◇ What Computational Thinking is *Not*

- **Not the same as Computer Science (CS):**
  While CT is related to CS, it is not the same. CS is a full academic discipline focused on theories, systems, and programming. CT is a way of thinking that draws from some CS concepts but is meant to be broadly applicable across different fields.
- **Not just Programming:**
  Teaching programming focuses on writing code and building software. CT, on the other hand, focuses on solving problems in a way that a computer *could* implement, but doesn't necessarily involve writing actual code.
- **Not limited to 'Thinking Like a Computer Scientist':**
  While some have described CT this way, critics argue that this definition makes CT too narrow and overly connected to CS. CT should be seen as a general problem-solving method useful in many contexts—not just computer science.
- **Not Exclusive to Computing:**
  Many of CT's core techniques—like abstraction, modelling, and evaluation—are used in fields such as science, law, education, and language. CT does not "own" these ideas; it simply organizes and applies them in a particular way.

◇ Current Shortcomings of CT

- **Still Immature as a Concept:**
  Although ideas behind CT have existed for decades (e.g., procedural thinking by Papert in the 1980s), the term "computational thinking" only gained major attention around 2006. Only recently have clearer definitions (like Selby's 2013 framework) been proposed.
- **Lack of Long-Term Evidence:**
  CT is not yet widely taught in a consistent way, and there isn't strong data to show that it reliably teaches transferable skills across disciplines. More research and studies are needed.
- **Perceived Intellectual Imperialism:**
  Some critics argue that CT advocates risk sounding territorial—suggesting that any effective method of thinking becomes "computational" once it's used in computing. This can make CT seem elitist or exclusionary if not presented carefully.

☑ Key Takeaway

CT is a powerful and flexible problem-solving framework, drawing from computing but useful in many domains. However, it's important not to confuse it with computer science or programming, and to recognize its current limitations as an emerging educational concept.

Chapter 2

🔍 What is Logical Thinking?

Logical thinking is central to **computational thinking**. It involves forming structured, reasoned arguments based on premises that can be evaluated as either **true** or **false**. It's essential when designing computer-based solutions, because computers follow strict logical rules.

📌 Key Concepts in Logical Thinking

☑ Logical Argument

- A **logical argument** consists of premises that lead to a conclusion.
- A **premise** is a statement with a clear truth value (true or false).
- Logic helps ensure conclusions are valid if the reasoning and premises are correct.

🔄 Deductive vs. Inductive Reasoning

- **Deductive reasoning:** Guarantees the truth of the conclusion if the premises are true.
  Example:
  - All men are mortal.
  - Socrates is a man.
  - Therefore, Socrates is mortal.
- **Inductive reasoning:** The conclusion is *likely* but not guaranteed.
  Example:

- 99 of 100 balls are red.
- Sarah picks one ball.
- Probably, Sarah picked a red ball.

🔢 Boolean Logic

Boolean logic is the foundation of how computers make decisions: statements are **true (1)** or **false (0)**. It helps transform everyday reasoning into something computers can process.

➕ Propositions

- Propositions are statements that are either **true** or **false**, never both.
- They must be **clear** and **unambiguous**.

🔗 Logical Operators

These connect or modify propositions to form compound logical expressions:

Operator Name	Meaning	Symbol	Example (P, Q are propositions)
AND	Both must be true	∧ or AND	P AND Q → true only if both P and Q are true
OR	At least one is true	∨ or OR	P OR Q → true if either P or Q is true
NOT	Negation (flips truth value)	¬ or NOT	NOT P → true if P is false, and vice versa
IMPLIES	If P, then Q (not reversible)	→	P → Q → false only if P is true and Q is false
IFF (IF AND ONLY IF)	Both must have same value	↔	P ↔ Q → true only if both are true or both are false

🧮 Symbolic Logic

Symbolic logic uses **letters** (e.g., P, Q, R) to represent propositions and **symbols** for operators. This eliminates ambiguity and simplifies complex arguments.

Example:
"If at least one square is empty AND no player has a row, THEN the game is still in progress"
becomes:
P AND Q → S

📊 Truth Tables

Truth tables show how combinations of truth values affect a logical expression. For example:

AND (P ∧ Q):

P	Q	P AND Q
True	True	True
True	False	False
False	True	False
False	False	False

Similar tables exist for OR, NOT, IMPLIES, and IFF, helping formalize logic for programming and reasoning.

🔄 Application in Computational Thinking

- Computers automate logical reasoning, so humans must design logic that's sound and valid.
- Understanding **inductive vs deductive** reasoning helps manage certainty in computer outputs.
- **Boolean logic and symbolic representation** translate real-world reasoning into binary, computable forms.

Would you like a diagram or table summarizing all logical operators with examples and symbols?

4o

Hai detto:

more detail

ChatGPT ha detto:

Certainly! Here's a **more detailed summary** of the *Logical Thinking* section, expanding key points with examples and explanations, while keeping it structured and clear:

🔍 What is Logical Thinking in Computational Thinking (CT)?

Logical thinking is the process of using structured reasoning to arrive at valid conclusions. In computational thinking, this skill is essential because computers follow logic exactly—any flaws in your reasoning or input can lead to incorrect results.

🧠 Logic as a Foundation for Reasoning

- **Logic** is not about arguments in conversation, but about **formal arguments**—chains of reasoning that lead to conclusions.
- It helps distinguish **valid** from **invalid** reasoning and is crucial when designing algorithms or computer systems.

📐 Structure of a Logical Argument

1. **Premises** – statements assumed to be true.
2. **Conclusion** – a result derived logically from the premises.
3. **Truth Value** – each premise must be either **true** or **false**, never both.

Example:

```
vbnet
Copia
Premise 1: Socrates is a man.
Premise 2: All men are mortal.
```

```
Conclusion: Therefore, Socrates is mortal.
```

This is a **deductive** argument, meaning if the premises are true, the conclusion *must* be true.

🧭 Deductive vs. Inductive Reasoning

◇ Deductive Reasoning:

- Produces **certain** conclusions.
- Follows strict logical form.
- Breaks if **any** premise is false, or if the reasoning is invalid.

