

# *Computational Thinking and Problem Solving with Python*

*By*

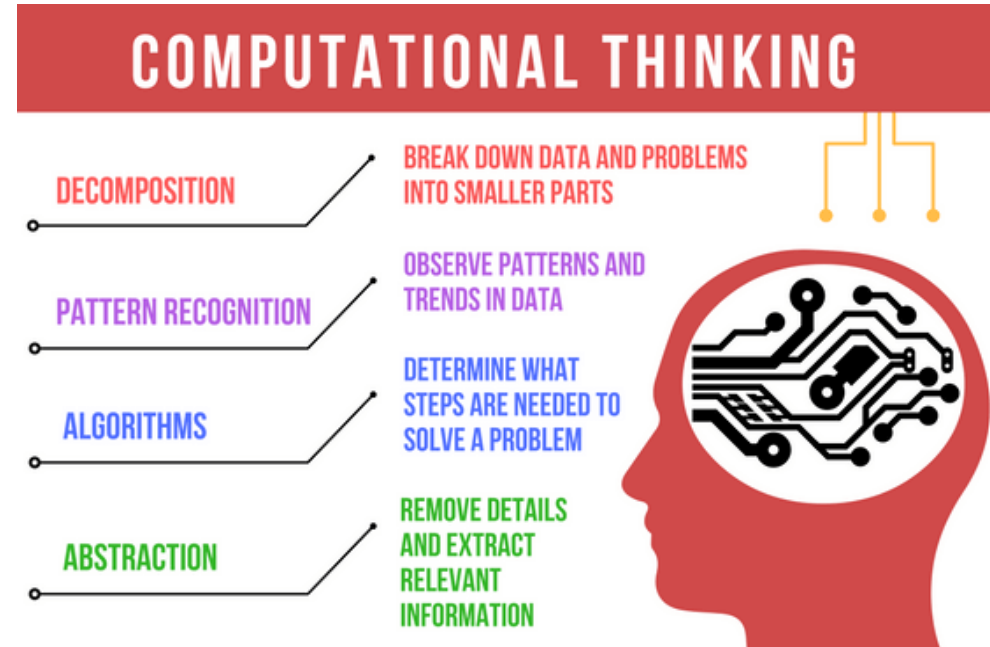
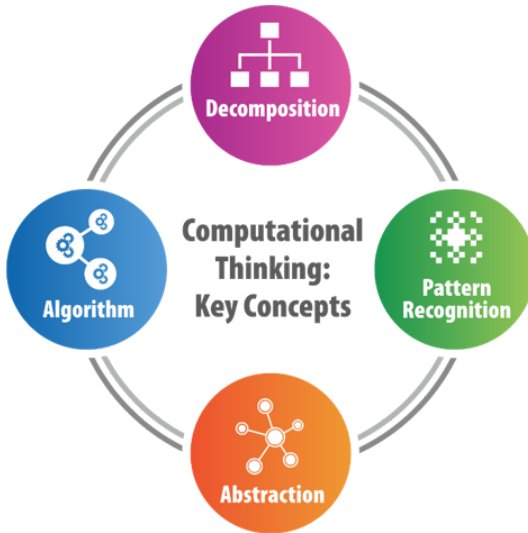
*Rahul Kumar Gupta*

*rahul.gupta@woxsen.edu.in*

# Computational Thinking

- *What is Computational Thinking?*
- *Computational thinking is a systematic approach to solving problems by drawing upon principles from computer science.*
- *It involves breaking down problems into smaller components, recognizing patterns, devising algorithms, and Extracting relevant data.*

# Computational Thinking



# Computational Thinking Steps



## Problem Specification

- **Abstraction:** Removing aspects of a problem that are not needed for its solution.
- **Decomposition:** Breaking a big problem down into smaller, more manageable sub-problems.
- **Pattern Recognition:** Analyzing any kind of repeating elements or sequences in the problem.

## Algorithmic Expression

- **Algorithm Design:** Creating step-by-step instructions for solving the problem.



## Solution Implementation & Evaluation

- **Generalization:** Extending a solution from a particular problem to other kinds of problems.



