

1. Define problem-solving steps. List any three characteristics of a good algorithm.

Problem-solving in computing is a structured process that transforms a real-world or abstract problem into a sequence of well-defined steps that a computer (or human) can execute to produce the desired outcome. Good problem solving begins by fully understanding the problem and the expected results, then proceeds through planning, designing an algorithm, implementing it in code, and finally testing and refining the solution. Each stage reduces ambiguity and helps ensure the solution is correct, efficient and maintainable.

(mix of steps and characteristics):

1. Problem identification & understanding — clarify inputs, outputs, constraints and success criteria.
 2. Analysis of requirements — specify what resources, data and edge cases must be handled.
 3. Algorithm design — create a step-by-step logical solution (often with pseudocode/flowchart).
 4. Implementation (Coding) — translate the algorithm into a programming language.
 5. Testing & debugging — validate with normal/edge/invalid inputs and fix errors.
 6. Characteristics of a good algorithm: Definiteness (clear steps), Finiteness (terminates), Effectiveness (each step is doable), Efficiency (time/space reasonable), Correctness (produces expected result), Robustness (handles edge cases).
-

2. Pseudocode and flowchart to determine whether a number is even or odd.

To decide if an integer is even or odd we examine its remainder when divided by 2. If the remainder is zero the number is even; otherwise it is odd. This check is efficient — $O(1)$ time — and directly maps to modulus or remainder operators present in most languages. Below is clear pseudocode and a simple flowchart representation.

Pseudocode (clear, line-by-line):

START

READ n

IF $n \bmod 2 == 0$ THEN

 PRINT "n is even"

ELSE

 PRINT "n is odd"

END IF

STOP

Flowchart (textual diagram + description):

[Start]

|

[Input: n]

|

[Decision: $n \% 2 == 0$?]

/ \

Yes No

| |

[Print "n is even"] [Print "n is odd"]

\ /

[Stop]

(explanation + steps + properties):

1. Input: single integer n .
2. Operation: compute $n \% 2$ (modulus).
3. Decision: if result equals 0 \rightarrow even; else \rightarrow odd.
4. Correctness: modulus yields exact remainder; classification is exhaustive.
5. Complexity: constant time $O(1)$ and constant space.
6. Edge cases: negative integers behave the same using $\%$ semantics in most languages; zero is even.
7. Pseudocode clarity: shows definite, ordered, finite steps.
8. Flowchart utility: good for visualizing branches and ensuring every path ends in stop.

3. Symbolic representation of flowcharts, their importance, limitations, and flow of control with neat diagrams.

Flowcharts are graphical representations of algorithms or processes that use standard symbols to represent operations, inputs/outputs, decisions and the flow of control. These standardized symbols help people (and teams) quickly understand program logic without reading code — they're especially useful in planning, teaching, and communicating designs. However, flowcharts have limitations such as poor scalability for very large systems and potential ambiguity if not carefully drawn. Flow of

control refers to the sequence in which statements or blocks of an algorithm are executed — through linear steps, conditional branching, loops, and subroutine calls.

Common flowchart symbols (with short descriptions):

1. Start/Stop (Oval): marks beginning and termination.
2. Process (Rectangle): statements or operations (assignments, calculations).
3. Input/Output (Parallelogram): read or write operations.
4. Decision (Diamond): branching — yes/no or true/false with multiple exits.
5. Flow line (Arrow): indicates the order of execution.
6. Connector (Small circle): connects separated parts of a chart (used when chart spans pages).
7. Subroutine / Predefined process (Double-lined rectangle): calls a named process or function.
8. Annotation (Bracket): explanatory comments attached to part of chart.
9. Merge / Junction: combining flows back together after branches.
10. Looping structure (combination of decision + arrows): represents repetition (while, for loops).

Flow of control — explanation & small diagrams:

- Sequential control: steps executed in order ($A \rightarrow B \rightarrow C$).
Diagram: [Start] → [Process A] → [Process B] → [Stop]
- Selection (branching): choose one of multiple paths based on a condition.
Diagram: [Decision] → (Yes) → [Process X] and (No) → [Process Y] then merge.
- Iteration (looping): repeat a block while a condition holds (pre-test or post-test).
Diagram (while loop): [Decision] with arrow back to [Process] on true; false goes onward.
- Subroutine / modular control: flow jumps into a subroutine symbol and returns — improves readability and reuse.

(importance, benefits, limitations):

1. Clarity: visually conveys control flow, making logic easier to grasp.
2. Communication: useful across teams and stakeholders (non-coders included).
3. Design aid: helps find logic errors early, before coding.
4. Documentation: serves as enduring design documentation.
5. Debugging: simplifies identifying incorrect or unreachable paths.
6. Education: excellent tool for teaching programming constructs.
7. Scalability limitation: charts get cluttered for complex systems.
8. Maintenance issue: frequent code changes require flowchart updates; otherwise they diverge.

9. Ambiguity if inconsistent symbols used: requires adherence to standard symbols.
10. Not ideal for dynamic behavior: runtime issues like concurrency and exception handling are harder to represent.