Example of a failed deduction (false premise):

```
pgsql
Copia
1. All dogs are brown. (False)
2. Missie is a dog.
3. Therefore, Missie is brown. (Invalid conclusion)
```

Example of invalid form (logical fallacy):

```
python
Copia
1. All tennis balls are round.
2. The Earth is round.
3. Therefore, the Earth is a tennis ball. (Invalid logic)
```

◇ Inductive Reasoning:

- Produces **probable** conclusions based on observation.
- Common in real-world problems, where knowledge is uncertain or incomplete.

Example:

```
markdown
Copia
```

```
1. A bag has 99 red balls and 1 black ball.
2. 100 people each draw one ball.
3. Sarah is one of them.
4. Therefore, Sarah probably drew a red ball.
```

Though not guaranteed, it's a strong conclusion based on probability.

🧮 Boolean Logic: The Language of Computers

Since computers work in **binary** (true/false or 1/0), we need a form of logic that fits this. Boolean logic simplifies reasoning into clear, binary outcomes.

✔️ Boolean Propositions

- A **proposition** is a statement with a clear **truth value**.
- Must be **unambiguous**: e.g., "It is travelling fast" is vague without context.
- Example: "The light is on" → can be true or false.

🔗 Logical Operators

1. AND (∧ / conjunction)

- Both conditions must be true.
- Example:
  "If the weather is sunny AND I'm on holiday, THEN I will lie in the garden."

P	Q	P AND Q
T	T	T
T	F	F
F	T	F
F	F	F

2. OR (∨ / disjunction)

- At least one condition must be true.

Example:
"If Player 1 OR Player 2 gets a row, the game is over."

P	Q	P OR Q
T	T	T
T	F	T
F	T	T
F	F	F

3. NOT (¬ / negation)

- Reverses the truth value.

Example:
"If a square is NOT occupied, a player can move."

P	NOT P
T	F
F	T

4. IMPLIES (→ / implication)

- If P is true, then Q must also be true.
- False **only** when P is true and Q is false.

Example:
"If it is raining, then the grass is wet."

P	Q	P → Q
T	T	T
T	F	F
F	T	T
F	F	T

Explanation: If it's **not raining**, the grass might still be wet from a sprinkler—so we can't disprove the implication.

5. IF AND ONLY IF (↔ / biconditional)

- True only if both sides are the **same** (both true or both false).

Example:
"All squares are filled IF AND ONLY IF no moves are possible."

P	Q	P ↔ Q
T	T	T
T	F	F
F	T	F
F	F	T

✍️ Symbolic Logic

To eliminate ambiguity, logic uses **symbols** instead of language:

Example in symbols:

- Let:
  - P = One square is empty
  - Q = No player has a row
  - S = Game is still in progress

Then:
P ∧ Q → S
(If P AND Q, then S)

📊 Truth Tables and Visuals

Truth tables list all combinations of propositions and their outcomes. These are critical tools in logic and programming.

Example of an AND Truth Table:

P	Q	P ∧ Q
T	T	T
T	F	F

F	T	F
F	F	F

Venn Diagrams

- Used to visually represent logical operators like AND, OR, and NOT.

🔑 Key Takeaways

- Logical thinking is foundational to computational thinking.
- Computers follow exact logic—errors in reasoning lead to bad outcomes ("garbage in, garbage out").
- Boolean logic and symbolic notation help us reason clearly and communicate precisely with machines.
- Understanding the difference between **deductive** (certain) and **inductive** (probable) reasoning is crucial when interpreting computer output.

🧠 What Is Algorithmic Thinking?

Algorithmic thinking is the mental process of solving problems by breaking them into **step-by-step instructions** that can be executed **precisely** by a computer or another person. It is foundational in computer science and computational thinking.

📘 Definition of an Algorithm

An **algorithm** is:

"A **finite**, **well-defined sequence** of **unambiguous steps** that takes an input and produces an output, typically used to solve a problem."

Key terms:

- **Finite** – must end after a certain number of steps.
- **Well-defined** – no vague instructions.
- **Unambiguous** – every step is clear, only interpretable in one way.

- **Input/output** – it takes in data, processes it, and returns a result.

🔍 Analogy: Algorithm vs. Recipe

Recipe Step	Algorithm Step
“Preheat oven to 180°C”	Set temperature = 180
“Add 2 eggs and mix”	input(eggs); mix(eggs)
“Bake for 30 mins”	wait(30 minutes)

Just like a recipe, an algorithm needs:

- **Exact instructions**
- **Clear order**
- **No guessing**

🧱 Core Properties of Algorithms

1. Step-by-step instructions

Each action must be a **simple**, **independent step** (e.g., "Add 1", "Multiply by 2").

2. Definiteness

Every step must have **only one meaning**. For example, "stir for a while" is **too vague**.

✅ "Stir for 2 minutes"
❌ "Stir until it looks right"

3. Sequencing

Steps must happen in a **specific order**, or the result may be wrong.

Example:

- **Correct**: Chop onions → Fry onions
- **Incorrect**: Fry onions → Chop onions 😅

🧮 State in Algorithms

State = the current values of all the variables at a given point in time.

Example:
Let's say you're counting how many apples are in a basket:

```
python
Copia
apples = 0
for apple in basket:
    apples += 1
```

The **state** changes with each loop — apples go from 0 to 1, to 2, and so on.

Why it matters:

- Algorithms execute step by step.
- Each step **depends on the state** at that moment.
- There's no "global overview" — only a **snapshot at each point**.

🧮 Variables in Algorithms

Variables are **named memory locations** used to store values during execution. They:

- Track changing values (e.g., a counter, a score, user input)
- Are essential for loops, conditionals, and tracking state.

🧠 Think of variables as the **scratchpad** computers use to "remember" things.

🔁 Controlling Algorithm Flow

There are **two primary ways** to control the flow of execution in an algorithm:

1. Iteration (Loops)

Iteration = repeating a set of steps multiple times.

Why use it?

- Avoids repeating the same code manually.
- Efficient for repeated tasks (e.g., counting, printing, checking each item).

Example:

```
python
Copia
x = 99
while x > 0:
    print(f"{x} bottles of beer on the wall...")
    x = x - 1
```

Here:

- x is a **variable** controlling the loop.
- The **condition** `x > 0` stops the loop when x reaches 0.

