DAY 1 – Python & Computer Science Foundations
Detailed Notes with Diagrams & Flowcharts

1. How a Computer Executes a Program

At the lowest level, a computer does not understand Python, C, or any programming language. It understands only machine instructions (binary operations). All programs, regardless of language, are ultimately converted into a sequence of simple instructions executed one at a time. This execution is governed by the Fetch–Decode–Execute cycle, which repeats continuously as long as the system runs.

Execution Flow (Conceptual)

Program Counter (PC)	→ Points to next instruction
Fetch	→ Instruction loaded from memory into CPU
Decode	→ Control Unit interprets instruction
Execute	→ ALU / Control logic performs operation
Update PC	→ Move to next instruction

2. Von Neumann Architecture

The Von Neumann architecture is the foundational computer design where both data and program instructions reside in the same memory. This unified memory model allows programs to be stored, modified, and executed dynamically. It is the reason modern operating systems and high-level languages exist.

Von Neumann Logical Layout

CPU	ALU + Control Unit
Registers	Ultra-fast temporary storage
Memory (RAM)	Stores data + instructions
Input Devices	Keyboard, Mouse, Sensors
Output Devices	Monitor, Printer, Network

Von Neumann Bottleneck: Because instructions and data share the same memory path, the CPU can only fetch one at a time, limiting performance.

3. CPU Cache (L1, L2, L3)

CPU cache exists to bridge the massive speed gap between the CPU and RAM. Caches store recently and frequently accessed data closer to the CPU, reducing access latency.

Level	Location	Speed	Size	Shared
L1	Inside core	Fastest	KB	No
L2	Near core	Very fast	KB–MB	Usually No
L3	On chip	Fast	MBs	Yes
RAM	Off chip	Slow	GBs	Yes

4. Big-O Notation (Growth Visualization)

Big-O notation describes how an algorithm's time or space requirements grow as input size increases. It ignores constants and focuses purely on scalability.

Complexity	Growth Meaning	Example
O(1)	Constant work	Index access
O(n)	Linear growth	Single loop
$O(n^2)$	Quadratic growth	Nested loops

5. Python List Internals & Time Complexity

Python lists are implemented as dynamic arrays. This allows fast indexing but makes insertions and deletions at the beginning or middle expensive due to shifting elements.

Operation	Time Complexity	Reason
lst[i]	O(1)	Direct index calculation
append()	O(1) amortized	Extra capacity
pop()	O(1)	Remove from end
insert(0,x)	O(n)	Shift elements
x in lst	O(n)	Linear search

6. Python Memory Model (Stack vs Heap)

Python uses stack memory for function call frames and heap memory for objects. Variables are references to objects, not containers of values. This model explains mutability, aliasing, and object lifetime.

7. Mutability vs Immutability

Mutable objects can change their internal state without changing identity. Immutable objects cannot be altered once created and require new object creation for any change. Immutability enables safe sharing, hashing, and caching.

8. Identity vs Equality (is vs ==)

The 'is' operator checks whether two names refer to the same object, while '==' checks whether two objects are equal in value. Using 'is' for value comparison is a logical error.

9. Integer Caching & String Interning

CPython caches small integers and interns some strings as an optimization. These behaviors affect identity comparisons but must never be relied upon in program logic.

10. Practical Python System Utilities

Libraries like psutil allow Python programs to monitor system resources such as CPU and memory usage. Formatted output using f-strings improves readability and precision in applications.

Mentor Insight: Understanding execution, memory, and growth transforms Python from a scripting language into a predictable engineering tool.