©Ricky J. Sethi

# Computational Thinking

## Decomposition

Take ideas and problems apart



## Pattern Recognition

Look for similarities or trends



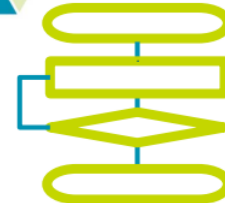
Focus on what's important

## Abstraction



Create step-by-step instructions

## Algorithmic Design



# Logic Building

- *Logic building is a very important part of programming. Whether it's a basic logic of a basic game or advanced logic of complex algorithms. Your logic building skills determine whether you're a good developer or a great one*
- *Programming is all about logic, and the better your logical reasoning skills, the better programmer you become*
- *Sudoku, Chess, etc. games improve logic building*
- *Solve coding challenges, contests, hackathons to improve your logical skills*
- *Understand the basics – data types, functions, loops, algorithms*
- *Practice by solving problems to build conceptual clarity*
- *Collaborate with others*
- *Write pseudo code*
- *Analyze other's code*
- *Break problems down*
- *Practice, practice, practice*

# Problem Solving Techniques

- *Step 1: Understand the problem*
- *Step 2: Create a step-by-step plan for how you'll solve it*
- *Step 3: Carry out the plan and write the actual code*
- *Step 4: Look back and possibly refactor your solution if it could be better*

# Algorithms

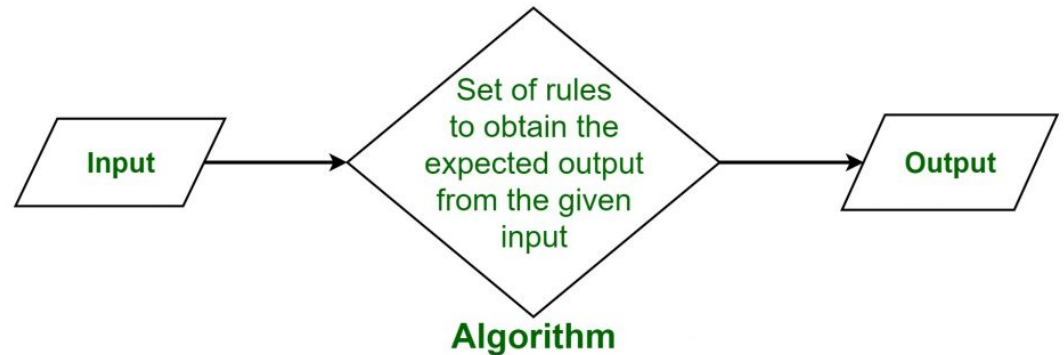
- An algorithm is a finite set of instructions designed to solve a specific problem or perform a particular task.
- It's a well-defined sequence of operations or actions that, when followed in the correct order, leads to a desired outcome.
- Algorithms are used in various fields, including computer science, mathematics, engineering, and everyday life, to outline systematic procedures for solving problems.

# Algorithms

An **algorithm** is a set of step-by-step instructions that describe how to perform a task.

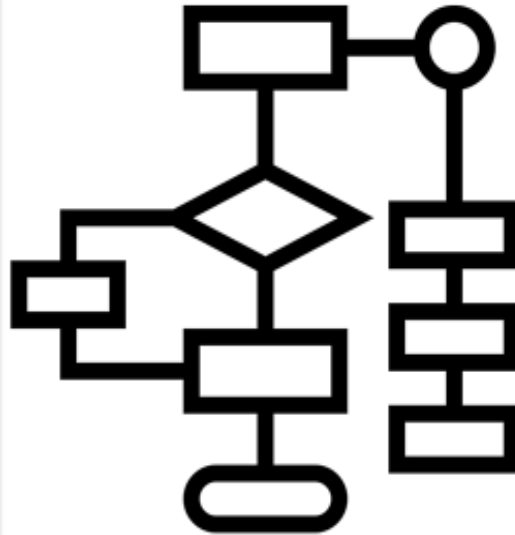


## What is Algorithm?





## WHAT IS **AN ALGORITHM**?



An **algorithm** is a set of step-by-step instructions that describe how to perform a task. It is made up of three basic building blocks: sequencing, selection, and iteration.

An **algorithm** is used for solving a problem or performing a computation. They are widely used throughout all areas of IT.

Some **characteristics of algorithms** are: they have well-defined inputs and outputs, they are clear and unambiguous, and they must terminate after a finite number of steps.