2. Selection (Conditionals)

Selection lets algorithms make **decisions**.

If-then statements evaluate conditions and perform actions based on them.

Example:

```
python
Copia
if temperature > 100:
    print("Boiling!")
else:
    print("Not boiling.")
```

🧪 Pseudocode Example (Tic-Tac-Toe)

```
plaintext
Copia
1. Start game
2. Loop:
3.   Ask player to choose square
4.   If square is free, place symbol
5.   Check for win
6.   If win, end game
7.   If no moves left, it's a draw
8.   Switch player
9. Repeat until win or draw
10. Print "Game over"
```

Used features:

- **Sequence**: Line-by-line execution
- **Selection**: Conditions to check win/draw
- **Iteration**: Loop until the game ends
- **State**: Current board, player, game status

⚠️ Common Mistakes in Algorithmic Thinking ("Gotchas")

1. Assuming the computer has common sense

It doesn't. If you don't **specify** it, it won't do it.

2. Incorrect logic

Example:

```
plaintext
Copia
If surname starts with A and B → Group 1
```

No name starts with A **and** B at the same time. This will never be true.

3. Forgetting the “else”

Example:

```
plaintext
Copia
If score > 50: grade = pass
```

What happens if score = 50? Nothing! Add `else`:

```
plaintext
Copia
Else: grade = fail
```

🔍 Complex Conditions and Precedence

Just like in math, logic follows **rules of precedence**:

Order	Operator
1	Parentheses ( )
2	NOT ¬
3	Comparison < >
4	AND ∧
5	OR ∨

Example Error:

```
plaintext
Copia
If score is not < 51 or > 80 → Pass
```

Looks right, but isn’t! A student with **85** gets a pass (wrong!).

Correct:

```
plaintext
```

```
Copia
If NOT(score < 51 OR score > 80) → Pass
```

This ensures only scores **between 51 and 80** are allowed.

🧠 Summary

Concept	Meaning
Algorithm	Clear steps to solve a problem
Sequence	Order matters; each step builds on the last
Selection	Use conditions to make decisions (if/then/else)
Iteration	Repeat steps using loops
State	Tracks current values during execution
Variables	Store and update information
Precision	Computers need exact instructions—no assumptions or ambiguity allowed

Chapter 3

📘 Getting Started with Problem-Solving

❗ Tip: Don't Panic

- Large problems can feel overwhelming.
- Most large problems are actually **collections of smaller problems**.
- Start by breaking them down instead of jumping into writing code.

🎯 Defining the Problem

Problem-solving starts by moving from an **undesirable** situation (start point) to a **desirable** one (goal).

✔️ What to do:

- **Understand the start state**: What's wrong or inefficient?

- **Define the goal** clearly: What does “done” look like? Use measurable criteria.
- **Avoid premature solutions**: Don’t think about *how* to solve it yet. Focus on *what* needs to be done.
- **Use visuals**: Diagrams and drawings can clarify complex issues.
- **Restate the problem in your own words** to ensure understanding.
- **Identify knowns and unknowns**.

🛠 Devising a Solution

Once the problem is defined:

- Realize there are often **many solutions** (some better than others).
- Focus on finding a **good solution**, not necessarily a perfect one.
- Solutions often involve **trade-offs** (e.g., speed vs. cost).

🤝 Collaboration and Iteration

- **Collaboration** helps by offering new perspectives.
- **Brainstorming** encourages creative, even wild ideas—don’t dismiss them too early.
- **Iterate**: Your first solution probably won’t be the best. Refine over time.

🔍 Decomposition: The Core Strategy

Decomposition means:

Breaking down a big problem into smaller, simpler parts that can be solved independently.

Why is it important?

- Computers often solve large, complex tasks by working on smaller chunks.
- It makes problems **manageable**, **understandable**, and easier to **solve or delegate**.

💡 Real-world examples:

- **Military strategy**: Divide enemy forces and conquer one group at a time.
- **Dissertation writing**: Break "write dissertation" into intro, methods, results, etc.
- **Tree structure**: Problems can be visually structured like a tree of tasks (e.g., Figure 3.3–3.4).

🔁 Recursive Decomposition

Sometimes sub-tasks are still too complex:

- Apply decomposition **again** (recursively).
- Eventually, you reach tasks simple enough to complete directly (called **leaves** in a tree).

🧠 Other Problem-Solving Strategies

1. Critical Thinking

- Challenge your assumptions.
- Ask: *Does this make sense? What if it fails?*

2. Solve a Concrete Instance

- Choose a **specific example** of the problem and try to solve it.
- Helps you understand abstract problems better.

3. Find a Related Problem

- Has something similar been solved before?
- Adapt an existing solution, but don't assume it works perfectly.

4. Work Backwards

- Start from the goal and figure out what steps lead to it.
- Useful when the final state is clear (e.g., planning a route).

🧩 Patterns and Generalisation

Once you have a solution:

Look for **repeating elements** to make your solution **simpler, reusable, and scalable**.

🔎 How to spot patterns:

- Repeated **nouns** = objects (e.g., "circle", "line")
- Repeated **verbs** = actions (e.g., "draw")
- Repeated **adjectives or numbers** = variable properties (e.g., “red”, “radius 30”)

🎨 Example: Drawing a Smiley Face

Original solution:

```
plaintext
Copia
Draw circle (head)
Draw two circles (eyes)
Draw a curve (mouth)
```

→ Generalise into:

- A **Draw(Shape)** operation
- Shapes like **Circle** and **Line**
- Parameters: **position, radius, colour**, etc.

🔄 Types of Generalisation

1. Loops – for repeated patterns

```
plaintext
```

```
Copia
For each (x, y) in coordinates:
   Draw eye at (x, y)
```

2. Subroutines – reusable blocks

```
plaintext
Copia
Subroutine: DrawEye(radius, x, y)
   Draw outer circle
   Draw inner circle
```

Call with different parameters to draw different eyes.

3. Rules – logical relationships

```
plaintext
Copia
If circle is not filled, then use background colour
```

or

```
plaintext
Copia
Inner radius = ½ outer radius
```

Rules help automate and enforce **consistency**.

