** — In match, the first case whose pattern matches runs and the statement ends. There is no fall-through and no break is needed. A wildcard case _ is optional; if nothing matches, nothing happens.
308. **C** — Python's conditional expression is written value_if_true if condition else value_if_false. Python has no ?: operator and no then keyword or built-in iif function.
309. **C** — A labelled break terminates the statement with that label, here the outer loop, so control goes to the first statement after it. continue outer would instead skip to the next pass of the outer loop. C++ has no labelled break.
310. **D** — while True guarantees the body runs, and the break check at the end plays the role of the do-while condition. The other patterns still test before the first pass or change nothing about the count.
311. **A** — JavaScript switch uses strict equality, so case 1 does not match the string "1". It is not Object.is: NaN never matches a case NaN, and 0 matches -0.
312. **B** — && short-circuits: if p != null is false, the whole condition is already false and p.size() is skipped. Swapping the order, or using the non-short-circuit &, would throw a NullPointerException.
313. **C** — In C++ and Java a variable declared in the for header is scoped to the loop. Python has no block scope: the for target is an ordinary variable in the enclosing function or module and keeps the last value assigned to it.
314. **C** — range(2, 11, 3) starts at 2 and adds 3 while the value stays below 11: 2, 5, 8. The next value, 11, equals the stop and is excluded.
315. **B** — The inner loop runs i times for i = 0, 1, 2, 3, 4, giving 0 + 1 + 2 + 3 + 4 = 10. 25 would be a full 5 x 5 grid.
316. **B** — x is 2, so execution starts at case 2: y += 2 makes y = 2. There is no break, so it falls through to case 3: y += 3 makes y = 5, then break leaves the switch before default.
317. **D** — Even values hit continue and print nothing. Odd values 1, 3 and 5 print. At i = 7 the test i > 6 is true, so break ends the loop before 7 is printed.
318. **D** — n-- > 0 compares the old value and then decrements. The test is true for n = 5, 4, 3, 2, 1, so c reaches 5. The sixth test compares 0 > 0 (false) but still decrements, leaving n = -1.

319. **C** — i takes the values 1, 3, 9, 27, 81. The loop checks i < 50 before each pass; 27 < 50 lets it multiply once more to 81, then 81 < 50 is false and the loop stops.
320. **C** — No number in [3, 5, 7] is even, so break never runs. A for loop's else runs when the loop ends without break, so it prints no even.
321. **D** — A do-while runs the body before testing. i becomes 11, then 11 < 5 is false and the loop ends, so 11 is printed.
322. **B** — break exits only the inner loop. For i = 0 the inner loop breaks at j = 0 (0 added). For i = 1 it counts j = 0 then breaks (1 added). For i = 2 it counts j = 0 and 1 then breaks (2 added). Total 3.
323. **A** — An if/elif chain stops at the first true condition. 85 >= 50 is true, so grade is set to C and the later, stricter tests are never checked. The tests are in the wrong order for a grading scheme.
324. **A** — The else belongs to the nearest unmatched if, which is if (b > 1). The outer test a > 10 is false, so the whole inner if/else is skipped and only Z is printed.
325. **B** — switch compares with ===. The string "1" is not strictly equal to the number 1, so the first case is skipped; it matches case "1" and prints string, then break ends the switch.
326. **D** — Assigning to i inside the body does not affect the loop. Each pass, for takes the next value from range(3) and rebinds i, so the output is 0 1 2.
327. **A** — i takes 1, 2, 4, 8, 16, 32, 64, each <= 100, so count reaches 7. The next value, 128, fails the test.
328. **C** — Only the indented print(i) is inside the while body. i never changes inside the loop, so i <= 5 stays true and 1 is printed endlessly. Indenting i += 1 fixes it.
329. **D** — x = 0 is an assignment, not a comparison. It stores 0 in x and its value, 0, converts to false, so the else branch prints nonzero. In Java the same line will not compile, because an int is not a boolean.
330. **B** — The semicolon right after the for header is an empty loop body, so the loop runs three times doing nothing. The braces that follow are an ordinary block that runs once. It compiles because the block does not use i.
331. **C** — found becomes True at -1, but the next items 4 and 2 run found = False. The flag ends as the result for the last item only. Remove the else branch, or break once found is True.
332. **C** — var is function-scoped, so all three callbacks close over the same i. They run after the loop finishes, when i is 3. Declaring the loop variable with let creates a fresh binding per iteration and prints 0 1 2.
333. **A** — An unsigned int can never be negative, so the condition i >= 0 cannot fail. Unsigned arithmetic is defined modulo 2^n, so 0 - 1 becomes UINT_MAX and the loop keeps going. Use a signed int or a different condition.
334. **A** — range(5, 0) counts upwards from 5 and stops before 0, so it is empty. A countdown needs a negative step: range(5, 0, -1).
335. **A** — n goes 10 to 7 (no break) then 7 to 4, where n == 4 triggers break. Because the loop ended by break, the while's else is skipped, so only 4 is printed.
336. **B** — if (score > 50); has an empty statement as its body. The cout line is a separate statement that runs unconditionally. Compilers often warn about this, so read their warnings.
337. **B** — The original returns 1 when both are positive, 2 when a > 0 but b is not, and 3 whenever a <= 0. Only the first expression keeps a > 0 as the test for 2. Check the others with a = 0: they ignore that a must be positive.
338. **A** — A counted for loop keeps the initialisation, test and update in one header, which suits a known count such as 10 repetitions or every index of an array. while and do-while fit loops whose end depends on something discovered during the loop, and switch is selection, not iteration.
339. **B** — break leaves only the innermost loop (or switch) that contains it, and execution continues with the statement after that loop. Skipping to the next iteration is what continue does, and leaving several loops needs a labelled break in Java or a flag/return in C++.
340. **C** — Statements run one after another (sequence), a choice picks a path (selection: if, switch) and a loop repeats (iteration: for, while). These three are enough for any computable algorithm, which is why goto is not needed.
341. **D** — Python spells the extra branch elif, as in if x > 0: ... elif x == 0: ... else: .... Writing else if on one line is a syntax error, elseif is not a keyword, and case belongs to match statements.
342. **A** — Control jumps to default only when no case constant equals the switch value. Because labels are just entry points, a case without break placed above default can also fall through into it. default is optional and can appear anywhere in the switch.

343. **B** — Every part of a for header is optional, and an empty condition counts as true, so for (;;) repeats until a break, return or exception. while (false) never runs its body, and the two counted loops stop after 10 iterations.
344. **C** — range(stop) starts at 0 and stops before stop, so range(5) gives five values, 0 to 4. That matches the C-style loop for (int i = 0; i < 5; i++).
345. **D** — return leaves the whole function immediately, whatever loops it is inside. break leaves only the innermost loop, continue skips to the next iteration, and Python's pass does nothing.
346. **A** — C++ inherits C's rule that a scalar condition is true when non-zero, so if (1) always runs. Java has no int-to-boolean conversion, so the condition must be a boolean expression; if (true) or if (x == 1) compile, if (1) does not.
347. **B** — Empty containers, 0, 0.0, None and False are falsy. A non-empty string is truthy even if it contains "0" or a space, and a list with one element is truthy even if that element is 0.
348. **C** — With ->, only the code on the right of the matching label runs; there is no fall-through to the next case. Colon labels keep the old behaviour, where execution continues into later cases until a break.
349. **D** — Case labels in one switch must be distinct constant expressions, and the compiler rejects duplicates such as two case 3: labels. Each value can therefore map to only one entry point.
350. **A** — A labelled continue jumps to the next iteration of the labelled loop. The inner loop is abandoned for this round, the outer loop's update and test run, and the inner loop then starts again from scratch. Exiting both loops is what break outer; does.
351. **B** — In a range-based for loop, int x declares a new variable initialised with a copy of each element, so doubling it changes only the copy. Declaring int& x makes x a reference to the element itself. Python's for x in lst: behaves like the copy version for ints, since rebinding x does not change the list.
352. **C** — _ is the wildcard pattern: it always matches and binds nothing, so it is placed last as the catch-all branch. Python checks cases from top to bottom and runs only the first one that matches.
353. **D** — continue jumps straight back to the condition test, skipping everything below it in the body, including a manual increment. In a for loop the update part of the header still runs, which is why the same continue is safe there.
354. **A** — A switch compares one value for equality against constant labels; it cannot test ranges such as marks >= 80. Menu numbers, chars and enum constants are exactly the discrete values switch is designed for.
355. **B** — enumerate pairs each element with a counter that starts at 0 by default, so i is the position and x the element. It replaces the C-style pattern for i in range(len(items)): x = items[i]. Dict key-value pairs come from items.items().
356. **D** — A String switch matches by content, as if by equals() (the compiler uses hashCode() first for speed, then equals()). A string built at run time therefore still matches its case. Switching on a null String throws NullPointerException.
357. **A** — A bare name in a pattern captures, it does not compare: case limit: matches anything and rebinds limit to n. Python even rejects it with a SyntaxError if other cases follow, because they become unreachable. To compare with a variable, use a guard (case x if x == limit:) or a dotted name such as Config.LIMIT.
358. **B** — A list iterator keeps an index. After lst.remove(x) every later element moves one place left, and the iterator's next index jumps over the element that moved into the gap. No error is raised for lists (dicts and sets do raise RuntimeError). Iterate over a copy, lst[:], or build a new list instead.
359. **C** — Java's definite-reachability rules say no statement can follow break, continue, return or throw in the same block, and the compiler reports "unreachable statement" as an error. C++ compilers accept such dead code, at most with a warning.
360. **D** — The whole switch body is one scope, so y is visible at case 2. Jumping there would skip y's initialisation, which C++ forbids ("jump to case label crosses initialization"). Putting { } around case 1's statements gives y its own scope and the error disappears.
361. **B** — i runs 1, 2, 3, 4, 5 because the test is i <= 5, and each value is added: 1 + 2 + 3 + 4 + 5 = 15. Stopping at i < 5 would give 10.
362. **C** — 0 is falsy in Python, so if x: is false and the else branch prints no. Any non-zero number would be truthy.
363. **D** — i goes 0, 4, 8, 12. At 8 the test 8 < 10 is true, so i becomes 12; then 12 < 10 is false and the loop stops. The final value overshoots 10.

364. **A** — The loop visits each character; only the digits 1, 2 and 3 pass isdigit(). int(ch) turns each into a number before adding, so the total is 1 + 2 + 3 = 6, not the string 123.
365. **B** — i takes 0, 1 and 2, and each is appended to out. When i is 3, break runs before the append and ends the loop, so 3 and 4 are never added.
366. **C** — 5 matches no case, so control jumps to default, wherever it sits. There is no break after it, so execution falls through into case 1, appends 1, and stops at that break. case 2 never runs.
367. **D** — The continue skips every pair where i >= j, so only pairs with i < j are counted. Out of the 16 pairs of values 0 to 3, there are 4 * 3 / 2 = 6 such pairs.
368. **A** — Integer division halves n each pass: 100, 50, 25, 12, 6, 3, 1. That is 6 divisions before n > 1 becomes false. A halving loop runs about log2(n) times.
369. **B** — Each pass adds 1 to i first, so continue cannot cause an infinite loop. Multiples of 3 are skipped by continue, and the loop breaks at the first i above 7 that is not a multiple of 3, which is 8. The printed values are 1 2 4 5 7.
370. **C** — The body runs before the test: n becomes 2, 4, 6, then 8. After n is 6 the test 6 < 7 is true, so one more pass makes it 8, and 8 < 7 ends the loop.
371. **D** — The | in case "go" | "move": is an OR pattern, so "move" matches the first case. match runs only the first matching case and never falls through, so case _ is not reached.
372. **A** — Each value is appended first. For k = 2, continue skips the "-" for that pass only, so the output builds as 1-, 1-2, 1-23-.
373. **B** — ?: groups from the right, so the expression is a % 2 == 0 ? "even" : (a > 5 ? "big odd" : "small odd"). 7 is odd, so the inner test runs, and 7 > 5 selects "big odd".
374. **C** — range(len(lst) - 1, -1, -1) is range(2, -1, -1), which gives 2, 1, 0. The stop -1 is excluded, so index 0 is still visited and the list is printed backwards.
375. **D** — For each i, j = 1 is counted, then j = 2 triggers continue outer, which abandons the inner loop and moves to the next i. So each of the 3 outer iterations adds exactly 1.
376. **A** — The inner loop starts at j = i. For i = 0 every term is 0. For i = 1, j is 1 and 2, adding 1 + 2 = 3. For i = 2, j is 2, adding 4. The total is 7. Summing over all nine pairs would give 9.
377. **B** — i takes 0, 3, 6, 9. Odd values make i % 2 non-zero (true), so continue skips 3 and 9, but the update i += 3 still runs. Only 0 and 6 are added, giving 6. The loop stops when i reaches 12, and i is declared outside the loop, so it can be printed.
378. **C** — k = 3 enters at case 3, adding 3. There is no break, so it falls through into case 5 and adds 5, then stops at break. case 7 is not reached, so t is 8. The empty case 1 label does not affect a value of 3.
379. **D** — A while loop's else runs when the condition becomes false without a break. continue only skips the rest of one pass; it does not stop the else. n climbs to 5, n < 5 becomes false, and the else prints done 5.
380. **A** — The comma lets the header update two variables. Pass 1 runs with (0, 10), pass 2 with (2, 7), then the update gives (4, 4) and 4 < 4 is false. So count is 2 and the final values are i = 4, j = 4.
381. **B** — range's stop value is excluded, so the loop adds 1 to 9, which is 45. The fix is range(1, 11). This off-by-one error is one of the most common loop bugs.
382. **C** — Without braces, the if controls only the next statement. The second println is outside the if despite its indentation, so it always runs. With x = -2 the condition is false, so only checked is printed.
383. **D** — while is entry-controlled: the condition 10 < 5 is false before the first pass, so the body never runs and only end is printed. A do-while style loop would have run once.
384. **A** — i takes 0, 1, 2, 3, 4 and 5, because the test is i <= 5. That is 6 values and 6 lines. Using i < 5 would give 5 lines.
385. **B** — Removing 2 shifts 4 into index 0, but the iterator moves on to index 1, which is now 5, so 4 is skipped. At index 2 it finds 6 and removes it; the list is then [4, 5] and the loop ends. Build a new list, such as [n for n in nums if n % 2], instead.
386. **C** — When i reaches 2, continue jumps back to the test before i++ runs. i stays 2, the test is still true, and the same continue runs again forever. Moving i++ above the if, or using a for loop, fixes it.
387. **D** — break leaves only the innermost loop. Each outer value i = 1, 2, 3 has one j that gives a sum of 4, so found reaches 3. To stop everything, use a flag checked by the outer loop, a for-else, or put the search in a function and return.

388. **A** — total = 0 sits inside the outer loop, so it is reset for every row. After the last row it holds only 3 + 4 = 7. Moving the initialisation above the outer loop gives 10.
389. **B** — n takes 10, 7, 4, 1, -2, -5 and so on; it is never exactly 0, so n != 0 stays true. A range test such as while n > 0: stops safely. Loop conditions built on == or != are fragile when the step can jump past the target.
390. **C** — pass is a no-op placeholder; it does not skip anything. The print below the if runs on every pass, so 0 1 2 is printed. continue would have skipped 1.
391. **D** — for...in iterates over property keys, and array keys are the strings "0", "1", "2". 0 + "0" becomes "00", then "001", then "0012". Use for...of to get the values 1, 2, 3 and a sum of 6.
392. **A** — 0.1 has no exact binary form, so ten additions give 0.9999999999999999, not 1.0, and later values never hit 1.0 either. The loop only stops through the guard, which fires once n reaches 21. Use an int counter, or a test like d < 1.0 - 1e-9.
393. **B** — range(2, 22, 5) gives 2, 7, 12, 17; 22 is the excluded stop. The matching while loop starts at 2, tests i < 22 before each pass, and prints before it updates.
394. **C** — The else belongs to the inner for loop and runs only when that loop finishes without break, i.e. when no divisor was found. That happens for the primes. For n = 2 the inner range is empty, so it finishes at once and 2 is printed too.
395. **D** — i % 3 is 0, 1, 2, 0. Case 0 falls through into case 1, adding 1 + 10 = 11; case 1 adds 10; default adds 100; i = 3 adds 11 again. The total is 132. The break exits only the switch, not the for loop.
396. **D** — In def area(r): ..., r is a parameter (also called a formal parameter). In area(5), 5 is the argument (actual parameter). When the call runs, the parameter is initialised from the argument.
397. **C** — Java is strictly pass-by-value. For a primitive the value itself is copied. For an object the reference is copied, so the method can change the object's fields but cannot make the caller's variable point to a different object.
398. **C** — Python passes object references, often called pass-by-object-reference or call by sharing. x.append(1) changes the shared list, but x = [] only rebinds the local name. Integers and strings are immutable, so a function cannot change them in place.
399. **D** — int &x is a reference parameter: x becomes another name for the caller's variable, so assignments to x change it. int x receives a copy, and the two const versions forbid modification.
400. **B** — The base case stops the recursion. Every recursive call must move the input towards it; otherwise calls pile up until the stack overflows.
401. **D** — CPython guards the call stack with a recursion limit (sys.getrecursionlimit(), 1000 by default). Exceeding it raises RecursionError. Java reports a similar problem as StackOverflowError.
402. **B** — Overloads must differ in their parameter lists (the method signature). Two methods that differ only in return type, parameter names or modifiers cause a compile error.
403. **C** — def is an assignment of a function object to a name, so the later definition rebinds the name. Python gets flexible calls from default values, *args and **kwargs instead. typing.overload only describes signatures for type checkers.
404. **B** — A prototype such as int add(int, int); tells the compiler the function's interface. The body can then appear later in the file or in another translation unit.
405. **C** — Arguments fill parameters from the left, so defaults must be trailing. In C++, void f(int a = 1, int b); is an error; in Python, def f(a=1, b): is a SyntaxError. (Python's keyword-only parameters after * are the exception.)
406. **A** — *args packs surplus positional arguments into a tuple. **kwargs packs surplus keyword arguments into a dict.
407. **A** — Default expressions are evaluated once, when the function object is created. Every call that omits bag shares that same list, which is why a mutable default seems to remember earlier calls.
408. **D** — Falling off the end of a function, or writing a bare return, returns None.
409. **C** — A static method belongs to the class, not to an instance, so you call it as ClassName.method(). It has no this reference and cannot use instance fields directly.
410. **A** — In a tail call nothing remains to be done after the recursive call returns, so a compiler could reuse the frame. CPython and the JVM do not do this; C++ compilers may do it as an optimisation but the standard does not require it.
411. **D** — Locals live in the call's stack frame (activation record). A recursive or repeated call gets a fresh copy. Keeping a value between calls needs a static local in C++, a field in Java, or a global/closure in Python.

412. **B** — a = 1 by position, b keeps its default 2 and c is set to 4 by keyword. The result is 1 + 2 * 4 = 9.
413. **B** — x and nums refer to the same list object. append mutates it in place, so the caller sees four items.
414. **D** — x = [9] rebinds the local name x to a new list; the caller's nums still refers to the original [1]. Rebinding a parameter never affects the caller in Python.
415. **D** — Java passes the int values 1 and 2 as copies. swap exchanges its own copies a and b, and the caller's x and y are untouched.
416. **C** — a and b are references, so they are aliases for x and y. Swapping them swaps the caller's variables.
417. **A** — b starts as a copy of the reference in x, so b.v = 5 changes the caller's object. b = new Box() then points the local copy elsewhere; b.v = 9 changes only the new object. x.v is 5.
418. **A** — f computes the factorial: 5 * 4 * 3 * 2 * 1 = 120. The base case n <= 1 returns 1.
419. **A** — The recursive call comes before print, so p(3) waits for p(2), which waits for p(1), which waits for p(0). The prints happen as the calls unwind: 1, then 2, then 3.
420. **B** — Let C(n) be the number of calls: C(0) = C(1) = 1 and C(n) = 1 + C(n-1) + C(n-2). So C(2) = 3, C(3) = 5, C(4) = 9 and C(5) = 15. The repeated subproblems are why naive recursive Fibonacci is exponential.
421. **B** — global x makes the function update the module-level x. The first call sets x to 6; the second sets it to 7 and returns 7. The print then shows the returned 7 and the current x, 7.
422. **D** — The default list is created once. The first call appends 1 to it, and the second call appends 2 to the same list, so it prints [1, 2].
423. **A** — JavaScript, like Java and Python, passes a copy of the reference. obj.n++ changes the shared object to n = 2. obj = {n: 100} only rebinds the parameter, so o.n stays 2.
424. **D** — 1 binds to a. The extra positional values 2 and 3 go into args (a tuple of length 2), and x=4 goes into kw (a dict of length 1).
425. **D** — &a passes a's address, and (*p)++ increments the int stored at that address, which is a itself. Passing a pointer is how C-style code simulates pass-by-reference.
426. **B** — The recursive line computes n + total(n - 1) but never returns it, so that branch returns None. total(1) returns None; then total(2) evaluates 2 + None and raises TypeError. Writing return n + total(n - 1) fixes it.
427. **B** — a holds a copy of the reference to arr. a = new int[a.length] points the copy at a new array, leaving arr unchanged. To clear it, write into the existing array (for example Arrays.fill(a, 0)) or return the new array.
428. **C** — From 5 the calls go 3, 1, -1, -3 ... and never equal 0, so the base case is missed. A safer base case is if n <= 0: return.
429. **A** — None is immutable, so it is a safe sentinel; the new list is created on every call that needs one. list() as a default is still evaluated once. Clearing at the end would also wipe the list that was just returned to the caller.
430. **A** — toUpperCase returns a new String; assigning it to s changes only the method's copy of the reference. The caller's t still refers to "pk". Return the new string, or use a mutable StringBuilder, to see a change.
431. **C** — Even n reaches 0 exactly (4, 2, 0). Odd n goes 3, 1, -1 ... and never equals 0, ending in RecursionError. Testing n <= 0 would fix it.
432. **C** — Python decides scope at compile time: count += 1 assigns to count, so count is local throughout inc. Reading it before any local value exists raises UnboundLocalError. Add global count (or pass and return the value).
433. **A** — f(1, 2) passes two ints. Each overload needs one widening conversion (int to double), so both apply, and neither signature is more specific than the other. javac reports a reference to f as ambiguous.
434. **B** — fib(n) calls fib(n-1) and fib(n-2), which repeat the same work, giving roughly O(phi^n) calls. Caching results (functools.lru_cache) or iterating computes each value once, so the time is O(n).
435. **C** — JavaScript evaluates default parameter expressions at call time, so each call that omits bag gets a new array. The second call returns an array holding only 2. Python evaluates defaults once at def time, which is why the same pattern behaves differently there.
436. **D** — void means the function produces no return value; a bare return; may end it early, but return 5; would be a compile error. It does not mean 0, a null pointer, or 'any type'.
437. **A** — A lambda builds a function object without a def name. Its body must be one expression, whose value is returned automatically; statements such as loops or assignments are not allowed inside it.

438. **A** — Each call pushes a frame on the call stack holding its parameters, locals and where to return. The frame is popped when the call returns. Code lives in the code segment, globals in static storage and new objects on the heap.

439. **D** — **kwargs gathers every keyword argument that matches no named parameter into a dict mapping names to values. *args, by contrast, gathers extra positional arguments into a tuple.

440. **A** — When two or more functions call each other in a cycle, the recursion is indirect, also called mutual recursion. Tail recursion is about where the self-call sits; overloading and nesting are unrelated.

441. **C** — A pure function depends only on its arguments and changes nothing outside itself (no printing, no global writes, no mutation of arguments). Purity has nothing to do with the number of parameters, the return type or where the function is defined.

442. **D** — A comma-separated return builds a tuple, so the caller gets (total, count) and can unpack it with a, b = f(). Python does not drop the second value and does not build a list.

443. **D** — Overriding happens across an inheritance relationship: the subclass method has the same name and parameter list, and the call is chosen at run time from the object's real class. Same name with different parameters in one class is overloading.

444. **B** — Varargs are compiled as an array parameter, so nums is an int[] and can be looped over or indexed. Calling sum() with no arguments passes an empty array of length 0.

445. **B** — First-class means a function is a value like any other: you can write g = len, pass a function to sorted(key=...), or return an inner function. That is what makes callbacks, closures and decorators possible.

446. **B** — & makes s an alias of the caller's string, so nothing is copied; const forbids changes through s. This is the usual way to pass large objects that a function only reads.

447. **C** — Without nonlocal, assigning to a name inside the inner function creates a new local. nonlocal n binds n to the enclosing function's variable, so n += 1 updates it. global targets the module level instead.

448. **A** — Python searches the current function first, then any enclosing function scopes, then the module's globals, and finally the built-ins such as len and print. The first match wins.

449. **C** — Parameters before / are positional-only and parameters after * are keyword-only, while b may be either. So a must be positional and c must be named. f(a=1, ...) names a, f(1, 2, 3) passes c positionally, and f(1, c=3) omits b.

450. **C** — A common Java idiom is void greet(String n) { greet(n, "Hello"); } next to void greet(String n, String g). Java rejects = value in a parameter list, and final or static do not supply defaults.

451. **D** — An array parameter decays to a pointer, so int a[] means int *a. Writes through a[i] change the caller's array, and sizeof(a) inside f gives the size of a pointer, not the array, which is why the length is usually passed separately.

452. **C** — inline is a hint that can remove call overhead for small functions; compilers decide for themselves and also inline functions that are not marked. inline also allows the same definition in several translation units, which is why such functions often live in headers.

453. **B** — Each call waits for the next one, so frames pile up until the base case: n frames at once, giving O(n) stack space. A loop doing the same work needs O(1) extra space.

454. **D** — A static method belongs to the class and runs without any object, so there is no this and no instance fields to read. It can call static methods, use locals, and create objects, then use those objects' fields.

455. **A** — The call tree doubles at each level and has n levels, so it holds about 2^n calls. One call on n - 1 would give O(n); two calls on n/2 would give O(n).

456. **C** — **kwargs must close the parameter list. def f(**kw, a): is a SyntaxError. The allowed order is normal parameters, *args, keyword-only parameters, then **kwargs.

457. **A** — For a parameter passed by value, the caller's object is copied either way, so the standard ignores top-level const when forming the function type. Both lines define void f(int), which is a redefinition. const does matter with references or pointers, such as int& versus const int&.

458. **C** — A non-void Java method must return a value on every path the compiler can see. When n is 0 neither branch runs, so execution could fall off the end; javac reports 'missing return statement' before the program ever runs.

459. **D** — Keyword arguments can be given in any order, so w = 2 and h = 3, and 2 * 3 = 6. The default h = 1 is unused because h is passed.

460. **C** — return 1, 2 packs the two values into a tuple, so x is (1, 2) and its type name is tuple.
461. **B** — sq(2) is 4, and sq(4) is 16. The inner call runs first and its result becomes the outer call's argument.
462. **A** — *args counts arguments, not the items inside them. The call passes three arguments: 1, the list [2, 3] and the string 'ab', so args has length 3.
463. **C** — sq(3) is 9. For sq(sq(2)) the inner call runs first and gives 4, so the outer call is sq(4), which is 16. 9 + 16 = 25.
464. **C** — A varargs method may be called with zero arguments; nums is then an empty array and the sum is 0. sum(1, 2, 3) is 6, so 0 + 6 prints 6.
465. **A** — *nums unpacks the list into two positional arguments, so the call is add(3, 4), which returns 7.
466. **B** — **d passes each key as a keyword argument, matched by name and not by the dict's order. So a = 10, b = 2, c = 1, and 10 - 2 - 1 = 7.
467. **B** — Each call to make() creates a new n and a new closure. a has been called twice, so the third call returns 3. b has its own n, so its first call returns 1.
468. **D** — Each call prints n, recurses, then prints n again after the deeper calls return. The first prints go 3 2 1 on the way down; the second prints run as the stack unwinds, giving 1 2 3.
469. **B** — Euclid's algorithm: g(48, 18) -> g(18, 12) -> g(12, 6) -> g(6, 0), which returns 6, the greatest common divisor.
470. **D** — n is static, so all objects share one counter. Three constructor calls make it 3. Reading a static field through an instance (c.n) is legal, though style tools warn about it, so the sum is 3 + 3 = 6.
471. **A** — f(1) uses both defaults: 1 + 10 + 100 = 111. f(1, 2) replaces only b: 1 + 2 + 100 = 103. Default arguments fill in from the right.
472. **B** — Overload resolution picks the exact match for each argument type: 5 is an int, 5.0 is a double, and '5' is a char. No conversion is needed, so each call reaches its own version.
473. **C** — A static local is initialised once and keeps its value between calls. The three calls raise c to 1, 2 and 3, and the last value is printed.
474. **C** — Each lambda refers to the variable i, not to its value when the lambda was made. The lookup happens when the lambda is called, after the comprehension has finished with i = 2, so every call returns 2. Writing lambda i=i: i captures each value.
475. **B** — A JS default applies when the argument is missing or exactly undefined, so the first two calls use b = 5. null is a real value, so b = null and 1 + null is 1, because null converts to 0 in arithmetic.
476. **A** — Positional arguments bind first, so 1 goes to a. Then a=2 tries to bind a again, and Python raises TypeError about multiple values for 'a' before checking b. Nothing is printed.
477. **B** — A default value is evaluated once, when the def statement runs. At that moment x is 10, so a defaults to 10 for ever; rebinding x to 20 later does not change it.
478. **D** — flat returns a one-item list for a non-list and otherwise joins the flattened results of each element, at any depth. The leaves are 1, 2, 3, 4 and 5, so the flat list has length 5.
479. **D** — biggest prints 7 but has no return statement, so it returns None. r is None, and the second print shows None. Replacing print(...) with return ... fixes it.
480. **A** — int n receives a copy, so the assignment changes only the copy. A reference parameter int &n is an alias for x, so n = 10 writes to x.
481. **B** — The base case is the empty list, but s(lst) passes the full list again, so the problem never shrinks and Python stops at its recursion limit. Passing lst[1:] makes progress toward the base case.
482. **D** — Python makes a name local only when the function assigns to it. add only reads total, so the lookup falls through to the global (0) and returns 5. UnboundLocalError appears only when the function also assigns, as in total += n.
483. **B** — The method always calls f(n - 1) and never stops: 3, 2, 1, 0, -1 and so on. Each call adds a frame until the JVM stack overflows. A base case such as if (n <= 1) return 1; fixes it.
484. **A** — scores and raw name the same list, so item assignment mutates the caller's data. Building a new list (or copying first with scores = scores[:]) leaves raw alone.

