Lecture 1: Introduction to Computing & Python

COMP101

October 17, 2025

Course Description

This course introduces:

- The fundamental principles of computing and programming.

Course Description

This course introduces:

- The fundamental principles of computing and programming.
- Problem–solving skills using Python.

Course Description

This course introduces:

- The fundamental principles of computing and programming.
- Problem–solving skills using Python.
- Basic data structures, algorithms, and an overview of computer systems.

Course Learning Outcomes

By the end of the course, **You** will be able to:

- Write, test, and debug programs in Python.

Course Learning Outcomes

By the end of the course, **You** will be able to:

- Write, test, and debug programs in Python.
- Apply basic data structures such as lists, dictionaries, and sets to solve computational problems.

Course Learning Outcomes

By the end of the course, **You** will be able to:

- Write, test, and debug programs in Python.
- Apply basic data structures such as lists, dictionaries, and sets to solve computational problems.
- Understand and implement algorithms for sorting, searching, and recursion.

Course Learning Outcomes

By the end of the course, **You** will be able to:

- Write, test, and debug programs in Python.
- Apply basic data structures such as lists, dictionaries, and sets to solve computational problems.
- Understand and implement algorithms for sorting, searching, and recursion.
- Explain the fundamentals of computer systems, including hardware and software components.

Course Learning Outcomes

By the end of the course, **You** will be able to:

- Write, test, and debug programs in Python.
- Apply basic data structures such as lists, dictionaries, and sets to solve computational problems.
- Understand and implement algorithms for sorting, searching, and recursion.
- Explain the fundamentals of computer systems, including hardware and software components.
- Develop problem–solving skills and computational thinking techniques applicable to various disciplines.

Course Materials

Primary reference:

- *Python Programming: An Introduction to Computer Science* (3rd Ed.), John Zelle.

Assessment and Grading

- **Programming Assignments (30%):** Weekly exercises to practice coding and problem solving in Python.

Assessment and Grading

- **Programming Assignments (30%):** Weekly exercises to practice coding and problem solving in Python.
- **Midterm Exam (30%):** Assess understanding of programming basics, control structures, and data structures.

Assessment and Grading

- **Programming Assignments (30%):** Weekly exercises to practice coding and problem solving in Python.
- **Midterm Exam (30%):** Assess understanding of programming basics, control structures, and data structures.
- **Final Project (20%):** Programming project applying course concepts in a practical setting.

Assessment and Grading

- **Programming Assignments (30%):** Weekly exercises to practice coding and problem solving in Python.
- **Midterm Exam (30%):** Assess understanding of programming basics, control structures, and data structures.
- **Final Project (20%):** Programming project applying course concepts in a practical setting.
- **Final Exam (20%):** Comprehensive exam covering course material, including algorithms and basic OOP.

Today

- What Is a Computer?

Today

- What Is a Computer?
- What Is Computer Science?

Today

- What Is a Computer?
- What Is Computer Science?
- What Is Programming?

What is a computer?

What is a computer?

- Why can one device do so many different tasks?

What is a computer?

- Why can one device do so many different tasks?
- Big idea: *stored information* $+$ *changeable program* $\Rightarrow$ many behaviors.

Working definition

Modern computer

A machine that **stores and manipulates information** under the control of a **changeable program**.

Working definition

Modern computer

A machine that **stores and manipulates information** under the control of a **changeable program**.

- Input $\rightarrow$ processing $\rightarrow$ output.

Working definition

Modern computer

A machine that **stores and manipulates information** under the control of a **changeable program**.

- Input $\rightarrow$ processing $\rightarrow$ output.
- Examples of information: numbers, text, images...

Not just any machine

- Is a calculator a computer? what about a washing machine?

Not just any machine

- Is a calculator a computer? what about a washing machine?
- Calculators and washing machines *manipulate information*, but are built for one fixed task.

Not just any machine

- Is a calculator a computer? what about a washing machine?
- Calculators and washing machines *manipulate information*, but are built for one fixed task.
- Missing ingredient: **changeable program**.

Programs: the lever

- A program is a **detailed, step-by-step** set of instructions.

Programs: the lever

- A program is a **detailed, step-by-step** set of instructions.
- The hardware stays the same; the program changes.

Programs: the lever

- A program is a **detailed, step-by-step** set of instructions.
- The hardware stays the same; the program changes.
- Word processor $\rightarrow$ budget tool $\rightarrow$ game: only the program switched.

Universality (why this matters)

- Across desktops, phones, and many theoretical models: **same computational power** (given enough time/memory).