☑ Summary

Concept	Description
Problem Definition	Focus on *what* needs to be solved, not *how*.
Decomposition	Break down problems into smaller, simpler tasks.
Iteration	Refine your solution over time.
Collaboration	Talk to others for ideas and feedback.

Pattern Recognition	Identify repetition to simplify and generalise.
Generalisation	Combine similar parts into one flexible solution.
Strategies	Include: critical thinking, concrete examples, analogies, working backward.

☑ ABSTRACTION

◇ What is Abstraction?

Abstraction is the process of **simplifying complexity** by focusing on the **relevant details** of a situation and **ignoring the irrelevant** ones in a given context. It helps manage complicated problems by reducing them to their essential components.

Definition: *"A way of expressing an idea in a specific context while at the same time suppressing details irrelevant in that context."*

◇ Why Abstraction Is Important

- It's a **core principle in computational thinking** and computer science.
- **Real-world problems** are too complex to model in full detail; abstraction allows us to make **models or programs** that represent them.
- Computers operate using simplified models of reality.
- Human cognition is limited — we can hold only a few ideas in mind at once. Abstractions help us cope.

◇ Examples of Abstraction

- **Drawing a face**: From lines and circles → to an eye → to a general eye of any size.
- **Metro map**: Shows station order and lines, not exact distances or directions.
- **Email**: We view an email as text, but in reality it's electrical signals or binary data — abstraction hides this.
- **Vehicle rental**: Cars, vans, motorcycles can all be grouped as "vehicles" depending on the level of detail needed.

◇ Key Concepts

- **Levels of abstraction**: You can move up (more general) or down (more detailed).
- **Instantiating**: To use an abstraction, you must eventually give it **concrete details** (e.g., “draw a circle” rather than “draw a shape”).
- **Leaky abstractions**: Sometimes hidden details affect behavior (e.g., worn brakes, cold weather stopping a car engine).

☑ MODELLING

◇ What is a Model?

A model is a **type of abstraction** — a simplified representation of a part of reality. It focuses only on what’s **relevant to the current problem or context**.

◇ Types of Models

1. **Static models**: Show a system at a fixed point in time (e.g., a metro map).
2. **Dynamic models**: Show how things change over time using:
    a. **States**: Conditions at a moment.
    b. **Transitions**: Changes from one state to another.
    c. **Events**: What triggers a transition.
    d. **Actions**: What happens during the transition.

Example: A turnstile that unlocks when you insert a coin, then locks again after you push it.

◇ Elements of Models

- **Entities**: The key components (e.g., student, vehicle).
- **Relationships**: How entities are connected (e.g., a student *has* an address).
- **Properties**: Attributes (e.g., name, weight).
- **Rules**: Constraints (e.g., “closed from 10–15 Nov”).

- **Behaviours**: What an entity can do (e.g., how a metro line slows down in a section).
- **Cardinality**: Number of instances allowed in a relationship (e.g., a student may have up to 3 phone numbers).

◇ Practical Uses of Modelling

1. **Clarifying complex problems**: Like Euler's bridge problem, which was solved using a simplified graph.
2. **Simulating future outcomes**: Like a climate model or an orrery (mechanical solar system).
3. **Designing data structures**: E.g., building a student record system by modelling data fields.
4. **Designing user interaction**:
    a. Define what data users can see/change.
    b. Plan navigation between screens.
    c. Create low-fidelity prototypes (e.g., paper mock-ups).

☑ Guidelines for Good Modelling

- **Purpose**: Ensure the model is useful for thinking or communication.

- **Formality**:
    - Use **informal models** for brainstorming.
    - Use **formal models** (with strict rules and standards) when correctness matters.
- **Accuracy**: All models simplify reality — don't aim for perfection; aim for **usefulness**.
- **Precision**: Computers deal with discrete data; model real-world values appropriately.

"**All models are wrong, but some are useful.**" — George Box

🐞 UNDERSTANDING ERRORS

◇ What is an Error?

An **error** (or **bug**) is any behaviour during execution that **does not match expected behaviour**.

◇ When Do Errors Appear?

- Some bugs are visible early (e.g., typos).
- Others only appear **after the system is partially or fully running**—these require **testing** and **debugging**.

📐 DESIGNING OUT BUGS EARLY

1. Typos and Surface-Level Mistakes

Even during early design (diagrams, pseudocode, etc.), errors like:

- **Spelling mistakes**
- **Wrong words** (e.g., “phase” instead of “phrase”)
- **Incorrect capitalisation or numbering**

...can lead to **misunderstandings or wrong logic** later in development.

2. Grammar & Ambiguity

- Ambiguities like “After inserting a coin, the user can push the turnstile” can be **interpreted incorrectly**.
- Sentences must be **clear and precise**.
- Example of vague language: “After login fails *a number of times*...” — How many times, exactly?

Tip: Maintain a **glossary of terms** and stick to it to avoid using different words for the same concept (e.g., “smartphone,” “handheld device,” “tablet”).

3. Inconsistencies

Statements that **conflict**:

- "After 5 failed logins, report incident."
- "After 3 failed logins, lock account."
  → These can't both be true; **inconsistencies break logic**.

4. Logical & Mathematical Errors

- Wrong logic (e.g., printing the **minimum** instead of the **maximum**).
- **Not handling equal values** in comparisons.
- **Mathematical faults**, like division by zero.

5. Being Exhaustive

Computers are literal. They don't fill in missing steps. You must:

- Account for **every possible input, outcome, or path**.
- Handle all conditional branches: `if`, `else if`, `else`.

"Bugs lurk in corners and congregate at boundaries" – Beizer (1990)

Examples:

- Boundary case: If `score < 50` is OK and `score > 50` is Good, what happens at `score = 50`?
- **Loops**: Ensure loops don't go infinite, can exit early, and skip when needed.

🛡 DEFENSIVE PROGRAMMING

◇ Principle

Before running critical parts of your code, **check conditions first** (called **preconditions**).

Example: Before dividing numbers, check if the divisor is zero.

Real-world analogy: Elevators are designed to stop safely when power is lost — **fail-safe by default**.