485. **C** — Both approaches do O(n) additions, but recursion keeps one frame per pending call. CPython has no tail-call optimisation and a default recursion limit of 1000, so the recursive version crashes long before n = 1,000,000.

486. **C** — The only base case is n == 1. fact(0) calls fact(-1), fact(-2) and so on, never reaching 1, until the recursion limit stops it. if n <= 1: return 1 handles 0 as well.

487. **D** — Assigning sum inside total creates a local that hides the built-in only inside that function. At module level sum still means the built-in, so sum([3, 4]) is 7. Shadowing a built-in at module level would break it.

488. **A** — 5 is an int, and neither overload takes int. int to long is an integral conversion and int to double is a floating-integral conversion; both have conversion rank, so neither is better and the compiler reports an ambiguous call. Adding f(int), or calling f(5L), resolves it.

489. **D** — Static methods are not overridden; B.n() hides A.n(). A static call is bound at compile time using the declared type of obj, which is A, so A.n() runs and prints A. An instance method would print B.

490. **A** — null fits both String and Object, so both overloads apply. Java picks the most specific one, and String is more specific because it is a subtype of Object. The call would be ambiguous only if two unrelated types, such as String and Integer, both matched.

491. **A** — Two calls on e/2 give T(e) = 2T(e/2) + O(1), which is O(e), so halving buys nothing. Computing h = pw(b, e // 2) once and returning h * h gives T(e) = T(e/2) + O(1), which is O(log e).

492. **C** — CPython does not optimise tail calls, so every call still adds a frame and 5000 frames pass the default limit of 1000. A loop carries the same state in variables with constant stack use. Caching does not reduce depth, the call is already in tail position, and an early return breaks the result.

493. **B** — Integer is immutable. Inside inc, n++ unboxes the value, adds 1, boxes a new Integer and binds the local n to it. The caller's a still refers to the original 5, as Java passes a copy of the reference.

494. **C** — input() reads one line from standard input, strips the trailing newline and returns it as a string. Convert it yourself, for example int(input()). (Python 2's input() evaluated the text, which is a common source of confusion.)

495. **A** — std::cin is an istream connected to standard input and std::cout is an ostream connected to standard output; both are declared in <iostream>. ifstream and ofstream are the file-stream classes from <fstream>.

496. **C** — Append mode keeps the existing content and sends every write to the end of the file. A missing file is created. Mode "w" truncates, "r" is read-only and fails if the file is missing.

497. **D** — "x" is exclusive creation, useful when you must not overwrite anything. "w" silently truncates an existing file, "a" appends to it, and "r+" needs the file to exist.

498. **B** — The with statement uses the file object as a context manager: on leaving the block, normally or by an exception, it calls f.close(), which also flushes buffered writes.

499. **C** — Reader/Writer subclasses (FileReader, BufferedReader, PrintWriter ...) work with chars and handle character encoding. InputStream/OutputStream and their Data/Object subclasses work with bytes; InputStreamReader bridges bytes to characters.

500. **D** — std::endl inserts '\n' and then calls flush(). Flushing on every line can slow down output-heavy programs, so '\n' is preferred unless you need the data written immediately.

501. **B** — cerr is tied to stderr and has unitbuf set, so each output operation is flushed at once, which suits error messages. clog also writes to stderr but is buffered. cout writes to standard output.

502. **C** — Each call into the operating system is expensive. A buffer gathers many small reads or writes into one larger transfer. The trade-off is that buffered output may not appear until the buffer is flushed or the stream is closed.

503. **B** — nextLine() returns everything up to the line separator and consumes it. next() returns one whitespace-separated token, nextInt() one integer token, and hasNext() only checks whether a token is available.

504. **A** — try (BufferedReader br = ...) { } calls close() on every declared resource when the block ends, in reverse order of declaration. The compiler requires each resource type to implement java.lang.AutoCloseable.

505. **A** — getline reads characters up to the newline and discards the newline. cin >> s stops at the first whitespace. cin.get works with a char or a char array, not a std::string, and cout is an output stream.

506. **B** — print separates its arguments with one space and finishes with a newline. Change them with, for example, print(a, b, sep=",", end="").

507. **D** — In text mode Python decodes with the given (or platform default) encoding and applies universal newlines, so \r\n becomes \n. Binary mode returns bytes objects exactly as stored, which you need for images, zip files and the like.

508. **C** — File streams live in <fstream>. <iostream> declares the console streams, <cstdio> the C-style FILE* functions such as fopen, and <sstream> the string streams.

509. **D** — readlines() returns a list of strings, each normally ending in \n. read() returns one string, readline() returns a single line, and file objects have no lines() method.

510. **D** — sep="-" puts a hyphen between a and b, and end="!" replaces the newline with !. The next print continues on the same line with c.

511. **A** — input() returns the string "5". Multiplying a string by an int repeats it, giving "555". Use int(input()) to get arithmetic.

512. **C** — :.2f rounds to two decimals, giving 3.14. :5d prints the integer in a field 5 wide, right-aligned for numbers, so 42 gets three leading spaces.

513. **D** — The "w" write leaves two complete lines. "a" adds ef at the end without a newline, so the file is ab, cd and ef: three lines. readlines() counts a final line even if it lacks \n.

514. **C** — Opening an existing file with "w" truncates it to zero length before writing, so only hi remains.

515. **A** — Each readline() continues from the current file position. The first call returns 'one\n', the second 'two\n'. readline() keeps the newline character, and repr shows it.

516. **D** — process.stdout.write prints exactly what it is given, with no newline added. So x and y join, the explicit \n ends the line, and console.log prints z followed by a newline.

517. **C** — int() ignores leading and trailing whitespace, so int(" 42 ") is 42 and the sum is 43. Spaces inside the number, as in "4 2", would raise ValueError.

518. **D** — input() returns the whole line "3 4". int() rejects the inner space and raises ValueError: invalid literal for int(). Use input().split() and convert each part.

519. **A** — split() breaks the line on whitespace into ["7", "8"], map(int, ...) converts both, and they unpack into a and b. 7 * 8 = 56.

520. **C** — cin >> s skips leading whitespace and stops at the next whitespace, so s is "Ali" and its length is 3. "Khan" stays in the input buffer for the next read.

521. **B** — nextInt() reads 25 but leaves the newline after it in the buffer. nextLine() then returns the rest of that line, which is empty. Ali is still waiting to be read.

522. **B** — fixed switches to fixed-point notation and setprecision(2) then means two digits after the decimal point. 3.14159 rounds to 3.14. setprecision comes from <iomanip>.

523. **C** — *nums unpacks the list into three separate arguments, and sep=", " places a comma and a space between them.

524. **B** — A text-mode file's write() accepts only str. Passing the int 0 raises TypeError on the first pass. Write str(i), or use print(i, file=f), which converts and adds a newline.

525. **A** — The first read() consumes the file and leaves the position at the end. The second read() starts from there and returns an empty string. Call f.seek(0) to read again.

526. **A** — The with block closed the file when it ended. The name f still exists (Python has no block scope) but refers to a closed file, so read() raises ValueError.

527. **B** — nextInt() leaves the line break in the buffer, and nextLine() returns the empty rest of that line. Discarding it with an extra nextLine(), or reading the number as Integer.parseInt(sc.nextLine().trim()), fixes the problem.

528. **A** — Iterating over a file keeps each line's newline, so the first line is "yes\n" and fails the comparison. Only the final "yes" has no newline and matches. Use line.strip() == "yes" to get 2.

529. **D** — cin >> n stops before the newline, so getline immediately reads an empty line. cin.ignore(..., '\n') discards everything up to and including that newline. getline(cin >> ws, name) is another fix.

530. **B** — The stream sets eofbit only when an extraction runs into the end. With a trailing newline, the last good read does not set it, so the loop body runs once more with a failed read. Testing the extraction itself stops exactly when a read fails.

531. **B** — BufferedWriter collects characters in memory and passes them on only when the buffer fills, flush() is called or close() is called. Use try-with-resources so close() always runs.

532. **A** — A file object is an iterator that reads buffered chunks and yields one line at a time, so memory use stays small. read() and readlines() load the whole 10 GB into memory first.
533. **D** — Both modes allow reading and writing, but any "w" mode empties the file at open. "r+" opens an existing file for reading and writing with the position at the start, so the data survives.
534. **A** — Opening a non-existent file for reading (the default mode "r") raises FileNotFoundError, a subclass of OSError. EOFError comes from input() at end of input, not from open().
535. **B** — readLine() returns null at end of stream, which is why the usual loop is while ((line = br.readLine()) != null). An empty string means an empty line, and -1 is what read() returns for a single character.
536. **C** — print() writes to sys.stdout by default; file= redirects it to any object with a write() method, such as sys.stderr or an open file.
537. **D** — f.seek(offset) moves the file position, and f.tell() returns where it is now. read() and write() also advance the position, but they do not set or report it directly.
538. **D** — ios::app means append: output always goes to the end of the file, and existing data is kept. Without it, an ofstream opens with truncation and erases the old contents.
539. **C** — printf (and String.format) take a format string with specifiers such as %d, %s and %.2f, like C's printf. println and print only print a value as it is.
540. **C** — The shell's < redirects standard input to the file, and input() reads from stdin, so each call gets the next line of in.txt. The program code does not change.
541. **C** — >> extracts data from an input stream into a variable; << inserts data into an output stream. For streams these are overloaded versions of the bit-shift operators.
542. **B** — "w" creates the file if it is missing and empties it if it exists. Only "r" (and "r+") raise FileNotFoundError for a missing file, and only "x" raises FileExistsError.
543. **A** — The compiler checks that every checked exception is either caught in a try/catch or declared with throws. Unchecked exceptions such as NullPointerException have no such rule.
544. **D** — At end of input, input() raises EOFError. An empty line typed by the user returns "" instead. Contrast sys.stdin.readline(), which returns "" at end of file.
545. **D** — InputStreamReader wraps an InputStream (bytes) and decodes it into characters using a charset, which gives a Reader. BufferedReader adds buffering and readLine(); Scanner parses tokens; OutputStreamWriter goes the other way.
546. **C** — Text mode decodes bytes with an encoding. If none is given, Python uses the locale's preferred encoding, which differs between systems (cp1252 on many Windows machines). Naming UTF-8 makes the program behave the same everywhere.
547. **B** — With universal newlines, text mode turns \r\n and \r into \n on reading, so code can test for \n on any platform. Binary mode ("rb") returns the bytes unchanged.
548. **B** — setw is the one common manipulator that is not sticky: it applies to the next formatted output and then resets. setfill, setprecision, fixed and left or right stay in effect until changed.
549. **A** — sys.stdin.readline() returns the line including \n, so you usually call .strip() or int(...) on it. input() removes the newline. readline() is popular in contest code because it avoids input()'s per-call overhead.
550. **C** — FileWriter(String name, boolean append) appends when the flag is true. With the one-argument constructor the file is overwritten. Auto-flush is a PrintWriter option, not a FileWriter one.
551. **A** — The trailing s stands for string: dumps returns a str, while dump(obj, f) writes to an open file. The readers mirror this: load(f) reads a file and loads(s) parses a string.
552. **B** — > redirects only standard output. Standard error is a separate stream and still goes to the terminal, which is exactly why errors are written to it: they stay visible when normal output is redirected. 2> or 2>&1 redirects it too.
553. **D** — FileReader already produces characters, but each read() can hit the operating system. BufferedReader fills a memory buffer in big blocks and serves reads from it, and it adds the convenient readLine() method.
554. **B** — A failed extraction sets failbit; since C++11 x is also set to 0. The bad characters are not consumed, and every later extraction fails at once while failbit is set. The fix is cin.clear() followed by cin.ignore(...) to discard the bad input.
555. **B** — Output to a terminal is flushed at each newline. With end="" no newline is written, so the text stays in the buffer until a later newline, a flush or program exit. print(..., flush=True) or sys.stdout.flush() forces it out.

556. **B** — By default C++ streams stay synchronised with printf/scanf, and cin is tied to cout so cout is flushed before every input. Turning both off lets the streams buffer freely, which matters with millions of reads. After that, do not mix printf and cout.
557. **D** — C functions receive arguments by value. To let scanf write into the caller's variable, you pass a pointer to it. printf only reads the value, so a copy is enough. Passing x to scanf by mistake is undefined behaviour.
558. **D** — print() converts each argument with str() and joins them with the default separator, one space, so the output is Total: 5 items.
559. **D** — "a" + "b" is one argument, "ab". With sep="" nothing is placed between "ab" and "c", giving abc.
560. **B** — In text mode, write() returns the number of characters written. "hello\n" is 5 letters plus the newline, so n is 6.
561. **A** — 1 + 2 is evaluated as an int expression, 3, and then inserted. Insertion adds no spaces, so the output is A3B.
562. **C** — print does not add a newline, and println does after its text. So x and y share the first line, the newline follows y, and z starts the second line.
563. **D** — str(3) + "4" joins two strings into "34", while 3 + 4 adds two ints into 7. print separates them with a space.
564. **A** — writelines writes each string exactly as given and adds no newlines, so the file holds abc. To get separate lines you must include \n in the strings.
565. **B** — seek(2) moves to offset 2, which is the character c (offsets start at 0). read(3) then returns three characters: cde.
566. **A** — :03d pads 7 with zeros to width 3. :<4 left-aligns "ab" in 4 characters. :6.1f gives 2.5 with one decimal, right-aligned in 6 characters.
567. **C** — setw(5) pads 42 to five characters (three spaces then 42). setw is not sticky, so the next setw(3) pads 7 to three characters (two spaces then 7). The | follows with no width.
568. **B** — setfill('*') sets the padding character and left puts the value on the left, so 12 is padded on the right to width 6 with four stars.
569. **D** — %5.2f rounds to two decimals and right-aligns in 5 characters, giving a space then 3.14. %-4d left-aligns 7 in 4 characters. %s prints ok, and %n ends the line.
570. **A** — split(",") returns the strings ['10', '20', '30']. Adding two strings joins them, so a[1] + a[2] is "2030". int() is needed to get 50.
571. **A** — nextInt() reads the token 12 and next() reads the next token, abc. String concatenation puts s first, so the output is abc12.
572. **C** — console.log separates its arguments with one space and formats an array with util.inspect, which shows [2, 3] with inner spaces. String(arr) would give 2,3 instead.
573. **B** — print(..., file=f) writes to the file instead of the screen, using sep="," between items and the default end="\n". Reading the file back gives 'a,b\n'.
574. **D** — "r+" opens for reading and writing without truncating, with the position at 0. write("J") overwrites the first character in place, so the file becomes Jello.
575. **D** — "a+" opens for appending and reading with the position at the end, so the first read() returns ''. write("c") appends, and after seek(0) the whole file 'abc' is read.
576. **A** — read(1) consumes the a. readline() then reads from the current position to the end of that line, including the newline, so it returns 'b\n'.
577. **C** — >> into an int reads digits and stops at the first non-digit, so n is 12 and abc stays in the buffer. The next >> into a string reads abc. The read succeeds because at least one digit was found.
578. **A** — The inner print("x") runs first and writes x. It returns None, and the outer print shows that value, so None appears on the next line.
579. **B** — input() always returns a string, and Python 3 cannot compare str with int using >, so it raises TypeError. Converting with int() makes the comparison numeric.
580. **C** — "w" truncates the file every time it is opened. Opening the file once before the loop, or using "a" to append, keeps every result.
581. **A** — Iterating a file yields strings such as "5\n". sum() starts at 0 and 0 + "5\n" raises TypeError. Converting each line with int() works, and int() ignores the surrounding whitespace.

582. **D** — An ofstream opened with no mode uses out|trunc, which empties the file on every run. ios::app keeps the old contents and appends.
583. **D** — try-with-resources calls close() automatically when the block ends, normally or by exception. A plain close() after the read is skipped when an exception jumps out first.
584. **C** — readline() includes the trailing \n, so the comparison is "done\n" == "done", which is False. Comparing line.strip() fixes it; input() would also strip the newline.
585. **C** — write() puts "hi" into an in-memory buffer. The second open reads the file on disk, which is still empty. Closing f (ideally with a with block) or calling f.flush() first makes the text visible.
586. **D** — Byte streams move data unchanged. Reader and Writer classes decode and encode characters, which corrupts bytes that are not valid text, and line-based copying also changes line endings.
587. **B** — A JPEG holds arbitrary bytes. Text mode tries to decode them with an encoding and may translate newlines, so the copy fails or comes out damaged. Binary mode reads and writes bytes objects unchanged.
588. **A** — The failed read sets failbit and leaves abc in the buffer. Every later >> fails at once, so the condition never changes. clear() resets the state and ignore() discards the bad line, so the next read waits for new input.
589. **A** — split(",") also cuts inside the quoted field, giving Ali, "Lahore, PK" (as two pieces) and 20, which is 4 parts. csv.reader understands quoting and returns 3 fields.
590. **A** — hasNextInt() only looks at the next token without removing it. abc stays at the front, so every check returns false. Calling sc.next() in the body throws the token away and lets the scanner reach the next one.
591. **C** — When stdout is a file it is block-buffered, so A waits in memory until the program exits. stderr is flushed at each line, so B reaches the file first. print("A", flush=True) keeps the source order.
592. **C** — é is stored in UTF-8 as the bytes C3 A9. Decoding with cp1252 maps each byte to its own character, Ã and ©, producing mojibake. Reading with encoding="utf-8" returns café.
593. **B** — In text mode with universal newlines, each \r\n becomes \n, so 6 bytes read as 4 characters: a, \n, b, \n. "rb" returns all 6 bytes unchanged.
594. **C** — Python strings are immutable, so assigning to s[0] raises TypeError: 'str' object does not support item assignment. To get "bat" you build a new string, for example s = "b" + s[1:].
595. **D** — StringBuilder (and the synchronised StringBuffer) can be changed in place with append, insert and reverse. String objects are immutable, and Character and Integer are immutable wrapper classes.
596. **B** — A C string is a char array terminated by the null character '\0'. strlen() counts characters until it reaches that terminator; it has no access to the array's declared size.
597. **B** — equals() compares content. == on objects compares references, so two separate String objects with the same text can give false. Equal hash codes do not prove equality, and compareTo returns 0 (not 1) for equal strings.
598. **A** — With no argument, strip() removes leading and trailing whitespace (spaces, tabs, newlines). It never touches characters inside the string; lstrip() and rstrip() trim one side only.
599. **D** — Because String is immutable, every + builds a new object and copies the old contents, so building n characters this way copies O(n^2) characters in total. A single StringBuilder appends in amortised constant time per character.
600. **B** — std::string is a mutable class: you can assign to s[i], append with += and erase characters. It manages its own length and memory, and operator== compares contents. C-style char arrays are the ones that need '\0'.
601. **A** — The f prefix (Python 3.6+) evaluates expressions inside braces at run time. Without it the braces stay as literal text; r marks a raw string and b a bytes literal.
602. **C** — The Java Language Specification requires string literals to be interned in the string pool, so identical literals share one object and == compares equal references. Strings built at run time (for example with new String or concatenation of variables) are separate objects.
603. **B** — find() signals a missing substring with -1. index() behaves the same when the substring is present but raises ValueError when it is not, so it needs a try/except.
604. **A** — isdigit() is True only when every character is a digit, so the decimal point in "20.5" makes it False. isalpha() would be False for both, as would isspace() and islower().
605. **A** — join() computes the final length once and copies each piece one time, giving linear work. Repeated += may copy the growing string again and again because str is immutable.

606. **D** — \t is the tab escape in all three languages. A newline is \n, a literal backslash is written \\, and only C-style strings use '\0' as a terminator.
607. **B** — ord() returns the Unicode code point of a single character. 'A' is 65 (lowercase 'a' is 97); chr(65) goes the other way and returns 'A'.
608. **C** — casefold() applies full Unicode case folding, mapping 'ß' to "ss", so both become "strasse". lower() leaves 'ß' unchanged, so the lowered strings differ.
609. **C** — A Python 3 str is a sequence of Unicode code points, and len() counts them. The byte count depends on the encoding, so you only get it after calling encode(), for example len(s.encode("utf-8")).
610. **C** — toUpperCase() returns a new String and leaves s unchanged because Strings are immutable. The result is discarded, so s still prints "nsct". You would need s = s.toUpperCase();.
611. **C** — \u00e9 is the single code point é, so the str has 4 code points. In UTF-8, é needs 2 bytes while c, a and f need 1 each, giving 5 bytes.
612. **D** — A slice with step -1 and no start or stop walks the whole string backwards, so it returns the reversed string.
613. **C** — substring(begin, end) includes begin and excludes end, so it takes indices 3 to 6: g, r, a, m. Python's s[3:7] follows the same half-open rule.
614. **A** — The format spec .2f rounds to two digits after the decimal point, giving 3.14.
615. **B** — 03d pads an integer with zeros to width 3, giving 007. >5 right-aligns the string in a field of width 5, adding three spaces before "ab".
616. **B** — %5.1f prints the number with one decimal place in a field 5 characters wide, right-aligned. "3.1" is 3 characters, so two spaces are added in front.
617. **C** — count() counts non-overlapping occurrences, scanning left to right. "banana" contains "an" twice. In "aaaa" the matches are positions 0-1 and 2-3, so the answer is 2, not 3.
618. **C** — The third argument to replace() limits the number of replacements, so only the first 'o' becomes '0'.
619. **D** — split(",") splits on every comma, so the two adjacent commas produce an empty string: ['a', 'b', '', 'c'] has 4 items. split() with no argument splits on runs of whitespace and drops empty strings, giving ['a', 'b'].
620. **D** — Strings sort by code point. Uppercase letters (65-90) come before lowercase letters (97-122), so every capital sorts ahead of every small letter. Use key=str.lower for dictionary-style order.
621. **A** — s[1:5:2] takes indices 1 and 3: "bd". s[-2:] is the last two characters: "ef". s[5:1:-2] starts at index 5 and steps back by 2 while staying above 1, taking indices 5 and 3: "fd".
622. **B** — strlen() counts characters before the terminator: 5. sizeof on the string literal gives the size of the whole char array, which includes the '\0', so 6.
623. **A** — \d+ matches one or more consecutive digits, and findall() returns every non-overlapping match as a string. The digit runs are 12, 3 and 456; the results stay strings, not integers.
624. **D** — lower() does return a new lowercased string, which is assigned back to s. The problem is the space: "race car" reversed is "rac ecar". Removing non-letters first, for example with "".join(c for c in s if c.isalnum()), fixes it.
625. **D** — A string read at run time is a separate object from the pooled literal "yes", so == compares two different references and gives false. equals() compares the characters and gives true.
626. **A** — Membership in "aeiou" is case-sensitive, so only u, i and o are counted. Lowercasing first (for ch in word.lower()) gives the correct 5.
627. **D** — A string literal of 5 letters occupies 6 chars because of the terminating '\0'. C++ forbids a string-literal initialiser longer than the array, so the compiler rejects it. C accepts it but leaves the array without a terminator, which is a common bug.
628. **B** — Step i copies about i characters into a new String, and 1 + 2 + ... + n is O(n^2). StringBuilder keeps a growable buffer with amortised O(1) appends, so the whole loop is O(n).
629. **A** — new String("hi") always creates a new heap object, so a == b is false. intern() returns the pooled instance for "hi", which is the same object as the literal held by a, so a == c is true.
630. **B** — Evaluation is left to right. 'a' + 1 is char plus int, which is int arithmetic: 97 + 1 = 98, then 98 + "b" is "98b". In the second line "b" + 'a' is already a String, so 'a' and 1 are each appended as text: "ba1".
631. **A** — isdigit() returns False for "-12" because '-' is not a digit, yet int("-12") works. "12", "007" and "2026" are all digits only. A try/except around int(s) is the reliable check.

632. **D** — Valid indices run from 0 to s.length() - 1. With <=, the last iteration calls charAt(s.length()), which is out of range. Using < keeps every index valid and still visits every character.
633. **C** — Python does not convert numbers to strings automatically with +, so "Total: " + 5 raises TypeError. str(5) or an f-string such as f"Total: {5}" does the conversion explicitly.
634. **A** — strcmp() returns 0 for equal strings, a negative value if a sorts before b, and a positive value if a sorts after b. So the usual equality test is strcmp(a, b) == 0.
635. **B** — find() returns the special constant std::string::npos (the largest value of size_t) when there is no match. Code should test pos != std::string::npos rather than pos != -1 or pos > 0.
636. **C** — std::string overloads operator== to compare contents character by character. This differs from Java, where == on String compares references and equals() compares contents.
637. **D** — String.join(delimiter, elements) (Java 8+) places the delimiter between each element. split() does the reverse, breaking a string into pieces.
638. **A** — isalpha() is True when the string is non-empty and all its characters are alphabetic. isalnum() also accepts digits, isdigit() accepts only digits, and istitle() checks capitalisation.
639. **B** — UTF-8 is a variable-length encoding that is backward-compatible with ASCII: code points 0-127 take exactly one byte. Other characters take 2 to 4 bytes.
640. **C** — A raw string (r prefix) does not process escape sequences, so r"\n" is the two characters '\' and 'n'. Raw strings are handy for regular expressions and Windows paths.
641. **D** — encode() turns text (str) into a bytes object, here b'abc'. decode() goes the other way, turning bytes back into str.
642. **A** — Outside a character class, ^ is an anchor for the start of the input. Inside brackets, as in [^0-9], it means negation, and $ is the end anchor.
643. **B** — ", ".join([1, 2]) raises TypeError because join only accepts strings. Convert first, for example ", ".join(map(str, nums)).
644. **C** — The extraction operator skips leading whitespace and reads one word. getline() reads every character up to (and discards) the newline, so "Ali Khan" is read whole only by getline.
645. **D** — Both are mutable character sequences with the same API. StringBuffer (Java 1.0) synchronizes its methods; StringBuilder (Java 5) drops that overhead and is preferred when only one thread uses it.
646. **A** — Because no code can change a String, the JVM can let many references share one pooled object, use it safely across threads and as a HashMap key, and cache its hashCode. Concatenation still creates new objects.
647. **B** — == compares values; is compares identity. Short literals may be interned so is happens to work in tests, but a string read from input or built by concatenation is usually a separate object with the same text.
648. **C** — partition() splits at the first occurrence of the separator and always returns a 3-tuple: before, separator, after. rpartition() would split at the last '='.
649. **D** — startswith() and endswith() accept a tuple of alternatives and return True if any one matches. They always return a bool.
650. **A** — !r calls repr() and !s calls str(). For name = "Ali", f"{name!r}" gives 'Ali' with quotes, which is useful in debugging output. Right-alignment uses > in the format spec.
651. **B** — gets() writes as many characters as the user types, with no limit, which causes buffer overflows. fgets(buf, size, stdin) takes the buffer size and is the safe replacement.
652. **C** — strcat() finds the null terminator of dest and copies src after it, then adds a new terminator. It does no bounds checking, so dest needs strlen(dest) + strlen(src) + 1 bytes. Writing into a string literal is undefined behaviour.
653. **D** — c_str() gives a pointer to a null-terminated character array owned by the string, so it can be passed to C APIs such as printf("%s") or fopen(). The pointer becomes invalid if the string is modified or destroyed.
654. **A** — A narrow string literal is an array of const char that includes the terminating null, so "hi" is const char[3]. It converts to const char* when passed to a function, and it is not a std::string unless you write "hi"s.
655. **B** — Java strings are sequences of UTF-16 code units. A character outside the Basic Multilingual Plane, like U+1F600, is stored as a surrogate pair, so length() is 2. codePointCount() would give 1.
656. **C** — Quantifiers are greedy by default. Adding ? makes them lazy, so in "<a><b>" the pattern <.*> matches the whole string while <.*?> matches "<a>" and then "<b>".

657. **D** — Immutability means a String's contents cannot be wiped, and the object may linger in the heap (or the pool) until collected. A char[] can be cleared with Arrays.fill(pw, '\0') as soon as it is no longer needed; this is why Console.readPassword() returns char[].

658. **A** — A string is an iterable of its characters, so join() places "-" between 'a', 'b' and 'c'. No separator is added at the ends.

659. **B** — title() capitalises the first letter of each word and lower-cases the rest. capitalize() would give "Nsct exam" and upper() would give "NSCT EXAM".

660. **C** — With + and a string operand, JavaScript concatenates, giving "53". The - operator has no string meaning, so "5" is converted to the number 5 and the result is 2.

661. **D** — trim() returns a new string without leading and trailing whitespace, "NSCT", whose length is 4. The original string is unchanged.

662. **A** — replace() first turns the string into "a,b,c", and split(",") then breaks it into three parts. Chaining works because each method returns a new string.

663. **B** — s is a modifiable char array, and in ASCII 'h' (104) minus 32 is 'H' (72). strlen counts characters up to the null terminator, so it is still 5.

664. **C** — zfill(5) pads with zeros on the left to width 5. center(6, "*") adds 4 fill characters split evenly, 2 on each side.

665. **D** — The second argument, maxsplit=1, allows at most one split, so the string is cut at the first '=' only. rsplit("=", 1) would cut at the last one.

666. **A** — In the format spec, ',' adds thousands separators and '.2f' rounds to two decimal places in fixed-point notation.

667. **B** — index() returns the first match, at position 2. rfind() searches from the right and returns the start of the last match, at position 5.

668. **C** — slice(-6) counts from the end and gives "Script". substring() swaps its arguments when start > end, so substring(4, 1) is substring(1, 4), which is "ava".