4. Explain type conversion in Python (implicit and explicit) with suitable examples.

Type conversion (casting) in Python is the process of converting a value from one data type to another. Python supports implicit (automatic) conversion performed by the interpreter when it safely promotes types (for example, from int to float during arithmetic), and explicit (manual) conversion where the programmer uses conversion functions like `int()`, `float()`, `str()`, etc. Explicit conversion is necessary when mixing types in contexts (like concatenation) that require a single common type, or when converting user input (strings) to numeric types for computation.

Examples with explanation and sample outputs:

Implicit conversion example:

```
a = 5      # int
b = 2.5    # float
c = a + b  # int promoted to float, result is float
print(type(a), type(b), type(c)) # <class 'int'> <class 'float'> <class 'float'>
print(c)   # 7.5
```

Explanation: Python automatically converts `a` to float before addition; no programmer action needed.

Explicit conversion example:

```
s = "123"
n = int(s)    # convert string to integer
f = float(n)  # convert integer to float
t = str(f)    # convert float to string
print(n, f, t) # 123 123.0 '123.0'
```

Explanation: We explicitly change types using built-in constructors. This can raise exceptions (e.g., `int("abc")` causes `ValueError`).

(details, rules, pitfalls):

1. Implicit conversion is automatic when no precision will be lost (e.g., `int` → `float`).
2. Explicit conversion uses built-ins: `int()`, `float()`, `str()`, `bool()`, `list()`, `tuple()`.

3. Numeric promotions: expressions with mixed numeric types produce the “widest” type (int + float → float).
 4. Strings & numbers: require explicit conversion to combine (e.g., concatenating string + number needs `str()` or f-string).
 5. Loss of information: converting float → int truncates fractional part (possible data loss).
 6. Invalid conversions raise exceptions: e.g., `int("12.3")` raises `ValueError`.
 7. Boolean conversions: `bool(0)` is `False`, `bool("")` is `False`, `nonzero/ nonempty` are `True`.
 8. Parsing user input: `input()` returns a string — always convert explicitly before arithmetic.
 9. Use decimal or fractions for precise decimal or rational conversions when float precision matters.
 10. Check with `isinstance()` before converting if input source is uncertain to avoid runtime errors.
-

5. What are Python keywords? Give any five examples.

Python keywords are reserved words that have special meaning in the language grammar; they cannot be used as identifiers (names for variables, functions, classes, etc.). Keywords define the language structure: flow control, declarations, and other syntactic constructs. The keyword module in Python can list all keywords available in the current version.

(definition + five examples with short meaning):

1. Definition: Reserved words that are part of Python’s syntax and cannot be used as variable names.
2. Example — `if`: starts a conditional block.
3. Example — `else`: alternative branch for `if`.
4. Example — `for`: loop over sequence.
5. Example — `while`: loop while condition is true.
6. Example — `def`: declare a function.

(Other common keywords: `class`, `import`, `return`, `try`, `except`, `with`, `lambda`.)

6. Explain identifiers, variables, and data types with suitable examples.

Identifiers are programmer-defined names used to identify variables, functions, classes and other objects in code. A variable is a named reference or label that points to data stored in memory; in Python variables are dynamically typed, meaning a variable can hold different types over its lifetime.

Data types classify the values — e.g., integers, floating points, strings, booleans, lists — and each type defines the set of operations that can be performed on its values.

(definitions + rules + examples):

1. Identifier: name chosen by programmer; rules: start with a letter or underscore `_`, contain letters/digits/underscores, case-sensitive, cannot be a keyword. Example: `student_name`.
2. Variable: name bound to a value. Example: `age = 21`. Variables are references to objects.
3. Dynamic typing: `x = 5` then `x = "hello"` is allowed in Python.
4. Primitive data types: `int` (e.g., `5`), `float` (e.g., `3.14`), `bool` (`True/False`), `str` (`"abc"`).
5. Compound data types: `list` (`[1,2,3]`), `tuple` (`(1,2)`), `dict` (`{'a':1}`), `set` (`{1,2}`).
6. Type checking: use `type()` or `isinstance()`; e.g., `type(3) → <class 'int'>`.
7. Naming conventions: use meaningful names, `snake_case` for variables and functions, avoid single letters except in loops.
8. Example snippet:

```
name = "Asha"    # identifier 'name' bound to a str
```

```
marks = 87      # int
```

```
pi = 3.14159    # float
```

```
passed = True   # bool
```

7. Menu Driven Program to generate a calculator using match-case statement.

A menu-driven calculator prompts the user to choose an operation and enter operands, then performs the selected operation. `match-case` (introduced in Python 3.10) provides a structured way to branch on the user's choice, improving readability over long `if-elif` chains for discrete options. Below is a complete, user-friendly example with input validation and sample outputs.