◇ Reacting to Failures

The **reaction depends on severity**:

- ❗ Serious: Shut down system (e.g., water leak in a vending machine).
- ⚠️ Minor: Show a warning or error message.

◇ User Input Validation

Always validate input from users:

- Is it the **correct type** (e.g., number)?
- Is it in the **correct range**?
- Is the **format valid** (e.g., phone number, date, email)?

🧪 TESTING

◇ Purpose

Use the actual system to:

- Check if it behaves as intended.
- Uncover hidden bugs.

◇ Two Main Approaches

Method	Description	Pros	Cons

Top-Down	Test the full system (even with placeholders).	Good for finding design flaws early.	May require simulating unfinished parts.
Bottom-Up	Test small parts first, then integrate.	Localises bugs easily.	Doesn't detect high-level integration flaws.

Best practice: **Combine both** — build a basic structure early, then implement and test parts progressively.

◇ Unit Testing

Test **small, individual pieces** of the solution.

Challenge: Units usually rely on other parts of the system.

Solution: Simulate inputs, outputs, and dependencies.

◇ Equivalence Class Testing

Group inputs that should produce similar outputs:

- Test one value from each group instead of all values.

Example: FizzBuzz

- **Fizz**: divisible by 3 → test with 3
- **Buzz**: divisible by 5 → test with 10
- **FizzBuzz**: divisible by both → test with 15 or 30
- **Normal numbers**: test with 7
- **Invalid inputs**: text like "VII" or negative numbers

Also test:

- **Boundary values** (e.g., 1 – valid, 0 – invalid)

🔧 DEBUGGING

◇ General Approach (Sherlock Holmes style)

1. **Observe** the error.
2. **Form a hypothesis** about what might be wrong.
3. **Test your hypothesis**.
4. **Fix the bug**, then re-test.

◇ Techniques

☑ *Use Hunches*

Try your gut feeling first — you know your code. But be ready to **discard wrong assumptions**.

☑ *Apply Occam's Razor*

Simpler explanations are more likely to be correct.

☑ *Divide and Conquer*

Eliminate large parts of the system as possible causes.

☑ *Change One Thing at a Time*

Only modify one element between tests to isolate cause and effect.

☑ *Use Logging*

Insert print statements to:

- Track variable values.
- Confirm which parts of the code are being run.

☑ *Tracing*

Step through the program line by line (manually or with a **debugger**).

☑ *Talk Aloud*

Explaining the problem to someone (or even yourself) often leads to breakthroughs.

⚖ MANAGING BUGS: NOT ALL ARE EQUAL

◇ All software has bugs.

Perfection is impossible. Prioritise which bugs to fix based on:

◇ Two Key Factors:

1. **Severity**: How badly does it affect functionality?
    a. **Critical**: crashes the system
    b. **Major**: breaks key features
    c. **Moderate**: limits functionality
    d. **Minor**: cosmetic
2. **Priority**: How urgently should we fix it?
    a. Depends on:
        i. **Frequency**
        ii. **Feature importance**
        iii. **Effort required to fix**

◇ Matrix: Severity vs Priority

	High Severity	**Low Severity**
High Priority	🔴 *Fix immediately* (e.g., login crash)	🟡 *Fix soon* (e.g., distorted logo on homepage)
Low Priority	🟠 *Maybe fix later* (e.g., rare upload issue)	⚪ *Fix last* (e.g., minor spelling error)

☑ FINAL TIPS

- **Iterate**: Build and test in small steps.
- **Modularize**: Keep parts of your solution **independent** to localise errors.
- **Peer review**: Ask others to review your work.
- **Document tests**: Keep a **test plan** that others can follow.

Chapter 6

🎯 Objectives of Solution Evaluation

By the end of solution development, you should:

- Evaluate the **quality** of your solution.
- Examine it across four main dimensions:
  - **Correctness** – Does it solve the problem?
  - **Efficiency** – How well does it use time and space?
  - **Elegance** – Is it simple yet powerful?
  - **Usability** – Is it effective and pleasant for people to use?
- Understand and manage **trade-offs** between these measures.

☑ 1. Correctness

◇ Key Question:

Does the solution solve the original problem?

◇ Approach:

- **Empirical Testing** (common in industry)
  - Create a test plan aligned with requirements.
  - Test expected outputs using a variety of input types.
- **Mathematical Proof** (used in critical systems or academia)
  - Offers a formal, indisputable guarantee of correctness.

◇ Levels of Confidence in Correctness:

1. No syntax/runtime errors.
2. Passes basic test cases.
3. Passes random or typical cases.
4. Passes difficult or edge cases.
5. Passes all valid input scenarios.
6. Gracefully handles erroneous input.
7. Responds correctly to **any** input.

One failing test means the solution is incorrect.

❁ 2. Efficiency

◇ Key Question:

How well does the solution use resources (time and memory)?

◇ Two Main Metrics:

- **Time complexity** – How fast it runs.
- **Space complexity** – How much memory it uses.

◇ Efficiency Measurement: Big-O Notation

Describes worst-case performance in relation to input size (N):

Class	Notation	Growth Example
Constant	O(1)	Check if number is even
Logarithmic	O(logN)	Binary search
Linear	O(N)	Loop through a list
Quadratic	$O(N^2)$	Nested loops over same list
Factorial	O(N!)	Brute-force solutions

Efficient algorithms are typically **O(logN)**, **O(N)**, or **$O(N^2)$**.
Inefficient ones include **O(N!)**, **O(2^N)**, etc.

◇ Example:

Linear search:

- Best case: O(1) (item is first)
- Average case: O(N/2)
- Worst case: O(N) (item is last or missing)

🎨 3. Elegance

◇ Key Question:

Is the solution both simple and effective?

◇ Characteristics of Elegant Solutions:

- Minimal yet powerful
- Solves multiple problems at once
- Easy to understand (when explained properly)

◇ Examples:

- **Gauss's shortcut** to summing numbers:
  Instead of `1 + 2 + ... + 100`, use `50 × 101 = 5050`
- **Plumbing trap**: Blocks sewage odours using water, no moving parts.

◇ Trade-Off:

- Elegant solutions can be **non-obvious**, requiring **clear comments** and documentation for future users (even yourself).