Universality (why this matters)

- Across desktops, phones, and many theoretical models: **same computational power** (given enough time/memory).
- With suitable programming, one machine can emulate another.

Universality (why this matters)

- Across desktops, phones, and many theoretical models: **same computational power** (given enough time/memory).
- With suitable programming, one machine can emulate another.
- Practical takeaway: learn to precisely describe the task; the machine can do it.

Software VS Hardware

Software rules the hardware

- Hardware is inert without software: programs **define** behavior.

Software rules the hardware

- Hardware is inert without software: programs **define** behavior.
- Same machine, many roles: change the program, change the task.

Software rules the hardware

- Hardware is inert without software: programs **define** behavior.
- Same machine, many roles: change the program, change the task.
- Your leverage this semester: learn to express processes precisely.

What is programming?

- Creating precise, step–by–step instructions that transform data.

What is programming?

- Creating precise, step–by–step instructions that transform data.
- Balancing two views: **big picture** (decomposition) and **details** (edge cases).

What is programming?

- Creating precise, step–by–step instructions that transform data.
- Balancing two views: **big picture** (decomposition) and **details** (edge cases).
- A craft: iterative, testable, readable; not magic.

Who can learn it?

- Talent helps; **practice** matters more.

Who can learn it?

- Talent helps; **practice** matters more.
- Virtually anyone can become productive with patience and effort.

Who can learn it?

- Talent helps; **practice** matters more.
- Virtually anyone can become productive with patience and effort.
- Mindset: break complex systems into understandable subsystems.

Why learn programming?

- **Control:** stop being a passenger; automate and extend tools.

Why learn programming?

- **Control:** stop being a passenger; automate and extend tools.
- **Computational literacy:** understand strengths & limits of computing.

Why learn programming?

- **Control:** stop being a passenger; automate and extend tools.
- **Computational literacy:** understand strengths & limits of computing.
- **Problem–solving:** analysis, abstraction, decomposition.

Why learn programming?

- **Control:** stop being a passenger; automate and extend tools.
- **Computational literacy:** understand strengths & limits of computing.
- **Problem–solving:** analysis, abstraction, decomposition.
- **Creativity & fun:** build useful, even beautiful, things.

What Is Computer Science?

Not the study of computers

- Computers are to CS what telescopes are to astronomy (tool, not the subject).

Not the study of computers

- Computers are to CS what telescopes are to astronomy (tool, not the subject).
- Core question: **What processes can we describe and execute?**

Not the study of computers

- Computers are to CS what telescopes are to astronomy (tool, not the subject).
- Core question: **What processes can we describe and execute?**
- In short: **What can be computed?**

Three ways we study computation

- **Design** — create an algorithm (a precise recipe) that solves the problem.

Three ways we study computation

- **Design** — create an algorithm (a precise recipe) that solves the problem.
- **Analysis** — prove properties: correctness, cost, limits (unsolvable / intractable).

Three ways we study computation

- **Design** — create an algorithm (a precise recipe) that solves the problem.
- **Analysis** — prove properties: correctness, cost, limits (unsolvable / intractable).
- **Experimentation** — build and measure systems to validate behavior in practice.

Design: algorithms

- Algorithm = finite, unambiguous, effective sequence of steps.

Design: algorithms

- Algorithm $=$ finite, unambiguous, effective sequence of steps.
- Design answers "*there exists a method*" (positive evidence of computability).

Design: algorithms

- Algorithm $=$ finite, unambiguous, effective sequence of steps.
- Design answers "*there exists a method*" (positive evidence of computability).
- Failure to find an algorithm $\neq$ proof of impossibility.

Analysis: limits & costs

- **Unsolvable** problems: no algorithm can exist (e.g., halting problem).

Analysis: limits & costs

- **Unsolvable** problems: no algorithm can exist (e.g., halting problem).
- **Intractable** problems: algorithms exist but are impractical (time/memory blow up).

Analysis: limits & costs

- **Unsolvable** problems: no algorithm can exist (e.g., halting problem).
- **Intractable** problems: algorithms exist but are impractical (time/memory blow up).
- We study correctness, complexity, lower bounds, reductions.

Experimentation: systems in the wild

- Some problems are too complex/ill-defined for pure analysis.

Experimentation: systems in the wild

- Some problems are too complex/ill-defined for pure analysis.
- Build prototypes, run benchmarks, collect empirical evidence.

