Use a systematic approach to problem solving including the use of decomposition and abstraction

Using a **systematic approach to problem solving** means tackling a problem in an **organized, logical, and step-by-step way** rather than randomly guessing solutions. It involves breaking the problem down, analyzing it carefully, and applying structured methods to reach a solution.

Two key techniques often used in this approach are **decomposition** and **abstraction**:

**1. Decomposition**

* **Definition:** Breaking a complex problem into smaller, more manageable parts or sub-problems.
* **Why it helps:** Smaller parts are easier to understand, solve, and test.
* **Example:**  
  Suppose you want to create a game. Instead of trying to make the whole game at once, you could decompose it into:
  + Character movement
  + Score system
  + Levels or stages
  + User interface  
    Each of these can be tackled individually.

**2. Abstraction**

* **Definition:** Focusing on the **important details** while ignoring irrelevant or unnecessary details.
* **Why it helps:** It simplifies complex problems and helps you see the underlying structure or pattern.
* **Example:**  
  When designing the character movement in the game, you might abstract away things like character color or background scenery and just focus on how the character moves and responds to controls.

**Putting it together**

A **systematic approach using decomposition and abstraction** might look like this:

1. Understand the problem.
2. Break it into smaller parts (**decomposition**).
3. Identify the key elements of each part (**abstraction**).
4. Solve each part in a logical sequence.
5. Combine the solutions to solve the original problem.

**Problem:**

Create a program that manages a small library of books. Users should be able to:

1. Add new books.
2. Search for a book by title.
3. Remove a book.
4. Display all books.

**Step 1: Decomposition (Break it down)**

We break the main problem into smaller, manageable parts:

1. **Book Storage** – How will books be stored? (e.g., a list or database)
2. **Add Book** – Function to add a new book to storage.
3. **Search Book** – Function to search books by title.
4. **Remove Book** – Function to delete a book.
5. **Display Books** – Function to show all books.

**Step 2: Abstraction (Focus on key details)**

We ignore unnecessary details at first, such as:

* Book cover images, publication date, or author biography.
* Fancy user interface design.

We only focus on the **essential features**: title, author, and basic operations.

**Step 3: Solve each part systematically**

**Book Storage:**

library = [] # A simple list to store books

**Add Book:**

def add\_book(title, author):

book = {'title': title, 'author': author}

library.append(book)

**Search Book:**

def search\_book(title):

for book in library:

if book['title'] == title:

return book

return None

**Remove Book:**

def remove\_book(title):

for book in library:

if book['title'] == title:

library.remove(book)

return True

return False

**Display Books:**

def display\_books():

for book in library:

print(f"{book['title']} by {book['author']}")

**Step 4: Combine solutions**

Now all functions can be used together to manage the library:

add\_book("1984", "George Orwell")

add\_book("To Kill a Mockingbird", "Harper Lee")

display\_books()

**Output:**

1984 by George Orwell

To Kill a Mockingbird by Harper Lee

✅ **Key Points in this Worked Example:**

* **Decomposition:** Each part of the library system was treated separately.
* **Abstraction:** Only the essential information (title and author) was considered.
* **Systematic approach:** Each function is logically designed, tested, and combined.

Use abstraction effectively to model selected aspects of the external world in an algorithm or program

**Definition:**

* **Abstraction** in programming means **ignoring unnecessary details** and focusing on the important features of a real-world object or system.
* When you model the external world in an algorithm or program, you **simplify reality** to only include the aspects that are relevant to the problem you are solving.

**Steps to Use Abstraction Effectively**

1. **Identify the real-world system or object** you want to model.
2. **Decide which details are relevant** for your program (these become your “attributes” or “properties”).
3. **Ignore irrelevant details** to keep the model simple and manageable.
4. **Create an algorithm or program** that uses the simplified model to perform tasks or solve problems.

**Worked Example**

**Problem:**  
You want to make a program that manages a simple zoo. Users can see information about animals and their feeding times.

**Step 1: Identify the system**

* Real-world system: a zoo with many animals.