"Simple can be harder than complex." – *Steve Jobs*

🧑‍💻 4. Usability

◇ Key Question:

Can real users use the solution easily and effectively?

◇ Nielsen's Usability Components:

1. **Learnability** – Easy for first-time users?
2. **Efficiency** – Fast for frequent users?
3. **Memorability** – Easy to re-learn?
4. **Errors** – Few, recoverable, and low-impact?
5. **Satisfaction** – Enjoyable to use?

◇ Evaluation Method:

- **Usability Testing**:
  - Recruit representative users.
  - Give them **pre-defined tasks** (e.g., "Buy a book under £10").
  - Observe, listen, and record outcomes.
  - Follow up with **questionnaires** or interviews.

◇ Optional: Early Feedback with Paper Prototypes

⚖️ Trade-Offs

◇ Definition:

You **can't optimise everything**. Improving one aspect can worsen another.

◇ Common Trade-Offs:

Trade-Off Type	Example
Space vs. Time	Compressed images save space, but take longer to render.
Cleverness vs. Clarity	Non-obvious tricks save time, but hurt readability.

Usability vs. Flexibility	Highly flexible systems may confuse casual users.

◇ Compression Example:

- **Uncompressed image**: Fast rendering (simple logic), uses **more space**.
- **Compressed image**: Smaller file size, **slower rendering** (complex logic).

◇ Making the Right Trade-Off:

Choose based on **context and priority**:

- Optimise **space** if bandwidth is limited.
- Optimise **time** if speed is crucial.
- Optimise **usability** for wide user adoption.

📝 Final Advice

- Always **evaluate** your solution across all 4 quality measures.
- Use **systematic testing**, real user feedback, and clear metrics.
- Be aware of **trade-offs** and make informed, purpose-driven compromises.
- Document everything, especially if your solution uses non-obvious logic.

Chapter 7

💻 First Steps in Python

You can run Python code in two ways:

1. Interpreter Prompt

- Start by typing `python` in your terminal.
- You'll see the `>>>` prompt where you can enter code line by line.

```
python
CopiaModifica
```

```
>>> x = 1 + 2
>>> print(x)
3
```

2. Source File

- Write code in a `.py` file (e.g., `my_code.py`) and run it from the command line:

```
bash
CopiaModifica
> python my_code.py
```

🔢 Basic Data Types

Type	Example	Notes
Number	7, 42, `3.14`	Integers and floats
String	`'Hello'`, `"World"`	Must be in quotes (single or double)
Boolean	`True`, `False`	Logical values
List	`[1, 2, 3]`	Ordered group of values

◇ Variables

You can store values in variables:

```
python
CopiaModifica
the_number = 3
message = 'Hello, World'
done = True
```

◇ Printing Variables

```
python
CopiaModifica
>>> print(the_number)
3
>>> age = 35
```

```
>>> print("I am", age, "years old.")
I am 35 years old.
```

➕ Basic Operations

◇ Arithmetic Operators

Operation	Symbol	Example
Add	+	`x = y + 2`
Subtract	-	`x = y - 2`
Multiply	*	`x = y * 2`
Divide	/	`x = y / 2`
Power	**	`x = y ** 2`
Modulo	%	`x = y % 2`

◇ Comparison Operators

Comparison	Symbol	Result
Equal to	==	`x == 3` → `True`
Less than	<	`3 < 4` → `True`
Greater than	>	`3 > 4` → `False`
Not equal	!=	`3 != 4` → `True`

🧱 Functions

Functions are **named blocks of code** you can run when needed.

◇ Basic Example:

```
python
CopiaModifica
def output_hello():
    print('Hello, World!')

def double_2():
    num = 2 * 2
    return num
```

◇ Calling a Function:

```
python
CopiaModifica
output_hello()
result = double_2()
print(result)  # Outputs: 4
```

◇ Parameters and Return Values:

```
python
CopiaModifica
def output_message(msg):
    print(msg)

def multiply(x, y):
    return x * y

output_message('Hello, World!')
product = multiply(2, 3)
print(product)  # Outputs: 6
```

🔠 Comments

Use # to write a **comment**—Python ignores everything after # on that line.

```
python
CopiaModifica
# This is a comment
x = 5  # This is also a comment
```

📐 Indentation

Python **uses indentation** (spaces or tabs) to define blocks of code. It's **not optional**.

Example:

```
python
CopiaModifica
def greet():
    print("Hello")  # This line is part of the function
print("Outside function")  # This is not part of the function
```

Indentation after colons (`:`) is required in functions, loops, and conditionals.

Chapter 8

📘 1. Python Basics

📄 The Interpreter Prompt

- Start Python with `python` in the terminal.
- You can enter commands interactively:

```
python
CopiaModifica
>>> x = 1 + 2
>>> print(x)  # Output: 3
```

🚀 Running Python Scripts

- Save commands in a `.py` file and run:

```
bash
CopiaModifica
python my_code.py
```

🔡 2. Data Types & Variables

Type	Description	Example
Number	Integer or float	`x = 42, y = 3.14`
String	Sequence of characters	`"Hello"` or `'Hi'`
Boolean	True or False	`done = True`
List	Ordered sequence of values	`[1, 2, 3]`

📝 Assigning & Using Variables

```
python
CopiaModifica
name = 'Alice'
print("Hello", name)
```

➕ 3. Basic Operators

🔢 Arithmetic

Operation	Symbol	Example
Addition	+	`x = a + b`
Subtraction	-	`x = a - b`
Multiplication	*	`x = a * b`
Division	/	`x = a / b`

Exponentiation	`**`	`x = a ** 2`
Modulo	`%`	`x = a % 2`

🧠 Comparisons

```
python
CopiaModifica
print(x == 3)  # Equal
print(x != 4)  # Not equal
print(x > 2)   # Greater than
```

🧩 4. Control Structures

📋 Sequence

- Python executes code line-by-line.