# Key characteristics of algorithms include

1. **Well-Defined:** Algorithms are precisely defined, leaving no ambiguity about the steps to be taken.
2. **Input and Output:** They take input values or data as parameters and produce output or results.
3. **Finite:** Algorithms must have a finite number of steps, meaning they eventually terminate and provide an answer.
4. **Deterministic:** Given the same input, an algorithm will produce the same output every time it's executed.
5. **Effective:** Algorithms are designed to be practical and feasible, using resources like time and memory efficiently.
6. **Problem-Specific:** Each algorithm is tailored to solve a particular problem or perform a specific task.

# Building Blocks of Algorithms:

- *Statements*
- *State*
- *Control flow*
- *Functions*

# Statements

- *Statements are the fundamental units of code that carry out specific actions or operations.*
- *They can be simple or complex, and they form the building blocks for creating algorithms.*
- *In most programming languages, statements end with a semicolon. Some common types of statements include:*
- **Assignment Statements:** *Assign values to variables.*
- **Arithmetic Statements:** *Perform mathematical operations.*
- **Conditional Statements:** *Make decisions based on conditions (if-else statements, switch statements).*
- **Looping Statements:** *Repeat a block of code multiple times (for loops, while loops).*

# State

- *State refers to the snapshot of the program's data at any given moment.*
- *It's essentially the values of variables and data structures that represent the program's current state.*
- *State can change as the program executes statements and progresses through its logic.*

# Control Flow

- Control flow determines the order in which statements are executed in a program.
- Key concepts related to control flow include:
- **Conditional Execution:** Executing specific statements based on conditions (if-else statements).
- **Looping:** Repeatedly executing a block of code while a condition is true (while loops, for loops).
- **Branching:** Choosing between multiple paths based on different conditions.

# Functions

- Functions are blocks of code that can be defined and reused to perform specific tasks. They encapsulate logic, making code more modular, readable, and maintainable. Functions take inputs (parameters) and produce outputs (return values).
- **Function Declaration:** Defining a function and specifying its name, parameters, and behavior.
- **Function Call:** Invoking a function with specific arguments.
- **Return Statement:** Sending a value back from a function to the calling code.

# Example: Sum of Even Numbers

- *# Function to calculate the sum of even numbers in a range*
- *def sum\_of\_even\_numbers(start, end):*
  - *total = 0*
  - *for num in range(start, end + 1):*
    - *if num % 2 == 0:*
      - *total += num*
  - *return total*
- *# Main program*
- *start\_range = 1*
- *end\_range = 10*
- *result = sum\_of\_even\_numbers(start\_range, end\_range)*
- *Print("Sum of even numbers from", start\_range, "to", end\_range, "is", result)*



# Basic Logic Structure in Computer Programming

- *Algorithms can be constructed from basic building blocks namely,*
  - *Sequence*
  - *Selection*
  - *Iteration*
- *Sequence: Executing commands in order.*
- *Selection: Making decisions based on conditions.*
- *Iteration: Repeating actions.*

# Sequence, Selection, Iteration

- *Sequencing: An algorithm is a step-by-step process, and the order of those steps are crucial to ensuring the correctness of an algorithm*
- *Selection: Algorithms can use selection to determine a different set of steps to execute based on a Boolean expression*
- *Iteration: Algorithms often use repetition to execute steps a certain number of times or until a certain condition is met*

# Example Selection

- *Algorithm: Check Even Number*

1. *Start*

2. *Input the number (num) to be checked*

3. *If num is divisible by 2 with no remainder, then it's even*

4. *Otherwise, it's odd*

5. *End*

# Example Sequence

- *Algorithm: Addition of Two Fixed Numbers*

1. *Start*

2. *Initialize the first number (num1) with a specific value*

3. *Initialize the second number (num2) with a specific value*

4. *Add num1 and num2 and store the result in a variable sum*

5. *Output the sum*

6. *End*




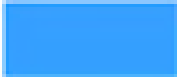

# Example Iteration

- *Algorithm: Print Name 20 Times*
- *Start*
- *Set a variable name to your name*
- *Set a variable count to 0*
- *While count is less than 20, do the following:*
  - *Print name*
  - *Increment count by 1*
- *End*

# Flowchart

- *A flowchart is a graphical representation of the workflow of an algorithm using various conventional figures and arrows.*
- A flowchart helps us write an algorithm more efficiently and faster.
- **Why do we need a flowchart?**
  - To develop a visual representation of the logic of the program.
  - To determine the required constraints.
  - To plan a project.
  - A flowchart is a better way of communicating the logic of the program.