669. **D** — padStart(3, "0") pads on the left to length 3, giving "007". repeat(2) gives "abab", and + joins the two strings.

670. **A** — StringBuilder is mutable and its methods change the same object. append gives "abcde", reverse gives "edcba", and insert(0, 'X') puts X at the front.

671. **B** — The first 'a' in "banana" is at index 1. "an" occurs at 1 and 3, and lastIndexOf returns the start of the last occurrence, 3.

672. **C** — substr(7) with no length takes everything from index 7 to the end, "ming". find('m') returns the first 'm', at index 6.

673. **D** — This is a Caesar shift of 3 on lowercase letters. % 26 wraps past 'z', so x->a, y->b and z->c. Without the modulus the result would be the characters after 'z' in ASCII.

674. **A** — swapcase() turns every lowercase letter upper and vice versa. In "level up" the space is at index 5, so "up" starts at index 6.

675. **B** — \s+ matches one or more whitespace characters (spaces and tabs), and re.sub replaces each run with a single space. This is a common way to normalise spacing.

676. **C** — ^7 centres the text in a field of width 7, leaving 5 spaces. When padding cannot split evenly, Python puts the extra space on the right, so it is 2 on the left and 3 on the right.

677. **D** — compareTo returns the difference of the first differing chars, or the length difference if one string is a prefix of the other. "Java" vs "Jav" gives 4 - 3 = 1; 'a' - 'b' gives -1.

678. **A** — std::string is mutable. += gives "abcd", insert(1, "XY") gives "aXYbcd", and erase(0, 2) removes 2 characters from index 0, leaving "Ybcd" of size 4.

679. **B** — split() treats its argument as a regular expression, where . means any character. Every character is a delimiter, so all pieces are empty, and Java drops trailing empty strings, giving length 0. Use split("\\.") to split on a literal dot.

680. **C** — String literals may live in read-only memory, and writing to one is undefined behaviour (often a segmentation fault). char p[] = "hello" copies the literal into a modifiable array.

681. **D** — When the source is at least n characters long, strncpy writes exactly n characters and no '\0'. d is then not a valid string, so %s keeps reading memory. Use a buffer of size 4 and set d[3] = '\0'.

682. **A** — "hello world".split(" ") gives ['hello', '', 'world']. split() with no argument splits on runs of whitespace and ignores empty pieces, giving 2 words.
683. **B** — .* is greedy, so it runs from the first < to the last >. The lazy .*? stops at the first >, giving each tag separately. A negated class such as r"<[^>]*>" also works.
684. **C** — String comparison is lexicographic: "10" < "2" because '1' < '2'. To sort numerically, pass a comparator such as (a, b) => a - b.
685. **D** — input() always returns a str, so + concatenates "3" and "4". Converting with int() (or float()) makes + add numbers, giving 7.
686. **A** — "a" + "b" is a compile-time constant, so it becomes the pooled literal "ab" and a == b is true. c + "b" uses a non-final variable, so it is built at run time as a new object and a == d is false. Use equals() for content.
687. **B** — find() returns the position of the match (0 here), which converts to false, and returns npos (non-zero, so true) when there is no match. The test is therefore backwards. Write if (s.find("x") != std::string::npos).
688. **C** — After lower() the text is "disk error at 10:02", which does not contain the capitalised "Error". Both sides must use the same case, for example "error" in log.lower().
689. **D** — The output is "False 4 5". \u00e9 is one precomposed character, while e + \u0301 is two code points that render the same. Unicode normalisation (NFC or NFD) makes them compare equal.
690. **A** — StringBuilder inherits equals() from Object, which is reference equality. Converting with toString() gives String objects, whose equals() compares characters.
691. **B** — A slice excludes its stop index, so [-3:-1] stops before the last character. Leaving the stop out, [-3:], runs to the end and gives "txt". In real code, os.path.splitext() is more robust.
692. **C** — A plain split cannot tell a delimiter comma from a comma inside quotes. csv.reader understands quoting and returns ['name', 'Khan, Ali', '21'].
693. **D** — sorted() compares exact characters, so 'D' differs from 'd' and the space counts. Removing spaces and lower-casing both strings before sorting makes the check pass.
694. **C** — A Java array has a fixed length set at creation. ArrayList wraps a resizable array and reallocates when it fills up. Both are zero-indexed, and ArrayList stores objects (Integer), not primitives.
695. **C** — Java initialises array elements to default values: 0 for numeric types, false for boolean, '\u0000' for char and null for references. Garbage values are a C/C++ issue with uninitialised local arrays.
696. **B** — Arrays expose length as a field, with no parentheses. String uses the method length(), and collections such as ArrayList use size().
697. **D** — A tuple cannot be changed after it is created: no item assignment, append or remove. Lists, dicts and sets are all mutable. (A frozenset is the immutable version of a set.)
698. **D** — With zero-based indexing the first element is at 0, so the last is at n - 1. Using n as an index is the classic off-by-one error.
699. **C** — A hash map hashes the key straight to a bucket, so lookup is O(1) on average. Many collisions can make the worst case worse; tree-based maps such as Java's TreeMap give O(log n).
700. **D** — TreeMap is a red-black tree ordered by the keys' natural order or a Comparator. HashMap and Hashtable make no ordering promise, and LinkedHashMap keeps insertion order, not sorted order.
701. **B** — Since Python 3.7, insertion order is part of the language specification for dict. Iteration returns keys in the order they were first inserted; updating a value does not move its key.
702. **C** — v.at(i) is bounds-checked and throws std::out_of_range for an invalid index. v[i] does no check, so an out-of-range index is undefined behaviour. Both return a reference to the element.
703. **C** — Raw C++ arrays have no bounds checking, so writing past the end is undefined behaviour. It might crash, corrupt another variable or seem to work. Java, by contrast, throws ArrayIndexOutOfBoundsException.
704. **B** — Both are backed by a contiguous array, so inserting at index 0 shifts all n elements right. Appending at the end is amortised O(1). A linked list or a deque supports O(1) insertion at the front.
705. **B** — Java 2D arrays are arrays of arrays. new int[3][] creates 3 row references set to null, and each can point to an array of any length, for example m[0] = new int[2]; m[1] = new int[5];.
706. **C** — A set stores each distinct value once; adding an existing value has no effect. Lists, arrays and queues all allow duplicates.

707. **A** — Dict keys (and set elements) must be hashable, which in practice means immutable. A list can change, so it has no hash. A tuple such as (1, 2) works as a key.
708. **D** — Negative indices count from the end in Python, so a[-1] is the last element and a[-2] the second last. Java and C++ arrays do not support negative indices.
709. **C** — std::map is ordered by key and gives O(log n) operations. std::unordered_map is a hash table with average O(1) operations and no ordering. Neither allows duplicate keys; std::multimap does.
710. **D** — A dict cannot hold the same key twice. In a literal, a later value for the same key silently replaces the earlier one, so 'a' maps to 2.
711. **B** — A set keeps one copy of each value, so {1, 2, 2, 3, 3, 3} becomes {1, 2, 3} and its length is 3.
712. **A** — Row-major means whole rows are stored one after another. a[2][3] comes after 2 full rows of 5 plus 3 more elements: 2 x 5 + 3 = 13 elements, so the address is 1000 + 13 x 4 = 1052.
713. **B** — b = a copies the reference, not the list, so a and b name the same object. Appending through b is visible through a.
714. **A** — a[:] creates a new (shallow) copy, so appending to b leaves a with 3 elements while b has 4.
715. **B** — append() adds its argument as one element, so the list becomes [1, 2, [3, 4]] with length 3. extend([3, 4]) would add two elements and give length 4.
716. **D** — get(key, default) returns the default when the key is missing instead of raising KeyError. d['z'] would raise KeyError.
717. **B** — Each word starts at 0 through get(w, 0) and is incremented, so 'to' and 'be' reach 2. Keys keep the order in which they were first inserted, not alphabetical order.
718. **A** — A - B keeps the elements of A not in B: {1, 3}. A ^ B (symmetric difference) keeps elements in exactly one of the sets: 1 and 3 from A and 6 from B.
719. **D** — The main diagonal is m[0][0], m[1][1] and m[2][2], which is 1 + 5 + 9 = 15.
720. **A** — The comprehension builds row j of the result from column j of m, which is the transpose. A 2 x 3 matrix becomes 3 x 2, with the first column (1, 4) as the first row.
721. **C** — A 3 x 4 array has 3 x 4 = 12 int elements. sizeof gives the size of the whole array in bytes: 12 x 4 = 48.
722. **D** — Arrays are objects in Java, so b = a copies the reference. a and b point to the same array, so the change through b shows up as a[0] = 9.
723. **A** — A Map holds each key once, so the second put replaces the value and size() stays 1. put() returns the previous value for that key, which was 1.
724. **D** — [[0] * 3] * 3 repeats a reference to one inner list three times, so changing grid[0][0] changes every row. Build independent rows with [[0] * 3 for _ in range(3)].
725. **B** — With an int argument, Java picks remove(int index) over remove(Object), so the element at index 1 (20) is removed. To remove the value 20 itself you must write l.remove(Integer.valueOf(20)).
726. **B** — The for-each loop uses an iterator, which detects that the list was modified structurally outside it and throws ConcurrentModificationException. Iterator.remove() or removeIf() change the list safely.
727. **A** — Removing an item shifts later items left while the loop's internal index moves on. After 2 is removed, 4 sits at index 0 and is skipped; 6 is removed next and 7 is checked. So 4 survives. Iterate over a copy or use [n for n in nums if n % 2].
728. **A** — sort() modifies the list in place and returns None, so nums = nums.sort() throws the sorted list away. Use nums.sort() on its own line, or nums = sorted(nums).
729. **C** — A hash set answers membership in average O(1) time and stores each name once. contains() on an ArrayList or a linked list scans linearly in O(n).
730. **A** — A map pairs each ID with its marks, and TreeMap keeps keys sorted, so iteration is already in ID order. HashMap has no ordering guarantee, a set has no values and a fixed two-element array cannot hold the records.
731. **A** — a and b are different array objects, so a == b is false. Arrays do not override equals(), so a.equals(b) also compares references and is false. Arrays.equals compares element by element and returns true.
732. **C** — Without overridden equals() and hashCode(), HashSet uses Object's identity versions, so two Point objects with the same coordinates count as different and both are stored. Override both methods to treat them as equal.

733. **D** — The JVM checks every array access. Index 3 is outside 0 to 2 for an array of length 3, so it throws ArrayIndexOutOfBoundsException. Java arrays never resize.
734. **A** — append(x) always adds exactly one item, so [4, 5] becomes a nested list. extend(x) iterates over x and adds each element. Both change a in place and return None.
735. **B** — Binary search halves the search range by comparing with the middle element, which only works when the elements are in sorted order. It needs O(1) indexing, so arrays suit it better than linked lists.
736. **C** — Java generics work only with reference types. Integer wraps int, and autoboxing converts between them automatically when you call add() or get().
737. **D** — LinkedHashMap keeps a linked list through its entries, so iteration follows insertion order. HashMap gives no order guarantee, and TreeMap sorts by key.
738. **A** — An array computes the address of element i directly as base + i x size. A linked list must follow i links from the head. Front insertion and removal are O(1) for a linked list.
739. **B** — std::set is an ordered container, usually a balanced binary search tree, that holds unique keys. std::multiset allows duplicates, and std::unordered_set uses hashing with no order.
740. **C** — Counter is a dict subclass for counting hashable items. Counter("banana") holds {'a': 3, 'n': 2, 'b': 1}, and most_common() lists the top counts.
741. **D** — getOrDefault returns the stored value if the key exists, otherwise the default given, without changing the map. Plain get() would return null for a missing key.
742. **A** — HashSet orders elements by hash bucket, which is an implementation detail. Use LinkedHashSet for insertion order or TreeSet for sorted order.
743. **B** — A dynamic array keeps spare capacity. When it fills, it allocates a larger array (for example 1.5x or about 2x) and copies everything, but because capacity grows geometrically, the total cost of n appends is O(n), or O(1) each on average.
744. **C** — The Deque interface, implemented by ArrayDeque, provides push/pop for stack use and offer/poll for queue use. Stack extends the synchronized Vector and is kept only for backward compatibility.
745. **D** — Removing the first element of a list shifts all the others left. deque is a double-ended structure with O(1) appends and pops at both ends.
746. **A** — capacity() >= size(). When size reaches capacity, the next push_back reallocates a larger buffer, which also invalidates existing iterators and pointers. reserve() raises capacity in advance.
747. **B** — A set locates elements by their hash, which must not change while stored. Tuples of immutable values are hashable; adding a list raises TypeError: unhashable type: 'list'.
748. **C** — frozenset has the same membership and set-algebra operations as set but no add() or remove(). Being immutable, it is hashable.
749. **D** — dict.keys(), values() and items() return view objects that reflect later changes. Wrap them in list() to take a snapshot. Changing the dict's size while iterating a view raises an error, but a view used afterwards is fine.
750. **A** — Column-major order stores the whole first column, then the second, and so on, so a[i][j] and a[i+1][j] are adjacent. C and C++ use row-major order, where rows are contiguous.
751. **B** — Zero-length arrays are valid objects and are often returned instead of null to mean 'no results'. Only a negative size throws NegativeArraySizeException.
752. **C** — Arrays.asList wraps the array, so set() works and writes through to the array, but the size cannot change. Use new ArrayList<>(Arrays.asList(...)) for a growable list.
753. **D** — HashMap allows one null key (stored in bucket 0) and null values. TreeMap with natural ordering must call compareTo on the key, so null throws NullPointerException; Hashtable and ConcurrentHashMap reject null keys and values.
754. **A** — The address of a[i][j] is base + (i x cols + j) x size, so cols must be known at compile time. Only the first dimension can be left out, because the array decays to a pointer to its first row, int (*)[4].
755. **B** — The Javadoc guarantees that Arrays.sort(Object[]) is stable (a TimSort-based merge sort, O(n log n)). For primitives it uses dual-pivot quicksort, where stability is meaningless for equal values anyway.
756. **C** — Java arrays are covariant, so a String[] can be referred to as Object[] and the assignment compiles. The array remembers it is a String[], and storing an Integer fails with ArrayStoreException. Generic collections avoid this by being invariant.

757. **D** — An array parameter is really a pointer, and the 10 is ignored. sizeof(arr) is sizeof(int*), so the length must be passed separately. Compilers such as GCC warn about this.

758. **A** — extend adds 4 and 5 (length 5), then append adds the list [6] as one element (length 6).

759. **B** — range(5) gives 0 to 4. x % 2 is truthy only for odd x, so the comprehension keeps 1 and 3 and squares them.

760. **C** — Adding 2 again changes nothing because sets hold unique values. discard() removes an element only if present and never raises; remove(5) would raise KeyError.

761. **D** — The enhanced for loop visits every element, so s = 4 + 7 + 1 = 12. a.length is a field giving 3. s is an int, so + " " starts string concatenation only after the sum is computed.

762. **A** — Address = base + index x size = 2000 + 7 x 4 = 2028. Index 7 is the eighth element, but the offset counts the seven elements before it.

763. **B** — std::set drops the duplicate 3 and iterates in ascending order, printing 1, 2 and 3 with no separator.

764. **C** — a[1:3] takes indexes 1 and 2. a[::-1] is the reversed list [1, 8, 3, 5], whose first element is 1.

765. **D** — Slice assignment replaces the elements at indexes 1 and 2 with the items of the right-hand list. The list can shrink or grow, so two elements become one.

766. **A** — setdefault inserts the default only when the key is missing, and always returns the stored value. So 'b' gets a new list that is then appended to, and 'a' keeps 1.

767. **B** — most_common(n) returns a list of the n (element, count) pairs with the highest counts. 'a' appears three times in "banana".

768. **D** — Iterating a dict gives its keys, and key=d.get sorts them by their values 1, 2 and 3. The result is a list of keys, not values.

769. **A** — & is intersection, the elements in both sets. <= tests whether the left set is a subset of the right, which it is.

770. **B** — const stops reassignment, not mutation, so push works and a is [1, 2, 3, 4]. map doubles to [2, 4, 6, 8], and filter keeps values above 4.

771. **C** — Reading past the end of a JavaScript array is not an error; it simply gives undefined. Unlike Java, there is no index exception.

772. **D** — add(0, "c") inserts at the front, shifting the others: [c, a, b]. set(2, "d") replaces index 2, which is "b".

773. **A** — Arrays.sort sorts in place, and Arrays.toString prints the elements. binarySearch returns the index of 5 in the sorted array, which is 2.

774. **B** — TreeMap keeps keys in sorted (alphabetical) order, and AbstractMap.toString prints entries as key=value. Sorting by value would give fig, pear, apple.

775. **C** — In column-major order the offset is (j x rows + i) x size = (2 x 3 + 1) x 2 = 14, so the address is 514. Row-major would give (1 x 4 + 2) x 2 = 12, that is 512.

776. **D** — m[::-1] reverses the rows to [[3, 4], [1, 2]], and zip(*...) transposes them, giving columns (3, 1) and (4, 2). Together this rotates the matrix 90 degrees clockwise.

777. **A** — push adds to the front, giving [2, 1], and offerLast adds to the back: [2, 1, 3]. pop removes from the front (2), and peekLast looks at the back (3).

778. **B** — list.copy() makes a new outer list but reuses the same inner list objects, so appending to a[0] shows up in b[0]. deepcopy copies the inner lists too.

779. **C** — For lists, += extends the existing object in place, so b sees 2. a = a + [3] builds a new list and rebinds only a, so b is unaffected.

780. **D** — Array.prototype.sort without a comparator converts elements to strings and sorts them in UTF-16 order, so "10" comes before "5". A numeric comparator fixes it.

781. **A** — map::operator[] default-constructs and inserts the value for a missing key, so testing m["c"] adds it. Use m.count("c") or m.find("c") to test without inserting; m.at() is the version that throws.

782. **B** — asList takes varargs T..., and T cannot be int, so T is inferred as int[] and the list holds one element: the array itself. Use Integer[] or IntStream.of(nums).boxed().toList() for a list of three numbers.

783. **C** — Autoboxing uses Integer.valueOf, which caches only -128 to 127 by default. 1000 is boxed twice into separate objects, so reference comparison is false. equals() or intValue() compares values.

784. **D** — Default values are evaluated once when the def runs. Both calls append to and return the same list object, and print shows it after both calls, so both print [1, 2]. The idiom is basket=None, then basket = [] if basket is None.

785. **A** — range's stop is exclusive, so range(3) gives 0, 1, 2 and a[3] is never added. range(len(a)) already covers every index.

786. **B** — First-in, first-out processing is exactly what a queue models: add at the back, remove from the front, both O(1) with a deque.

787. **C** — A hash map gives average O(1) update per word, so the whole pass is O(n). list.count makes the job O(n^2), and a set cannot store counts.

788. **D** — list.index() raises ValueError instead of returning -1. The in operator returns True or False safely. Lists have no find() method; that exists only on strings.

789. **A** — Python raises RuntimeError: dictionary changed size during iteration. Iterating over a copy of the keys, or building a new dict with a comprehension, avoids it.

790. **B** — The Javadoc says the array must be sorted and, if it is not, the results are undefined. Binary search does not check order; it just compares with middle elements, so it can miss values that are present.

791. **C** — g is an array of int[] objects. Arrays.toString calls toString() on each row, which prints the type code [I and a hash code. deepToString recurses into nested arrays and prints [[1, 2], [3, 4]].

792. **D** — The comprehension evaluates [0] * 3 afresh on each pass, creating three separate list objects. The other forms put the same row object in the list three times, so one change appears in every row.

793. **D** — A class defines the fields and methods that its objects will have. Each object created from it (for example with new in Java or C++, or by calling the class in Python) has its own copy of the instance data.

794. **C** — A constructor shares the class's name and declares no return type. Both languages allow several overloaded constructors. Python's equivalent is the __init__ method.

795. **D** — class Dog extends Animal implements Pet is the pattern. An interface that builds on another interface also uses extends, but a class always uses implements for interfaces.

796. **C** — Java allows single inheritance of classes to avoid the diamond problem with state. A class may implement any number of interfaces, for example class A extends B implements C, D.

797. **A** — Overloading is compile-time (static) polymorphism: the compiler picks a method by its argument types. Overriding is run-time (dynamic) polymorphism: the method called depends on the object's actual class.

798. **B** — Encapsulation keeps fields private and exposes controlled access through methods such as getters and setters, so the class can enforce rules on its own data. The other options describe inheritance, polymorphism and concurrency.

799. **D** — Abstraction shows the essential operations and hides implementation details, as abstract classes and interfaces do: a caller uses shape.area() without knowing the formula. Encapsulation is the mechanism that hides the data; the two ideas work together.

800. **B** — A Java method signature for overloading is the name plus the parameter types. Two methods differing only in return type would be ambiguous, so the compiler reports a duplicate method. C++ has the same rule.

801. **D** — An abstract class may mix abstract and concrete methods, hold instance fields and declare constructors, which subclasses call through super(...). Only new on the abstract class itself is forbidden.

802. **C** — Interface fields are constants: implicitly public, static and final, so every implementer shares one value that cannot change. Since Java 8, interfaces can also have default and static methods, and since Java 9 private methods.

803. **B** — A class with at least one pure virtual function cannot be instantiated. A C++ "interface" is by convention a class whose member functions are all pure virtual, usually with a virtual destructor.

804. **C** — With no modifier, access is limited to the same package. private limits it to the class, protected adds subclasses in other packages, and public opens it to all.

805. **C** — Java's protected includes package access plus subclass access. C++ protected is narrower: only the class, its friends and derived classes, with no idea of packages. This difference is a common trap.

806. **D** — The only difference between class and struct in C++ is the default: class members (and base classes) are private by default, struct members are public.

807. **B** — This is the diamond problem. C++ allows multiple inheritance, and without virtual inheritance D contains two A subobjects. Virtual base classes make B and C share a single A. Java avoids the issue by not allowing multiple class inheritance.

808. **C** — Only inherited, non-final instance methods can be overridden. private methods are not inherited, a static method with the same signature only hides the parent's, and final forbids overriding.

809. **A** — The variable's type is Animal but the object is a Dog. Java calls instance methods by the object's run-time class (dynamic dispatch), so Dog's overriding sound() runs.

810. **D** — count is a class attribute shared by all instances, and each __init__ increments Counter.count. Three objects are created, so it reaches 3.

811. **B** — Python uses duck typing: any object with a speak() method can be used in the loop, whatever its class. Each call runs the method of that object's own class.

812. **A** — A Dog is an Animal, because a subclass instance passes isinstance for its parent. An Animal is not necessarily a Dog, so the second test is False.

813. **A** — Every constructor first calls its parent's constructor (an implicit super()), so construction runs from the top of the hierarchy down: A, then B, then C.

814. **A** — new C().hi() runs C's version, which returns "C" + B's hi(). B's version returns "B" + A's hi(), which is "A". Joining them gives CBA.

815. **A** — Python resolves methods with the C3 linearisation (MRO). D lists B before C, and A comes after both because both inherit from it, so the order is D, B, C, A, object and B's who() is found first.

816. **B** — A class that inherits from ABC and still has an @abstractmethod cannot be instantiated; Python raises TypeError. A subclass that overrides area() can be instantiated normally.

817. **D** — print() calls str() on the object, which uses the class's __str__ method when it is defined. Without __str__ (or __repr__) you would see the default <__main__.Point object at 0x...> text.

818. **C** — A name starting with two underscores is mangled to _Account__balance inside the class. Outside, the mangled name still works, so the first print shows 100, but a.__balance does not exist and raises AttributeError. Python has no true private access.

819. **B** — Static methods are not overridden; they are hidden and chosen at compile time from the reference type P, so p.id() returns "P". name() is an instance method, so dynamic dispatch picks C's version.

820. **B** — Overloads are chosen at compile time. f(5) matches int exactly. 5L is a long; there is no long version, and long widens to double but not to int. 'A' is a char, which widens to both int and double, and Java picks the most specific, int.

821. **C** — describe() is defined in Shape, but self is a Square, so self.area() dispatches to Square's override and returns 9. super().__init__("square") sets name before side is stored.

822. **D** — Inside the constructor the parameter w shadows the field, so w = w assigns the parameter to itself. The field keeps its default value 0. The fix is this.w = w;.

823. **A** — Through a Base reference, a non-virtual call is bound at compile time to Base::f, while a virtual call is dispatched at run time to the object's actual type, Derived::g. In Java every non-static, non-private, non-final method behaves like g.

824. **D** — Fields are not polymorphic in Java: p.n uses the declared type P, so it reads P's field, "P". get() is an overridden instance method, so C's version runs and returns C's own field n, "C".

825. **B** — Python has no signature-based overloading: the second def simply rebinds the name add. Calling add(1, 2) then misses c and raises TypeError. Use default arguments or *args instead.

826. **D** — Defining __init__ in Student replaces Person's, and Python does not call the parent's automatically. Adding super().__init__(name) as the first line sets self.name.

827. **A** — tricks is a class attribute, so every Dog shares the same list object. a.add mutates that shared list, and b sees the change. Create the list per object in __init__ with self.tricks = [].

828. **A** — Downcasting is legal at compile time because an Animal reference might point to a Dog. At run time the JVM checks the real object, an Animal, and throws ClassCastException. Test first with instanceof.

829. **A** — Overriding may keep or widen access (protected to public) but never narrow it. Narrowing would break code that calls describe() through a Base reference, so the compiler reports weaker access privileges.

830. **C** — The C++ standard says deleting a derived object through a base pointer whose destructor is not virtual is undefined behaviour. Declaring virtual ~Base() makes delete call ~Derived() first, then ~Base().

831. **C** — Inheritance models "is-a"; composition models "has-a". A car is not an engine, it has one, so Car should hold an Engine object. This also lets you swap engine types without changing Car's parent class.
832. **B** — Abstract classes can hold instance state and constructors; interfaces cannot have instance fields (only constants). Use an interface when unrelated classes need a common capability, and an abstract class when related classes share state and code.
833. **A** — A static field belongs to the class, not to any object, so there is exactly one copy no matter how many objects exist. Every object (and the class name itself) reads and writes that same variable.
834. **A** — `this` is a reference to the current object, the one used in the call such as acc.deposit(5). It is often used to tell a field apart from a parameter with the same name, as in this.balance = balance.
835. **B** — The compiler supplies a default no-argument constructor only if the class declares no constructors. Once Book(String) is written, no Book() exists, so new Book() does not compile until one is declared.
836. **C** — A final class cannot be subclassed; String is final so its behaviour cannot be changed by a subclass. abstract means the opposite (the class must be extended to be used), and top-level classes cannot be static or private.
837. **D** — Python passes the instance explicitly as the first argument, so obj.m(5) becomes Class.m(obj, 5). The name self is a convention, not a keyword. cls is the convention for class methods, which receive the class.
838. **B** — Every Java class except Object itself extends exactly one class. When no extends clause is written, the compiler makes java.lang.Object the superclass, which is why every object has toString(), equals() and hashCode().
839. **D** — Polymorphism means one interface, many forms: the same call shape.area() runs different code depending on the actual object. Encapsulation is about hiding data, abstraction about hiding detail, and composition about building objects from other objects.
840. **B** — C++ (class D : public B, public C) and Python (class D(B, C)) support multiple inheritance of classes. Java allows a class to extend only one class, and gets a similar effect by implementing several interfaces.
841. **C** — The compiler picks among overloads using the declared argument types (static binding). Which override runs depends on the actual object, so it is chosen at run time (dynamic binding) through virtual dispatch.
842. **B** — Composition is strong ownership: the part's lifetime depends on the whole, shown with a filled (black) diamond on the whole's side. Aggregation is weaker shared ownership with a hollow diamond, and generalisation (inheritance) uses a hollow triangle pointing at the parent.
843. **C** — UML uses + for public, - for private, # for protected and ~ for package visibility. Static members are shown underlined, not with a symbol.
844. **D** — A class grants access with friend declarations, for example friend std::ostream& operator<<(std::ostream&, const Point&);. The friend is not a member and is not inherited, but it can read private data. A member callable without an object is a static member function.
845. **C** — Java 8 added default and static methods with bodies to interfaces. Interfaces still cannot have instance fields (fields are implicitly public static final) or constructors, and their abstract methods are implicitly public.
846. **A** — @classmethod passes the class (or subclass) the call was made through as cls, which makes it useful for alternative constructors. @staticmethod is a plain function inside the class namespace and gets neither self nor cls. Both can be called on the class or an instance.
847. **C** — Point b = a; declares and initialises a new object from an existing one, so the copy constructor Point(const Point&) runs. The copy assignment operator runs only when an already existing object is assigned, as in b = a; on a later line.
848. **D** — The Open/Closed Principle says you should be able to add new behaviour, for example a new Shape subclass, without editing existing, tested code. Polymorphism through abstract classes or interfaces is the usual way to achieve it.
849. **A** — High cohesion means each class has one focused job; low coupling means classes depend on each other as little as possible. Together they make classes easier to change, test and reuse.
850. **D** — Making the constructor private stops other classes from calling new, and a static getInstance() (or a static final field, or an enum) hands out the one shared object. final or abstract alone do not stop extra instances being created.
851. **A** — Java always initialises the parent part first. If you do not write super(...), the compiler adds super(); for you. If the parent declares only, say, Parent(int), that implicit call has no match and the subclass will not compile.

852. **B** — LSP says a subtype must honour the base type's contract: callers written against the base class should keep working when handed a subclass object. A subclass that throws on an inherited method or weakens guarantees violates it.

853. **A** — Since Java 5 an overriding method may return a subtype of the original return type (a covariant return type), so Dog create() overrides it. A wider type such as Object, a different type such as void, or adding a broader checked exception all fail to compile.

854. **B** — When a Derived is assigned or passed by value to a Base, the Base copy constructor copies just the Base subobject. The copy is a genuine Base, so virtual calls on it run Base's versions. Passing by reference or pointer avoids slicing.

855. **D** — abstract requires a subclass to override the method, while final forbids any override, so the combination is contradictory and rejected. The other pairs are all legal on a method.