Python program (complete):

```
# Simple menu-driven calculator using match-case (Python 3.10+)
```

```
def get_number(prompt):
```

```
    while True:
```

```
        try:
```

```
            return float(input(prompt))
```

```
        except ValueError:
```

```
            print("Invalid number. Please enter numeric value.")
```

```
def calculator():
```

```
    a = get_number("Enter first number: ")
```

```
    b = get_number("Enter second number: ")
```

```
    print("\nMenu:")
```

```
    print("1. Add")
```

```
    print("2. Subtract")
```

```
    print("3. Multiply")
```

```
    print("4. Divide")
```

```
    print("5. Modulus")
```

```
    print("6. Power (a^b)")
```

```
    print("7. Exit")
```

```
choice = input("Enter choice (1-7): ").strip()
```

```
match choice:
```

```
    case "1":
```

```
        result = a + b
```

```
        print(f"Result: {a} + {b} = {result}")
```

```
    case "2":
```

```
        result = a - b
```

```
        print(f"Result: {a} - {b} = {result}")
```

```
    case "3":
```

```
        result = a * b
```

```
        print(f"Result: {a} * {b} = {result}")
```

```
    case "4":
```

```
        if b == 0:
```

```
            print("Error: Division by zero.")
```

```
        else:
```

```
            result = a / b
```

```
            print(f"Result: {a} / {b} = {result}")
```

```
    case "5":
```

```

        result = a % b

        print(f"Result: {a} % {b} = {result}")

    case "6":

        result = a ** b

        print(f"Result: {a} ** {b} = {result}")

    case "7":

        print("Exiting calculator. Goodbye!")

    case _:

        print("Invalid choice. Please select 1-7.")

```

```

if __name__ == "__main__":
    calculator()

```

Explanation & sample run:

- The `get_number` function enforces numeric input, preventing `ValueError`.
- `match-case` branches by string choices.
- Division protects against division by zero.
- To run: use Python 3.10+.

Sample input/output (example):

Enter first number: 8

Enter second number: 2

Menu:

1. Add
2. Subtract
3. Multiply
4. Divide
5. Modulus
6. Power (a^b)
7. Exit

Enter choice (1-7): 4

Result: $8.0 / 2.0 = 4.0$

(design choices + features + edge cases):

1. Input validation: `get_number` loop prevents non-numeric input.
2. Clear menu: lists operations and exit option.
3. match-case: clean branching for discrete choices.
4. Error handling: division by zero check.
5. Floating arithmetic: accepts floats for generality.
6. Extensible: add operations (sin, cos) by new cases.
7. User experience: prints descriptive result strings.
8. Maintainability: small helper `get_number` separates concerns.
9. Compatibility: requires Python 3.10+ for match-case.
10. Security: no arbitrary code execution (no `eval` used).

8. What is a Python module? List any three commonly used modules.

A Python module is a file containing Python definitions and statements — functions, classes, and variables — that can be imported and reused in other Python programs. Modules enable modularity, code reuse, and namespace separation. The standard library provides a large collection of modules for different tasks (math, file I/O, networking, etc.), and third-party modules can be installed via `pip`.

(definition + three examples + purpose):

1. Definition: file ending in `.py` containing Python code (functions, classes, variables).
 2. How to use: `import module_name` or `from module_name import symbol`.
 3. Example — `math`: advanced math functions like `sqrt`, `sin`, constants `pi`.
 4. Example — `random`: random number generation and sampling.
 5. Example — `statistics`: mean, median, variance, and other stats helpers.
 6. Benefit: promotes code reuse and improves organization.
-

9. Algorithm and flowchart to find the smallest of three numbers.

Finding the smallest of three numbers can be done by pairwise comparisons: compare two numbers to find a candidate minimum, then compare that candidate with the third number. This approach uses at most two comparisons in the ideal flow (or up to three comparisons depending on branching), and is straightforward to implement.

Algorithm (stepwise):

1. Start
2. Read three numbers a, b, c
3. Let $\text{min} = a$
4. If $b < \text{min}$ then $\text{min} = b$
5. If $c < \text{min}$ then $\text{min} = c$
6. Print min as smallest
7. Stop

Flowchart (text diagram):

[Start]

|

[Input a, b, c]

|

[$\text{min} = a$]

|

[Decision: $b < \text{min} ?$] --Yes--> [$\text{min} = b$] --\

| No \

v

[Decision: $c < \text{min} ?$] --Yes--> [$\text{min} = c$] ---+--> [Print min]

| No /

v

[Print min] <-----/

|

[Stop]

(logic, complexity, edge cases):

1. Initiate min with one of the numbers to avoid extra comparisons.
2. Compare b with min , update if smaller.
3. Compare c with updated min .
4. Print the final min .
5. Comparisons: at most two effective comparisons (after initializing min).
6. Handles equal values: if numbers equal, first occurrence remains min .

7. Complexity: $O(1)$ time, constant operations.
 8. Alternative approaches: nested if-else chains or `min(a,b,c)` built-in in Python.
-

10. Discuss elif statement with suitable example.

The elif (else-if) statement in Python extends an if statement to check multiple mutually exclusive conditions in sequence. When an if condition is false, Python evaluates the first elif; if that is false it continues to the next elif, and so on, until an else (optional) or the end. elif allows clean, readable multi-branch logic without deeply nested if blocks. Only the first true branch executes; subsequent conditions are ignored.