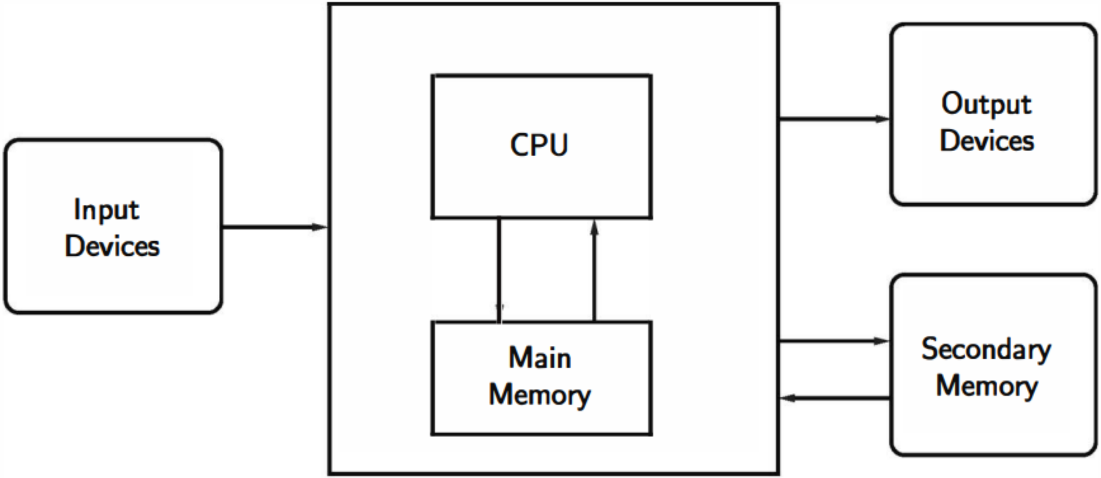
Experimentation: systems in the wild

- Some problems are too complex/ill-defined for pure analysis.
- Build prototypes, run benchmarks, collect empirical evidence.
- Bottom line: does a reliable, working system meet the requirements?

Why peek under the hood?

- Like driving: a little engine knowledge makes the controls make sense.
- Goal today: the **functional** view only (no microarchitecture).
- Payoff: clearer mental model when your programs run.

The big pieces



The big pieces

- **CPU** — arithmetic & logic, control, instruction sequencing.

The big pieces

- **CPU** — arithmetic & logic, control, instruction sequencing.
- **Main memory (RAM)** — fast, volatile; CPU reads/writes here.

The big pieces

- **CPU** — arithmetic & logic, control, instruction sequencing.
- **Main memory (RAM)** — fast, volatile; CPU reads/writes here.
- **Secondary storage** — persistent: HDD/SSD; plus removable media.

The big pieces

- **CPU** — arithmetic & logic, control, instruction sequencing.
- **Main memory (RAM)** — fast, volatile; CPU reads/writes here.
- **Secondary storage** — persistent: HDD/SSD; plus removable media.
- **I/O devices** — keyboards, mice, displays, network, sensors.

Memory: RAM vs storage

- RAM: holds running programs & their data; lost on power-off.

Memory: RAM vs storage

- RAM: holds running programs & their data; lost on power-off.
- Storage: long-term; HDD (magnetic, spinning) vs SSD (flash, electronic).

Memory: RAM vs storage

- RAM: holds running programs & their data; lost on power-off.
- Storage: long-term; HDD (magnetic, spinning) vs SSD (flash, electronic).
- Removable: USB flash drives, optical discs (DVD) for portability.

What happens when you launch a program?

- Program instructions are **loaded** from storage into RAM.

What happens when you launch a program?

- Program instructions are **loaded** from storage into RAM.
- CPU begins executing at the program's entry point.

What happens when you launch a program?

- Program instructions are **loaded** from storage into RAM.
- CPU begins executing at the program's entry point.
- As it runs, the program reads input, updates state, and writes output.

Fetch–decode–execute (the core loop)

- **Fetch**: get the next instruction from RAM.

Fetch–decode–execute (the core loop)

- **Fetch**: get the next instruction from RAM.
- **Decode**: figure out the operation and its operands.

Fetch–decode–execute (the core loop)

- **Fetch**: get the next instruction from RAM.
- **Decode**: figure out the operation and its operands.
- **Execute**: do it (ALU ops, memory load/store, branch, I/O).

Fetch–decode–execute (the core loop)

- **Fetch**: get the next instruction from RAM.
- **Decode**: figure out the operation and its operands.
- **Execute**: do it (ALU ops, memory load/store, branch, I/O).
- Repeat at GHz speeds: billions of simple steps per second.