# Flowchart components

Symbol	Details
Start / End 	The oval defines the start and end points of the flow.
Arrows 	Arrows are connectors. Their direction shows the relationships between different elements of the flow.
Input/Output 	A parallelogram represents the input or output of the flow.
Process 	A rectangle represents a process.
Decision 	A diamond is used to mark any decision points.

# Pseudo code

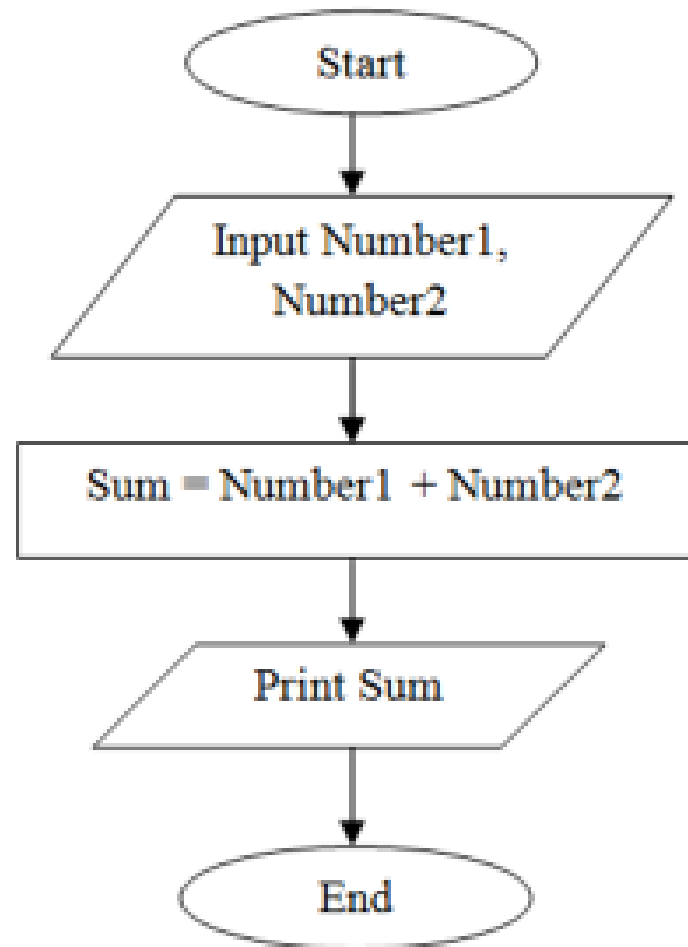
- Pseudocode is a high-level, human-readable description of the steps involved in solving a problem or implementing an algorithm.
- It's not tied to any specific programming language syntax and serves as an intermediary step between plain language and actual code.
- Pseudocode helps programmers plan and communicate their algorithms before writing the actual code.
- Pseudo code is a mixture of English statements, some mathematical notations and selected keywords from a programming language.
- It is one of the tools used to design and develop the solution to a task or problem.
- Pseudo code uses the structural conventions of a programming language, but is intended for human reading rather than machine reading.
- The purpose of using pseudo code is that it is easier for people to understand than conventional programming language code, and that it is an efficient and environment-independent description of the key principles of an algorithm.