Example — grading program:

```
marks = int(input("Enter marks (0-100): "))
```

```
if marks >= 90:
```

```
    print("Grade: A+ (Distinction)")
```

```
elif marks >= 75:
```

```
    print("Grade: A")
```

```
elif marks >= 60:
```

```
    print("Grade: B")
```

```
elif marks >= 50:
```

```
    print("Grade: C")
```

```
elif marks >= 35:
```

```
    print("Grade: Pass")
```

```
else:
```

```
    print("Grade: Fail")
```

Explanation: This sequence checks from highest to lowest; once a condition matches (e.g., marks=78 matches marks >= 75) the corresponding block runs and the remaining elifs are skipped.

(features, use-cases, rules):

1. Purpose: test multiple exclusive conditions in sequence.
2. Execution model: only first true branch executes.
3. Avoids nesting: improves readability vs nested if inside else.
4. Order matters: place most likely or highest priority checks first.

5. else optional: used as fallback if no if/elif matched.
 6. Use with comparisons and boolean expressions.
 7. Example use cases: grading, menu selection, category mapping.
 8. Performance: stops checking once a condition is true — efficient for prioritized checks.
 9. Maintainability: easy to add/remove branches, but very long chains may still be refactored into lookup tables.
 10. Pitfall: overlapping conditions must be ordered carefully to avoid incorrect earlier match.
-

11. Define flow of control. Explain its importance in programming.

Flow of control (sometimes “control flow”) refers to the order in which individual statements, instructions, or function calls are executed or evaluated in a program. It’s governed by constructs like sequential execution, branching (if/else), looping (for/while), and function calls — these determine how a program responds to data and decisions at runtime. Understanding and managing control flow is central to designing correct, efficient and predictable programs.

(definition + importance):

1. Definition: the execution order of statements and blocks in a program.
 2. Constructs: includes sequence, selection, iteration, and function/subroutine calls.
 3. Decision making: enables different outcomes for different inputs using branching.
 4. Repetition: loops allow repeated execution and are essential for processing sequences/collections.
 5. Modularity: control flow across functions allows decomposition and reuse.
 6. Importance: correct control flow ensures program correctness, handles edge cases, and determines program performance and responsiveness.
-

12. Describe the scope of variables with a clear comparison of local vs global.

Variable scope determines where a variable can be accessed within a program. Global variables are defined at the top level of a module and are accessible throughout that module (and importers if exported). Local variables are created inside functions or blocks and exist only during the execution of that function — they are not visible or accessible outside. Careful use of scope prevents name collisions and reduces unintended side effects; global variables can simplify shared state but increase coupling, while local variables promote modular, testable code.

(direct comparison and examples):

1. Global variable: declared outside functions; accessible by any code in the module.
 2. `x = 10 # global`
 3. `def f(): print(x)`
 4. Local variable: declared inside a function; lifetime limited to function call.
 5. `def f():`
 6. `y = 5 # local`
 7. Visibility: locals not visible outside their function; globals visible inside functions unless shadowed.
 8. Shadowing: a local with same name hides the global inside function scope.
 9. Modification of globals: use global keyword to reassign a global inside function (discouraged).
 10. `def inc():`
 11. `global x`
 12. `x += 1`
 13. Best practice: prefer local variables; limit use of globals to constants or well-encapsulated state.
 14. Testing & maintenance: locals ease unit testing; globals can create hidden dependencies.
 15. Example of error: unintentionally using a variable before assignment inside a function raises `UnboundLocalError` if Python treats it as local due to assignment in function.
-

13. Explain break, continue, and pass statements with examples.

`break`, `continue`, and `pass` are special control statements used in loops and general flow: `break` exits the nearest enclosing loop immediately, `continue` skips the remaining statements in the current loop iteration and jumps to the next iteration, while `pass` is a null operation — it does nothing and is used as a placeholder where syntactically a statement is required (e.g., in empty function bodies or temporary stubs).

Examples and explanation:

- `break`: terminate loop early when a condition is met.

for `i` in `range(1, 10)`:

 if `i == 5`:

`break`

`print(i)`

Output: 1 2 3 4 (then loop stops)

- `continue`: skip to next iteration when current iteration should be ignored.

```
for i in range(1, 6):
```

```
    if i % 2 == 0:
```

```
        continue
```

```
    print(i)
```

Output: 1 3 5 (even numbers skipped)

- `pass`: placeholder where code is required syntactically but not yet implemented.

```
def future_function():
```

```
    pass # will implement later
```

(behavior, use-cases, pitfalls):

1. `break`: exits nearest loop entirely; can be used in `for` and `while`.
2. Use of `break`: search until found, then stop (early exit improves efficiency).
3. Pitfall for `break`: may leave loop invariants or cleanup undone; use carefully.
4. `continue`: stops current iteration; control flows to loop's next cycle.
5. Use of `continue`: skip processing when a condition invalidates current iteration.
6. Pitfall for `continue`: can cause infinite loops if loop control is not updated correctly.
7. `pass`: no operation; useful as placeholder, for empty classes, functions, or `if` blocks.
8. Readability: `pass` signals intention to implement later; `break/continue` change flow and must be documented.
9. Nested loops: `break` only exits innermost loop. To exit outer loops, use flags or exceptions.
10. Example combination: `for ...: if condition: break; elif other: continue` — both can be used together to shape loop behavior.