856. **D** — The contract requires equal objects to have equal hash codes; otherwise HashMap and HashSet place equal objects in different buckets and treat them as different keys. Unequal objects are allowed to share a hash code (a collision), so distinct codes are not required.

857. **A** — c1.wheels = 3 creates an instance attribute on c1 that hides the class attribute. c2 has no instance attribute, so it still reads Car.wheels, which is unchanged at 4.

858. **B** — len(p) calls p.__len__(), which returns the length of the songs list, 3. Defining a special (dunder) method lets your own class work with built-in functions; without __len__ the call would raise TypeError.

859. **C** — The compiler picks the overload by the number of arguments: area(3) uses the one-parameter version (9) and area(2, 5) the two-parameter version (10). Both are ints, so + adds them to 19.

860. **A** — Construction builds the base part first, so A() prints before B(). Destruction runs in the reverse order: ~B() first, then ~A(). The output is ABba.

861. **D** — The + operator calls Vec(1, 2).__add__(Vec(3, 4)), which returns Vec(4, 6). print uses __str__, which falls back to __repr__ here. This is operator overloading in Python.

862. **B** — f is a property, so t.f calls the getter without parentheses and computes from the current _c. _c was changed to 0, giving 0 * 9 / 5 + 32 = 32.0; the / operator always returns a float in Python 3.

863. **C** — super() follows the MRO of the object's class, D: D, B, C, A, object. So super() inside B calls C.go, not A.go, and each class runs once: D + B + C + A = DBCA.

864. **D** — next is static, so all Tickets share it. Post-increment stores the old value then adds 1: a gets 1, b gets 2, c gets 3, and next ends at 4. Each id is an instance field, so a keeps 1.

865. **C** — this(...) must run first, so Rect() delegates to Rect(1), which delegates to Rect(1, 1). The innermost body prints C and sets both sides to 1, then control returns to print B and finally A. w * h is 1.

866. **A** — The static member is shared by all objects. Three objects exist inside the inner block, so it prints 3. At the closing brace b and c are destroyed and their destructors decrement live to 1, which is printed next.

867. **D** — A default method is inherited by implementing classes that do not override it. Polite overrides greet(), so g1 prints Good day; Plain inherits the default and prints Hello. Plain compiles because greet() is not abstract.

868. **D** — Called through MyDate, the classmethod receives cls = MyDate, so cls(y, m, d) builds a MyDate. The string splits into 2026, 11 and 25, and m is 11. This is why alternative constructors use @classmethod rather than hard-coding Date.

869. **B** — Without __eq__, == falls back to identity, and two separate P objects are different objects, so it is False. Q defines __eq__ to compare the x values, so Q(1) == Q(1) is True.

870. **A** — In Java an overridden method called from a superclass constructor dispatches to the subclass. But Child's field initialiser x = 5 runs only after Base() returns, so the first call sees the default 0. The second call, after construction, prints Child 5.

871. **D** — Passing by value copies only the Animal part of c (object slicing), so the parameter really is an Animal and its virtual call runs Animal::name. Passing by reference keeps the Cat object, so dynamic dispatch runs Cat::name.

872. **B** — Overload selection happens at compile time using the declared type of the argument. o is declared as Object, so print(Object) is chosen even though it holds a String. The literal "hi" has static type String, so print(String) is chosen.

873. **A** — Each object runs its own area(): 4 + 6 + 1 = 11. Python uses duck typing, so the objects need only a method with the right name; no common base class is required for this polymorphic call.

874. **A** — Manager inherits raised() but overrides the class attribute raise_pct. self.raise_pct looks up the attribute on the actual object's class first, so the Employee gets 10% (1100) and the Manager gets 20% (1200).
875. **D** — A subclass without a constructor gets a default one that passes its arguments to super(...), so name is "Rex". Dog.speak overrides Animal.speak and calls it through super.speak(), then appends " and barks".
876. **A** — A static initialiser runs once, when the class is first initialised. An instance initialiser block runs every time an object is created, just before the constructor body. So two objects give S I C I C.
877. **B** — Private methods are not inherited, so B.id() is a new method, not an override. Inside A, the call id() binds to A's private id(), and it prints A. Declaring a method with the same name in B is legal.
878. **D** — Callers written for Bird expect fly() to work, but a Penguin cannot be substituted for a Bird without breaking them. That is a Liskov Substitution violation; a better model separates flying birds (for example a Flyer interface) from birds in general.
879. **B** — Validation lives in the age setter, which runs only on assignments to age. The constructor writes to _age, the hidden field, so the check never runs. Routing construction through self.age = age enforces the rule everywhere.
880. **C** — A subclass of an ABC stays abstract until every abstract method is overridden by name. pay_amount is a different method, so pay is still abstract and instantiating Card raises TypeError. Renaming it to pay fixes the error.
881. **B** — DIP says high-level modules should depend on abstractions, not concrete classes. Injecting a Database interface lets production pass MySqlDatabase and tests pass a fake, without changing ReportService.
882. **C** — Calling parent initialisers by class name ignores the MRO, so both B and C call A.__init__ and the shared base is initialised twice. Using super().__init__() in every class follows the MRO D, B, C, A and runs each initialiser exactly once (printing D B C A).
883. **C** — private and final protect the reference, not the list object. The getter hands out the real list, so add("Ali") changes the team's internal state. Returning Collections.unmodifiableList(members) or List.copyOf(members) preserves encapsulation.
884. **C** — Object.equals takes an Object parameter, so equals(Pt) is a new overload, not an override. With static type Object only equals(Object) is visible, and since Pt never overrode it the identity check gives false. With static type Pt the more specific equals(Pt) is chosen, giving true. Adding @Override would have caught the mistake at compile time.
885. **B** — If every class implements area(), new shapes are added by writing a new class, and calling code never changes. The other options keep a central list of kinds that must still be edited for every new shape.
886. **B** — In C++ the object's dynamic type during Base's constructor is Base, because the Derived part does not exist yet; so hello() runs Base::hello. After construction d.hello() runs Derived::hello. Java, by contrast, dispatches to the subclass override even from the superclass constructor.
887. **D** — A Stack is not a list in the substitutable sense: it should not support insert, sort or slicing assignment. Composition (self._items = []) hides the list and exposes only stack operations, which keeps the invariant.
888. **C** — Dynamic polymorphism is overriding: the method that runs depends on the actual object at run time. Constructor and method overloading are resolved at compile time, and a getter is encapsulation.
889. **C** — Calling through the class gives the plain function, which still needs its self argument. Greeter().hello() (or marking it @staticmethod and removing self) fixes it.
890. **A** — @Override asks the compiler to confirm the method overrides an instance method. A static method with the same signature only hides the parent's version, so there is nothing to override and javac reports an error. Removing the annotation makes it compile as method hiding.
891. **A** — A property defined only with a getter is read-only; assigning to it raises AttributeError. Area is derived from r, so the design is right: change r, and area follows.
892. **C** — Python has no private keyword. _balance signals "internal, please don't touch", but obj._balance still works. __balance becomes _ClassName__balance to avoid accidental clashes, and can still be reached by that mangled name.
893. **A** — Non-static local variables live in the stack frame created when the function is called and are released automatically when it returns. The heap is only used when you allocate explicitly (new/malloc); static and global variables sit in the static data segment; the text segment holds machine code.

894. **D** — Memory from new[] must be released with delete[], which knows to destroy every element of the array. Plain delete on an array, or free() on memory from new, is undefined behaviour, and setting the pointer to nullptr only loses the address, causing a leak.

895. **D** — Every Java object is allocated on the heap. A local variable such as `StringBuilder sb` lives in the stack frame and holds a reference to that heap object. When no references remain, the garbage collector can reclaim the object.

896. **B** — A leak happens when allocated memory is never freed even though the program has stopped using it, typically because the last pointer to it was lost. Leaked memory accumulates, so long-running programs can exhaust memory. Out-of-bounds reads are a different bug (buffer overrun).

897. **C** — A dangling pointer points to memory that is no longer valid, for example after delete or after the local variable it pointed to went out of scope. Dereferencing it is undefined behaviour. A nullptr is the safe opposite: it clearly points to nothing.

898. **D** — Each call pushes a new frame on the thread's stack; unbounded recursion fills the stack and the JVM throws StackOverflowError. OutOfMemoryError is thrown when the heap (or another memory area) cannot satisfy an allocation. Both are Errors, not Exceptions.

899. **D** — CPython stores a reference count in each object; when the count falls to zero the object is deallocated immediately. Because reference counting cannot free cycles (objects that refer to each other), CPython also has a generational cyclic garbage collector in the gc module. del only removes a name, decreasing the count by one.

900. **A** — With RAII, a stack object owns a resource (memory, a file, a lock). When the object goes out of scope, normally or because an exception is thrown, its destructor runs and releases the resource. std::vector, std::unique_ptr, std::fstream and std::lock_guard all follow this idiom.

901. **B** — std::unique_ptr has a deleted copy constructor, so exactly one unique_ptr owns the object at a time; ownership can be transferred with std::move. shared_ptr allows many owners through a reference count, and weak_ptr is a non-owning observer of a shared_ptr.

902. **D** — weak_ptr observes a shared object without owning it, so it does not keep the object alive. You call lock() to get a temporary shared_ptr if the object still exists. This is the standard way to break shared_ptr reference cycles. auto_ptr is an old ownership pointer, removed in C++17.

903. **C** — Java's collectors trace reachability from GC roots (local variables on thread stacks, static fields, and so on). An object nobody can reach is eligible, even if other unreachable objects still point to it. Setting one variable to null is not enough if another reference exists, and an object created in a method may still be referenced after it returns.

904. **B** — A C++ reference is an alias that must be bound at declaration and stays bound to the same object; assigning to it assigns to that object. A pointer is a variable holding an address: it can be nullptr and can be pointed elsewhere. A reference does not copy the object.

905. **A** — The weak generational hypothesis says most objects become garbage soon after allocation. Collecting a small young generation frequently is cheap and frees most garbage; survivors are promoted to the old generation, which is collected rarely. Freeing at a zero count is reference counting, a different technique.

906. **A** — System.gc() is only a request: the Java documentation says the JVM makes a best effort, and flags such as -XX:+DisableExplicitGC make it do nothing. Reachable objects are never freed, so you cannot use it to reclaim memory you still reference.

907. **C** — Mark-and-sweep traces the object graph from the roots and marks what it reaches; the sweep then walks the heap and frees anything unmarked. Because it uses reachability rather than counts, it collects reference cycles. Counting references is reference counting; moving survivors describes copying or compacting collectors.

908. **C** — External fragmentation means free space exists but is not contiguous. Compacting (mark-compact) and copying collectors slide or copy live objects into one region, leaving a single large free area. Internal fragmentation is wasted space inside allocated blocks, and a leak is memory never freed.

909. **D** — b = a copies the reference, not the list, so a and b name the same list object on the heap. Appending through b changes that one object, so len(a) is 4.

910. **B** — Slicing with a[:] creates a new list with equal contents. == compares values, so it is True; `is` compares identity (the same object in memory), so it is False.

911. **A** — p holds the address of x, so *p is x itself. Assigning 20 through *p changes x, and the program prints 20.

912. **B** — r is a reference (alias) to a, so r = r + 3 makes a equal 8. int b = r copies the value 8 into a separate variable; changing b to 100 does not affect a or r. Both a and r print 8.
913. **A** — p starts at arr[0]. p += 2 moves it forward by two int elements (not two bytes), so *p is arr[2] = 30 and *(p - 1) is arr[1] = 20. Pointer arithmetic scales by the size of the pointed-to type.
914. **B** — Each call allocates an array with new[]. When early is true the function returns before delete[], so that array's address is lost. f(true) is called twice, so two arrays leak; the f(false) call frees its array.
915. **C** — std::move transfers ownership from p to q. After the move p is empty (it compares equal to nullptr, which prints as 1) and q owns the int 7. unique_ptr cannot be copied, but it can be moved.
916. **C** — a and b share ownership, so the count is 2. Inside the block c adds a third owner, so use_count() prints 3. When the block ends, c is destroyed and the count drops back to 2.
917. **B** — sys.getrefcount reports an object's reference count (plus a temporary reference for the call itself, which cancels out in the difference). y = x adds one reference, and the list z stores x twice, adding two more, so the count rises by 3.
918. **C** — del a removes the name a, not the list. The list still has one reference (b), so its count stays above zero and it is not freed. b still prints [1, 2].
919. **A** — After del r1 the object still has one reference (r2), so it survives and "after del" prints first. r2 = None drops the count to zero, and CPython's reference counting calls __del__ immediately, printing "free 1" before "end". Other implementations such as PyPy may run __del__ later.
920. **A** — f(3) calls f(2), which calls f(1), which calls f(0). f(0) is the base case, but at that moment all four frames (n = 3, 2, 1, 0) are on the stack. Each frame is popped as its call returns.
921. **C** — Object 1 is still referenced by c and, after line 4, also by b. Object 2 was referenced only by b; when b is reassigned to c, nothing refers to object 2, so it is eligible. Eligibility depends on reachability, not on whether the collector has run yet.
922. **D** — x and y were the only outside references. Once both are null, the two nodes still point to each other, but neither can be reached from any GC root. Java's tracing collector ignores such "islands of isolation", so both nodes are eligible.
923. **B** — Returning the address of a local variable produces a dangling pointer: the frame that held x is gone once the function returns, and that stack space may be reused. Compilers usually warn. Fixes include returning by value, or allocating with std::make_unique and returning the smart pointer.
924. **A** — The compiler-generated copy constructor performs a member-wise (shallow) copy, so b1 and b2 hold the same address and both destructors free it. The Rule of Three says a class that needs a custom destructor usually needs a custom copy constructor and copy assignment too; holding a std::vector instead avoids the problem (Rule of Zero).
925. **B** — When main ends, the local shared_ptrs a and b are destroyed, dropping each count from 2 to 1. The remaining owner of each node is the other node's next pointer, so neither count reaches zero: a reference-cycle leak. Making one of the links a std::weak_ptr breaks the cycle.
926. **D** — A garbage collector frees only unreachable objects. The static field items is reachable from a GC root and holds a reference to every array ever added, so memory grows without limit. This is the classic Java "leak": unwanted references, often in static caches. The fix is to remove entries or bound the cache.
927. **C** — StackOverflowError means a thread's call stack ran out of space, which almost always comes from very deep or infinite recursion. Heap problems such as growing collections or many objects lead to OutOfMemoryError instead. System.gc() only requests a collection and does not consume stack space.
928. **C** — When an exception leaves process(), execution never reaches delete[], so the allocation leaks. Stack unwinding runs destructors of local objects, but a raw pointer has no destructor that frees anything. std::vector<int> data(n) (or std::unique_ptr<int[]>) releases the memory in its destructor whether the function returns normally or by exception.
929. **A** — n.self_ref = n creates a reference cycle, so after del n the object's count is still 1 and reference counting cannot free it; "after del" prints first. gc.collect() runs the cycle detector, which finds the unreachable cycle and frees it, calling __del__ ("freed") before "end".
930. **D** — The list had three references: a, the dictionary value b["k"], and the element c[0]. Setting a to None removes one. Reference counting frees the list only when the count hits zero, so both the dictionary entry and the list element must go.

931. **B** — Default argument values are evaluated once, when the function is defined, and stored on the function object. Every call without bucket gets that same list, so items accumulate across calls. The usual fix is `bucket=None`, then `if bucket is None: bucket = []` inside the function.
932. **D** — The C++ standard says the form of delete must match the form of new; mismatching them is undefined behaviour, whatever the element type. It may appear to work, crash, or corrupt the heap. Compilers cannot generally detect it; std::vector or std::unique_ptr<int[]> avoids the issue.
933. **D** — Objects with static storage duration live in a fixed area set up before main runs: initialised ones in the data segment, zero-initialised ones in BSS. They exist until the program ends, unlike stack locals, which vanish on return.
934. **D** — new allocates storage and then constructs the object in it, returning a T*. malloc only returns raw bytes as void* and calls no constructor. Neither frees memory automatically, and new int, for example, leaves the value indeterminate unless you write new int() to value-initialise it.
935. **C** — Both allocate from the heap and both need free(). calloc takes a count and an element size and zero-initialises the block; malloc's contents are indeterminate, so reading them before writing is a bug.
936. **D** — After the first delete the block belongs to the allocator again, so a second delete corrupts its bookkeeping and the behaviour is undefined. Setting the pointer to nullptr after delete makes a repeated delete harmless, and smart pointers prevent it entirely.
937. **A** — nullptr converts to any pointer type but not to int, so it avoids the overload confusion that NULL (often just 0) causes. null is Java, None is Python, and void is a type, not a value.
938. **C** — The text segment stores the executable code and is usually read-only. The data and BSS segments hold static variables, the heap holds dynamic allocations, and the stack holds call frames.
939. **A** — realloc may move the data. That is why you must use its return value and never the old pointer afterwards. If it fails it returns NULL and leaves the old block untouched, so writing p = realloc(p, n) directly can leak the original block.
940. **B** — A wild pointer is declared without a value (int* p;) and may point anywhere. A dangling pointer, by contrast, was valid once but its target has been freed or gone out of scope. Initialising pointers, for example to nullptr, avoids wild pointers.
941. **C** — A class that manages a resource (such as a raw pointer from new) needs a destructor to free it, and then the compiler-generated copies would share the pointer. So it also needs a deep-copying copy constructor and copy assignment. C++11 extends this to the Rule of Five with move operations.
942. **A** — The JVM interns string literals, so two identical literals refer to the same object, saving memory. Strings created with new String(...) are separate objects unless intern() is called.
943. **B** — Java 8 removed the Permanent Generation. Class metadata now lives in Metaspace, outside the Java heap, which grows by default and can be capped with -XX:MaxMetaspaceSize.
944. **A** — -Xmx sets the maximum heap and -Xss sets each thread's stack size. -Xms is the initial heap size, and -Xmn sizes the young generation.
945. **A** — Many collectors need the object graph to stop changing during some phases, so they pause every application thread. Concurrent collectors such as G1 and ZGC aim to keep these pauses short, but most still have brief stop-the-world phases.
946. **B** — The heap is split into from-space and to-space. Live objects reachable from the roots are copied into to-space contiguously, then the roles swap. Cost depends on live data rather than heap size, at the price of keeping half the space empty. This idea underlies young-generation collectors.
947. **C** — Allocators often round requests up, for example to a multiple of 16 bytes, and the extra bytes inside the block are wasted. Free memory scattered in gaps between blocks is external fragmentation; unreachable memory is a leak or garbage.
948. **C** — A weakref does not keep its target alive. Once the last strong reference goes, CPython frees the object, and calling the weak reference returns None. It is used for caches and observer lists that must not cause leaks.
949. **D** — shared_ptr needs a control block for its counts. make_shared puts it next to the object in a single allocation, which is faster and more cache-friendly, and there is no moment when a raw new result is unowned. It does not prevent cycles; weak_ptr does.
950. **A** — Resources that implement AutoCloseable, such as streams and database connections, have close() called in reverse order of declaration on every exit path. It manages resources such as file handles; memory is still reclaimed later by the garbage collector.

951. **A** — The JVM clears soft references only when it needs memory (all are cleared before an OutOfMemoryError), which suits memory-sensitive caches. Weak references do not hold the object at all, so it goes as soon as a collection finds it only weakly reachable; WeakHashMap uses them.
952. **D** — Finalization was deprecated in Java 9 and marked for removal in Java 18. A finaliser runs at some unknown time after the object becomes unreachable, possibly never, and slows collection. Use try-with-resources or java.lang.ref.Cleaner instead.
953. **B** — Each call pushes a frame holding arguments, locals, the return address and saved registers; returning pops it. Objects from new live on the heap, globals in the data segment, and code in the text segment.
954. **C** — Moving transfers ownership instead of copying: steal the pointer, then null it out in the source so the source's destructor frees nothing. Copying the elements is what a copy constructor does; sharing the pointer without nulling it leads to a double delete.
955. **C** — CPython limits the depth of Python calls, by default to 1000, and raises RecursionError (a subclass of RuntimeError) when it is exceeded. StackOverflowError is the Java equivalent. The limit can be changed with sys.setrecursionlimit.
956. **A** — Members are aligned to their natural boundaries for efficient access, so 3 unused bytes follow c and i sits at offset 4, giving 1 + 3 + 4 = 8. The exact layout is implementation-defined, which is why the stem states the convention.
957. **D** — copy.copy makes a new outer list whose elements are the same inner list objects. Appending to b[0] changes the inner list shared with a, but appending [4] to b only changes the new outer list.
958. **C** — deepcopy copies the dictionary and, recursively, the list inside it. a and b share nothing, so appending to b's list leaves a's list as [1].
959. **C** — The parameter first refers to the caller's list, so append(1) changes x. lst = [99] then rebinds only the local name to a new list; later changes affect that new list, not x.
960. **D** — Java passes references by value. The swap only exchanges the local copies a and b, so x and y in main are unchanged. After the swap, a refers to y's object, so append("!") changes that object, giving x y!.
961. **B** — Identical string literals are interned, so a and b refer to the same pooled object and == is true. new String creates a separate heap object, so a == c is false, while equals compares content and is true.
962. **B** — Autoboxing calls Integer.valueOf, which returns cached objects for -128 to 127, so a and b are the same object. 1000 is outside the default cache, so c and d are two different objects and == compares references. Use equals() to compare values.
963. **A** — unique_ptr deletes its object when it goes out of scope. b dies at the end of the inner block, printing -2 before c is created. At the end of main, locals are destroyed in reverse order of creation: c, then a.
964. **D** — weak_ptr does not add to the strong count, so use_count() is 1 and expired() is false (0). After s.reset() the last owner is gone, the int is destroyed, use_count() becomes 0 and expired() is true (1).
965. **A** — The pointer itself is passed by value. grow changes only its local copy (and leaks the new int), so q still points at 1. set dereferences the copy, which holds the same address, so it changes the int q points to.
966. **C** — The static local s is initialised once and keeps its value between calls (static storage), so it reaches 3. t is an ordinary stack local, recreated as 0 on every call, so it is 1. 3 * 10 + 1 = 31.
967. **B** — Calling a weak reference returns the object while it is alive. The weakref does not count as an owner, so del b drops the reference count to zero and CPython frees the object at once; calling r() then returns None.
968. **A** — When make returns, its frame goes and tmp's count drops to zero, so free 2 prints first. r survives because the returned lambda captures it in a closure. Setting f = None frees the lambda and with it the last reference to r, printing free 1 before end.
969. **A** — For lists, += extends the existing object in place, so b sees [1, 2, 3] and c is the same object as b. a + [4] builds a new list and rebinds a to it, so a is no longer b.
970. **B** — Local variables disappear when work() returns. Object 2 is still referenced by the static field keep, so it stays. Object 1 lost its only reference when a = c ran, and object 3 was referenced only by locals, so objects 1 and 3 are eligible.
971. **D** — Setting the variable to null removes one reference, but the Map still references the key object strongly, so its size stays 1 and the object cannot be collected. WeakMap keys are held weakly, so its entry alone would not prevent collection.

972. **D** — The copy constructor allocates a new int with the same value (a deep copy), so a and b own separate ints. Changing *b.d leaves *a.d as 1, giving 19, and each destructor deletes its own int with no double free.
973. **A** — Passing a shared_ptr by value copies it, adding an owner for the duration of the call, so show prints 2. The copy is destroyed when show returns. Passing by const reference creates no new owner, so peek and main both print 1.
974. **B** — del a[0] removes an element from the list object itself, not the name a. b refers to the same list, so it sees [20, 30]. Compare del a, which would remove only the name.
975. **C** — new T[3] constructs three objects, printing ccc. delete[] calls the destructor for each of the three elements (in reverse order) and then frees the block, printing ddd.
976. **D** — * 3 repeats the reference to one inner list, so all three rows are the same object. Changing grid[0][0] shows up in every row. [[0] * 2 for _ in range(3)] builds three separate lists.
977. **B** — Each allocation is 1000 × 8 = 8,000 bytes, and none is freed, so 500 × 8,000 = 4,000,000 bytes leak. Overwriting p each pass loses the previous address, so the earlier blocks can never be deleted.
978. **B** — delete frees the block but leaves the old address in p. Resetting it means deleting a null pointer does nothing, and code can test if (p) before use. Smart pointers do this automatically.
979. **D** — A bound method keeps a reference to its instance. Bus.listeners lives as long as the class, so every Screen subscribed to it is still reachable and its 10 MB buffer cannot be freed. Unsubscribing, or storing weakref.WeakMethod objects, fixes the leak.
980. **C** — HashMap files an entry under the hash computed at put time (1). After k.v = 2, remove(k) hashes to 2 and never finds the entry, which the map still references strongly. Keys used in hash collections should be immutable.
981. **A** — The local name keeps the list alive until the function returns. Removing the last reference frees it immediately under reference counting, before the loop begins. gc.collect only handles cycles, and copying or making it global keeps it alive just as long.
982. **B** — Recursion depth equals list length, and every frame uses stack space. Java does not perform tail-call elimination, so a million calls overflow the thread stack. An iterative loop uses constant stack; raising -Xss is only a workaround.
983. **B** — Small, short-lived values belong on the stack, which is fast and freed automatically. A 200 MB block of run-time size would overflow a typical stack of a few megabytes, so it goes on the heap; std::vector frees it automatically through RAII.
984. **B** — Returning unique_ptr transfers sole ownership by move, and the Widget is deleted automatically when the caller's pointer goes out of scope. The caller can still convert it to a shared_ptr if shared ownership is needed later.
985. **C** — Each Task instance stores Activity.this, so as long as the queue references the Task, the Activity is reachable. Making Task a static nested class (and passing in only what it needs) removes the hidden reference.
986. **D** — Reassigning p loses the only address of the first int, so it can never be deleted: a leak. The second fragment leaves q dangling, the third returns the address of a dead local, and the fourth is a double free.
987. **C** — When a method returns, its frame and local references disappear, making the objects unreachable anyway. Explicit nulling helps only when a reference would otherwise outlive its use, such as a static cache or a long loop that no longer needs a large object.
988. **D** — Reference counting frees an object exactly when its count reaches zero, which is why CPython and shared_ptr give deterministic cleanup. The price is count updates on every copy and drop, plus an extra mechanism (a cycle detector or weak references) for cycles.
989. **B** — 40 bytes rounds up to 64 (two 32-byte units), wasting 24 bytes inside each block. Over 1,000 blocks that is 24,000 bytes of internal fragmentation.
990. **D** — When a vector grows past its capacity it allocates a bigger array, moves the elements and frees the old one. Pointers, references and iterators into the old array are then invalid, and dereferencing p is undefined behaviour. Reserve capacity first or take the address after the push.
991. **A** — The data simply does not fit in a 2 GB heap, and a single Java array is limited to about 2^31 elements anyway. Streaming with a BufferedReader or a fixed buffer keeps memory use constant regardless of file size.
992. **C** — del removes a name, not an object. The pool list holds another reference, so the count stays above zero and __del__ cannot run yet. It runs once pool releases the object, for example after pool.clear() or at shutdown.

993. **D** — A finally block runs after the try (and any matching catch) on every normal or exceptional exit, which makes it the place to release resources. catch runs only when a matching exception occurs; throws is part of a method signature; Java has no else clause on try (Python does).

994. **C** — Python uses raise, for example `raise ValueError("bad age")`. throw is the keyword in Java, C++ and JavaScript; throws appears in Java method signatures; except is Python's handler clause, not a way to raise.

995. **B** — IOException extends Exception but not RuntimeException, so it is checked: the compiler requires you to catch it or declare it with throws. The other three extend RuntimeException, so they are unchecked.

996. **A** — `throw new IllegalStateException("closed");` is a statement that raises one exception. `void read() throws IOException` is a declaration telling callers what checked exceptions may escape. Either kind of exception can be thrown with throw.

997. **D** — else runs when the try block completes with no exception, before finally. It is the place for code that should run only on success but whose own errors you do not want the except clauses to catch. finally runs in every case.

998. **A** — The ellipsis handler catch (...) matches every exception type, including thrown ints or strings. catch (std::exception&) only matches classes derived from std::exception. catch (void) is invalid, and C++ has no universal Object base class.

999. **C** — Only Throwable and its subclasses can be thrown and caught in Java. Throwable has two direct subclasses: Exception (conditions a program may handle) and Error (serious JVM problems). RuntimeException is a subclass of Exception; BaseException is Python's root class.

1000. **C** — Checked exceptions (Exception subclasses other than RuntimeException) are subject to the catch-or-declare rule at compile time. Unchecked exceptions (RuntimeException and Error subclasses) are exempt. The check is static, by the compiler, not at run time.

1001. **D** — Catching a derived exception by value copies only the base-class part (object slicing), so virtual calls such as what() would use the base version. A reference binds to the original object with its full type. Catching by value is legal, just worse; only catch (...) catches every type.

1002. **A** — try (Res r = new Res()) { ... } calls r.close() automatically when the block exits, so the resource type must implement java.lang.AutoCloseable. java.io.Closeable is a sub-interface used by I/O streams. The other interfaces have nothing to do with cleanup.

1003. **B** — The Python documentation recommends deriving user-defined exceptions from Exception (or a subclass of it). BaseException also covers SystemExit and KeyboardInterrupt, which `except Exception` deliberately does not catch. Python has no built-in class named Error.

1004. **D** — This is stack unwinding: each frame without a matching handler is exited (C++ runs destructors of its local objects, Java and Python run finally blocks) and the search continues in the caller. Only if no frame handles it does the program or thread end with an uncaught-exception report.

1005. **B** — Error and its subclasses (OutOfMemoryError, StackOverflowError) signal conditions a typical application cannot recover from. They are unchecked, are siblings of Exception under Throwable, and are usually thrown by the JVM itself.

1006. **A** — If the search for a handler reaches the end without a match, the runtime calls std::terminate(), which by default calls std::abort(). Whether the stack is unwound before that is implementation-defined. C++ has no compile-time catch-or-declare rule.