**Step 2: Select relevant aspects (abstraction)**

* Important details: animal name, species, feeding time
* Ignored details: animal weight, exact habitat location, age, color, diet specifics

**Step 3: Model it in code**

# Abstract model of animals in the zoo

class Animal:

def \_\_init\_\_(self, name, species, feeding\_time):

self.name = name

self.species = species

self.feeding\_time = feeding\_time

# Create some animals

lion = Animal("Leo", "Lion", "12:00")

penguin = Animal("Penny", "Penguin", "14:00")

# Store animals in a list

zoo = [lion, penguin]

# Algorithm to display feeding times

for animal in zoo:

print(f"{animal.name} the {animal.species} is fed at {animal.feeding\_time}")

**Output:**

Leo the Lion is fed at 12:00

Penny the Penguin is fed at 14:00

Learners should apply abstraction to remove irrelevant information from a real world scenario and model the simplified version into an algorithm or program. It is advantageous for abstraction to occur before decomposition.

**Scenario:**

You are asked to create a program to **manage a bus timetable system**. Users should be able to see when buses arrive at different stops.

**Step 1: Apply Abstraction (Remove irrelevant information)**

**Real-world system:** A bus network with many buses, drivers, routes, and stops.

**Irrelevant details for our program:**

* Driver names, bus colors, exact GPS location, traffic conditions, bus capacity.

**Relevant details to model:**

* Bus number
* Stop name
* Arrival time

This is **abstraction**: simplifying reality by focusing only on what is needed.

**Step 2: Decomposition (Break it into manageable parts)**

After abstraction, we can decompose the system into smaller tasks:

1. **Store bus timetable data** – A way to keep track of buses, stops, and times.
2. **Display timetable for a specific stop** – Show all buses arriving at a selected stop.
3. **Add a new bus schedule** – Allow adding new bus arrival information.

**Step 3: Model it in a Program (Algorithm)**

# Abstract model of a bus arrival

class Bus:

def \_\_init\_\_(self, bus\_number, stop\_name, arrival\_time):

self.bus\_number = bus\_number

self.stop\_name = stop\_name

self.arrival\_time = arrival\_time

# Decomposed: Store bus timetable

timetable = [

Bus("12A", "Central Station", "09:00"),

Bus("7B", "Central Station", "09:15"),

Bus("12A", "East Park", "09:30")

]

# Decomposed: Function to display timetable for a stop

def display\_timetable(stop\_name):

print(f"Bus timetable for {stop\_name}:")

for bus in timetable:

if bus.stop\_name == stop\_name:

print(f"Bus {bus.bus\_number} arrives at {bus.arrival\_time}")

# Example usage

display\_timetable("Central Station")

**Output:**

Bus timetable for Central Station:

Bus 12A arrives at 09:00

Bus 7B arrives at 09:15

**✅ Why This Demonstrates Abstraction Before Decomposition**

1. **Abstraction:** We removed unnecessary details (drivers, colors, GPS, capacity) and only kept essential information (bus number, stop, arrival time).
2. **Decomposition:** Only after simplifying the system did we break the problem into manageable functions: storing timetable, displaying a timetable, and adding schedules.
3. **Advantage:** The algorithm is simpler, clearer, and easier to implement because abstraction reduces complexity before we decompose it.

Use abstraction effectively to appropriately structure programs into modular parts with clear, well documented interfaces

**Definition:**

* **Abstraction** means hiding unnecessary details and focusing on essential features.
* **Modular programming** means dividing a program into **independent, reusable modules** (functions, classes, or components) with **clear interfaces** for communication between modules.
* By using abstraction, you can define **what each module does** without worrying about **how it does it internally**.

**Why It Helps**

1. Simplifies complex programs.
2. Makes code easier to maintain and test.
3. Encourages reuse of modules in other programs.
4. Improves collaboration—different programmers can work on different modules without conflicts.

**Worked Example**

**Scenario:** A program for a simple **calculator** that can perform addition, subtraction, multiplication, and division.