14. What is a nested loop? Provide a suitable example.

A nested loop is a loop inside another loop; the inner loop completes all of its iterations for every single iteration of the outer loop. Nested loops are commonly used for processing multi-dimensional data structures (like matrices), generating pairs, or building complex iteration patterns. They increase time complexity multiplicatively (e.g., two nested loops over $n \rightarrow O(n^2)$).

(definition + example + caveats):

1. Definition: loop A contains loop B inside it; B runs completely for each iteration of A.

2. Use-cases: 2D arrays, matrix algorithms, combinatorics (pairs/triples), nested menus.

3. Example (Python):

```
for i in range(3):      # outer loop
    for j in range(2):  # inner loop
        print(f'i={i}, j={j}')
```

4. Output: i=0,j=0; i=0,j=1; i=1,j=0; i=1,j=1; i=2,j=0; i=2,j=1.

5. Complexity: $\text{outer_size} * \text{inner_size}$; watch for performance issues.

6. Nesting depth: deeper nesting is valid but reduces readability and performance; consider refactoring.

15. Explain for loop with syntax and an example showing repetition.

The for loop iterates over a sequence (like list, tuple, string, range, or any iterable), executing the loop body once per element. Unlike languages that index loops numerically, Python's for iterates directly over items, which reduces indexing errors and improves readability. The repetition executes as many times as the iterable provides elements.

Syntax and example:

for item in iterable:

body using item

Example (repeat printing numbers):

```
for i in range(1, 6): # iterates 1,2,3,4,5
    print("Count:", i)
```

(features + usage patterns):

1. Iterates over items of iterable (list, tuple, string, dict keys, range()).
 2. range(start, stop, step) is commonly used for numeric repetition.
 3. No explicit index needed; enumerate() yields index and value when needed.
 4. Can be combined with else: runs after loop completes normally (no break).
 5. Supports break and continue for flow control.
 6. Example repetition: for i in range(5): repeats body 5 times.
 7. Performance: simple and readable; use comprehensions for mapping/filtering.
 8. Pitfall: modifying a list while iterating over it can cause unexpected behavior.
-

16. Discuss selection statements (if, elif, nested if, match) with examples.

Selection statements control program execution based on conditions. `if` checks a condition and executes a block if true. `elif` and `else` extend this to multiple alternatives. Nested `if` places an `if` inside another to express hierarchical decisions. `match` (pattern matching, Python 3.10+) allows matching complex structures or literal values in a clear, declarative style. Each selection mechanism is a fundamental building block for decision-making logic in programs.

Examples:

- `if / else:`

```
x = 10
```

```
if x > 0:
```

```
    print("Positive")
```

```
else:
```

```
    print("Non-positive")
```

- `if / elif / else:`

```
n = 0
```

```
if n > 0:
```

```
    print("Positive")
```

```
elif n == 0:
```

```
    print("Zero")
```

```
else:
```

```
    print("Negative")
```

- Nested `if:`

```
age = 20
```

```
if age >= 18:
```

```
    if age >= 21:
```

```
        print("Adult and can drink in some countries")
```

```
    else:
```

```
        print("Adult but under 21")
```

```
else:
```

```
    print("Minor")
```

- `match` (Python 3.10+) basic usage:

```
command = ("move", 10, 20)
```

```
match command:
```

```
    case ("move", x, y):
```

```
        print(f"Move to {x},{y}")
```

```
    case ("stop",):
```

```
        print("Stop")
```

```
    case _:
```

```
        print("Unknown command")
```

(capabilities and best practices):

1. if: single condition branch.
2. elif: add multiple mutual alternatives; only first true branch runs.
3. else: fallback when none of conditions match.
4. Nested if: supports hierarchical decision making but can reduce readability.
5. Use match for pattern matching: clearer when matching against structured data or many literal cases.
6. Order matters: earlier conditions take precedence.
7. Boolean expressions: combine conditions with and, or, not.
8. Readable conditions: prefer descriptive predicates (use variables like `is_valid`) for clarity.
9. Avoid deep nesting: use functions or early return/continue to flatten logic.
10. Error handling: selection statements should consider edge/invalid inputs.

17. What is a user-defined function? Why do we need functions?

A user-defined function is a block of code written by the programmer that performs a specific task and can be called by name from other parts of the program. Functions encapsulate logic, promoting reuse and abstraction. By dividing programs into functions you create modular, testable units that reduce duplication and make code easier to read, maintain, and debug.

(definition + benefits):

1. Definition: programmer-created callable block defined using `def` (or `lambda` for small anonymous functions).
2. Reusability: write once, call many times.
3. Modularity: separates concerns into named units.

4. Abstraction: hides implementation details behind a clean interface.
 5. Testing: smaller functions are easier to test and verify.
 6. Maintainability: changes confined to function internals reduce ripple effects.
-

18. Explain arguments and parameters with a suitable example.

Parameters are the variables listed in a function's definition; arguments are the actual values passed when the function is called. Python supports positional arguments, keyword arguments, default parameter values, variable-length argument lists (*args, **kwargs), and keyword-only arguments, enabling flexible function interfaces.