1007. **D** — noexcept is a promise checked at run time: if an exception tries to leave the function, std::terminate() is called. The compiler does not have to reject such code (it may warn). Converting to bad_exception relates to old dynamic exception specifications, removed in C++17.

1008. **A** — The alternatives in a multi-catch must not be subclasses of each other. FileNotFoundException extends IOException, so listing both is redundant and rejected; catching IOException alone is enough. The other pairs are unrelated by inheritance.

1009. **B** — "A" prints, then 1 / 0 raises ZeroDivisionError, so "B" is skipped. The matching except prints "C", and finally always runs, printing "D".

1010. **D** — int("abc") receives a string, which is the right type, but its value cannot be parsed as an integer, so Python raises ValueError. TypeError would mean the wrong type, such as int([1]).

1011. **D** — JSON.parse throws a SyntaxError when its input is not valid JSON. The catch block receives that error object, and e.name holds the constructor's name.

1012. **A** — The try block raises nothing, so except is skipped and else prints "L". finally then prints "F", and only afterwards does the function return r, which the caller prints: 10.
1013. **D** — The try block starts to return 1, but finally runs before the method exits, and its own return 2 replaces the pending return value. This compiles (lint tools warn that the finally block cannot complete normally). Returning from finally is bad practice because it also discards exceptions.
1014. **B** — return x evaluates x (10) and saves that value before finally runs. finally then changes the local variable x to 20, but the saved return value is not affected, so 10 is printed. Contrast this with returning from inside finally, which would replace the value.
1015. **A** — [1, 2][5] raises IndexError, which is a subclass of LookupError. Python tests except clauses from top to bottom and runs the first one that matches, so the LookupError handler wins and the IndexError handler is never reached.
1016. **B** — The throw jumps to catch, which logs "catch x" and starts to return "c". Before the function actually returns, finally runs and logs "finally". Then the caller logs the returned value "c".
1017. **C** — inner() raises KeyError. outer's except ValueError does not match, so the exception keeps propagating, but outer's finally runs on the way out and prints "F". The caller's except KeyError then catches it and prints "K".
1018. **C** — withdraw raises the custom LowBalance exception with a message. type(e).__name__ is the class name, and printing e gives the message passed to the constructor, so the output is "LowBalance need 30 more".
1019. **B** — The FileReader(String) constructor declares throws FileNotFoundException, a checked exception (subclass of IOException). Because read() neither catches it nor declares it, the compiler rejects the method. Closing the reader is good practice but is not enforced by the compiler.
1020. **A** — Handlers are tried in order, but catch matching does not apply arithmetic conversions: a thrown int does not match catch (double). The int handler matches exactly and prints "int 5"; catch (...) is never reached.
1021. **A** — finally runs every time control leaves the try block, including through continue. All three iterations increment count (the i == 1 iteration leaves early through continue, but finally still runs), so count is 3.
1022. **C** — int("x") raises ValueError, and `raise RuntimeError(...) from e` chains the new exception to it by setting __cause__. The outer handler catches the RuntimeError, and its __cause__ is the original ValueError object.
1023. **B** — sys.exit(1) raises SystemExit, which derives from BaseException, not Exception. except Exception therefore does not catch it, the exception propagates to the top level, and the interpreter exits with status 1 without running either print.
1024. **A** — An overriding method may declare the same checked exceptions, narrower ones (such as FileNotFoundException), or none, but not broader ones. Callers holding a Base reference only prepare for IOException, so Child cannot add Exception. Narrower is allowed, so the clauses need not be identical.
1025. **B** — Catching every Exception and returning a normal value swallows errors: bad user input (ValueError) and a programming mistake (TypeError from None) become indistinguishable from a real age of 0. Catch the specific exception you expect (ValueError), and report or re-raise it rather than inventing a value.
1026. **C** — During stack unwinding, local objects of f are destroyed in reverse order of construction, so b's destructor runs before a's. Both run before control reaches the handler in main, which then prints "caught". This is why RAII cleanup is exception-safe.
1027. **C** — try-with-resources closes resources automatically, in the reverse order they were opened, as soon as the try block exits, and before any catch or finally of the same statement runs. So B closes before A, and only then does the catch block run.
1028. **D** — The ValueError is pending when finally raises TypeError, so the TypeError replaces it as the exception that propagates. Python keeps the replaced exception in the new one's __context__ attribute, so the output is the class names TypeError and ValueError. The original error would be lost in Java, which has no implicit context.
1029. **C** — Joshua Bloch's Effective Java advice: use checked exceptions for recoverable conditions the caller should handle (a missing file), and runtime exceptions for programming errors such as precondition violations. A negative quantity from buggy code is a programming error, and IllegalArgumentException is the standard choice.
1030. **C** — Checking first ("look before you leap") leaves a window between the check and the use: a time-of-check to time-of-use race. Python's "easier to ask forgiveness than permission" (EAFP) style attempts the operation and handles the exception, which removes the window.
1031. **B** — The first throw is caught, but catch throws a new Error("B"). Before that exception leaves run(), finally logs "F". The outer catch then receives the second error and logs "B"; the original "A" error is discarded.

1032. **D** — The except clause is meant only for errors from load(). Code inside try is also covered by that handler, so a failure in use() would be reported as "could not load". Moving it to else keeps the try block minimal; use() still runs only after a successful load, and its errors propagate normally.
1033. **A** — Every built-in exception, including SystemExit and KeyboardInterrupt, derives from BaseException. Exception is one level below it and is the parent of ordinary application errors. Throwable is Java's root, and Python has no class named Error.
1034. **B** — KeyboardInterrupt (like SystemExit) is a direct subclass of BaseException so that broad `except Exception` handlers do not stop the user from interrupting a program. It can still be caught with `except KeyboardInterrupt` or `except BaseException`.
1035. **C** — Subclasses of RuntimeException are unchecked, so the compiler does not enforce catch-or-declare. Extending Exception (or IOException, or Throwable directly) makes the new class a checked exception.
1036. **D** — Java checks catch clauses top to bottom. Since IOException is a subclass of Exception, the second handler can never be reached, and javac reports that the exception has already been caught. Subclass handlers must come first.
1037. **A** — `from None` sets __suppress_context__, so Python does not print the "During handling of the above exception" chain. It is used when the original exception is an internal detail the caller does not need to see.
1038. **B** — Inside a handler, a bare raise re-raises the active exception unchanged. It is the usual way to log or clean up and then let the error continue to the caller.
1039. **C** — std::exception declares `virtual const char* what() const noexcept`, and derived classes such as std::runtime_error return the message passed to their constructor. getMessage() is the Java equivalent.
1040. **D** — Stack unwinding destroys every fully constructed automatic object between the throw point and the handler, in reverse order of construction. This is what makes RAII (resource release in destructors) safe in the presence of exceptions.
1041. **A** — JavaScript lets you throw any value; `throw "oops"` and `throw 42` are legal. Throwing Error objects is recommended because they carry a message, a name and a stack trace.
1042. **B** — printStackTrace() writes the exception's class name, message and stack frames (plus any cause) to System.err. It neither rethrows nor terminates the program; getStackTrace() is the method that returns the frames as data.
1043. **C** — A failing assert raises AssertionError with the given message. Because `python -O` removes assert statements entirely, asserts should check internal assumptions, not validate user input.
1044. **D** — A with statement calls the context manager's __exit__ however the block ends. For files, __exit__ closes the file and returns a false value, so the exception still propagates after cleanup.
1045. **A** — Python compiles the entire file before executing it, so a syntax error stops the program before any try statement exists. The other errors happen during execution and can be caught normally.
1046. **B** — `throw;` re-throws the current exception object itself, so a derived type caught as a base reference is not sliced. Writing `throw e;` would instead copy e using its static type.
1047. **C** — JavaScript uses the throw keyword with an expression, usually a new Error object. raise is Python's keyword, throws is Java's method-signature keyword, and Error has no throw method.
1048. **D** — ZeroDivisionError, OverflowError and FloatingPointError all derive from ArithmeticError, which derives from Exception. So `except ArithmeticError` also catches division by zero.
1049. **A** — StackOverflowError extends VirtualMachineError, which extends Error. Errors are unchecked and represent conditions such as running out of stack or heap that a normal application is not expected to recover from.
1050. **B** — An unhandled exception propagates until something handles it or the program stops, so an error cannot be lost the way an unchecked return code can. The handler can also sit far from the failure, keeping the main logic clean.
1051. **C** — Ordinary new reports allocation failure by throwing std::bad_alloc. Only the `new (std::nothrow)` form returns nullptr instead.
1052. **D** — A Java try must be followed by at least one catch, a finally, or both (try-with-resources may omit them). try-finally is legal and is used for cleanup while letting the exception propagate.
1053. **A** — ExceptionGroup bundles several unrelated exceptions, for example from concurrent tasks. An except* clause handles the members that match its type and leaves the others in a group that continues to propagate.

1054. **B** — Throwable has constructors that take a cause. Chaining keeps the original stack trace (printed as "Caused by:") while the caller sees an exception that fits its own abstraction layer.
1055. **C** — Since C++11, destructors are implicitly noexcept(true) unless a member or base says otherwise. An exception escaping a noexcept function calls std::terminate, which is why destructors should never let exceptions out.
1056. **D** — try-with-resources keeps the primary exception from the body and records exceptions from close() as suppressed ones. This fixes the old try/finally pattern, where an exception in close() replaced the original.
1057. **A** — int(None) raises TypeError (None is not a string or number). The handler lists a tuple of types, so it matches either ValueError or TypeError and prints "bad input"; else is skipped.
1058. **B** — Every exception stores its constructor arguments in the args tuple. With two arguments, args is ('x', 2), and printing a tuple shows its repr.
1059. **C** — On the first pass, return 0 starts leaving the function, but finally runs first and prints "fin 0". The function then returns 0, so the loop never reaches i = 1, and the caller prints 0.
1060. **D** — The handlers of one try statement only guard the try block. The IndexError raised inside the ValueError handler is not caught by the sibling except IndexError clause, so it propagates and ends the program after "V".
1061. **A** — DbError is a subclass of AppError, so `except AppError` catches it. str(e) is the message passed to the constructor, so the output is "app: db down".
1062. **B** — Unlike most exceptions, KeyError's str() returns the repr of the missing key, so the output includes the quotes: 'k'. The "KeyError:" prefix appears only in a traceback.
1063. **C** — Looking up the missing key "b" raises KeyError, so value is set to 0 in the handler. Execution continues after the try statement and prints 0 + 5 = 5.
1064. **D** — __exit__ receives the exception's type and prints "exit ValueError". Because it returns True, the with statement suppresses the exception, so execution continues and prints "after".
1065. **A** — Reading a property of null throws a TypeError in JavaScript. The catch block receives that object, so `e instanceof TypeError` is true.
1066. **B** — JavaScript can throw any value. The thrown value is the string "oops" itself, not an Error object, so typeof e is "string".
1067. **C** — An exception thrown inside an async function rejects the returned promise instead of propagating synchronously. The .catch callback runs later as a microtask, so "after" is logged first, then "caught a".
1068. **D** — Writing a[3] on a two-element array throws ArrayIndexOutOfBoundsException, so "S" is skipped. Only the first matching handler runs ("A"), then finally prints "F".
1069. **A** — parseInt("12a") throws NumberFormatException, which is a subclass of IllegalArgumentException. The first handler therefore matches and prints "I"; the RuntimeException handler is not reached.
1070. **B** — Calling a method on a null reference throws NullPointerException. It is a RuntimeException, so `catch (Exception e)` catches it, and getSimpleName() returns the class name without its package.
1071. **C** — vector::at() checks the index and throws std::out_of_range when it is invalid, so "set" is skipped and the handler prints "range". It is operator[] (v[5]) that has no check and gives undefined behaviour.
1072. **D** — C++ tries handlers in order and takes the first match. runtime_error derives from std::exception, so the first handler catches it and prints "base". Unlike Java, a later unreachable handler is legal in C++ (compilers may only warn).
1073. **A** — The inner handler prints "inner " and `throw;` re-throws the same int object. It propagates out of f() to main's handler, which prints "outer " and the value 1.
1074. **B** — While the exception leaves g(), stack unwinding destroys its locals in reverse order of construction (b, then a). Only after unwinding reaches main's try block does the handler run and print "caught".
1075. **C** — b() raises; a() has no except clause, but its finally block runs as the exception passes through, printing "a-fin". The exception then reaches the caller's handler, which prints "main".
1076. **D** — In Python 3, the target of `except ... as e` is deleted when the handler ends (to avoid reference cycles with the traceback). Because the same name was used, the earlier value is gone too, and print(e) raises NameError.
1077. **A** — "1" and "3" convert and are added in the else block. "x" raises ValueError, the handler continues to the next item, and else is skipped for it. The total is 1 + 3 = 4.

1078. **B** — Python tests except clauses from top to bottom and runs only the first match. ValueError is an Exception, so "general" prints and the specific handler is dead code. Specific handlers must come first.
1079. **C** — A bare `except:` is equivalent to `except BaseException:`, so it also swallows KeyboardInterrupt. Catching Exception (and logging it) would let Ctrl+C and sys.exit() work while still handling ordinary job failures.
1080. **D** — When finally executes a return, that return replaces whatever was leaving the try block, including an active exception, so the ValueError silently disappears. This is why Python 3.14 emits a SyntaxWarning for return in finally.
1081. **A** — An exception thrown from finally replaces the one already propagating, so the more useful readAll() error disappears. try-with-resources fixes this by keeping the body's exception and attaching the close() failure to it as suppressed.
1082. **B** — Python's convention is TypeError for an argument of the wrong type and ValueError for a correctly typed argument with an inappropriate value, as math.sqrt(-4) itself does.
1083. **C** — An exception jumps past f.close(), leaving the file open. `with open("data.txt") as f:` (or try/finally) guarantees the close however the block ends.
1084. **D** — throw 1 starts unwinding, which runs ~D(). That destructor throws 2 while exception 1 is still in flight, and C++ cannot handle two active exceptions, so std::terminate is called. noexcept(false) only allows the throw; it does not make this situation legal.
1085. **A** — Calls 1 and 2 raise, and continue moves to the next iteration. Call 3 returns "data"; with no exception, the break after the try statement ends the loop, so attempts is 3.
1086. **B** — setTimeout only schedules the callback; the try block finishes before it runs. When the callback throws later, no try/catch is on the stack, so the error is uncaught. The try/catch must go inside the callback.
1087. **C** — After the handler runs, save() returns normally and the caller prints "Order saved" even though nothing was saved. Re-raising after cleanup keeps the rollback and still tells the caller that the operation failed.
1088. **D** — Throwable is the root of both Exception and Error. Errors signal problems such as memory exhaustion that the application generally cannot repair; continuing as if nothing happened can corrupt data. Catch Exception (or specific types) in the loop instead.
1089. **A** — The C++ standard requires catch(...) to be the last handler, and compilers reject code that puts another handler after it. Order specific handlers first and the catch-all last.
1090. **B** — The expression after except is evaluated first: `ValueError or TypeError` returns ValueError, the first truthy operand. int(None) raises TypeError, which does not match, so it propagates. The correct form is a tuple: `except (ValueError, TypeError):`.
1091. **C** — The catch block prints and throws a new exception. finally still runs before that exception leaves the try statement, printing "finally ". No outer handler exists, so main ends with the uncaught RuntimeException "y".
1092. **D** — Handle an exception at the level that can actually do something about it. The low-level function should report the failure clearly; main() decides how to inform the user or exit.
1093. **B** — A module is one Python source file (or an extension module); its name is the file name without .py. A package is a directory of modules. setup.py relates to distributing a package, not to what a module is.
1094. **A** — Java's import statement brings a class name from another package into scope so you can use its simple name. #include is the C/C++ preprocessor, using is C++ and C#, and package declares which package the current file belongs to.
1095. **B** — #include is a preprocessor directive: it textually inserts the named header into the translation unit before the compiler proper runs. Linking is a separate, later step, and C++ has no package download built into the language.
1096. **C** — pip is Python's standard package installer, for example `pip install requests`. npm manages JavaScript packages, Maven manages Java dependencies and builds, and javac is the Java compiler.
1097. **D** — Classes in java.lang, such as String, Math, Integer and System, can be used without an import. java.util (ArrayList, HashMap) and java.io must be imported explicitly. java.base is a module name in the Java module system, not a package.
1098. **A** — A namespace such as std or a library's own namespace qualifies names (std::sort versus mylib::sort), so two libraries can use the same identifier without a clash. Access control is done with private/public, not namespaces.

1099. **B** — When a file is run as a script, Python sets its __name__ to "__main__"; when it is imported, __name__ is the module name. The guard lets a file serve both as a reusable module and as a script. Duplicate imports are prevented by the sys.modules cache, not by this check.

1100. **C** — A regular package is a directory containing __init__.py, which runs on first import and can define the package's public names. Since Python 3.3 a directory without it can still be imported as a namespace package. __main__.py runs with `python -m package`; setup.py and requirements.txt are packaging files.

1101. **C** — Linking a static library (.lib/.a) embeds the used object code, making a larger, self-contained executable. A dynamic library (.dll on Windows, .so on Linux) stays a separate file loaded when the program starts or on demand, so several programs share one copy and it can be updated independently.

1102. **D** — Maven reads pom.xml (Project Object Model) to download dependencies from repositories such as Maven Central and to run build phases like compile, test and package. Gradle plays the same role with build.gradle. Python's equivalents are requirements.txt or pyproject.toml.

1103. **B** — A virtual environment is a directory with its own interpreter link and site-packages, so one project can use library version 1.x while another uses 2.x without conflict. It does not change speed or compile code.

1104. **C** — SemVer 2.0.0: increment MAJOR for incompatible API changes, MINOR for backward-compatible new features, and PATCH for backward-compatible bug fixes. Package managers such as npm rely on this when resolving version ranges.

1105. **A** — With no modifier, Java uses default (package-private) access: any class in the same package can use the member, but classes in other packages cannot, even subclasses. Unlike protected, it gives no special access to subclasses outside the package.

1106. **A** — The caret allows changes that do not modify the left-most non-zero part, so for 1.x.x it accepts new MINOR and PATCH releases: >=1.4.2 <2.0.0. The tilde (~1.4.2) is narrower and allows only PATCH updates: >=1.4.2 <1.5.0.

1107. **D** — If a.h and b.h both include common.h, a file that includes both would otherwise see common.h's class definitions twice and fail with a redefinition error. Guards work only within one translation unit; they do not stop separate .cpp files including the header, and so they cannot fix multiple-definition link errors.

1108. **C** — When a module defines __all__, `from module import *` imports exactly the names listed. It does not make other names private: `import module` followed by module.name, or an explicit `from module import name`, still works for any name.

1109. **D** — `import math as m` binds the module to the alias m, so m.sqrt works. math.sqrt always returns a float, so the result is 4.0, not 4.

1110. **B** — `from math import pi, floor` binds the two names directly, so they can be used without the math. prefix. floor(3.14159...) rounds down to the int 3.

1111. **C** — The first import runs counter.py and stores the module in sys.modules; later imports of the same name reuse the cached module without running the file again. So "loading counter" prints once, and count is incremented once to 1.

1112. **D** — A module's __name__ is set to "__main__" only for the file being run as the script. tools.py is imported, so its __name__ is its module name, "tools".

1113. **C** — `from math import *` binds every public name of math in the current namespace, overwriting the sqrt defined above it. The call therefore uses math.sqrt and prints 4.0. Silent overwriting like this is why star imports are discouraged.

1114. **B** — Because shapes.py defines __all__ = ["area"], the star import brings in area but not perimeter. area(2) prints 12; the call perimeter(2) then raises NameError, since that name was never bound in main.

1115. **A** — Each v lives in a different scope: A::v, the nested A::B::v and the global ::v. Qualified names pick each one explicitly, so the program prints 1, 2 and 3 with no separators: 123.

1116. **A** — main.py imports shop.cart as part of the shop package, so the relative import `from .pricing import tax` resolves to shop.pricing. tax(200) is 200 // 10 = 20, and total returns 220.

1117. **A** — require caches a module after its first load, so a and b are the same exports object sharing one closure variable n. Two calls through a make n equal 2, and b.inc() raises it to 3.

1118. **B** — The -r flag tells pip to read requirement specifiers from a file. Without -r, pip treats requirements.txt as the name of a package to download. `pip freeze > requirements.txt` is the command that creates such a file from the current environment.

1119. **B** — The tilde range ~2.3.1 means >=2.3.1 and <2.4.0, so only PATCH updates within 2.3 are allowed. 2.3.9 qualifies; 2.4.0 is a new MINOR version, 3.0.0 a new MAJOR, and 2.3.0 is below the lower bound.
1120. **D** — `from config import level` copies the current binding (1) into main's own namespace. bump() rebinds level inside the config module to 2, but main's separate name still refers to 1. config.level reads the module attribute, so it shows 2.
1121. **D** — Two on-demand (wildcard) imports both supply a class named Date, so the simple name is ambiguous and javac rejects its use. Fix it with a single-type import such as `import java.util.Date;` (which takes precedence over wildcards) or the fully qualified name java.util.Date.
1122. **D** — A static import brings static members of a class into scope. java.lang being automatic only lets you write Math without an import; you still need Math.sqrt. `import java.lang.Math.*;` would try to import nested types of Math, not its static methods.
1123. **A** — Python puts the script's directory first on sys.path, so `import random` loads the student's file (as a new module named random) rather than the standard library. That module has no randint yet when the call runs, so Python raises AttributeError. Renaming the file fixes it; the import cache stops the self-import from repeating forever.
1124. **D** — main imports a; a's first line imports b; b runs `from a import x`, but a is still at its first line and has not yet defined x. Python raises ImportError ("cannot import name 'x' from partially initialized module 'a'"). Fixes: move shared values into a third module, or import inside the function that needs them.
1125. **D** — Run as a script, cart.py becomes __main__ with no parent package, so `from .pricing` cannot be resolved and Python raises ImportError ("attempted relative import with no known parent package"). Running `python -m shop.cart` from the project folder keeps the package context.
1126. **C** — Item is public, so the import is fine, but its field price has no modifier and is visible only inside com.shop. Making it public, or better adding a public getter, fixes the error. A wildcard import changes nothing about access.
1127. **C** — Include guards work per translation unit, so a.cpp and b.cpp each compile their own definition of sq. The One Definition Rule allows a non-inline function only one definition in the whole program, so the linker reports a multiple-definition error. Marking it inline, or keeping only the declaration in the header, fixes it.
1128. **C** — One environment can hold only one installed version of a package, so the global installation forces a conflict. Separate virtual environments isolate each project's site-packages, and pinned requirements (requests==2.31.0) make the set reproducible.
1129. **A** — With dynamic linking the library stays a separate file loaded at run time, so replacing it with a patched, ABI-compatible version fixes every application at once. Static linking or source copying bakes the old code into each binary, so each must be rebuilt; header-only code is also compiled into each program.
1130. **A** — `from settings import items` binds main's name items to the same list object, so append(1) changes that list. `settings.items = [9]` then rebinds the module attribute to a new list, which does not affect main's name, still pointing at [1].
1131. **B** — A using-directive in a header affects every translation unit that includes it, and the includer cannot undo it. A user function called count or distance can then clash with std::count or std::distance. Inside a .cpp file, or with explicit std:: prefixes, the effect stays local.
1132. **B** — A caret range accepts any later 4.x release, so installs at different times can resolve differently. The lockfile records the exact resolved version of every package, including transitive ones, and `npm ci` installs exactly those. Changing to ~ only narrows the range; it does not pin it.
1133. **A** — Python searches the module path held in sys.path in order and uses the first match. Its first entry is the running script's directory, which is why a local file can shadow a standard-library module. import never downloads anything.
1134. **B** — A repeated import returns the cached module from sys.modules. importlib.reload() re-runs the module's source in the existing module object. del only removes the local name; the module stays cached, so the next import still does not re-run it.
1135. **C** — json ships with every Python installation. requests, numpy and flask are third-party packages that must be installed from PyPI, usually with pip.
1136. **D** — pip freeze prints installed packages in name==version form. Redirecting it to requirements.txt records the environment so others can recreate it with `pip install -r requirements.txt`.

1137. **A** — Java package names use the organisation's internet domain reversed, in lower case, followed by the project or module name. Reversing the domain keeps package names globally unique.
1138. **B** — The directory structure mirrors the package name: each dot becomes a folder, and the file is named after the public class. javac and the class loader use this layout to find classes.
1139. **C** — A Java ARchive is a ZIP file of .class files, resources and a META-INF/MANIFEST.MF. Libraries are distributed as JARs and added to the classpath, and a JAR with a Main-Class entry can be run with java -jar.
1140. **D** — The classpath (set with -cp or CLASSPATH) tells the compiler and the class loader where to search for user classes and library JARs. A ClassNotFoundException or NoClassDefFoundError often means something is missing from it.
1141. **A** — A module descriptor states dependencies with `requires` and its public API with `exports`. Packages that are not exported stay hidden even if their classes are public, giving stronger encapsulation than packages alone.
1142. **B** — import is purely a compile-time naming convenience. java.util.List and List refer to the same class; classes are loaded lazily at run time. You can always use the fully qualified name instead of importing.
1143. **C** — Each .cpp file compiles on its own using only declarations. Matching calls to definitions happens at link time, so a missing definition (or an object file or library left out of the link) is an "undefined reference" linker error.
1144. **D** — Headers declare the interface that other files #include; the .cpp file implements it and is compiled once. Putting non-inline definitions in a header risks multiple-definition errors when several files include it.
1145. **A** — By convention and in all major compilers, quotes are used for your own headers and start the search next to the current file, falling back to the system paths. Angle brackets are for standard and library headers on the include path. The exact search is implementation-defined.
1146. **B** — Linux shared objects end in .so (e.g. libm.so). .dll is the Windows equivalent, .a is a static library archive on Unix-like systems, and .jar is a Java archive.
1147. **C** — An unnamed namespace gives its members internal linkage, the modern replacement for file-level static. Two .cpp files can each have their own helper without a multiple-definition link error.
1148. **D** — A using-declaration introduces one specific name. A using-directive makes all names from the namespace visible, which increases the chance of name clashes. Neither has anything to do with linking.
1149. **A** — std::sort, std::find, std::max_element and other generic algorithms are declared in <algorithm>. <vector> provides the container only, and there is no standard header called <sort>.
1150. **B** — A default export is imported without braces, under any name you choose; named exports go inside braces with their exported names. area is the default, so it cannot be requested as a named import.
1151. **C** — With no arguments, npm install reads package.json and installs everything it lists, respecting the lockfile when one exists. npm init creates package.json and npm publish uploads a package.
1152. **D** — Local installs go into the project's node_modules directory. package.json lists the dependencies and package-lock.json records exact versions; neither stores the code itself.
1153. **A** — With a library you stay in control and call its functions when you need them. A framework provides the overall structure and calls the code you plug into it, for example request handlers in a web framework.
1154. **B** — Since Python 3.3, a directory without __init__.py can still be imported as a namespace package, whose parts may even be spread over several directories. A regular package with __init__.py is still preferred when the package needs initialisation code.
1155. **C** — On import, CPython compiles source to bytecode and saves it as a .pyc in __pycache__, tagged by interpreter version. If the source has not changed, the next import loads the cached bytecode directly.
1156. **D** — A wheel is a ZIP-format built package, possibly with compiled extensions for a specific platform, so installation is mostly a copy. A source distribution (sdist) may need building on the target machine, which is slower and can require a compiler.
1157. **A** — `from math import sqrt` binds only the name sqrt in the current namespace; it does not bind math. Either call sqrt(4) directly or write `import math`.
1158. **B** — The random module's generator is deterministic: the same seed produces the same sequence. Reseeding with 1 restarts the sequence, so both calls return the same number.
1159. **C** — greet.name = "Sara" changes the variable inside the module object itself. hello() looks up name in its module's globals when it runs, so it sees the new value.

1160. **D** — An imported module's __name__ is its fully qualified name within the package, here shop.pricing. Only the file started directly by the interpreter is named __main__.
1161. **A** — Importing the package runs shop/__init__.py, which imports tax from the pricing submodule into the package namespace. So shop.tax is available, and 50 // 10 is the int 5.
1162. **B** — Importing a package runs only its __init__.py; submodules are not loaded automatically. Since __init__.py is empty, shop has no pricing attribute. `import shop.pricing` (or an import in __init__.py) would fix it.
1163. **C** — json.dumps converts a Python dict into a JSON string. JSON requires double quotes, and by default the separators are ", " and ": ", giving {"a": 1}.
1164. **D** — importlib.import_module imports a module whose name is given as a string at run time and returns the module object. m is the math module, and factorial(4) is 24.
1165. **A** — An alias only changes the local name bound to the module object. The module's own __name__ attribute is still math.
1166. **B** — The first require runs log.js and caches its exports; the second require returns the cached value without running the file again. So "load" appears once, and a + b is 1 + 1 = 2.
1167. **C** — exports is just a local variable that initially refers to module.exports. Reassigning it points the variable at a new object, but require returns module.exports, which is still the original empty object, so .a is undefined. Use module.exports = {...} instead.
1168. **D** — module.exports can be any value, including a single function. require returns that function, so sq(4) returns 16.
1169. **A** — The using-declaration brings only b::f into the global scope, so the unqualified call f() means b::f and returns 2. a::f() is qualified and returns 1, giving 3. There is no ambiguity because a::f was never made visible unqualified.
1170. **B** — cout is declared inside namespace std. Without std::cout, a using-declaration or a using-directive, the unqualified name is not found and compilation fails. The header was included, so linking is not the issue.
1171. **C** — Namespaces are open: a second `namespace N { ... }` block adds members to the same namespace instead of redefining it. Both a and b belong to N, so the sum is 3.
1172. **D** — A fully qualified name such as java.util.List can be used anywhere without an import; import only provides the short name. List.of(1, 2) creates a two-element list, so size() is 2.
1173. **A** — A wildcard import brings in the types of exactly one package. java.util.function is a separate package, so Function needs `import java.util.function.Function;` (or java.util.function.*).
1174. **B** — pip show prints metadata for an installed package, including Version, Location and Requires. pip has no info, version or get subcommand.
1175. **C** — --save-dev (or -D) records the package under devDependencies, meaning it is needed for development and testing but not in production. A plain install adds it to dependencies, and -g installs globally without touching package.json.
1176. **D** — A backward-compatible new feature raises MINOR and resets PATCH to 0, giving 3.5.0. MAJOR changes only for breaking changes, and PATCH alone is for backward-compatible bug fixes.
1177. **A** — -l<name> links the library lib<name> (libm.so or libm.a), so -lm links the maths library. -I adds a header search directory, -L adds a library search directory, and -o sets the output file name.
1178. **B** — With `import a`, b.py only needs the (partly initialised) module object, which already exists in sys.modules; main_value is looked up later, when helper() runs. The from-import fails because it asks for main_value before a.py has defined it.
1179. **C** — Without __all__, `from mod import *` skips names that start with an underscore, so _helper is not bound in main. calc still works, because it looks up _helper in mod's own globals, where it exists.
1180. **D** — Importing a runs it top to bottom; its import b runs b.py there, between "a start" and "a end". By the time main.py imports b, the module is already cached in sys.modules, so it does not print again.
1181. **A** — Importing a module executes all of its top-level code, whatever import form is used. Guarding the script-only work with the __main__ check runs it only when stats.py is started directly, so importing it becomes cheap.
1182. **B** — Static linking copies the needed library code into the binary, so it runs without the library on the target. Dynamic linking would fail to start when the shared library is missing. Headers are only needed at compile time.