📜 Conditionals

```
python
CopiaModifica
if age > 18:
    print("Adult")
elif age == 18:
    print("Just 18!")
else:
    print("Minor")
```

🌀 Loops

`while` loop:

```
python
CopiaModifica
```

```
while guess != secret:
    guess = int(input("Guess again: "))
```

for loop with range():

```
python
CopiaModifica
for i in range(1, 5):  # 1 to 4
    print(i)
```

🔁 Loop Control

- `break`: exit loop
- `continue`: skip current iteration

🧪 5. Expressions and Logic

- **Expression**: A statement that evaluates to a value.

```
python
CopiaModifica
1 + 2   # evaluates to 3
x > 5   # evaluates to True or False
```

🔗 Logical Operators

```
python
CopiaModifica
a and b
a or b
not a
```

🔧 6. Functions

- Define reusable blocks of code:

```
python
CopiaModifica
def greet(name):
    print("Hello", name)

greet("Sam")  # Output: Hello Sam
```

- Return values:

```
python
CopiaModifica
def square(x):
    return x * x
```

💬 7. Comments

- Use # to add comments:

```
python
CopiaModifica
# This line adds two numbers
x = a + b
```

🧠 8. State & Mutability

- **Variables** store program state.
- Python uses labels pointing to values.

🧊 Immutable vs 🔁 Mutable

Type	Mutable?	Example

Numbers	No	`x = 1; x = x + 1`
Strings	No	`s = "hi"`
Lists	Yes	`lst.append(5)` changes list

🧪 ***Danger of shared references:***

```
python
CopiaModifica
a = [1, 2]
b = a
a.append(3)
print(b)  # Also shows [1, 2, 3]
```

🧱 9. Declarative vs Imperative

Style	Description	Example
Imperative	Step-by-step instructions	Use `for` loops, manual appending
Declarative	Describe what to do, not how	Use `filter()`, `map()`, comprehensions

🔄 List Comprehension

```
python
CopiaModifica
squares = [n ** 2 for n in range(5)]
```

🧠 10. Higher-Order Functions

Functions that accept functions:

```
python
CopiaModifica
def test_numbers(fn, nums):
    return [fn(n) for n in nums]
```

```
def is_even(n):
    return n % 2 == 0

print(test_numbers(is_even, [1, 2, 3]))
```

🔧 11. Functional Primitives

Name	Use	Python Example
`map`	Apply function to each item	`map(len, words)`
`filter`	Keep items that match condition	`filter(is_even, nums)`
`reduce`	Combine items using function (e.g. sum)	`reduce(add, [1,2,3])` from `functools`

🧱 Conclusion & Best Practices

- Use **testing** to validate correctness.
- Prefer **declarative approaches** when clarity matters.
- Understand and control **mutable state**.
- Write **clear, concise, well-commented** code.
- Use **higher-order functions** and **comprehensions** to make code more Pythonic.

Chapter 9

🔍 1. Separation of Concerns (SoC)

📌 What is it?

Separation of concerns is a **software design principle** that advocates splitting a program into distinct sections, each responsible for a **single, well-defined task**.

🎯 Why it matters:

- Reduces **interdependencies**

- Improves **readability** and **maintainability**
- Facilitates **testing** and **reusability**

🧠 In Practice (Example):

Instead of one function doing *everything*:

```
python
CopiaModifica
def draw_smiley_face():
    # Code for canvas setup
    # Code for drawing face
    # Code for drawing eyes
    # Code for drawing mouth
```

Break it into:

```
python
CopiaModifica
def draw_face_outline(): ...
def draw_eyes(): ...
def draw_mouth(): ...
```

Then use:

```
python
CopiaModifica
def draw_smiley_face():
    draw_face_outline()
    draw_eyes()
    draw_mouth()
```

🖼️ 2. Using tkinter for Graphics

🧰 Components:

- **Window**: `tkinter.Tk()` – creates the main GUI window.
- **Canvas**: `tkinter.Canvas()` – area where shapes are drawn.

- **Shapes**:
  - `create_oval(x1, y1, x2, y2, fill='color')` – circle or ellipse.
  - `create_line(x1, y1, x2, y2)` – straight line.
  - `create_arc(...)` – curved lines.

🛠 Startup Code Template:

```
python
CopiaModifica
import tkinter

window = tkinter.Tk()
c = tkinter.Canvas(window, width=200, height=200)
c.pack()
window.mainloop()
```

All drawing commands must come **before** `mainloop()`.

📏 3. Coordinate System in tkinter

- Origin (0, 0) is at the **top-left**.
- X increases **rightward**, Y increases **downward**.
- Example:
  - (10, 10) is **near top-left**
  - (150, 150) is **bottom-right** in a 200x200 canvas

Understanding this is **crucial** for drawing shapes precisely.

🎯 4. Scope: Local vs Global Variables

◇ Global Variables

- Defined outside all functions
- Can be accessed anywhere *but dangerous*
- Example:

python
CopiaModifica
```
count = 0
def increment():
    global count
    count += 1
```

◇ Local Variables

- Exist only within a function
- Preferred for modular, safe code
- Example:

python
CopiaModifica
```
def greet():
    name = "Alice"
    print(name)
```

Best Practice: Avoid `global` unless absolutely necessary.

🧩 5. Parameter Passing (instead of globals)

Instead of relying on shared state (globals), pass values between functions:

python
CopiaModifica
```
def draw_oval(x1, y1, x2, y2, fill_color=None):
    if fill_color:
        c.create_oval(x1, y1, x2, y2, fill=fill_color)
    else:
        c.create_oval(x1, y1, x2, y2)
```

Makes function **reusable**, **testable**, and **self-contained**.

6. Modular Programming

What is a Module?

A **module** is a `.py` file that contains Python code (functions, variables, classes).

How to Use a Module:

```
python
CopiaModifica
import my_module
my_module.my_function()
```

Import Variations:

```
python
CopiaModifica
from my_module import my_function  # Direct use
from my_module import *            # Not recommended
```

Library vs Script:

- **Library**: defines functions, no auto-execution
- **Script**: can be run directly
- Use this in libraries to prevent auto-execution:

```
python
CopiaModifica
if __name__ == "__main__":
    run_my_program()
```

7. Packages and Project Structure

What is a Package?

A **package** is a folder that contains Python modules and a `__init__.py` file.

◇ Why use packages?