Example (illustrating types):

```
def greet(name, msg="Hello", *titles, polite=False, **extras):
```

```
    # name, msg: parameters (positional, with default)
```

```
    # *titles: captures extra positional args
```

```
    # polite: keyword-only parameter
```

```
    # **extras: captures extra keyword args
```

```
    greeting = f"{msg}, {name}"
```

```
    if titles:
```

```
        greeting += " (" + ", ".join(titles) + ")"
```

```
    if polite:
```

```
        greeting += " — pleased to meet you."
```

```
    print(greeting)
```

```
    if extras:
```

```
        print("Extras:", extras)
```

```
# Call examples (arguments):
```

```
greet("Asha")                # name="Asha"
```

```
greet("Ravi", "Hi", "Dr.", polite=True) # titles=("Dr."), polite=True
```

```
greet("Zoe", country="India")    # extras={'country': 'India'}
```

```
(rules + examples):
```

1. Parameter: placeholder variable in function signature.

2. Argument: concrete value passed during call.
 3. Positional arguments: matched by position.
 4. Keyword arguments: passed as name=value.
 5. Default parameters: provide fallback values if argument omitted.
 6. *args: variable number of positional args captured as tuple.
 7. **kwargs: variable keyword args captured as dict.
 8. Best practice: document parameter expectations and prefer keyword args for clarity.
-

19. Discuss Python's standard library functions including `math()`, `random()`, and `statistics()` modules.

Python's standard library includes many modules offering robust, optimized, and well-tested functionality. The `math` module provides mathematical functions and constants (trigonometry, logarithms, factorial), `random` supports pseudo-random number generation and sampling utilities (shuffling, uniform or normal distributions), and `statistics` offers high-level statistical computations (mean, median, variance). Using these modules avoids reimplementing common logic and ensures numerical correctness and performance.

Representative examples & usage:

- `math` module:

```
import math

print(math.sqrt(16))    # 4.0
print(math.pi)         # 3.141592653589793
print(math.factorial(5)) # 120
```

- `random` module:

```
import random

print(random.random())    # float in [0.0, 1.0)
print(random.randint(1, 6)) # integer in [1,6]

lst = [1,2,3]
random.shuffle(lst)        # in-place shuffle
print(lst)
```

- `statistics` module:

```
import statistics
```

```
data = [2, 5, 3, 8, 7]

print(statistics.mean(data))    # average
print(statistics.median(data))  # median
print(statistics.pstdev(data))  # population standard deviation
```

(features + cautions):

1. math: fast, low-level numeric operations; uses C under the hood for speed.
2. math constants: pi, e, useful for precise formulas.
3. random: pseudo-random; use random.seed() for reproducible results.
4. Random caution: not suitable for cryptographic needs — use secrets module for security.
5. statistics: high-level descriptive stats (mean, median, mode, variance).
6. Large datasets: statistics works but consider streaming or NumPy for performance on big arrays.
7. Interoperability: these modules complement each other (e.g., math for transforms, random for sampling, statistics for analysis).
8. Numerical accuracy: math functions follow IEEE semantics, but floating precision limits apply.
9. Convenience: reduce code size and bugs vs hand-rolled implementations.
10. Extendability: third-party libs (NumPy, SciPy, pandas) build on these for advanced scientific tasks.

20. Define strings. Mention any three string operations.

A string is an immutable sequence of Unicode characters used to represent text. In Python, strings are created using single, double, or triple quotes. String immutability means any operation that appears to modify a string actually returns a new string. Strings are central to input/output, data formatting, and text processing.

(definition + three operations):

1. Definition: sequence of characters, immutable, supports indexing and slicing.
2. Indexing: access characters by position, e.g., s[0].
3. Slicing: extract substrings, e.g., s[1:4].
4. Concatenation: s1 + s2 joins strings.
5. Repetition: s * 3 repeats string three times.
6. Common operations: len(s) (length), s.upper() (uppercase), s.replace("a","b") (replace).

21. Algorithm and flowchart to check if a number is zero, positive, or negative.

Classifying a number as positive, negative, or zero is a basic decision problem. The algorithm compares the number with zero using sequential condition checks and outputs the appropriate class. The checks are mutually exclusive and exhaustive.

Algorithm (stepwise):

1. Start
2. Read number n
3. If $n > 0$ print "Positive"
4. Else if $n == 0$ print "Zero"
5. Else print "Negative"
6. Stop

Flowchart (textual):

[Start]

|

[Input n]

|

[Decision: $n > 0$?

/ \

Yes No

| |

Print "Positive" [Decision: $n == 0$?