1183. **C** — Include guards stop repeated inclusion within one translation unit only. Each .cpp gets its own definition of counter, violating the one-definition rule at link time. An extern declaration in the header plus a single definition fixes it (C++17 inline variables are another option).

1184. **D** — The header gives the compiler a declaration, so main.cpp compiles. The definition lives in util.cpp, which was never compiled or passed to the linker, so the reference cannot be resolved.

1185. **A** — The classpath is searched in order and the first class with the requested name is loaded; the other copy is silently ignored. Such duplicate classes cause hard-to-trace bugs, which is why build tools manage dependency versions.

1186. **B** — --omit=dev skips devDependencies. Because the app needs express at run time, it belongs in dependencies; on the laptop it was present only because a full install includes dev packages.

1187. **C** — When several Pythons or virtual environments exist, pip and python can point to different interpreters, so the package lands where app.py's interpreter does not look. `python -m pip` installs into exactly the interpreter you run.

1188. **D** — ^2.9.3 means >=2.9.3 and <3.0.0. Under semantic versioning a MAJOR bump signals breaking changes, so the caret allows minor and patch updates but never 3.x.

1189. **A** — ES module imports are live, read-only bindings to the exporting module's variables, so main sees count change to 2. A CommonJS destructured copy, by contrast, would still show 0. main itself may not assign to count, but the exporter can.

1190. **B** — Each from-import simply binds the name f in main's namespace. The second import rebinds it to b.f, silently hiding a.f. Using `import a, b` and calling a.f() and b.f() avoids such clashes.

1191. **C** — `using namespace std;` makes std::count (from <algorithm>) visible for unqualified lookup alongside ::count, so the compiler reports a reference to count as ambiguous. Removing the directive or writing ::count fixes it.

1192. **D** — split(",") cuts inside quoted fields, so "Lahore, PK" becomes two values. csv.reader follows CSV quoting rules, which is a typical reason to prefer a tested standard-library module over hand-written parsing.

1193. **B** — Generics give compile-time type safety: adding an Integer to a List<String> is rejected by the compiler, and get() already returns String, so no cast is needed. They do not change memory layout, add thread safety or allow multiple class inheritance.

1194. **A** — A Python lambda is lambda parameters: expression. Its body must be a single expression, so return is not allowed. The => arrow is JavaScript or C# syntax, and lambda is a reserved keyword, so it cannot be a function name.

1195. **D** — Java implements generics by type erasure: the compiler checks types and inserts casts, then removes the type arguments. At run time both lists are plain ArrayList objects, so new ArrayList<String>().getClass() == new ArrayList<Integer>().getClass() is true. Per-type code generation is how C++ templates work.

1196. **D** — A template is a blueprint. When the compiler sees maxOf(3, 7) and maxOf(2.5, 1.5), it instantiates maxOf<int> and maxOf<double> as separate functions at compile time. Nothing is generated at run time, and there is no single void-pointer version.

1197. **C** — Python's iterator protocol needs __iter__ (return the iterator, usually self) and __next__ (return the next item or raise StopIteration). __enter__/__exit__ make a context manager, and hasNext/next is Java's Iterator interface.

1198. **A** — java.util.Iterator declares hasNext(), next() and a default remove(). A for-each loop over a collection calls these behind the scenes. begin()/end() are C++ container functions and __iter__/__next__ are Python's protocol.

1199. **C** — The for loop keeps calling next() and stops when StopIteration is raised. Returning None would just produce a None item. IndexError belongs to the older __getitem__ sequence protocol, and iterators need not have a length at all.

1200. **A** — Higher-order functions treat functions as values. map, filter, sorted(key=...) and Java's Stream.map all accept a function; a decorator both accepts and returns one. Being a method, having many parameters or being recursive has nothing to do with it.

1201. **D** — When an inner function refers to a variable of its enclosing function, the variable is captured. The inner function can still read (and, with nonlocal, change) it after the outer call has finished. Counter factories and decorators rely on this.

1202. **D** — Any function containing yield is a generator function. Calling it creates a generator object without executing any of the body. Each next() call runs the body up to the next yield and pauses there, which is why generators are lazy.

1203. **B** — The bound says T must implement Comparable<T>, so the compiler knows compareTo exists and permits a.compareTo(b). In exchange, callers can only pass lists of comparable types. Erasure still forbids new T(), and generics never take primitives.

1204. **C** — An upper-bounded wildcard means 'a list of some unknown subtype of Number'. Every element is at least a Number, so reading is safe. Adding is unsafe because the list might really be a List<Double>, so only null can be added. This is the 'producer extends, consumer super' (PECS) rule.

1205. **C** — A lambda supplies the body of one method, so it can only target a functional interface: an interface with exactly one abstract method, such as Runnable, Comparator<T> or Function<T,R>. Default and static methods do not count. Interfaces never have constructors.

1206. **A** — [&] captures used variables by reference, so the lambda sees later changes and can modify them. [=] captures by value (a copy at creation time), and [] captures nothing. Capturing by reference is risky if the lambda outlives those variables.

1207. **B** — Because of type erasure, the JVM does not know what T is when the code runs, so it cannot choose a constructor. A common workaround is passing a Supplier<T> or a Class<T> object. Generic classes can have constructors, and type arguments can never be primitives.

1208. **C** — Decorator syntax is shorthand: after the def statement runs, Python passes the function object to the decorator and rebinds the name to whatever it returns. So @timer means work = timer(work). The decorator usually returns a wrapper function.

1209. **C** — map applies the lambda to each element, giving 1, 4 and 9, and list() collects them. Printing the map object without list() would show something like <map object at 0x...>, but here it is converted first.

1210. **C** — key=len sorts by string length: fig (3), pear (4), banana (6). Without the key the list would be sorted alphabetically, which gives ['banana', 'fig', 'pear'].

1211. **B** — Each next() resumes the generator from where it paused. The first call runs until yield 0*10 = 0; the second resumes the loop with i = 1 and yields 10. The third value, 20, is never requested.

1212. **A** — filter keeps the items for which the lambda returns True. range(1, 10) is 1 to 9, and the multiples of 3 among them are 3, 6 and 9. Zero is not included because the range starts at 1.

1213. **B** — Each make_counter() call creates a separate count variable captured by its own inc closure. c1 is called twice before the print and a third time inside it, giving 3. c2 has its own count and is called once, giving 1.

1214. **D** — reduce folds left to right: first 4 * 10 + 0 = 40, then 40 * 10 + 7 = 407. The lambda builds a number from its digits. Adding the digits would give 11.

1215. **B** — @shout replaces greet with wrapper. Calling greet('ali') runs wrapper, which calls the original greet to get 'hi ali', upper-cases it and appends '!'.

1216. **D** — An iterator can be consumed only once. The first sum() reads 1, 2 and 3 and leaves the iterator exhausted, so the second sum() sees no items and returns 0. A list, by contrast, can be summed any number of times.

1217. **B** — map calls the arrow function x => add(x, 1) on each element, giving 2, 3 and 4 in a new array. Node prints arrays with spaces inside the brackets. The original array is not changed.

1218. **A** — reduce starts with acc = 0 and adds each element's square: 1 + 4 + 9 + 16 = 30. Summing the plain elements gives 10, and squaring that sum gives 100.

1219. **D** — A generator expression is lazy. It holds an iterator over the list object but does not read elements until list(g) runs, by which time 4 has been appended. So all four elements are doubled. Python raises no error for appending to a list before it is iterated.

1220. **D** — The compiler instantiates maxOf<int> for (3, 7) and maxOf<double> for (2.5, 1.5). Each returns the larger argument in its own type, so cout prints 7 and then 2.5. The int version is not used for the doubles, so nothing is truncated.

1221. **C** — The comparator lambda (a, b) -> b.compareTo(a) reverses natural (alphabetical) order, so the list is sorted in descending order: Sara, Bilal, Ali. ArrayList.toString() prints elements in brackets, separated by commas.

1222. **D** — The key (-t[0], t[1]) sorts by the first value descending (because of the minus sign), then by the letter ascending to break ties. So both 2s come first, with 'a' before 'b', and (1, 'z') comes last.

1223. **B** — Closures capture variables, not values (late binding). All three lambdas refer to the same i, which is 2 when they are finally called. The usual fix is a default argument: lambda i=i: i. The comprehension's i survives inside the closures, so there is no NameError.

1224. **A** — Template argument deduction needs both arguments to agree on one T. 3 deduces int and 2.5 deduces double, so deduction fails and the call does not compile. Writing maxOf<double>(3, 2.5) fixes T explicitly, after which 3 converts to 3.0. Ordinary (non-template) functions would allow the promotion.

1225. **A** — If List<String> were a List<Object>, you could add an Integer through objs and corrupt the string list. Java prevents this by making generic types invariant. Use List<? extends Object> (or List<?>) for read-only access. Assigning an ArrayList to a List is normal, and <> is optional.

1226. **B** — sum() consumes every value (0 + 1 + 4 + 9 = 14) and leaves the generator empty. max() of an empty iterable raises ValueError. Rebuild the generator or store the values in a list if you need two passes.

1227. **A** — The for-each loop uses the list's iterator. Removing through xs.remove(x) changes the list's modification count, so the iterator's next next() call detects it and throws ConcurrentModificationException. Use xs.removeIf(x -> x % 2 == 0) or Iterator.remove(). Because x is an Integer, remove(Object) is called, not remove(int index).

1228. **B** — @log rebinds area to wrapper, so area.__name__ is 'wrapper'. This breaks help() and logging. Adding @functools.wraps(fn) above def wrapper copies the original name and docstring onto the wrapper.

1229. **C** — __iter__ returns self, so the object is its own iterator and keeps its state. The first list() counts 3, 2, 1 down to n = 0. The second list() reuses the same exhausted object and gets nothing. To allow repeated loops, __iter__ should return a fresh iterator (for example, a generator).

1230. **D** — var is function-scoped, so all three arrow functions close over one shared i, which is 3 when the loop ends. Changing var to let creates a fresh binding per iteration and prints 0 1 2.

1231. **C** — w.lower without parentheses is the method object, not its result. sorted() then tries to compare method objects with <, which raises TypeError. The fix is key=str.lower or key=lambda w: w.lower().

1232. **A** — Unused templates are only checked for basic syntax. Calling maxOf(p1, p2) instantiates maxOf<Point>, and a > b then needs an operator> for Point, which does not exist. Defining bool operator>(const Point&, const Point&) fixes it. C++20 concepts can state such requirements up front.

1233. **D** — Predicate<T> has the single abstract method boolean test(T t), so it matches a lambda that takes one value and answers true or false. It is the type that Stream.filter() expects.

1234. **A** — Consumer<T> declares void accept(T t): it takes a value and only produces a side effect. Supplier takes nothing and returns a value, Predicate returns a boolean, and UnaryOperator returns a value of the same type.

1235. **D** — A reference to an instance method of an arbitrary object, Type::method, becomes a lambda whose first parameter is the receiver. So String::length means s -> s.length(), which takes a String and returns its length.

1236. **C** — Java lambdas capture the value of local variables, so the language requires those locals to be final or effectively final. Reassigning the variable anywhere in the method makes the lambda fail to compile.

1237. **B** — A lower-bounded wildcard, ? super Integer, matches lists of Integer or any supertype, and it is always safe to add an Integer to such a list. This is the 'consumer super' half of the PECS rule (Producer Extends, Consumer Super).

1238. **B** — Type erasure removes the type arguments, so both methods become print(List) in the compiled class. The compiler reports a name clash because they would have the same erasure. Overloading on parameter types is otherwise allowed.

1239. **C** — A class template is a blueprint. Each distinct set of template arguments is instantiated at compile time into its own class, so Stack<int> and Stack<double> are unrelated types with their own generated code.

1240. **A** — template <> followed by Box<bool> is a full specialisation: when the code uses Box<bool>, the compiler uses this definition instead of the general one. std::vector<bool> is the standard library's well-known example.

1241. **A** — A non-type template parameter is a value, not a type, and it is fixed when the template is instantiated. So the argument must be a constant expression such as 16 or a constexpr value, never a run-time variable.

1242. **A** — Round brackets create a lazy generator, which hands sum() one square at a time and keeps only its current state. Square brackets build a list holding all ten million results before sum() starts.

1243. **D** — yield from delegates to another iterable: each value the sub-generator yields is passed straight to the caller. When the sub-generator finishes, the return value it gave becomes the value of the yield from expression.

1244. **D** — STL ranges are half-open: begin() points to the first element and end() points one past the last. Loops therefore run while it != v.end(), and dereferencing end() is undefined behaviour.

1245. **B** — When push_back exceeds the capacity, the vector allocates a larger buffer and moves its elements there. Every existing iterator, pointer and reference into the old buffer is invalidated. Re-acquire iterators after operations that may reallocate.

1246. **A** — lru_cache is a decorator that memoises the function. The wrapper keeps a dictionary from arguments to results, so each fib(k) is computed once. That turns the exponential recursive Fibonacci into a linear one.

1247. **D** — In Python 3, map() returns a lazy map object that is its own iterator, so next() works on it directly. Lists, strings and tuples are iterables: they have __iter__ but no __next__, and you need iter() to get an iterator from them.

1248. **C** — Arrow functions bind this lexically: they take it from the scope where they are written. That suits callbacks inside methods, but makes arrow functions a poor choice for object methods that need this to be the object.

1249. **D** — A pure function always returns the same output for the same input and changes nothing outside itself: no global writes, I/O or mutation of its arguments. Pure functions are easy to test, cache and run in parallel.

1250. **D** — Currying turns a function of several arguments into a chain of one-argument functions: add(2) returns a function that waits for b. It relies on closures, because the returned function remembers a.

1251. **A** — Because Java generics are implemented by erasure to Object, a type argument must be a class or interface type. Primitives such as int are not, so you use Integer and rely on autoboxing.

1252. **A** — The Comparator contract is: negative means a < b (a goes first), zero means equal, and positive means a > b (b goes first). So (a, b) -> a.compareTo(b) sorts in ascending order.

1253. **B** — Stream intermediate operations build a pipeline but process nothing. A terminal operation pulls elements through the whole pipeline, one at a time. This laziness is what lets streams short-circuit with operations such as findFirst().

1254. **D** — Python's lambda allows exactly one expression, whose value is returned automatically. Statements such as assignment (=), return or if-blocks are not allowed, so logic like this needs a def.

1255. **C** — Decorators are applied from the bottom up: the one nearest def wraps the function first. So b(f) is built first and then passed to a, giving f = a(b(f)). When f is called, a's wrapper runs outermost.

1256. **B** — send() works like next(), but it also passes a value into the generator: the yield where g is paused evaluates to that value. A generator that has not started must first be advanced with next(g) or send(None).

1257. **A** — zip() stops as soon as the shortest input is exhausted. The string 'ab' has only two characters, so the 3 is never paired. Use itertools.zip_longest to pad instead.

1258. **B** — enumerate() yields (index, item) pairs, and start=1 makes the count begin at 1 instead of 0. The index always comes first in each tuple.

1259. **C** — functools.partial(power, e=2) returns a new callable with e fixed at 2, so sq(5) is 5 ** 2 = 25 and sq(3) is 3 ** 2 = 9. 32 and 8 would come from fixing the base at 2 instead (2 ** 5 and 2 ** 3).

1260. **C** — The default value k=k is evaluated when each lambda is created, so each function stores its own k (1, 2 and 3). Calling each with x = 10 gives 10, 20 and 30. Only k has a default, so passing one argument is fine.

1261. **C** — yield from passes on inner's values 1 and 2. When inner returns 'done', that value becomes the result of the yield from expression, so r is 'done', and outer then yields it. A return value is not itself yielded unless outer yields it.

1262. **D** — next(g) runs to the first yield, which gives 0. send(5) makes x = 5, so total becomes 5, and the generator pauses at the next yield. send(10) makes x = 10, total becomes 15, and the generator yields 15, which is printed.

1263. **D** — Decorators apply bottom-up, so g = a(b(g)). Calling g runs a's lambda, which calls b's lambda, which calls the original g. The strings are built inside out, giving a(b(g)).

1264. **A** — islice(iterable, 2, 5) skips the first two items and yields positions 2, 3 and 4 (counting from 0). nat() yields 1, 2, 3, ..., so those are 3, 4 and 5. islice pulls only what it needs, so the infinite generator is safe.

1265. **C** — key=abs sorts by absolute value (1, 3, 4, 5) but returns the original elements. So the output keeps the minus signs: [-1, 3, 4, -5].

1266. **A** — repeat(3) is called first and returns the real decorator deco, which then wraps hi. The wrapper calls the original function n = 3 times and returns the list of results.

1267. **B** — Because __iter__ contains yield, each call to iter(b) returns a fresh generator. Bag is therefore a reusable iterable, not a one-shot iterator, and both list() calls give [2, 4]. A class only needs __iter__ to be iterable.

1268. **C** — Array.prototype.map passes each element and its index to the callback. The products are 3 * 0, 1 * 1 and 2 * 2, giving [0, 1, 4].

1269. **C** — Each call to makeCounter creates a new c, which its arrow function closes over. a has been called twice before the log, so a() returns 3, while b has its own c and returns 1.

1270. **B** — filter keeps elements whose callback result is truthy; n % 2 is 1 for odd numbers, so 1, 3 and 5 remain. map multiplies each by 10, and join(',') produces 10,30,50.

1271. **A** — The first next() runs to yield 1. next(5) resumes it, so x = 5, and the generator yields 10. The third next() runs past the last yield, so the result has done: true.

1272. **B** — filter keeps 4, 7 and 10 (3 is not greater than 3). mapToInt doubles them to 8, 14 and 20, and sum() gives 42. 48 would include the 3.

1273. **C** — [x] captures a copy of x when the lambda is created, so byVal keeps 1. [&x] captures a reference, so byRef sees the later assignment x = 5.

1274. **B** — fact<4> calls fact<3>, then fact<2>, fact<1> and fact<0>. The explicit specialisation fact<0> returns 1 and stops the recursion at compile time. The result is 4 * 3 * 2 * 1 * 1 = 24.

1275. **A** — The third argument 10 is the initial accumulator. reduce then adds len('ab') = 2, len('cde') = 3 and len('') = 0, giving 10 + 5 = 15.

1276. **B** — map() with two iterables calls the function with one item from each and stops at the shorter one. That gives 1 + 10 and 2 + 20; the 3 has no partner.

1277. **C** — compose(f, g) returns x -> f(g(x)), so the right-hand function runs first. compose(inc, dbl)(5) is inc(dbl(5)) = 11, and compose(dbl, inc)(5) is dbl(inc(5)) = 12.

1278. **B** — Any assignment to a name inside a function makes that name local to the function, unless it is declared nonlocal. c += 1 therefore reads a local c that has not been assigned yet, raising UnboundLocalError. Adding nonlocal c fixes it.

1279. **D** — Calling a generator function only creates a generator object; none of its body runs until next() is called. Nothing iterates g, so neither print('start') nor 1/0 ever executes, and only 'made' appears.

1280. **A** — zip pulls from its arguments left to right. After two pairs it takes 3 from it, then finds range(2) exhausted and stops, so 3 is consumed but discarded. next(it) therefore returns 4. Put the shorter iterable first to avoid the silent loss.

1281. **D** — A raw List turns off generic type checking, so raw.add(42) compiles (with a warning) and succeeds at run time, because erasure leaves only a List of Object. The compiler inserts a cast to String when names.get(0) is assigned, and that cast fails. This is heap pollution.

1282. **B** — A lambda may only use local variables that are final or effectively final. count++ changes count, so the compiler rejects it. Use an AtomicInteger, an array element, or a stream operation such as count() instead.

1283. **C** — [&n] stores only a reference to the local n. When makeGetter returns, n's lifetime ends, so the reference dangles and g() reads dead memory, which is undefined behaviour. Capture by value, [n] or [=], to copy it.

1284. **B** — A regular method called as obj.regular() gets this = obj. An arrow function has no this of its own; it uses the module-level this (an empty module.exports object), so this.n is undefined. Use method syntax when you need this.

1285. **D** — A dict's iterator checks that the dictionary's size has not changed. Adding a key during the loop makes the next step raise RuntimeError: dictionary changed size during iteration. Iterate over list(d) instead. Java's ConcurrentModificationException is the same safeguard.

1286. **D** — In a generator expression only the outermost iterable, range(5), is evaluated immediately. The condition x % n == 0 runs later, when list() consumes g, and by then n is 3. Only 0 and 3 pass.

1287. **C** — __iter__ returns self, so the outer and inner loops share one iterator. The outer loop takes 1, the inner loop takes 2 and then hits StopIteration, and the outer loop then finds the iterator exhausted. Return a new iterator from __iter__ to make the class reusable.

1288. **A** — wrapper calls fn(*a) but throws its result away and has no return statement, so it returns None. The decorated add therefore returns None. The fix is return fn(*a).

1289. **B** — Integer.MIN_VALUE - 5 overflows int and wraps to a large positive number, so the comparator claims MIN_VALUE is greater than 5. Java int arithmetic wraps silently. Use Integer.compare(a, b) or Comparator.naturalOrder() instead.

1290. **C** — map calls its callback with (element, index, array). So the calls are parseInt('1', 0), parseInt('2', 1) and parseInt('3', 2). Radix 0 is treated as 10, radix 1 is invalid, and '3' is not a base-2 digit. Use .map(Number) or .map(s => parseInt(s, 10)).

1291. **D** — A generator's return value is stored in the StopIteration exception that ends it; list() catches that exception and does not include the value. Only the yielded 1 appears. Use yield 5, or yield from in another generator, to get the 5.
1292. **A** — vector::erase invalidates iterators at and after the erased position and returns an iterator to the next element. Incrementing the invalidated it is undefined behaviour. The standard idiom is the erase-remove idiom or it = v.erase(it) inside the loop.
1293. **D** — Concurrency means several tasks are in progress during the same period, which one core can achieve by interleaving them. Parallelism means tasks literally execute at the same instant, which needs several cores or processors. Parallel execution is one way of running a concurrent program.
1294. **C** — Threads of one process share its address space: code, heap, globals and open files. Each thread has its own stack, program counter and registers so it can run its own sequence of calls (Silberschatz).
1295. **B** — A race condition occurs when two or more threads access shared data, at least one writes, and the outcome depends on the order in which their operations happen to interleave. It is fixed by synchronisation, such as a lock around the shared update.
1296. **B** — A critical section is code that reads or updates shared resources. Correct synchronisation guarantees mutual exclusion: while one thread is in its critical section, no other thread may be in its critical section for the same data.
1297. **A** — A counting semaphore lets up to its value of threads proceed; further wait() calls block until someone calls signal(). A mutex is a lock with a single owner, and only that owner should unlock it. Semaphores have no owner, so any thread may signal.
1298. **D** — The four conditions are mutual exclusion, hold and wait, no preemption and circular wait. Deadlock is possible only when all four hold together. Bounded waiting is a requirement for a correct critical-section solution, not a deadlock condition.
1299. **C** — In the default CPython build, the GIL allows one thread to execute bytecode at a time, so CPU-bound threads take turns rather than running in parallel. I/O-bound threads still benefit, because the GIL is released while waiting. For CPU-bound work use multiprocessing; Python 3.13 and later also offer an optional free-threaded build.
1300. **A** — A synchronized instance method locks the object's monitor (this). Only one thread at a time can run any synchronized method on that same object; others block until the lock is released. Static synchronized methods lock the Class object instead.
1301. **A** — join() blocks the caller until the target thread terminates (or a timeout passes, if one is given). It is the standard way to wait for worker threads before using their results. start() is what begins a thread.
1302. **B** — run() is an ordinary method call, so its body runs synchronously in the caller's thread. Only start() asks the JVM to create a new thread, which then calls run(). Calling start() twice on the same thread is what throws IllegalThreadStateException.
1303. **C** — volatile ensures visibility and ordering: every read sees the most recent write. It does not make compound actions atomic, and count++ is a read, an add and a write. For an atomic counter use AtomicInteger or a lock.
1304. **A** — In livelock, threads are not blocked; they keep changing state in response to each other without progress. In deadlock, threads are blocked waiting forever. Starvation is one thread being denied a resource indefinitely while others proceed.
1305. **A** — An atomic operation appears to happen all at once. No other thread can observe an intermediate state. Examples include AtomicInteger.incrementAndGet() in Java and compare-and-swap instructions in hardware.
1306. **A** — empty counts free slots and makes producers wait when the buffer is full. full counts filled slots and makes consumers wait when it is empty. The mutex (a binary semaphore) protects the buffer while an item is inserted or removed.
1307. **B** — Creating and destroying a thread for every small task is expensive. A pool keeps worker threads alive, queues tasks and caps how many run at once. Shared data still needs synchronisation, and completion order is not guaranteed.
1308. **D** — Amdahl's law gives speed-up = 1 / (s + (1 − s)/N). As N grows, (1 − s)/N tends to 0, so speed-up approaches 1/s. A program that is 10% serial can never run more than 10 times faster.
1309. **C** — Speed-up = 1 / (0.2 + 0.8/4) = 1 / (0.2 + 0.2) = 1 / 0.4 = 2.5. The serial 20% stops it reaching the ideal 4.
1310. **B** — The limit is 1 / serial fraction = 1 / 0.1 = 10. However many processors run the 90% part, the 10% serial part still takes the same time.

1311. **B** — Speed-up = 1 / (0.25 + 0.75/3) = 1 / (0.25 + 0.25) = 1 / 0.5 = 2.

1312. **D** — Every increment happens inside with lock, so the read-add-write of count is mutually exclusive and no update is lost. All four threads are joined before printing, so the output is always 4 × 1000 = 4000.

1313. **C** — t.join() makes the main thread wait until task() has finished, so 'child' is always printed before 'main'. Without the join() call the order could vary.

1314. **D** — Each successful acquire() decrements the count: 3 → 2 → 1 → 0. The fourth and fifth calls find it at 0 and block until another thread calls release(). Blocking, not an error, is the normal behaviour.

1315. **C** — Each P (wait) decrements the value by 1 and each V (signal) increments it by 1, so the final value is 7 − 20 + 15 = 2. The order does not matter for the final count once all operations have completed.

1316. **D** — A global lock order breaks the circular-wait condition: if everyone takes A before B, no thread can hold B while waiting for A. Reentrant locks only help one thread re-acquire a lock it holds. Priorities and sleeps change the timing but leave the deadlock possible.

1317. **D** — If one thread's read-add-write finishes before the other reads, the result is 7. If both read 5 before either writes, both write 6 and one update is lost. A value of 5 is impossible because at least one write of 6 always happens.

1318. **A** — Executor.map returns results in the order of the input iterable, whichever worker finishes first. So the squares of 3, 1 and 2 are listed as 9, 1, 4. Use as_completed() if you want completion order.

1319. **C** — JavaScript runs on one thread with an event loop. Synchronous code runs first (A, D). Then the microtask queue runs, which holds promise callbacks (C). Finally the timer callback from the macrotask queue runs (B), even with a 0 ms delay.

1320. **D** — Speed-up = 120 / 40 = 3. Efficiency = speed-up / cores = 3 / 4 = 0.75, or 75%. An efficiency of 100% would mean perfect linear scaling.

1321. **D** — Each eating philosopher holds two forks, and 5 forks allow only floor(5/2) = 2 pairs. The two eaters cannot be neighbours, because neighbours share a fork.

1322. **B** — Both coroutines run concurrently on one thread. y sleeps for less time, so it prints first. asyncio.gather still returns results in the order the awaitables were passed, so the list is ['x', 'y'].

1323. **A** — count++ is not atomic. If both threads read the same value before either writes it back, one increment is lost. Use synchronized, a lock or AtomicInteger.incrementAndGet() to make the update atomic.

1324. **C** — Thread 1 may take a.lock while Thread 2 takes b.lock. Each then waits for the lock the other holds: deadlock. A fix is to always lock accounts in a fixed global order, for example by account id, which breaks circular wait.

1325. **B** — The test and the update are separate steps. Both threads can see balance >= 80 as true while it is still 100, then both subtract, giving −60. The check and the subtraction must be in one critical section, for example inside the same lock.

1326. **B** — With the GIL, only one thread executes Python bytecode at a time, so CPU-bound threads just take turns. Separate processes each have their own interpreter and GIL, so ProcessPoolExecutor or multiprocessing can use several cores. More threads add switching overhead. join() waits for threads; it does not start them.