**Step 1: Use Abstraction**

* Each operation can be abstracted as a **module**.
* Users don’t need to know **how the operation is performed internally**, just that the module can perform it.

**Step 2: Structure Program into Modular Parts**

# Module 1: Addition

def add(a, b):

"""Returns the sum of a and b."""

return a + b

# Module 2: Subtraction

def subtract(a, b):

"""Returns the difference between a and b."""

return a - b

# Module 3: Multiplication

def multiply(a, b):

"""Returns the product of a and b."""

return a \* b

# Module 4: Division

def divide(a, b):

"""Returns the quotient of a divided by b. Handles division by zero."""

if b == 0:

return "Error: Division by zero"

return a / b

# Module 5: User Interface

def calculator():

"""Interface for the calculator program."""

print("Simple Calculator")

x = float(input("Enter first number: "))

y = float(input("Enter second number: "))

print("Select operation: +, -, \*, /")

op = input("Enter operation: ")

if op == "+":

print(add(x, y))

elif op == "-":

print(subtract(x, y))

elif op == "\*":

print(multiply(x, y))

elif op == "/":

print(divide(x, y))

else:

print("Invalid operation")

# Run the calculator

calculator()

**✅ How Abstraction Is Used**

1. Each operation is a **separate module** (add, subtract, multiply, divide).
2. **Internal details** of how the operation works are hidden from the user interface.
3. The **user interface module** only calls the functions—it doesn’t care how they calculate results.
4. **Clear interfaces:** Each function has clear input (parameters) and output (return value).
5. Functions are **well-documented** with docstrings, explaining their purpose without exposing internal logic.

**Activity A**

**Problem:**  
Your school wants to organize a sports day. Students need to create a program that helps manage the event. The program should:

1. Keep track of participants in each sport
2. Record scores for each participant
3. Determine winners for each event
4. Display overall results

**Step 1: Apply Abstraction (Focus on essential details)**

**Essential details to include:**

* Participant name
* Sport/event name
* Score

**Irrelevant details to ignore:**

* Participant height, weight, age
* Exact weather conditions
* Seating arrangements
* Snack preferences

This simplifies the real-world event into a **model that is manageable for the program**.

**Step 2: Apply Decomposition (Break the problem into smaller parts)**

1. **Store participants** – Keep a record of who is competing in each event.
2. **Record scores** – Update scores for participants after each event.
3. **Determine winners** – Find the participant with the highest score in each event.
4. **Display results** – Show scores and winners for each sport.

Breaking it into modules makes the program easier to plan, implement, and test.

**Step 3: Model in a Program (Algorithm)**

# Step 1: Abstract model of a participant

class Participant:

def \_\_init\_\_(self, name, sport):

self.name = name

self.sport = sport

self.score = 0

# Step 2: Store participants

participants = [

Participant("Alice", "100m"),

Participant("Bob", "100m"),

Participant("Charlie", "Long Jump")

]

# Step 3: Record scores

def record\_score(name, score):

for participant in participants:

if participant.name.lower() == name.lower():

participant.score = score

print(f"{participant.name}'s score updated to {score}")

return

print("Participant not found")

# Step 4: Determine winners

def determine\_winner(sport):

sport\_participants = [p for p in participants if p.sport == sport]

if not sport\_participants:

print("No participants in this sport")

return

winner = max(sport\_participants, key=lambda p: p.score)

print(f"The winner of {sport} is {winner.name} with a score of {winner.score}")

# Step 5: Display results

def display\_results():

for p in participants:

print(f"{p.name} - {p.sport} - Score: {p.score}")

# Example usage

record\_score("Alice", 12.3)

record\_score("Bob", 11.8)

record\_score("Charlie", 5.4)

display\_results()

determine\_winner("100m")

determine\_winner("Long Jump")

**How This Demonstrates Systematic Problem Solving**

1. **Abstraction:** Only important details (name, sport, score) are included; unnecessary details are ignored.
2. **Decomposition:** The program is divided into clear modules—storing participants, recording scores, determining winners, displaying results.
3. **Systematic Approach:** Each module can be implemented, tested, and refined independently before combining them into the full program.