- Organizes modules into logical groups
- Avoids clutter and naming conflicts
- Encourages modular design

🗁 Example Folder Structure:

```
markdown
CopiaModifica
emoji/
├── __init__.py
├── drawing/
│   ├── __init__.py
│   └── core_draw.py
├── rendering/
│   ├── __init__.py
│   ├── faces.py
│   ├── weather.py
│   └── vehicles.py
```

To use:

```
python
CopiaModifica
import emoji.drawing.core_draw as cd
cd.draw_line(...)
```

🔐 8. Information Hiding and Interfaces

💡 Interface = Contract

- A module exposes **function signatures**, not internal details.
- Clients of the module use **only what's documented**.

💬 Analogy:

Think of using a TV remote:

- You press "Volume Up"
- You don't care how the internal circuits work

🔧 Implementation Hiding Example:

You can switch from `tkinter` to `ultra_draw` inside `core_draw`, and the rest of the program won't break—as long as `core_draw`'s interface stays the same.

🔄 9. Refactoring for Reuse and Scalability

🧹 Steps:

1. Identify repeated code (e.g. drawing eyes, face).
2. Move it into reusable functions.
3. Move reusable functions into modules.
4. Group related modules into packages.

This results in:

- **Cleaner** code
- **Less duplication**
- **Easier updates**

📋 10. Best Practices Recap

Principle	Description
DRY	Don't Repeat Yourself – avoid duplication
SoC	Separation of Concerns – each function/module does one thing
Local Scope	Minimize use of global variables
Interface Design	Code to contracts (input/output), not internals
Documentation	Every function should be clearly documented
Modularization	Break large problems into small, reusable parts
Package Organization	Use folders and `__init__.py` to structure large projects
Main Guard	Use `if __name__ == '__main__':` in scripts

Chapter 10

Patterns in Programming

✳ What is a Pattern?

A **pattern** is a repeated structure or solution that can be **reused** across problems. Patterns are key to avoiding repetition and improving code clarity.

🔍 Two Key Principles:

1. **Don't Repeat Yourself (DRY)**: Every piece of logic should live in **one place only**.
2. **Find what varies and encapsulate it**: Isolate the changing parts, generalize the rest.

☑ Refactoring Example:

From:

```
python
CopiaModifica
core_draw.draw_oval(60, 60, 80, 80)
core_draw.draw_oval(90, 60, 110, 80)
```

To:

```
python
CopiaModifica
circle_data = [(60, 60, 80, 80), (90, 60, 110, 80)]
for d in circle_data:
    core_draw.draw_oval(*d)
```

And generalized into reusable logic:

```
python
CopiaModifica
```

```
def render_mouthless_face(circle_params):
    for d in circle_params:
        core_draw.draw_oval(*d)
```

🎭 PART 2: Abstraction in Python

◇ What is Abstraction?

Abstraction hides unnecessary detail and lets us focus on **what** a system does, not **how**.

🧱 Python Abstractions You've Already Used:

- **Variables**: Hide memory details
- **Functions**: Hide implementation logic
- **List Comprehensions, Loops**: Simplify common logic
- **Modules/Packages**: Encapsulate reusable logic

🔡 PART 3: Built-in Types (and Their Power)

Type	Purpose	Example
`int`	Whole numbers	`x = 5`
`float`	Real numbers	`x = 3.14`
`bool`	Boolean logic	`ready = True`
`str`	Text	`"hello"`
`list`	Ordered collection	`[1, 2, 3]`
`tuple`	Immutable collection	`('a', 1)`
`set`	Unordered, no duplicates	`{1, 2, 3}`
`dict`	Key-value mapping	`{"age": 30}`

🛠 Built-in methods like `list.append()`, `str.split()`, and `float.is_integer()` provide **ready-made solutions** for many tasks.

🧰 PART 4: Custom Types via Classes

🧪 Creating Your Own Type:

python
CopiaModifica
```
class Car:
    def __init__(self, model, mileage):
        self.model = model
        self.mileage = mileage

    def update_mileage(self, new_mileage):
        if new_mileage >= self.mileage:
            self.mileage = new_mileage
```

🧠 Key Concepts:

- `class` defines a **blueprint**
- `__init__` is the **constructor**
- `self` refers to the **current object**
- Methods = **Operations**
- Attributes = **State**

👯 PART 5: Inheritance & Polymorphism

⬆️ Inheritance:

python
CopiaModifica
```
class Vehicle:
    ...
```

```
class Car(Vehicle):
    ...

class Van(Vehicle):
    ...
```

Avoid repeating shared logic in subclasses. Put it in a **base class** and **inherit** it.

🔁 Polymorphism:

Same method, different behavior:

```
python
CopiaModifica
def find_parking_bay(self):
    print("Looking for a car bay...")

# Can loop over any Vehicle and call this:
for v in vehicles:
    v.find_parking_bay()
```

🧩 PART 6: Composition

Use when relationships are “has-a” rather than “is-a”.

```
python
CopiaModifica
class ParkingSection:
    def __init__(self):
        self.bays = [ParkingBay(), ParkingBay()]
```

Preferred for **flexibility** over inheritance.

🧠 PART 7: Design Patterns

Design patterns = *Named, reusable solutions to common problems.*

Categories:

Type	Example	Purpose
Creational	**Factory**	Manage complex object creation
Structural	**Facade**	Simplify interface to complex system
Behavioral	**Method chaining**	Streamline sequence of method calls

🧱 Example 1: Facade

Simplifies ignition system:

```
python
CopiaModifica
class IgnitionSystem:
    def start_car(self):
        self.battery.draw_power()
        self.coil.send_power()
```

🏭 Example 2: Factory

Centralizes complex construction logic:

```
python
CopiaModifica
class EngineFactory:
    def get_new_battery():
        return Battery(600)
```

⛓️ Example 3: Method Chaining

```
python
CopiaModifica
result = obj.method1().method2().method3()
```

📚 Final Takeaways

Concept	Why It Matters
Patterns	Reduce code duplication, speed up development
Abstraction	Simplifies complexity, clarifies intention
Composition & Inheritance	Two strategies for managing complexity
Custom Types	Organize real-world logic in code
Polymorphism	Write flexible, adaptable code
Design Patterns	Learn from decades of proven best practices