1327. **A** — Speed-up(4) = 1 / (0.4 + 0.6/4) = 1 / 0.55 ≈ 1.82. Speed-up(8) = 1 / (0.4 + 0.075) ≈ 2.11. Doubling the cores gives only about 16% more speed, because the 40% serial part caps speed-up at 2.5.

1328. **B** — A plain Lock has no owner tracking, so a second acquire() by the same thread blocks; with timeout=0.1 it gives up and returns False. An RLock (reentrant lock) counts acquisitions by its owner, so the same thread can acquire it again and gets True. Without the timeout the Lock call would self-deadlock.

1329. **C** — volatile guarantees that each read sees the latest write, but hits++ is still read, add, write, so two threads can read the same value and lose an update. AtomicInteger.incrementAndGet() or a synchronized block makes the whole step atomic.

1330. **C** — The writer is never permanently blocked by a cycle; readers keep arriving and the reader-preference policy keeps admitting them, so the writer is postponed indefinitely. That is starvation. Writer-preference or fair (queue-ordered) locks fix it.

1331. **A** — With one core, only one instruction stream executes at any instant. The scheduler switches between the threads, so both make progress over the same period: that is concurrency without parallelism. Threads in one process still share memory.

1332. **D** — Neither thread is blocked: each keeps releasing and retrying in step with the other, so this is livelock. A random back-off delay breaks the symmetry, and a fixed lock order avoids the conflict entirely.
1333. **A** — One instruction operating on several data elements at the same time is SIMD. Vector units and GPUs work this way. A multicore CPU running different threads on different data is MIMD.
1334. **A** — Data parallelism applies the same operation to different parts of the data on different cores. Task parallelism would run different operations (for example, parsing on one core and compressing on another) at the same time.
1335. **B** — Amdahl's law asks how much faster a fixed job gets, so the serial fraction caps the speed-up. Gustafson's law asks how much more work fits in the same time when the problem is scaled up, which gives the scaled speed-up S = N - s(N - 1).
1336. **A** — A condition variable lets a thread wait for a condition on shared data. wait() atomically unlocks the mutex and blocks, so other threads can change the data. When signalled, the thread re-locks the mutex before continuing.
1337. **C** — Being woken only means the condition might now be true. Another thread may have consumed the item first, or the wake-up may be spurious, which POSIX and Java both allow. Re-testing in a loop makes the thread wait again if needed.
1338. **A** — wait(), notify() and notifyAll() must be called by a thread that owns the object's monitor, because wait() has to release it. Calling them outside a synchronized block or method on that object throws IllegalMonitorStateException at run time.
1339. **C** — Thread.sleep pauses the current thread but releases no locks, so other threads stay blocked on obj. obj.wait releases obj's monitor while waiting and re-acquires it before returning.
1340. **B** — notify() wakes one arbitrary waiting thread; notifyAll() wakes them all, and they then compete for the monitor. notifyAll() is safer when threads wait for different conditions on the same object.
1341. **D** — queue.Queue is built for threads: it locks internally, so producers and consumers need no extra mutex. get() waits until an item is available, and a maxsize makes put() wait when the queue is full, which is the bounded-buffer pattern.
1342. **D** — The interpreter waits for all non-daemon threads before exiting, but not for daemon threads. When only daemon threads remain, the program ends and they are killed without running their remaining code, so they suit background helpers only.
1343. **B** — Thread-local storage gives each thread a private copy of a variable, such as Python's threading.local() or Java's ThreadLocal. Threads can read and write it without locks, because no other thread sees their copy.
1344. **C** — CAS reads, compares and conditionally writes in one atomic hardware step. Lock-free code, such as Java's AtomicInteger.incrementAndGet, loops: read the value, compute the new one, and retry the CAS if another thread changed it in between.
1345. **A** — A spinning thread wastes CPU while it waits, so a spinlock pays off only when the wait is shorter than the cost of blocking and rescheduling. On one core it is especially wasteful, because the holder cannot run while the waiter spins.
1346. **A** — Java uses BLOCKED for a thread waiting to acquire a monitor lock. WAITING is for threads in wait(), join() or park() with no timeout, and TIMED_WAITING is for sleep() or waits with a timeout.
1347. **C** — Race conditions need at least one write to shared data. An immutable object is never written after it is constructed, so any number of threads can read it at the same time with no synchronisation.
1348. **C** — With one kernel thread, the kernel sees one schedulable entity. A blocking call stops all the user threads, and only one can run at a time even on a multicore machine. The one-to-one model, used by Linux and Windows, removes both limits.
1349. **D** — Mutual exclusion keeps at most one process in the critical section. Progress means the choice of who enters next cannot be postponed indefinitely. Bounded waiting limits how many times others can enter before a waiting process gets its turn.
1350. **C** — Node starts an I/O operation and moves on instead of waiting. When the operation finishes, its callback or promise is queued, and the event loop runs it. That gives concurrency on one thread, though not parallel JavaScript execution.
1351. **A** — An async function always returns a Promise. A return value becomes the Promise's fulfilled value, and a thrown error rejects it. Callers use await or .then() to get the result.

1352. **A** — Processes, unlike threads, do not share an address space. The child gets its own copy of the module's globals. To pass data back, use a multiprocessing.Queue, Pipe, Value or Manager.
1353. **C** — Cache coherence works per cache line, not per variable. Each write invalidates the other core's copy of the whole line, so the line keeps moving between cores even though no data is shared. Padding or aligning the variables apart fixes it.
1354. **C** — synchronizedMap guards every call with one lock on the whole map, so all threads queue up. ConcurrentHashMap uses fine-grained locking and CAS on individual bins, and reads are mostly lock-free, so threads seldom block each other.
1355. **D** — A barrier is a meeting point. Each thread that calls wait() blocks until the set number of threads have arrived, and then all continue. It is used between phases of a parallel algorithm.
1356. **D** — A Future represents a result that will be available later. get() blocks until the task completes, then returns the result or throws ExecutionException if the task failed. isDone() checks without blocking.
1357. **D** — Speed-up = 1 / ((1 - 0.5) + 0.5 / 8) = 1 / 0.5625 ≈ 1.78. The serial half alone takes 0.5 of the original time, so the speed-up can never reach 2, however many processors are added.
1358. **A** — The limit is 1 / s, where s is the serial fraction. 1 / s = 5 gives s = 0.2, so 80% of the program is parallel. 20% is the serial part, not the parallel part.
1359. **D** — S = 16 - 0.1 × 15 = 16 - 1.5 = 14.5. Amdahl's law with the same serial fraction would give only about 6.4, because it keeps the problem size fixed.
1360. **D** — Speed-up = T1 / Tp = 200 / 25 = 8. Efficiency = speed-up / cores = 8 / 10 = 0.8, or 80%. The last option subtracts the times instead of dividing them.
1361. **A** — Speed-up = 1 / (0.1 + 0.9 / 10) = 1 / 0.19 ≈ 5.26. Even with 10 cores the 10% serial part keeps it well below 10. 9 comes from wrongly multiplying 0.9 by 10.
1362. **A** — The worker blocks on ready.wait() until the event is set. The main thread appends 'main' before calling ready.set(), so 'worker' is always appended second. join() then waits for the worker to finish before printing.
1363. **D** — queue.Queue is first in, first out, so the consumer receives 0, 1 and 4 in the order they were put. The None sentinel ends the loop and is not appended. Starting the consumer first does not matter, because get() blocks until an item arrives.
1364. **D** — gather schedules A and then B on one event loop. Each task prints once and then await asyncio.sleep(0) hands control back to the loop, so the tasks take turns. This is concurrency through cooperative switching on a single thread.
1365. **B** — gather runs both coroutines concurrently, so their waits overlap. Total time is set by the longest sleep, 2 s, not the sum, 3 s. Awaiting them one after another would take 3 s.
1366. **A** — Calling f() runs its body synchronously up to the first await, so f1 prints right after start. await suspends f and queues the rest as a microtask, so end prints next and f2 prints once the synchronous code finishes.
1367. **B** — Promise.race settles with the first promise to settle, b at 10 ms, so b prints first. Promise.all waits for all three (30 ms) and keeps the results in the order the promises were passed, not the order they finished, so it prints abc.
1368. **D** — With S = 0, X's wait(S) blocks until Y calls signal(S), and Y signals only after printing 'first'. A semaphore initialised to 0 is the standard way to force one event to happen before another.
1369. **D** — getAndIncrement() returns the old value (5) and then sets c to 6. incrementAndGet() sets c to 7 and returns the new value (7). c.get() is 7.
1370. **C** — The wait-for chain T1 -> T2 -> T3 has no cycle, because T3 waits for nothing. With single-instance resources a deadlock needs a cycle. T3 runs and releases R3, which unblocks T2, and T2's release of R2 unblocks T1.
1371. **C** — The child process gets its own copy of count and adds 100 to that copy. The parent's count is untouched, so it prints 0. Shared state between processes needs multiprocessing.Value, a Queue or a Manager.
1372. **C** — The futures are kept in the list in submission order, and f.result() waits for each one. So the results follow the order 3, 1, 2 regardless of which thread finished first.
1373. **B** — threading.local() gives every thread its own attributes. The worker sets its own data.x, which is a separate copy from the main thread's. After join(), the main thread still sees its own value, 'main'.
1374. **C** — Semaphore(2) allows two acquisitions, so the first two non-blocking acquire() calls return True and the third returns False. release() adds one permit back, so the next acquire() succeeds.

1375. **B** — The tasks run in waves of at most 4: 4 in the first second, 4 in the second and the last 2 in the third. That is ceil(10 / 4) = 3 s. 2.5 s would need tasks to be split, which they cannot be.
1376. **B** — Each step halves the number of values: 16 -> 8 -> 4 -> 2 -> 1, which is log2(16) = 4 steps. 15 is the total number of additions, which is the number of steps a single processor would need.
1377. **B** — Only the parallel part shrinks: 60 / 4 = 15 s. The serial 40 s stays the same, so the total is 55 s. 25 s wrongly divides the whole 100 s by 4.
1378. **C** — This is a check-then-act race. Thread A tests None, is switched out, Thread B also tests None, and both construct objects. The test and the assignment must happen as one critical section under a lock.
1379. **A** — Without volatile, the JVM may reorder the writes so another thread sees the reference before the constructor's writes. That thread passes the unsynchronised outer check and uses a half-built object. volatile forbids the reordering; a holder class is the other common fix.
1380. **D** — When woken, a consumer re-acquires the lock, but the other consumer may already have taken the only item, and wake-ups can be spurious. With if, the woken thread pops without re-checking. while not items: cond.wait() re-tests the condition.
1381. **D** — The background thread is a daemon, and the main thread does not join it. When the main thread finishes, only a daemon thread is left, so the interpreter exits without waiting the 1 s, and 'bg done' never prints.
1382. **B** — Java monitors are reentrant: the JVM counts how many times the owning thread has acquired the lock, and releases it when the count returns to zero. A thread can therefore call one synchronized method from another on the same object.
1383. **C** — A lock only excludes threads that acquire the same lock. Because each thread takes a different one, both can be in the read-add-write sequence at once. Every access to a shared variable must be guarded by one agreed lock.
1384. **B** — Superlinear speed-up is real and usually comes from caches. Splitting the data shrinks each core's working set until it fits in cache, so each core runs faster than the single core did on the whole data set.
1385. **C** — Holding a lock during slow I/O forces every thread to queue for it, so the threads run one at a time. Keep critical sections short: do the I/O outside the lock, and lock only around the shared data that actually changes.
1386. **B** — H is effectively blocked by the lower-priority M, which is priority inversion. Under priority inheritance L inherits H's priority while holding the mutex, so M cannot pre-empt it. L then releases the lock quickly.
1387. **A** — asyncio switches tasks only at an await. time.sleep(1) is a blocking call with no await, so it freezes the single event-loop thread, and the second coroutine cannot start until the first finishes. Use await asyncio.sleep(1) instead.
1388. **B** — The GIL stops two threads executing Python bytecode at the same instant, but network waits happen outside the interpreter, with the GIL released. The 20 downloads overlap their waiting time, so total time drops sharply. The GIL limits CPU-bound threads.
1389. **A** — Hold and wait means a thread holds some resources while waiting for others. Taking all locks at once, or none, means no thread ever holds one lock while blocked on another, so that condition cannot occur.
1390. **B** — Each thread locks its own Counter instance, so the two threads never contend for the same monitor. Both can run total++ (read, add, write) at once and lose updates. Use a static synchronized method, synchronized (Counter.class), or an AtomicInteger.
1391. **B** — Node runs all JavaScript callbacks on one thread. A long synchronous computation never yields, so the event loop cannot dispatch other callbacks. CPU-bound work belongs in worker_threads or a child process. Making the function async does not move it off the thread.
1392. **D** — The Java memory model does not guarantee that a plain field written by one thread becomes visible to another. The JIT may hoist the read out of the loop. Declaring running volatile creates a happens-before edge, so the worker sees false.
1393. **B** — When execution reaches a breakpoint, the debugger suspends the program before running that line. You can then inspect variables and the call stack and step through the code. It does not change what the program does.
1394. **C** — Step into enters the called function and pauses at its first line. Step over runs the whole call and pauses on the next line of the current function. Step out finishes the current function and returns to its caller. Continue runs to the next breakpoint.

1395. **C** — A conditional breakpoint pauses only when its expression is true, so you reach the failing iteration directly. An ordinary breakpoint would stop 7,342 times first, and stepping through every iteration is impractical.
1396. **A** — A logic error is a mistake in the algorithm, such as a wrong formula or loop bound. The program is valid and runs to completion but gives incorrect output. Syntax and type errors stop compilation, and a runtime exception stops the program while it is running.
1397. **D** — Unit tests exercise a single function, method or class with chosen inputs and assert the expected results. They are fast and automated, and dependencies are often replaced by test doubles. End-user and performance testing are other levels and types.
1398. **A** — TDD repeats a short loop known as red, green, refactor. Write a small test that fails (red), write the minimum code to make it pass (green), then clean up the code with the test as a safety net. Each loop adds one small behaviour.
1399. **B** — Both are test doubles that replace real dependencies. A stub supplies fixed responses so the code under test can run. A mock is set up with expectations, and the test fails if the expected calls (for example, send_email called once) did not happen. Some libraries, such as Python's unittest.mock, use 'mock' loosely for both.
1400. **B** — Regression testing checks that changes have not broken previously working behaviour. An automated unit-test suite makes it cheap to run after every change. Acceptance testing checks requirements with the customer, and load testing checks behaviour under heavy use.
1401. **A** — White-box (structural) testing uses knowledge of the code, for example to cover every branch. Black-box (functional) testing chooses inputs and expected outputs from the requirements without looking inside, for example with equivalence partitioning and boundary values. Either can be manual or automated.
1402. **D** — A profiler records how often each function is called and how long it takes (or how much memory it uses). This shows the real hotspots, so you optimise based on measurement rather than guessing. Examples include Python's cProfile and Java's VisualVM.
1403. **A** — Taking every branch outcome means every statement on those branches runs, so branch coverage subsumes statement coverage. The reverse fails: an if with no else can have every statement run while its false outcome is never tested. Full coverage shows code was executed, not that its results were correct.
1404. **D** — Arrange sets up the inputs and objects. Act calls the method being tested, usually in a single line. Assert checks the outcome. Keeping these three parts separate makes a test easy to read.
1405. **D** — Integration testing combines units and tests their interfaces, such as a service calling a repository. Unit testing isolates single units, and acceptance testing checks the finished system against user requirements. Static analysis inspects code without running it.
1406. **B** — Most run time is spent in a small part of the code, and guessing which part is often wrong. Optimising early makes code harder to read for little gain. Profile first, then optimise the measured hotspots, especially by choosing better algorithms.
1407. **C** — Caching is only correct if the same arguments always give the same result and skipping a call loses nothing. Time-dependent or random functions would return stale values, and side effects such as file writes would silently stop happening.
1408. **B** — The call stack lists the active function frames, from the current function back to main. Selecting a frame shows that function's local variables. A watch window tracks chosen expressions, and a stack trace printed on a crash gives the same chain in text form.
1409. **D** — Boundary value analysis tests values at and just beyond each edge, because off-by-one mistakes cluster there. 17 and 61 should be rejected, while 18 and 60 should be accepted. Middle values add little.
1410. **B** — There are three partitions: below the range (≤ 0), inside it (1–100) and above it (≥ 101). One representative from each, such as −5, 50 and 150, covers them all. Boundary value analysis would add tests at the edges.
1411. **C** — For a single function, McCabe's cyclomatic complexity is the number of decision points plus 1: 2 + 1 + 1 = 4. It is also the number of independent paths, which gives a guide to how many tests are needed.
1412. **D** — Each if has a true and a false outcome. Test (1, 1) takes both true branches and test (0, 0) takes both false branches, so 2 tests cover all four outcomes. Covering all four combinations (4 tests) would be path coverage, which is stronger than needed.
1413. **A** — avg([2, 4]) is 6 / 2 = 3.0, and 3.0 == 3 is True in Python, so the first assertion passes. avg([1, 2]) is 1.5, which is not equal to 1, so the second raises AssertionError.

1414. **B** — The call tree for fib(5) has 15 nodes, because calls(n) = calls(n−1) + calls(n−2) + 1 with calls(0) = calls(1) = 1. With memoisation (for example @lru_cache) only fib(0) to fib(5) are computed, giving 6 calls. This exponential repetition is why naive recursive Fibonacci is slow.

1415. **A** — cumtime includes time spent in functions called from a function, so main and parse look expensive mainly because they call lookup. tottime is time spent in the function's own code, and lookup accounts for 7.4 of the 10 seconds. Making lookup faster, or calling it less often, gives the biggest gain.

1416. **C** — Membership in a list is a linear scan, O(m), done n times. Building a set costs O(m) once, and each lookup is O(1) on average because it uses hashing, so the total becomes O(n + m). The worst case for hashing is still linear, but average case is what this question asks.

1417. **A** — best starts at 0, and none of −5, −2 or −9 is greater than 0, so it is never updated and 0 is returned. The bug is the initial value; starting with best = xs[0] (or float('-inf')) gives the correct answer, −2.

1418. **B** — range(1, len(xs)) gives indices 1 and 2, so the loop skips xs[0] and adds 5 + 6 = 11. This is an off-by-one bug; range(len(xs)) or sum(xs) gives the intended 15.

1419. **D** — The square root gives the same value on every iteration, so computing it n times is wasted work. Storing it once before the loop (loop-invariant code motion) removes n − 1 calls without changing the result. Recursion or threads would add overhead, not remove it.

1420. **C** — 0.1 and 0.2 have no exact binary representation, so their sum is 0.30000000000000004 and == returns False. math.isclose compares with a relative tolerance, so it returns True. In tests, use assertAlmostEqual or pytest.approx for floats.

1421. **C** — Each test halves the remaining range, so the number of steps is ceil(log2 1000) = 10, because 2^10 = 1024 ≥ 1000. A linear search could need up to 1,000 tests.

1422. **D** — Gregorian leap years are divisible by 4, except century years, which must also be divisible by 400. 1900 is not a leap year, but the function returns True. 2024 and 2023 are handled correctly, and 2000 really is a leap year, so it does not reveal the bug either.

1423. **A** — Such 'heisenbugs' change behaviour when observed because printing or pausing changes thread timing (or memory layout, in C/C++). Race conditions and uninitialised memory are typical causes. A wrong formula in a pure function would give the same wrong result every time.

1424. **A** — The test calls add but never checks result, so nothing can make it fail except an exception. Adding assert result == 4 would expose the bug, because 2 − 2 is 0. A passing test is only as strong as its assertions.

1425. **D** — An order-dependent result usually means a test relies on state left by another test, or another test leaves state behind. Each test should create its own fresh data in setup and clean it up in teardown. A syntax error would fail in both cases.

1426. **B** — Amdahl's law applies to any partial optimisation: 1 / (0.1 + 0.9/2) = 1 / 0.55 ≈ 1.82. The 10% outside X is unchanged, so the whole program cannot become twice as fast.

1427. **C** — lru_cache stores results in a dictionary keyed by the arguments, so the arguments must be hashable. A list is mutable and unhashable, so the call raises TypeError. Pass a tuple instead, which also guards against the list changing after its result was cached.

1428. **D** — Each concatenation creates a new String and copies all previous characters, so total copying grows with the square of the length. A StringBuilder appends into a growable buffer, making the whole build O(n). Loop style makes no real difference here.

1429. **C** — A Python function that ends without return returns None. id 7 is not in users, so get_user(7) is None and subscripting it fails. Fix it by returning a default, raising a clear error, or checking for None before use.

1430. **C** — A useful test must be able to fail when the code is wrong. Unsorted inputs such as [3, 1, 2], duplicates, an empty list and negative numbers would expose the broken function. Python compares lists element by element with ==.

1431. **B** — With lo = 0 and hi = 1, mid = 0 and a[0] = 1 < 3, so lo is set to 0 again. Nothing changes, and the loop repeats forever. Using lo = mid + 1 guarantees progress, because mid < hi whenever lo < hi.

1432. **A** — Nested loops compare every pair, which is $O(n^2)$. A single pass that checks whether target − x is already in a hash set is O(n) on average. Sorting first and then using two pointers would give O(n log n), but sorting after the nested loops adds work rather than removing it.

1433. **C** — Step over runs the whole call on the current line as one step and pauses on the next line of the same function. Step into would enter save_record, step out would leave the current function, and restart begins the program again.

1434. **B** — Step out (also called step return) finishes the current function and stops in the caller right after the call. Continue would run on to the next breakpoint, which may be far away or nowhere at all.
1435. **C** — A watch evaluates the chosen expression at every pause and shows its value, so you can see it change as you step. A conditional breakpoint decides when to pause, and the call stack shows which functions led to the current line.
1436. **D** — The code is valid and starts running, but it fails during execution for a particular input, which is a runtime error. A syntax error stops the program before it runs, and a logic error gives a wrong answer without crashing.
1437. **D** — Python lists frames from the outermost call to the innermost, so the most recent call, the line that raised the exception, is the last frame, directly above the exception type and message. The first frame is usually the top-level call in your script.
1438. **C** — Rubber duck debugging means explaining code aloud, step by step, to any listener, even an object. Saying what each line should do forces you to compare intent with the code, which often exposes the bug.
1439. **D** — Static analysis reads the source code (or bytecode) and flags suspicious patterns without executing it. Profilers, debuggers and test runners are dynamic tools: they all need the program to run.
1440. **B** — The pyramid puts many fast, cheap, precise unit tests at the base, fewer integration tests in the middle and only a few slow, brittle end-to-end tests at the top. Manual testing is not part of the automated pyramid's base.
1441. **C** — Smoke testing is a quick, shallow check that the most important functions of a build work at all, so it is worth testing further. Regression testing re-runs the full existing suite, stress testing pushes load past normal limits, and acceptance testing checks user requirements.
1442. **B** — Both are forms of acceptance testing near release. Alpha testing is done by internal staff or selected users at the developer's site, and beta testing hands the near-final product to real users in their own environments to find problems the team did not foresee.
1443. **C** — unittest calls setUp before every test method, so each test starts from the same known state (the fixture). tearDown runs after each test to clean up. Test methods are found by their test_ prefix, not by setUp.
1444. **A** — Mutation testing plants small faults (mutants) and checks whether the tests kill them by failing. A surviving mutant means no test distinguishes > from >=, so a boundary test at x = 0 is missing. It says nothing on its own about whether the original code is wrong.
1445. **D** — A fuzzer generates huge numbers of unexpected, random or corrupted inputs and watches for crashes, hangs or failed assertions. Boundary value analysis picks a few chosen edge values by hand, and code review does not run the program at all.
1446. **A** — A fake actually works but is simplified and unsuitable for production, like an in-memory database. A stub only returns canned answers, a mock also verifies how it was called, and a dummy is passed in but never used.
1447. **B** — To judge a test you must know the right answer. The oracle is whatever supplies it: a specification, a trusted older version, a mathematical property or a hand-worked result. Without a reliable oracle, a test can run but cannot tell pass from fail.
1448. **A** — Covering every path through the code also takes every branch outcome, which in turn runs every statement and calls every function. The reverse does not hold: two tests can cover all branches of two if statements without covering all four paths.
1449. **B** — The levels rise in severity: DEBUG, INFO, WARNING, ERROR, CRITICAL. DEBUG is the most detailed, for diagnosing problems, and is normally filtered out in production, where the threshold is often INFO or WARNING.
1450. **B** — With optimisation enabled (python -O), the interpreter strips assert statements entirely. Asserts are for catching programmer mistakes during development; input checks should use if statements that raise ValueError or similar.
1451. **A** — A sampling profiler interrupts the program periodically and notes where it is; hot spots show up in more samples. An instrumenting profiler like cProfile hooks every call and return, so counts are exact but overhead is higher.
1452. **A** — A leak happens when a program loses the last pointer to heap memory without freeing it, so the memory stays reserved until the process ends. The other options are different bugs: buffer over-read, double free and use of an uninitialised variable.
1453. **B** — Valgrind's Memcheck and AddressSanitizer track which bytes are valid and report out-of-bounds reads and writes, use-after-free and leaks. gprof measures time, clang-format only reformats code, and git bisect finds the commit that introduced a bug.

1454. **D** — A driver stands in for the missing caller: it invokes the module under test and passes in test data. Stubs do the opposite job in top-down integration, standing in for lower-level modules that the tested code calls.

1455. **D** — The square brackets build the whole list in memory before sum starts. The generator expression yields each square on demand, so only one value exists at a time. Both give the same total.

1456. **D** — The threshold is WARNING, so only messages at WARNING or above are shown: 'WARNING: c' and 'ERROR: d'. DEBUG and INFO are below the threshold and are dropped.

1457. **C** — The default list is created once, when def runs, and shared by every call that omits basket. Both calls return the same list object, and print shows it only after both appends, so it appears twice as ['pen', 'ink']. Use basket=None and create a new list inside.

1458. **C** — Each lambda looks up i when it is called, not when it is created (late binding). By then the comprehension has finished with i = 2, so every call returns 20. Writing lambda i=i: i * 10 captures each value.

1459. **B** — Removing the first 2 shifts the later items left, so the second 2 moves into the position the loop has just visited and is skipped. The result keeps one 2. Build a new list instead: [n for n in nums if n != 2].

1460. **D** — var gives one function-scoped i shared by all three callbacks. The callbacks run after the loop ends, when i is 3, so each prints 3. Declaring the loop variable with let creates a new binding per iteration and prints 0, 1, 2.

1461. **B** — == converts the string "0" to the number 0 before comparing, so it is true. === compares type as well as value, and a number is never strictly equal to a string, so it is false.

1462. **B** — Multiplying the outer list copies the reference, so both rows are the same inner list object. Changing one row changes 'both'. Use [[0] * 3 for _ in range(2)] to create separate rows.

1463. **A** — total / count divides two ints, so Java performs integer division and gets 3. Only then is 3 widened to the double 3.0. Casting one operand, (double) total / count, gives 3.5.

1464. **A** — There are two valid classes (0 to 49 and 50 to 100) and two invalid classes (below 0 and above 100), so 4 partitions, each tested with one representative value such as -5, 30, 75 and 120.

1465. **C** — One test with m = 60 executes every statement, including the assignment inside the if. Branch coverage would need a second test, such as m = 30, so the false outcome of the if is also taken.

1466. **D** — V(G) = 14 - 13 + 2 x 2 = 5. Each separate component adds 2, which is the same as summing E - N + 2 for each function. Taking P = 1 gives 3, and forgetting the 2P term gives 1.

1467. **A** — By Amdahl's law the new run time is 0.6 + 0.4 / 4 = 0.7 of the original, so the speed-up is 1 / 0.7, about 1.43. The 60% that was not optimised limits the gain.

1468. **A** — With quadratic growth, multiplying n by 4 multiplies the time by 4^2 = 16, so 2 x 16 = 32 seconds. 8 seconds would be linear growth.

1469. **D** — With the cache, the body runs once for each distinct argument 0 to 30, which is 31 times; every repeat call is answered from the cache. Without memoisation fib(30) would make 2,692,537 calls.

1470. **A** — Dictionary keys are case-sensitive, so 'Ali' is not the same key as 'ali'. Indexing with a missing key raises KeyError. scores.get('Ali') would return None instead.

1471. **C** — 20 / 2 = 10 gaps, and a row of gaps needs one more post than gaps, so 11 posts. Counting gaps instead of posts is the classic fencepost or off-by-one error.

1472. **C** — The context-manager form runs int("abc") inside the with block, where assertRaises can catch the exception. In the other forms int("abc") is evaluated as an argument before any assertion method runs, so the ValueError escapes and the test errors.

1473. **C** — A Mock records every call. call_count is 2, and call_args holds the arguments of the most recent call, which was send('b@x.pk'). call_args_list would hold both calls.

1474. **C** — With no compare function, sort converts elements to strings and orders them as text: '100' < '25' < '9' because '1' < '2' < '9'. The array looks unchanged. marks.sort((a, b) => a - b) sorts numerically.

1475. **C** — Python 3's round uses round-half-to-even (banker's rounding): an exact .5 goes to the nearest even integer. So 0.5 becomes 0, 1.5 becomes 2 and 2.5 becomes 2, and the test's expectation is wrong.

1476. **C** — p[r + 1] is the sum of a[0..r] and p[l] is the sum of a[0..l-1], so their difference is exactly a[l..r]. The other expressions are off by one at one end; p[l - 1] even wraps to the last element in Python when l = 0.

1477. **A** — compute_total raised an exception, but except Exception: pass discarded it, so total was never assigned. The only visible symptom is a later NameError far from the cause. Catch specific exceptions and log or re-raise them.