/ \

Yes No

| |

Print "Zero" Print "Negative"

\ /

[Stop]

(steps, correctness, edge cases):

1. Input: single numeric value n .

2. Comparison order: test `n>0` first, then `n==0`, else negative.
 3. Exhaustive classification: covers all real number cases.
 4. Complexity: $O(1)$ time, constant space.
 5. Type flexibility: works for integers and floats; watch NaN (not-a-number) in floats — comparisons behave oddly.
 6. Output clarity: descriptive strings help user understand result.
 7. Edge case: for floating NaN, all comparisons are false — special handling required if needed.
 8. Implementations: can map to one if-elif-else block in code.
-

22. Explain different string methods with suitable examples.

Python strings come with many built-in methods that simplify common text manipulation tasks: case conversion, searching, splitting, joining, trimming whitespace, replacing substrings, and formatting. Because strings are immutable, these methods return new strings rather than modifying the original. Below are frequently used methods and examples showing how they behave.

Selected string methods and examples:

1. `str.upper()` / `str.lower()` — change case.
2. `s = "Hello"`
3. `print(s.upper())` # "HELLO"
4. `print(s.lower())` # "hello"
5. `str.strip()`, `lstrip()`, `rstrip()` — remove whitespace or specified characters.
6. `s = " hi "`
7. `print(s.strip())` # "hi"
8. `str.replace(old, new)` — replace occurrences.
9. `s = "apple banana"`
10. `print(s.replace("a", "@"))` # "@pple b@n@n@"
11. `str.split(sep=None)` — split into list by separator.
12. `s = "one,two,three"`
13. `print(s.split(","))` # ['one', 'two', 'three']
14. `sep.join(iterable)` — join list into string using separator.
15. `words = ["one", "two"]`
16. `print("-".join(words))` # "one-two"
17. `str.find(sub)` / `str.index(sub)` — find substring index (index raises `ValueError` if not found).

18. `str.startswith(prefix) / str.endswith(suffix)` — boolean checks for prefix/suffix.
19. `str.format()` and f-strings — formatting variables into strings.
20. `name = "Asha"`
21. `print(f'Hello, {name}!')` # f-string
22. `str.isdigit()`, `str.isalpha()`, `str.isspace()` — tests for character composition.
23. `str.count(sub)` — count occurrences of substring.

(summary & best practices):

1. Immutable results: methods return new strings.
 2. Case methods: `upper()`, `lower()`, `title()`, `capitalize()`.
 3. Whitespace trimming: `strip()` variants for cleaning input.
 4. Search & test: `find()`, `index()`, `in` operator, `startswith()`.
 5. Splitting/joining: `split()` and `join()` for tokenization and reconstruction.
 6. Replacement: `replace()` for simple substitutions.
 7. Validation tests: `isdigit()`, `isalnum()` for input validation.
 8. Formatting options: `format()` and f-strings (preferred for readability).
 9. Performance note: joining large lists is efficient with `join` vs repeated concatenation.
 10. Unicode awareness: methods handle Unicode; be aware of normalization for equivalence in complex scripts.
-

23. Write a program to find the average of three numbers.

To compute the average of three numbers, sum them and divide by 3. Be careful about integer division in languages where `/` may perform integer division; in Python, `/` yields a float result. Validate inputs if necessary.

Program (Python) & sample output:

```
# average of three numbers
```

```
a = float(input("Enter first number: "))
```

```
b = float(input("Enter second number: "))
```

```
c = float(input("Enter third number: "))
```

```
avg = (a + b + c) / 3
```

```
print(f"Average = {avg}")
```

Sample run:

Input: 10, 20, 30 → Output: Average = 20.0

(explain steps and edge cases):

1. Read three values (use floats to accept decimals).
 2. Sum the values $a + b + c$.
 3. Divide by 3 using floating division for accurate result.
 4. Print formatted average.
 5. Edge cases: large numbers might need higher precision (use decimal if needed).
 6. Validation: optionally check for non-numeric input to avoid exceptions.
-

24. Explain if-else ladder with example.

An if-else ladder (if → elif → elif → ... → else) is a chain of conditional checks where each subsequent condition is evaluated only if all previous ones were false. This structure is useful when checking ordered or mutually exclusive conditions and results in readable, linear control flow.

Example (age classification):

```
age = int(input("Enter age: "))
```

```
if age < 13:
```

```
    print("Child")
```

```
elif age < 20:
```

```
    print("Teenager")
```

```
elif age < 60:
```

```
    print("Adult")
```

```
else:
```

```
    print("Senior")
```

(explanation & rules):

1. Sequential evaluation: checks each condition top to bottom.
2. First true condition executes; rest ignored.
3. Use else as fallback if none matched.

4. Order conditions by priority (most specific/highest priority first).
5. Avoid overlapping ranges that could cause incorrect earlier matches.
6. Better readability than nested ifs for linear branching.
7. Example use-cases: grading, classification, multi-option menus.
8. Pitfall: very long ladders can be refactored into mapping/dictionaries.

25. Write a program to check whether a student is pass or fail. If pass, then give grade (Distinction / First Class / Second Class / Pass Class).

This program reads a student's marks (single score or aggregate percentage), checks pass/fail threshold, and assigns a grade based on defined ranges. We'll implement input validation, clear grading logic, and print structured output. Grading thresholds are typical examples — adjust as per specific rules.