**Activity B**

**Problem:**  
A school wants a simple program to monitor the weather in their town. Students need to create a program that can:

1. Record the temperature and rainfall for each day
2. Display the recorded data
3. Identify the hottest and wettest day

**Step 1: Apply Abstraction (Focus on essential details)**

**Real-world system:** Weather includes many factors like temperature, rainfall, wind speed, humidity, air pressure, cloud cover, UV index, etc.

**Essential details for the program:**

* Temperature (°C)
* Rainfall (mm)
* Date

**Irrelevant details (ignored):**

* Wind speed, humidity, UV index, air pressure, sunrise/sunset times

By using **abstraction**, students only focus on the data relevant to the goals of the program.

**Step 2: Model in a Program (Algorithm)**

# Abstract model of a day's weather

class WeatherDay:

def \_\_init\_\_(self, date, temperature, rainfall):

self.date = date

self.temperature = temperature

self.rainfall = rainfall

# Store weather data

week\_weather = [

WeatherDay("2025-09-01", 22, 5),

WeatherDay("2025-09-02", 25, 0),

WeatherDay("2025-09-03", 20, 12)

]

# Display all weather data

def display\_weather():

print("Weather Data:")

for day in week\_weather:

print(f"{day.date} - Temp: {day.temperature}°C, Rainfall: {day.rainfall}mm")

# Find hottest day

def hottest\_day():

hottest = max(week\_weather, key=lambda day: day.temperature)

print(f"Hottest Day: {hottest.date} with {hottest.temperature}°C")

# Find wettest day

def wettest\_day():

wettest = max(week\_weather, key=lambda day: day.rainfall)

print(f"Wettest Day: {wettest.date} with {wettest.rainfall}mm of rain")

# Example usage

display\_weather()

hottest\_day()

wettest\_day()

**Output:**

Weather Data:

2025-09-01 - Temp: 22°C, Rainfall: 5mm

2025-09-02 - Temp: 25°C, Rainfall: 0mm

2025-09-03 - Temp: 20°C, Rainfall: 12mm

Hottest Day: 2025-09-02 with 25°C

Wettest Day: 2025-09-03 with 12mm of rain

**How This Demonstrates Abstraction**

1. **External world modeled:** The real-world weather system is simplified into just **temperature, rainfall, and date**.
2. **Irrelevant details ignored:** Factors like wind, humidity, and air pressure are not modeled because they are not needed for this program.
3. **Algorithm uses simplified model:** The program can display, analyze, and compare weather efficiently without unnecessary complexity.

**Activity C**

**Problem:**  
Students need to create a program to track daily fitness activities. The program should allow users to:

1. Record activities (running, cycling, swimming) and duration in minutes
2. Calculate total calories burned per activity
3. Display a summary of all activities
4. Identify the activity with the most calories burned

**Step 1: Apply Abstraction**

**Real-world system:** Fitness tracking involves many details like heart rate, GPS tracking, calories burned per step, distance, etc.

**Essential details for the program:**

* Activity type (running, cycling, swimming)
* Duration (minutes)
* Calories burned (calculated using simple formula)

**Irrelevant details:**

* Heart rate, speed, distance, elevation, GPS tracking

By abstracting away unnecessary details, the program focuses only on the **core data needed to track fitness progress**.

**Step 2: Structure Program into Modular Parts**

Using **modular programming**, we can split the program into modules with **clear, well-documented interfaces**:

1. **Activity Module** – Stores data for an activity
2. **Record Module** – Adds a new activity to the log
3. **Calories Module** – Calculates calories burned
4. **Summary Module** – Displays all activities
5. **Analysis Module** – Finds activity with most calories burned

**Step 3: Model in a Program**

# Module 1: Activity

class Activity:

"""Represents a fitness activity with type, duration, and calories burned."""

def \_\_init\_\_(self, activity\_type, duration):

self.activity\_type = activity\_type

self.duration = duration

self.calories = self.calculate\_calories()

def calculate\_calories(self):