1478. **A** — strlen counts characters until the terminating '\0', which takes O(n), and the condition i < strlen(s) calls it n times. Computing size_t len = strlen(s) once before the loop makes it O(n).
1479. **D** — Java int arithmetic silently wraps on overflow. 3,000,000,000 - 2^32 = -1,294,967,296, which is what prints. Declaring total as long (and adding with long arithmetic) gives the correct sum.
1480. **B** — The test is non-deterministic because greeting reads the real clock. Injecting the hour (for example greeting(now)) or mocking datetime lets the test fix the time and check both outcomes deterministically.
1481. **D** — By Amdahl's law, the new time is 0.98 + 0.02 / 10 = 0.982 of the original, a gain under 2%. Effort should go to the functions that the profile shows take most of the time.
1482. **D** — Every line ran, so coverage is full, but the crashing input (b = 0, ZeroDivisionError) was never tried. Coverage measures execution, not correctness; edge-case tests and a decision on how to handle zero are needed.
1483. **A** — Inside the with block, billing.compute_total is replaced by a mock that returns 150 whatever it is given, so the real code never runs. Mock the dependencies of the unit under test (a database, a price service), never the unit itself.
1484. **D** — Garbage collection only frees objects that are unreachable. Every entry stays reachable from _cache, and the number of (page, user) pairs keeps growing. A bounded cache such as functools.lru_cache(maxsize=1024) or a time-based eviction policy fixes the leak.
1485. **B** — Each addition carries a tiny rounding error, so x skips past 1.0 without ever equalling it. Use x < 1.0 as the condition, or loop over an integer counter and compute x = i / 10.
1486. **D** — For a 3-item list, i reaches 2 and the code reads a[3], which does not exist. Only n - 1 adjacent pairs exist, so the loop should stop one index earlier.
1487. **A** — Memoisation is only safe for functions whose result depends solely on their arguments. get_rate depends on time as well, so cached values go stale. A cache with expiry (a time-to-live) or no cache is needed.
1488. **B** — A Python list is a dynamic array, so inserting at the front moves all current elements one place: O(n) each time and O(n^2) overall. Appending is amortised O(1), and deque supports O(1) inserts at both ends.
1489. **B** — For objects, == checks whether two references point to the same object. A string built at run time is a new object even with the same characters. equals compares contents.
1490. **B** — The base case only stops at exactly 0. Starting below 0, each call moves further away until Python's recursion limit (about 1000) is hit. Validate n >= 0 or raise ValueError for negative input.
1491. **A** — Python evaluates arguments before calling a function, so the f-string, and expensive_dump inside it, runs on every loop pass. logging then drops the message. Checking isEnabledFor(logging.DEBUG) first skips the work when DEBUG is off.
1492. **C** — git add stages a snapshot of the named files, so the next commit will include them. git commit records what is staged, git push uploads commits to a remote, and git init creates a new repository.
1493. **A** — Version control records every committed change with its author and message. That lets you see who changed what, roll back mistakes, work on branches in parallel and merge everyone's work. It does not compile or debug code.
1494. **A** — git pull is git fetch followed by git merge (or a rebase, if configured). git fetch alone downloads commits without touching your branch, and git push uploads your commits.
1495. **A** — git revert keeps history intact by adding an inverse commit, so it is safe on shared branches. git reset rewrites the branch so it points at an earlier commit, which is fine locally but disrupts anyone who already pulled the removed commits.
1496. **D** — DRY (from The Pragmatic Programmer) says each piece of knowledge should have a single representation in a system. Duplicated logic means a fix must be applied in several places, and one is usually missed.
1497. **D** — YAGNI ('You Aren't Gonna Need It') comes from Extreme Programming. Speculative features add code to test and maintain, and often turn out to be wrong when the real need arrives. Build for today's requirements and keep the code easy to change.
1498. **B** — PEP 8 uses snake_case (lowercase with underscores) for functions and variables, CapWords (PascalCase) for classes and UPPER_CASE for constants. camelCase is the Java convention for methods.
1499. **D** — Java constants (static final fields) use UPPER_SNAKE_CASE. Classes use PascalCase (StudentRecord), and methods and variables use camelCase (maxStudents). snake_case is not the Java convention.

1500. **B** — A magic number is a bare literal whose purpose the reader must guess. Replacing it with a named constant, such as SECONDS_PER_DAY = 86400, documents its meaning and gives one place to change it.
1501. **C** — A docstring is a triple-quoted string placed first in a module, class or function body. Python stores it in the object's __doc__ attribute, so help() and documentation tools can show it. Comments starting with # are discarded and are not docstrings.
1502. **B** — Well-named code already shows what it does. Comments add value by giving the reasons that cannot be seen in the code, such as a regulation, a workaround or a performance trade-off. Change history belongs in version control, and dead code should be deleted.
1503. **C** — Reviews catch bugs, unclear code and security issues before they are merged, when they are cheapest to fix. They also spread knowledge of the codebase and its standards. They complement automated tests, and good reviews criticise code rather than people.
1504. **C** — Semantic versioning is MAJOR.MINOR.PATCH. An incompatible API change increments MAJOR and resets MINOR and PATCH to 0. A backward-compatible feature bumps MINOR (2.5.0), and a bug fix bumps PATCH (2.4.2).
1505. **D** — git rebase replays your commits on a new base, creating new commits with new hashes. Collaborators who built on the old commits then have divergent history, and pushing the result needs a force-push. Rebasing private, unpushed work is fine and keeps history linear.
1506. **B** — HEAD is a pointer to what you have checked out. Normally it points to a branch name, which in turn points to that branch's latest commit. If you check out a commit hash directly, HEAD points to that commit (a 'detached HEAD').
1507. **A** — In CI, small changes are integrated frequently, and an automated server builds the code and runs the test suite on each push. Integration problems then appear within minutes rather than at a painful big-bang merge. Two people at one computer is pair programming.
1508. **A** — git switch -c feature (or the older git checkout -b feature) creates the branch and checks it out. git branch feature only creates it, leaving you on the current branch. commit has no -b option.
1509. **D** — git add stages the file as it is at that moment (v1). The later edit to v2 is not staged, so the commit stores v1 and git status then shows a.txt as modified. Use git add again, or git commit -a, to include v2.
1510. **D** — ?? marks an untracked file: Git sees it but it is neither committed nor staged. After git add it would show A (added to the index). M means modified and D means deleted.
1511. **B** — main now contains A, C, B (reachable through the merge) and the new merge commit, which has two parents: 4 commits in total.
1512. **D** — When the current branch has no commits that the other lacks, Git just moves the branch pointer forward. main then holds A, B and C, with no merge commit. Use --no-ff if you want a merge commit anyway.
1513. **D** — Git cannot choose between two edits of the same line, so it writes both into the file with conflict markers: your side (HEAD) first, then the other branch's side. You edit the file, remove the markers, then git add and commit to finish the merge.
1514. **D** — git revert does not remove anything. It adds a new commit that applies the inverse of the last commit's changes, so the history grows to 4 commits and the files return to their state after commit 2.
1515. **A** — --soft moves only the branch pointer, leaving the index and working files untouched, so the undone commit's changes appear as staged (ready to recommit). --mixed (the default) also unstages them, and --hard discards them. Creating an inverse commit is what revert does.
1516. **B** — A good name says what the value means without needing a comment. days_since_borrowed reveals its unit and purpose. Single letters, generic words like data and noise like tmp force readers to trace the code.
1517. **C** — Extract Function removes the duplication (DRY). When the tax rule changes, you edit one place instead of five and cannot miss a copy. Comments and file moves leave the duplication in place.
1518. **A** — SRP says a class should have one reason to change. Here a new tax rule, a new PDF layout and a database migration would all change the same class. Splitting the responsibilities gives each one a single home.
1519. **A** — The triple-quoted string is the first statement in the function body, so Python stores it as the docstring in area.__doc__. Printing that attribute shows the text. The function's name is in __name__, not __doc__.
1520. **C** — A backward-compatible feature increments MINOR and resets PATCH: 1.9.3 becomes 1.10.0. Version parts are separate numbers, so after 9 comes 10, not 2.0. 1.9.4 would mean a bug fix only.

1521. **B** — git stash saves your uncommitted changes onto a stack and restores the files to the last commit, so status shows no changes. git stash pop later re-applies them.
1522. **B** — One function doing four jobs is hard to read, test and change: the Long Method smell. Extracting read_records, validate, calculate_totals and write_report gives small functions that each do one thing and can be unit-tested separately.
1523. **C** — Rebasing gives the commits new hashes. Teammates still have the old commits, so Git treats both versions as separate work, causing duplicates and conflicts. Use merge (or revert) on shared branches and keep rebase for local, unpublished branches.
1524. **C** — Every commit is a full snapshot, so anyone with the repository can check out the earlier commit and read the key. The safe response is to revoke or rotate the key, then remove it from history (for example by amending or rewriting the unpushed commits) and keep secrets out with .gitignore or environment variables.
1525. **A** — The code runs as written, so customers get 15% off, while the comment promises 10%. An out-of-date comment misleads every future reader. Check the requirement, fix whichever is wrong and replace the magic number with a named constant such as DISCOUNT_RATE.
1526. **A** — A reader cannot tell what True, False, True mean without opening the definition. Keyword arguments (validate=True), an options object or separate functions make the call self-explanatory. Readability is a core purpose of review, and adding speculative flags goes against YAGNI.
1527. **C** — Git compares both branches with their common ancestor. Each file was changed on only one side, so the three-way merge takes each change and records a merge commit. Conflicts arise only when both sides change the same lines. It is not a fast-forward, because main has its own new commit.
1528. **B** — Checking out a hash gives a detached HEAD. Commits made there are not on any branch, so once you switch away nothing points to them. git reflog still records the hash, and git branch rescue <hash> keeps it. Unreachable commits are eventually removed by garbage collection.
1529. **B** — Repeated magic numbers combine two smells: unclear meaning and duplication. When the value changes, every copy must be found by hand. A single constant such as Status.SHIPPED makes the code self-documenting and gives one place to change.
1530. **D** — git cherry-pick applies one commit's changes on top of the current branch as a new commit. Its parent and commit time differ, so its hash differs from the original, which stays on feature. Only that one commit is copied, not the whole branch.
1531. **C** — Ignore rules tell Git not to start tracking files. A file already in the index stays tracked. git rm --cached removes it from the index (and from the next commit) while leaving it on disk, after which the ignore rule applies.
1532. **A** — git init turns the current folder into a repository by creating .git, which will hold the objects, refs and configuration. Nothing is staged or committed yet. Copying an existing remote repository is git clone.
1533. **D** — In a distributed system each clone is a complete repository. Work such as committing and browsing history happens locally, and changes are exchanged with push and pull. Central-only history describes centralised systems such as Subversion.
1534. **A** — fetch only updates your copy of the remote's state (origin/main and similar). Your own branch moves only when you merge or rebase, which is what git pull adds on top of fetch. Uploading is git push.
1535. **C** — Plain git diff shows unstaged changes. git diff --staged (or --cached) compares the staging area with the last commit, and git diff HEAD compares the working directory with the last commit.
1536. **C** — --amend builds a fresh commit from the previous commit's content plus the current staging area, and the branch points to it instead. Because the hash changes, amending a commit that others have already pulled rewrites shared history.
1537. **C** — clone records the source URL as a remote called origin, which is why you push with git push origin main and see branches such as origin/main. The name is a convention and can be changed.
1538. **B** — A pull request proposes merging a branch. It shows the diff, collects review comments and approvals, and usually shows CI results, so the change is checked before it reaches the main branch.
1539. **A** — KISS warns against needless complexity, because simple code is easier to read, test and change. Small, client-specific interfaces describe the Interface Segregation Principle, a different idea.
1540. **A** — The OCP (the O in SOLID) aims for designs where adding a feature, such as a new shape or payment type, means adding a new class or module rather than changing existing, working code. Polymorphism and interfaces are the usual tools.

1541. **B** — Refactoring keeps behaviour the same while improving structure, for example by extracting a function or renaming a variable. Tests should pass before and after. Adding features or fixing bugs changes behaviour, so it is separate work.

1542. **A** — @throws (or its older synonym @exception) names the exception class and explains when it is thrown. @param describes a parameter, @return the return value, and @see adds a cross-reference.

1543. **B** — A README is the front page of a project. It should tell a newcomer what the project is for and how to get it running, often with usage examples and contribution notes. History lives in Git, and code lives in the source files.

1544. **C** — High cohesion means each module's parts belong together around one purpose. Low coupling means modules depend on each other as little as possible. Together they make changes local and code easier to test and reuse.

1545. **A** — Both build on CI. Continuous delivery keeps the main branch always releasable, with a manual release step. Continuous deployment removes that step, so every change that passes the pipeline is released automatically.

1546. **D** — Pair programming is two people at one workstation. The driver handles the keyboard and the navigator reviews continuously and plans ahead. Pairs swap roles often, which spreads knowledge and catches defects early.

1547. **D** — Formatters change whitespace, line breaks and quotes so that style debates disappear from code review. Linters inspect code for problems such as unused imports, undefined names or risky patterns, and usually report rather than rewrite.

1548. **C** — Like financial debt, shortcuts such as duplicated code or missing tests save time now but charge 'interest': every later change is slower until the debt is repaid by refactoring.

1549. **A** — An atomic commit does one thing fully, such as fixing one bug together with its test. That keeps history readable and makes revert, cherry-pick and bisect precise. One logical change may touch several files.

1550. **C** — ~n follows first parents n steps back, so HEAD~1 is the commit main pointed to before the merge. ^n picks the nth parent of a merge, so HEAD^2 is the merged-in feature tip. The grandparent is HEAD~2.

1551. **C** — SemVer reserves major version 0 for initial development, where the usual compatibility promises do not apply. Version 1.0.0 defines the first stable public API. Pre-releases are marked with a hyphen suffix, such as 1.0.0-beta.

1552. **C** — The caret allows changes that SemVer says are compatible: new minor and patch versions within major version 1. The tilde (~1.4.2) is the narrower range that allows only patch updates below 1.5.0.

1553. **D** — Version ranges in the manifest can resolve to different releases over time. The lock file pins the exact resolved versions so that a teammate's laptop, CI and production all install identical dependencies. It should be committed.

1554. **D** — git cat-file -t shows 'tag' for an annotated tag and 'commit' for a lightweight one, because the lightweight tag refers directly to the commit. Annotated tags are recommended for releases. Neither kind moves when new commits are made.

1555. **A** — --mixed moves the branch back to A and resets the staging area to match, but leaves the working file alone. B's edit therefore appears as an unstaged modification. --soft would leave it staged, and --hard would discard it.

1556. **D** — -a stages modifications to files Git already tracks, so the edit to a.txt is committed. It never adds untracked files, so new.txt stays untracked until you git add it.

1557. **D** — The first column describes the staging area (the first edit is staged) and the second describes the working tree (the second edit is not). A commit now would include only the first edit. AM would mean the file was newly added, not modified.

1558. **A** — -d is the safe delete: it refuses when the branch has commits not reachable from the current branch or its upstream, so work is not lost by accident. -D deletes anyway, leaving the commit reachable only through the reflog.

1559. **B** — Rebase replays B and D as new commits B' and D' on top of C, giving the straight history A, C, B', D'. No merge commit is created; that is the main difference from git merge.

1560. **B** — apply restores the most recent stash into the working tree but keeps it in the stash list. git stash pop applies it and then drops it, which would leave 1 entry.

1561. **D** — Amend replaces the last commit rather than adding one, so the count stays at 2. The last commit now contains the forgotten file and has a new hash.
1562. **B** — --squash combines feature's changes into the staging area, and the commit you make is an ordinary one-parent commit. Git does not record any link to F1 to F3, so it still considers feature unmerged.
1563. **B** — *.log ignores every .log file, and the later rule !important.log re-includes that one file. So a.log is ignored, while important.log and b.txt show as ?? untracked.
1564. **B** — -n is a dry run: it prints 'Would remove ...' for each untracked file. git clean -f actually deletes them, and -x also removes ignored files. Changes to tracked files are discarded with git restore, not git clean.
1565. **A** — git diff compares the working tree with the staging area, and only c is unstaged. --staged shows only b (staging area against the last commit), git diff HEAD shows both b and c, and git show displays the last commit.
1566. **B** — A compatible bug fix increments PATCH, and each part is a separate integer, so 9 becomes 10. MINOR is for new compatible features and MAJOR for breaking changes. SemVer has exactly three numeric parts.
1567. **B** — Each part is compared as a number, so minor 10 comes after 9. A pre-release has lower precedence than the normal version with the same numbers, so 1.10.0-beta comes just before 1.10.0. Sorting as text would wrongly put 1.10 before 1.2.
1568. **B** — Java class names use PascalCase (UpperCamelCase). camelCase starting with a lower-case letter is for methods and variables, and UPPER_SNAKE_CASE is for constants. snake_case is the Python style for functions and variables.
1569. **C** — Only a string literal that is the first statement of the function body becomes its docstring. Here an assignment comes first, so the string is just an unused expression and __doc__ is None. Every function has a __doc__ attribute, so there is no error.
1570. **B** — Type hints document the intended types and help tools such as mypy and IDEs, but the interpreter does not enforce them at run time. "ab" * 2 is string repetition, giving abab.
1571. **B** — doctest runs both examples in the docstring. square(3) gives 9 as documented, but square(-2) gives 4, not -4, so one of two examples fails. Doctests keep documentation honest because wrong examples fail.
1572. **D** — The original charges 300 to anyone under 18, then 500 to adult members and 800 to other adults. Checking age < 18 first and returning early keeps that logic flat. Checking member first charges young members 500, <= 18 misprices 18-year-olds, and not member swaps the adult prices.
1573. **D** — HEAD@{1} is the reflog entry for where HEAD was one move ago, which is D. Resetting there restores the branch to A, B, C, D. Commits are not deleted by a reset; they stay reachable through the reflog until garbage collection.
1574. **C** — restore --staged unstages the file by copying the committed version back into the staging area. The edit itself stays in the working tree, so the file shows as modified but not staged. git restore a.txt (without --staged) would discard the edit.
1575. **B** — Git leaves conflict markers in the file and refuses to commit while paths are unmerged. You resolve the content yourself, git add marks the file as resolved, and git commit creates the merge commit. git merge --abort cancels instead.
1576. **D** — The remote has a commit you do not have, so your push would discard it. Integrating it first keeps both changes, and the next push is a fast-forward.
1577. **D** — Commented-out code clutters the file, confuses readers about what runs and goes stale quickly. Version control is the safe place for old code, and it can be recovered with git log or git show.
1578. **A** — Guard clauses handle each failure case first and exit, so the remaining code reads as the normal path at one indentation level. It also makes it clear which error belongs to which check.
1579. **C** — Reviewers find defects well only in small, single-purpose diffs; in a huge mixed diff they skim and approve. Separate pull requests for formatting, refactoring and the feature are each easy to review and to revert.
1580. **A** — Without versions, pip installs the latest releases. The laptop still has the older versions installed months ago, so the two environments differ. Pinning exact versions (requests==2.32.3) or using a lock file makes installs reproducible.
1581. **C** — The Law of Demeter (principle of least knowledge) says a method should talk to its immediate collaborators, not reach through them. Adding order.getShippingCity() keeps the navigation inside Order, so OrderPrinter is not coupled to Address.

1582. **C** — Generated files change whenever anyone builds, so two branches always disagree about them. They can be rebuilt from source, so they do not belong in version control. Because they are already tracked, .gitignore alone is not enough.

1583. **A** — Months later, the message is often the only explanation of a change. A short imperative summary such as 'Fix rounding in invoice total', with a body for the reason if needed, makes history useful.

1584. **D** — SemVer promises that minor releases are backward compatible, which is exactly what ^ relies on. Removing public API breaks that promise. A lock file would have protected your build, but the versioning error is the library's.

1585. **C** — Conflicts grow with the time branches spend apart. Integrating small changes often keeps each merge small, and the automated build catches breakage within minutes, while the cause is still fresh.

1586. **A** — A long parameter list is a code smell: calls are hard to read and arguments are easy to swap by mistake. Grouping values that belong together into objects (Address, ContactInfo) shortens the list and gives the concepts names.

1587. **A** — Git's three-way merge needs unchanged context between the two edits to apply both safely. Edits that touch or neighbour each other form one conflicting region. Edits on lines 1 and 6 of the same file merge cleanly.

1588. **B** — A good comment is about the code, not the person. It names a specific problem and its impact, and suggests a fix. Personal remarks put people on the defensive, and rubber-stamp approvals let defects through.

1589. **D** — ahead counts local commits not on the remote-tracking branch, and behind counts the reverse. With both non-zero, a push is not a fast-forward and would be rejected. Merge or rebase onto origin/main first.

1590. **D** — Shotgun surgery means one change forces many small edits across the codebase. Collecting each payment method's behaviour in one place (polymorphism) means a new method becomes one new class, in line with the Open/Closed Principle.

1591. **A** — Overriding needs the same name and the same parameter list (signature). print(String) differs from print(), so ColourPrinter inherits print() and adds an overload. A ColourPrinter object can call both print() and print("red").

1592. **D** — Java checks which methods may be called using the declared (static) type of the reference, here Animal. Animal has no fetch, so line 2 is a compile-time error even though the object is a Dog. A cast, ((Dog) a).fetch(), would compile and run.

1593. **A** — show is not virtual, so the call is bound at compile time to the reference's static type, Base. hello is virtual, so the call is dispatched at run time to the object's real type, Derived. In C++, polymorphism through a reference or pointer needs virtual.

1594. **B** — Python searches the method resolution order (MRO), which for class C(A, B) is C, A, B, object: left to right through the base list. A defines hello first, so 'A' is returned. Python does not report an ambiguity error the way C++ does.

1595. **D** — Static methods are not overridden; B.who() only hides A.who(). A static call is resolved from the declared type, A, giving 'A'. Instance methods are overridden, so obj.me() dispatches to B and returns 'B'. The output is AB. Calling a static method through an instance compiles, but with a warning.

1596. **B** — A protected member is accessible in subclasses, so line 1 compiles. A private member is accessible only inside the class that declares it, so line 2 fails even though the object physically contains secret. Parent would need a getter or a protected field.

1597. **C** — Each constructor first calls super(), so Vehicle runs first. Before Car's constructor body runs, Car's field initialisers run, printing 'wheels'. Then the Car body runs, and finally the SportsCar body. Construction goes from the top of the hierarchy down.

1598. **A** — Each call x.area() is dispatched on the real object: Circle gives 3 × 2 × 2 = 12, Square gives 3 × 3 = 9 and Shape gives 0. The total is 21. Because total is a double, println shows 21.0, not 21.

1599. **D** — Java picks overloads in phases. Phase 1 allows only identity and primitive widening, and it beats boxing. For 5 (an int), both long and double qualify by widening, and the most specific, long, wins over Integer. For 5.0f (a float), only double is reachable by widening.

1600. **A** — self.count += 1 reads the class attribute the first time, then creates an instance attribute that shadows it. So a.count becomes 2 and b.count becomes 1, both stored on the instances. Changing Counter.count to 10 affects only objects without their own count, such as a new Counter().

1601. **C** — Leaf.greet adds 'Leaf>' and calls super().greet(), which is Mid's version. Mid adds 'Mid>' and calls Base's version, which returns 'Base'. The pieces join in call order, giving Leaf>Mid>Base.

1602. **C** — Base's constructor runs first and calls show(). Calls are dynamically dispatched even inside constructors, so Derived.show runs, but Derived's field initialiser (x = 7) has not executed yet, so x still holds its default 0. Then x is set to 7 and Derived() calls show() again: 7.

1603. **B** — pay is virtual, so it uses each object's real type: 100 for e and 300 for m. bonus is not virtual, so through an Emp* it always calls Emp::bonus, giving 10 each time. Total: (100 + 10) + (300 + 10) = 420. Making bonus virtual would give 460.

1604. **C** — super() follows the MRO of the actual object's class, not the class where the method is written. D's MRO is D, B, C, A, object (C3 linearisation), so B's super().f() calls C, not A. A appears once, at the end. This is how Python solves the diamond problem.

1605. **D** — Fields are never polymorphic in Java: B.x hides A.x, and ref.x is resolved at compile time from ref's type, A, giving 1. getX is overridden, so the call dispatches to B.getX, which reads B's x, giving 2. This is why fields are usually private, with access through overridable getters.

1606. **A** — null fits both overloads, and Java chooses the most specific applicable one. String is a subtype of Object, so write(String) wins. For write(o), overloads are chosen at compile time from the declared type Object, so write(Object) runs even though the object is a String. Overloading is static; only overriding is dynamic.

1607. **C** — Passing a Cat by value to an Animal parameter constructs a new Animal object from c, so the Cat part is sliced off and its dynamic type really is Animal. Virtual dispatch then correctly calls Animal::speak. Passing by reference keeps the original Cat object, so Cat::speak runs. Polymorphism in C++ needs a pointer or a reference.

1608. **B** — members = [] runs once, when the class is defined, creating a single list stored on Team. self.members.append(...) mutates that shared list; it never creates an instance attribute, unlike self.count += 1 on an int. Fix it by writing self.members = [] inside __init__.

1609. **D** — equals(Point) overloads Object.equals(Object) instead of overriding it. a.equals(b) has a Point argument, so the overload runs and returns true. ArrayList.contains calls equals(Object), which still resolves to Object's identity check, so it returns false. Declaring equals(Object o) with @Override fixes it.

1610. **B** — print(x) calls str(x), which uses the overridden __str__. Printing a list calls repr on each element, and Item did not override __repr__, so object's default representation (with a memory address) appears. Overriding __repr__ as well fixes both.

1611. **D** — &x is the address of x, and p stores that address. The dereference operator * follows the address to the value stored there, so *p is 25. p and &x are both the address, and &p is the address of the pointer variable itself.

1612. **D** — Pointer arithmetic is scaled by the size of the pointed-to type. p + 1 means 'the next int', which is sizeof(int) = 4 bytes further on: 2000 + 4 = 2004. A char* would move by 1 byte.

1613. **B** — The language defines arr[i] as *(arr + i): the array decays to a pointer to its first element, which is moved i elements forward and then dereferenced. *arr + i adds i to the first value instead, and *(&arr[i] + 1) is arr[i + 1].

1614. **C** — Read the declarations right to left. const int *p is a pointer to a constant int: the value is read-only through p, but p itself can move. int *const q is a constant pointer to an int: q is fixed to a, but *q can change a. const int *const r would forbid both.

1615. **C** — Assignment in Python binds a new name to the same object; it never copies. a and b refer to one list, so appending through b is visible through a, and a is b is True. Use a.copy() or a[:] to get a separate list.

1616. **B** — An array parameter is adjusted to a pointer, so int a[10] in a parameter list means int *a; the 10 is ignored. sizeof(a) is the size of a pointer, 8 bytes here. This is why C functions take the array length as a separate argument. GCC even warns about it.

1617. **D** — p points at arr[1] = 14. *(p + 2) is two elements further on, arr[3] = 28. p[-1] means *(p − 1), which is arr[0] = 7. A negative index is fine while it stays inside the array.

1618. **A** — &d[4] is 4 elements of 8 bytes past the start: 0x3000 + 32 = 0x3020 (32 is 0x20 in hex). Casting to char* changes the scale to 1 byte, so (char*)d + 4 is 0x3004. Remember to convert the offset to hex.

1619. **D** — Subtracting two pointers into the same array gives the distance in elements, not bytes: 6 − 1 = 5. The byte distance would be 5 × 4 = 20. %td is the printf format for ptrdiff_t.

1620. **B** — a is a pointer, so *a += b changes the caller's variable; b is a copy, so doubling it has no effect outside. First call: x = 4 + 3 = 7. Second call: y = 3 + 7 = 10, using the updated x. Output: 7 10.

1621. **A** — pp points at p. **pp = 9 writes through p into x, so x = 9. *pp = &y changes p itself, which now points at y. *p += 10 then changes y to 12, and **pp follows pp, then p, to y, giving 12. Output: 9 12 12.

1622. **C** — *(*(m + 2) + 1) is m[2][1] = 21. C stores 2D arrays row by row, so &m[0][0] + 6 is the 7th int in memory: row 1 (6 ÷ 4), column 2 (6 mod 4), which is m[1][2] = 12.

1623. **B** — A shallow copy makes a new outer list but shares the inner lists, so changing grid[0][0] shows through shallow (99). deepcopy also copies the inner lists, so deep keeps 1. Appending a row changes only grid's outer list, so shallow still has 2 rows.

1624. **B** — s += 4 moves the pointer past 'N', 'S', 'C' and 'T', so printing s with %s gives 2026. *(s + 1) is the character '0'. s[3] is '6', and subtracting '0' turns the digit character into the number 6. The string literal is never modified, so this is safe.

1625. **A** — Pointers are passed by value too. make changes its own copy p, so ptr stays nullptr and the program prints 'still null'. The new int is also leaked, because no one keeps its address. Passing &ptr to an int** parameter, or using an int*& reference, lets make update the caller's pointer.

1626. **D** — *p++ parses as *(p++): it reads 5, then moves p to a[1]. (*p)++ reads 10 and then increments a[1] to 11; p does not move. ++*p increments a[1] to 12 and yields 12. So x = 5, y = 10, z = 12, a[1] = 12, and p − a = 1. Each statement modifies one object once, so nothing is undefined.

1627. **C** — The default copy is memberwise, so b.data receives the same address as a.data: a shallow copy. Writing *b.data = 42 changes v, which a also sees. Re-pointing b.data to w affects only b, so a.data still points at v (42), and w becomes 99. If Box owned heap memory, this shared pointer would lead to a double delete; that is the rule of three.

1628. **A** — [[0] * 3] * 2 copies the reference to one inner list twice, so both rows are the same object and a change to one row shows in both. The list comprehension builds a new inner list on each iteration, so only row 0 changes.

1629. **C** — A reference is bound once, at initialisation, and cannot be re-seated. r = y copies y's value (2) into x; it does not make r refer to y. Changing y to 5 leaves x alone, and r += 10 makes x = 12. A pointer, unlike a reference, could be re-pointed.

1630. **A** — sizeof(arr) is the whole array, 40 bytes, and 40 / 4 = 10 elements. sizeof(p) is just the pointer, 8 bytes, and 8 / 4 = 2, which is meaningless as a length. The 'count elements' idiom works only on a real array in scope, never on a pointer.