# Flowchart & Pseudo code

- **Pseudo code: Addition of Two Numbers**

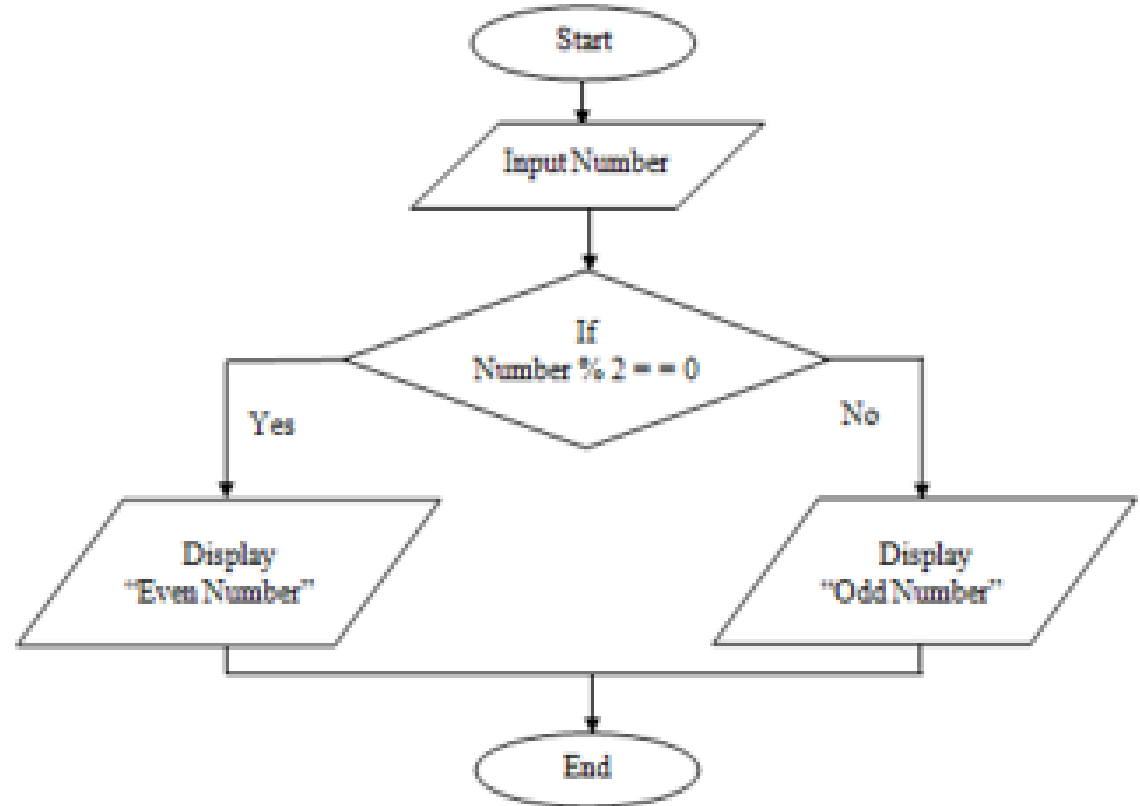
1. Start
2. Number1=10,
3. Number2=20;
4.  $\text{Sum} = \text{Number1} + \text{Number2}$
5. Print sum
6. End



# Flowchart & Pseudo code

- **Pseudo code: Check Odd or Even Number**

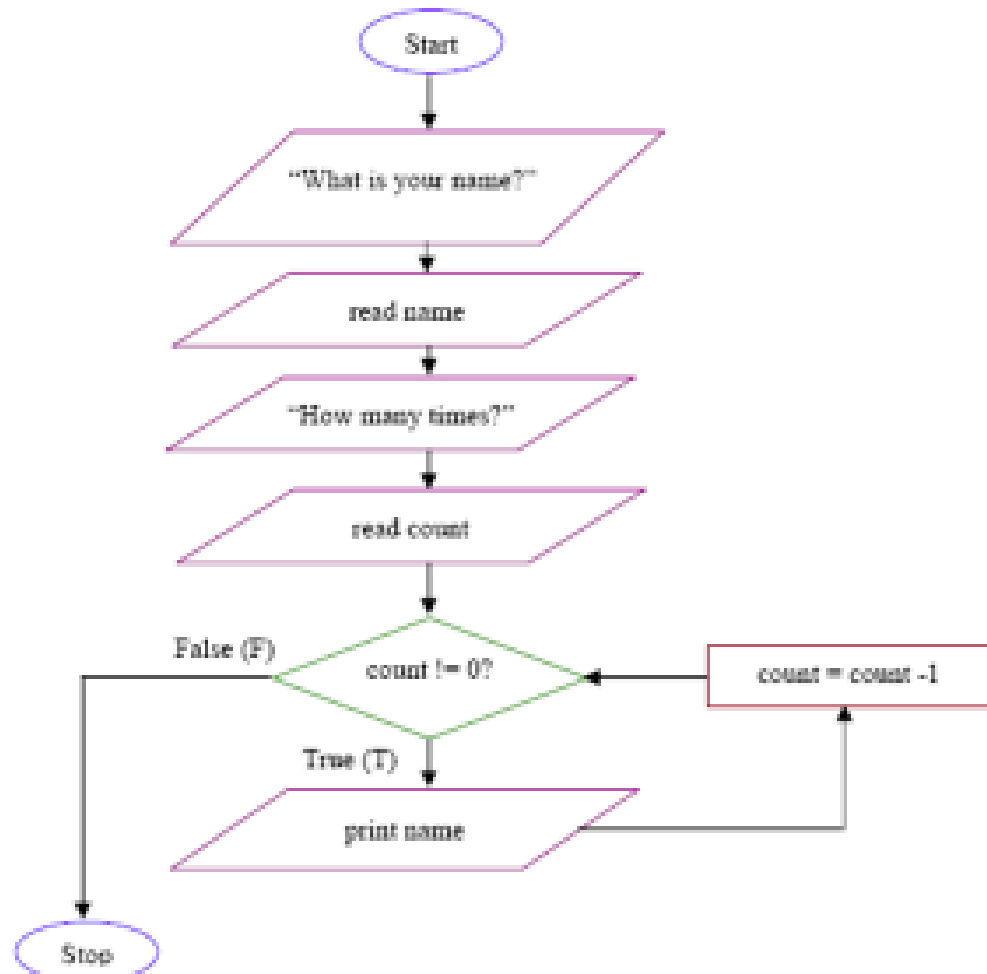
1. Start
2. Read a number from user
3. If number  $\%2 = 0$
4. Display "Even number"
5. Else Display "Odd number"
6. End