Why programming languages?

- Programs are sequences of **precise** instructions a computer can execute.

Why programming languages?

- Programs are sequences of **precise** instructions a computer can execute.
- Natural language is ambiguous (*“man in the park with the telescope”*).

Why programming languages?

- Programs are sequences of **precise** instructions a computer can execute.
- Natural language is ambiguous (*"man in the park with the telescope"*).
- We need notations with clear **syntax** (form) and **semantics** (meaning).

Syntax & semantics

- **Syntax**: the grammar/structure of valid programs.

Syntax & semantics

- **Syntax**: the grammar/structure of valid programs.
- **Semantics**: what those programs *mean* (their effect on state).

Syntax & semantics

- **Syntax**: the grammar/structure of valid programs.
- **Semantics**: what those programs *mean* (their effect on state).
- Coding = writing algorithms in a language with precise syntax & semantics.

High-level vs. machine language

High-level (Python)

```
c = a + b
```

Machine-level (conceptual)

- load [2001] -> R1

High-level is for humans; hardware only directly executes machine language.

High-level vs. machine language

High-level (Python)

```
c = a + b
```

Machine-level (conceptual)

- load [2001] -> R1
- load [2002] -> R2

High-level is for humans; hardware only directly executes machine language.

High-level vs. machine language

High-level (Python)

```
c = a + b
```

Machine-level (conceptual)

- load [2001] -> R1
- load [2002] -> R2
- add R1,R2 -> R3

High-level is for humans; hardware only directly executes machine language.

High-level vs. machine language

High-level (Python)

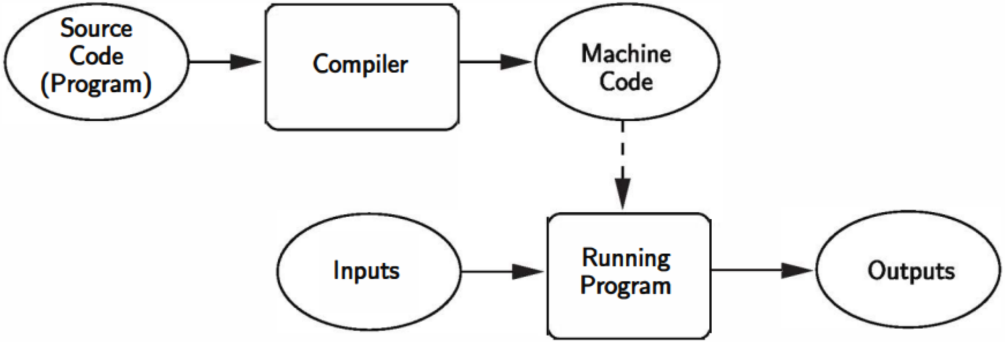
```
c = a + b
```

Machine-level (conceptual)

- load [2001] -> R1
- load [2002] -> R2
- add R1,R2 -> R3
- store R3 -> [2003]

High-level is for humans; hardware only directly executes machine language.

Compilation pipeline



Compilation pipeline

- **Compile**: translate source code once into machine code (executable).

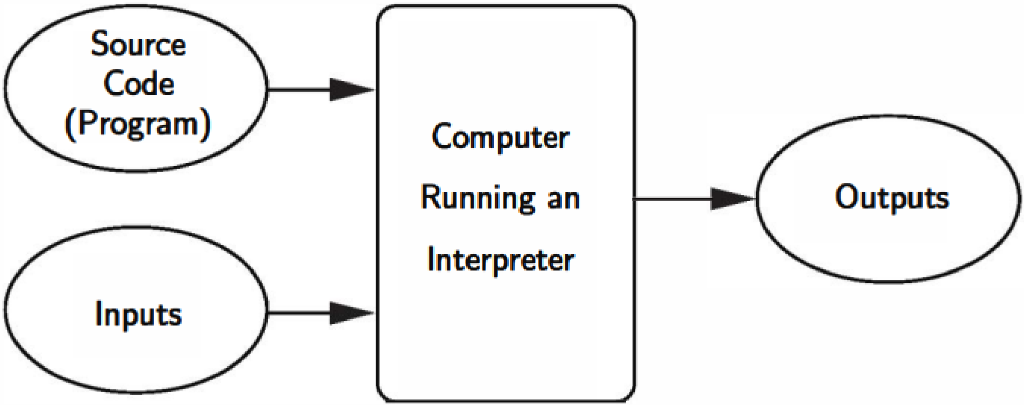
Compilation pipeline

- **Compile**: translate source code once into machine code (executable).
- Re-run without the compiler; typically faster at runtime.