Program (Python) with explanation & sample outputs:

Student pass/fail and grade determination

try:

```
marks = float(input("Enter marks (0-100): "))
```

```
if not (0 <= marks <= 100):
```

```
    raise ValueError("Marks must be between 0 and 100.")
```

```
except ValueError as e:
```

```
    print("Invalid input:", e)
```

```
else:
```

```
    if marks < 35:
```

```
        print("Result: FAIL")
```

```
    else:
```

```
        # Passed; determine grade
```

```
        if marks >= 75:
```

```
            grade = "Distinction"
```

```
        elif marks >= 60:
```

```
            grade = "First Class"
```

```
        elif marks >= 50:
```

```
            grade = "Second Class"
```

```
else:
    grade = "Pass Class"

print(f"Result: PASS - Grade: {grade}")
```

Sample runs:

- Input 82 → Result: PASS - Grade: Distinction
- Input 58 → Result: PASS - Grade: Second Class
- Input 30 → Result: FAIL

(logic & design):

1. Input validation: ensure numeric and in 0–100.
 2. Pass threshold: used 35% (adjustable).
 3. Grade ranges: Distinction (≥ 75), First Class (60–74.99), Second (50–59.99), Pass Class (35–49.99).
 4. if-elif-else ladder determines grade in descending order.
 5. Edge cases: boundary marks exactly equal to thresholds handled by \geq .
 6. Error handling: informs user for invalid inputs.
 7. Extensible: incorporate multiple subject aggregation or CGPA mapping.
 8. Usability: clear messages for pass/fail and grade.
 9. Separation of concerns: input, validation, grading, output are logically distinct.
 10. Testing: test boundary values (35, 50, 60, 75) to confirm correct classification.
-

26. Explain implicit and explicit type conversion.

Implicit type conversion (coercion) is performed automatically by the language interpreter when it safely converts one data type to another (e.g., integer to float in arithmetic) without programmer intervention. Explicit type conversion (casting) is when the programmer requests a type change using built-in functions (e.g., `int()`, `float()`, `str()`), necessary when combining types that the interpreter will not convert automatically.

(definition + examples + cautions):

1. Implicit conversion: automatic, e.g., $3 + 2.5 \rightarrow 5.5$ (int promoted to float).
2. Explicit conversion: using functions, e.g., `int("123") → 123`.
3. When needed: explicit conversion required for concatenating string and number.
4. Risk: explicit float→int may truncate decimal part (data loss).

5. Safety: implicit conversions avoid data loss by promoting to wider type.
 6. Best practice: convert explicitly when input types are ambiguous (e.g., reading from input()).
-

27. Explain list methods (append, insert, remove, pop) with examples.

Python lists are mutable sequences with methods to modify their contents. `append` adds an item to the end, `insert` places an item at a specified index, `remove` deletes the first occurrence of a value, and `pop` removes and returns an element (by index or the last element by default). Understanding these allows efficient in-place list manipulation.

Examples & explanation:

```
lst = [1, 2, 3]
```

```
lst.append(4)      # [1,2,3,4]
```

```
lst.insert(1, 1.5) # [1,1.5,2,3,4]
```

```
lst.remove(2)      # removes first '2' -> [1,1.5,3,4]
```

```
val = lst.pop()    # removes last element, val = 4, list becomes [1,1.5,3]
```

```
val2 = lst.pop(1)  # removes element at index 1 -> val2=1.5
```

(behavior + complexity + cautions):

1. `append(x)`: add `x` at the end (amortized $O(1)$).
 2. `insert(i, x)`: insert `x` at index `i` ($O(n)$ for shifting elements).
 3. `remove(x)`: removes first matching `x`, raises `ValueError` if not found ($O(n)$).
 4. `pop([i])`: remove and return element at index `i` (default last), raises `IndexError` if empty.
 5. Mutability: all modify list in place (no new list created).
 6. Use `pop` for stack behavior (LIFO).
 7. Use `append` in loops to build lists efficiently.
 8. Caution: `remove` only removes first occurrence; use list comprehensions to filter all.
-

28. Discuss symbolic representation of flowcharts, their importance, and limitations with a neat diagram.

(Effectively a focused restatement of Q3 with emphasis on symbolic representation and practical pros/cons.) Flowchart symbols encode program elements: ovals for start/stop, rectangles for processing steps, parallelograms for input/output, diamonds for decisions, arrows for control flow,

and connectors to join separated parts. These symbols form a small visual language that communicates execution order and logic. Flowcharts are invaluable in early design and teaching, but for very large or dynamic systems they become unwieldy and may fail to express concurrency or asynchronous behavior clearly.

(symbols, importance, limitations):

1. Oval: Start / Stop.
2. Rectangle: Process / Statement.
3. Parallelogram: Input / Output.
4. Diamond: Decision / Branching (two or more exits).
5. Arrow: Flow of control / sequencing.
6. Connector: link separated flowchart sections.
7. Importance: visual clarity for algorithmic steps and branching.
8. Importance: reduces ambiguity and communicates design across teams.
9. Limitation: poor scalability — charts become complex and unreadable for large systems.
10. Limitation: less suited for concurrency, dynamic behavior, or highly data-driven flows (better modeled with state machines or sequence diagrams).