# Flowchart & Pseudo code

- **Pseudo code: Print Name 20 Times**

1. Start
2. Read name
3. Read count
4. Loop (If count  $\neq 0$ )
  1. Display "Name"
  2. *Count = Count - 1*
5. *End*



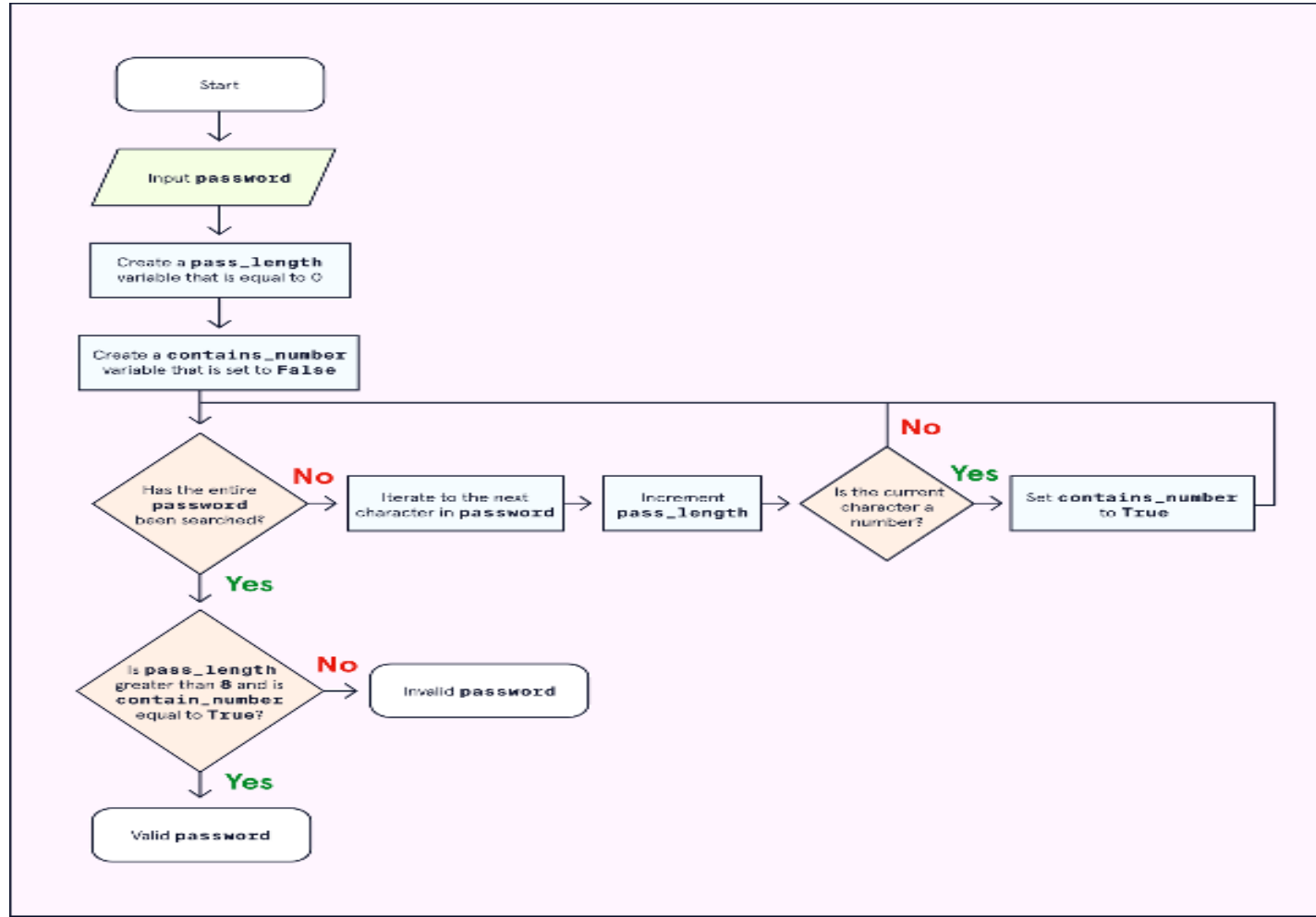
# Example(Password validator)