Compilation pipeline

- **Compile**: translate source code once into machine code (executable).
- Re-run without the compiler; typically faster at runtime.
- Good fit for deployment where startup cost is amortized.

Interpretation pipeline



Interpretation pipeline

- **Interpret**: execute source step-by-step via an interpreter/VM.

Interpretation pipeline

- **Interpret**: execute source step-by-step via an interpreter/VM.
- Requires source/interpreter each run; great for interactivity and rapid dev.

Interpretation pipeline

- **Interpret**: execute source step-by-step via an interpreter/VM.
- Requires source/interpreter each run; great for interactivity and rapid dev.
- Python commonly interpreted (often via bytecode + VM).

Compile vs. interpret (at a glance)

	Compiled	**Interpreted**
Translation	One-time	On-the-fly
Run speed	Typically faster	Typically slower
Iteration speed	Slower build cycle	Fast, interactive
Artifacts	Executable/binary	Source + interpreter/VM
Examples	C/C++, Rust	Python, Ruby (also mixed modes)

Many languages use hybrids (e.g., bytecode + JIT).

Portability

- Machine code is tied to a CPU/ISA (e.g., x86 vs ARM).
- High-level source can run on many platforms with a suitable compiler/interpreter.
- Same Python code on different devices with a Python runtime.

Big ideas at a glance

Computer

A universal information-processing machine that can carry out *any* process describable in sufficient detail.

Algorithm

A finite, unambiguous sequence of steps for solving a problem. Algorithms become **programs** that drive the hardware.

What is Computer Science?

Core question

What can be computed?

- **Design**: invent algorithms (positive evidence of computability).

What is Computer Science?

Core question

What can be computed?

- **Design**: invent algorithms (positive evidence of computability).
- **Analysis**: reason about correctness, cost, and limits (unsolvable / intractable).

What is Computer Science?

Core question

What can be computed?

- **Design**: invent algorithms (positive evidence of computability).
- **Analysis**: reason about correctness, cost, and limits (unsolvable / intractable).
- **Experimentation**: build systems and evaluate behavior empirically.

Functional view of a computer

Components

CPU (compute & control), Main Memory (RAM), Secondary Storage (HDD/SSD), I/O (keyboard, screen, network, sensors).

- CPU performs arithmetic/logic; accesses **main memory** directly.

Functional view of a computer

Components

CPU (compute & control), Main Memory (RAM), Secondary Storage (HDD/SSD), I/O (keyboard, screen, network, sensors).

- CPU performs arithmetic/logic; accesses **main memory** directly.
- RAM is fast but **volatile**; storage is persistent (magnetic/flash/optical).

Functional view of a computer

Components

CPU (compute & control), Main Memory (RAM), Secondary Storage (HDD/SSD), I/O (keyboard, screen, network, sensors).

- CPU performs arithmetic/logic; accesses **main memory** directly.
- RAM is fast but **volatile**; storage is persistent (magnetic/flash/optical).
- Programs and data live in memory; I/O moves information in/out.

Programming languages & translation

Definition

A programming language provides precise **syntax** (form) and **semantics** (meaning) for writing algorithms.

- Hardware directly understands only **machine language**.

Programming languages & translation

Definition

A programming language provides precise **syntax** (form) and **semantics** (meaning) for writing algorithms.

- Hardware directly understands only **machine language**.
- Humans write in **high-level** languages (e.g., Python).

Programming languages & translation

Definition

A programming language provides precise **syntax** (form) and **semantics** (meaning) for writing algorithms.

- Hardware directly understands only **machine language**.
- Humans write in **high-level** languages (e.g., Python).
- To run: **compile** to machine code *or* **interpret** the source.

Programming languages & translation

Definition

A programming language provides precise **syntax** (form) and **semantics** (meaning) for writing algorithms.

- Hardware directly understands only **machine language**.
- Humans write in **high-level** languages (e.g., Python).
- To run: **compile** to machine code *or* **interpret** the source.
- High-level languages are generally more **portable**.

What you should remember

- Computers execute precise instructions; *intent doesn't count*.

What you should remember

- Computers execute precise instructions; *intent doesn't count*.
- Algorithms $\rightarrow$ programs $\rightarrow$ software that drives hardware.

What you should remember

- Computers execute precise instructions; *intent doesn't count*.
- Algorithms $\rightarrow$ programs $\rightarrow$ software that drives hardware.
- CS studies computability, cost, and construction of systems.