"""Calculates calories burned based on activity type and duration."""

rates = {"running": 10, "cycling": 8, "swimming": 12} # calories per minute

return self.duration \* rates.get(self.activity\_type, 0)

# Module 2: Record activities

activity\_log = []

def record\_activity(activity\_type, duration):

"""Adds a new activity to the activity log."""

activity = Activity(activity\_type, duration)

activity\_log.append(activity)

print(f"Recorded {activity\_type} for {duration} minutes ({activity.calories} calories)")

# Module 3: Display summary

def display\_summary():

"""Displays all recorded activities."""

print("Activity Summary:")

for act in activity\_log:

print(f"{act.activity\_type.title()} - {act.duration} min - {act.calories} cal")

# Module 4: Find activity with most calories burned

def top\_activity():

"""Finds and displays the activity with the highest calories burned."""

if not activity\_log:

print("No activities recorded")

return

top = max(activity\_log, key=lambda act: act.calories)

print(f"Top Activity: {top.activity\_type.title()} - {top.calories} calories burned")

# Example usage

record\_activity("running", 30)

record\_activity("cycling", 45)

record\_activity("swimming", 20)

display\_summary()

top\_activity()

**Output:**

Recorded running for 30 minutes (300 calories)

Recorded cycling for 45 minutes (360 calories)

Recorded swimming for 20 minutes (240 calories)

Activity Summary:

Running - 30 min - 300 cal

Cycling - 45 min - 360 cal

Swimming - 20 min - 240 cal

Top Activity: Cycling - 360 calories burned

**How This Demonstrates Abstraction and Modular Structure**

1. **Abstraction:** Only essential details are modeled: activity type, duration, and calories. Irrelevant details like heart rate, distance, or GPS are ignored.
2. **Modular Structure:** The program is divided into **independent modules**: Activity class, record\_activity, display\_summary, top\_activity.
3. **Clear Interfaces:**
   * record\_activity(activity\_type, duration) clearly accepts two inputs and adds an activity.
   * display\_summary() outputs a summary without needing internal details.
   * top\_activity() returns the activity with the highest calories burned.
4. **Well-documented:** Each module has a docstring explaining its purpose.

**1. Sports Day Scenario – Algorithm Questions**

**Scenario Recap:** Students track participants in events, record scores, determine winners, and display results.

**Questions:**

1. Write an algorithm to **store participants** in different sports events. Include participant name and sport.
2. Write an algorithm to **update the score** of a participant after an event.
3. Write an algorithm to **determine the winner** of a specific sport by comparing participants’ scores.
4. Write an algorithm to **display all participants** with their sport and current score.
5. Combine your algorithms to **simulate one event** of the sports day, including adding participants, recording scores, displaying results, and announcing winners.

**2. Weather Monitoring Scenario – Algorithm Questions**

**Scenario Recap:** Students record temperature and rainfall for each day, display data, and find hottest/wettest days.

**Questions:**

1. Write an algorithm to **store weather data** for each day, including date, temperature, and rainfall.
2. Write an algorithm to **display all recorded weather data** for the week.
3. Write an algorithm to **find the hottest day** by comparing the temperatures of all recorded days.
4. Write an algorithm to **find the wettest day** by comparing the rainfall of all recorded days.
5. Combine your algorithms to **input daily weather data for a week**, then display the week’s data and the hottest and wettest days.

**3. Fitness Tracker Scenario – Algorithm Questions**

**Scenario Recap:** Students track fitness activities, calculate calories burned, display summaries, and find the top calorie-burning activity.

**Questions:**

1. Write an algorithm to **record a fitness activity**, including type (running, cycling, swimming) and duration.
2. Write an algorithm to **calculate calories burned** based on activity type and duration.
3. Write an algorithm to **display a summary** of all recorded activities, showing activity type, duration, and calories burned.
4. Write an algorithm to **identify the activity with the highest calories burned**.
5. Combine your algorithms to **simulate recording multiple activities**, then display a summary and the top calorie-burning activity.