- If we want to set a couple of restrictions, such as that the password must be at least 8 characters and contain a number, then the following passwords would be valid:
  - Supers3cure
  - James1510
  - Meandmy2dogs
- But these would not:
  - password
  - Dog
  - Hunter2
- We have all seen plenty of passwords like these, so let's come up with a simple algorithm to validate passwords like this!

# The solution(Algorithm)

1. Input the password that we plan to validate.
2. To keep track of the password length, establish a `pass_length` variable and set it to 0.
3. To keep track of whether the password contains a number, establish a `contains_number` variable and initially set it to `False`.
4. Has the entire password been searched?
  1. If not, continue to step 5.
  2. If so, skip to step 8.
5. Iterate, or move to, to the next character in password.
6. Increase the value of `pass_length` by 1.
7. Is the current character a number?
  1. If not, go straight back to step 4 and continue to iterate over the entire password.
  2. If so, set the `contains_number` variable to `True` and then go back to step 4.
8. Is the `pass_length` greater than 8 and is `contains_number` equal to `True`?
  1. If not, then the password is invalid.
  2. If so, then the password is valid!

# Flow Chart



# Pseudocode

- *define password*
- *create a pass\_length variable and set it to 0*
- *create a contains\_number variable and set it to False*
- *if the entire password hasn't been searched:*
  - *iterate to the next character of the password*
  - *increment the pass\_length variable*
  - *if the current character of the password contains number:*
    - *set contains\_number to True*
- *if pass\_length is greater than 8 and if contain\_number is equal to True:*
  - *valid password*
- *otherwise:*
  - *invalid password*

# Code

```
• # 1. Input the `password` that we plan to validate
• password = "c0decademy"
• # 2. To keep track of the password length, establish a `pass_length` variable and initially set it to `0`
• pass_length = 0
• # 3. To keep track of if the password contains a number, establish a `contains_number` variable and initially set it to `False`
• contains_number = False
• # 4. Has the entire `password` been searched?
• while pass_length is not len(password):
•     # 5. Iterate to the next character in `password`
•     current_character = password[pass_length]
•     # 6. Increment `pass_length`
•     pass_length = pass_length + 1
•     # 7. Is the current character a number?
•     if current_character.isdigit():
•         # If so, set the `contains_number` variable to `True` and then go back to step 4
•         contains_number = True
• # 8. Is the `pass_length` greater than `8` and is `contains_number` equal to `True`?
• if pass_length > 8 and contains_number is True:
•     # If so, then the `password` is valid!
•     print("Valid Password!")
• else:
•     # If not, then the `password` is invalid
•     print("Invalid Password")
```



## *Write an algorithm to find area of a rectangle*

- *Step 1: Start*
- *Step 2: get l,b values*
- *Step 3: Calculate  $A=l*b$*
- *Step 4: Display A*
- *Step 5: Stop*

# Write an algorithm for Calculating area and circumference of circle

- Step 1: Start
- Step 2: get  $r$  value
- Step 3: Calculate  $A = 3.14 * r * r$
- Step 4: Calculate  $C = 2 * 3.14 * r$
- Step 5: Display  $A, C$
- Step 6: Stop

# Write an algorithm for Calculating simple interest

- Step 1: Start
- Step 2: get P, n, r value
- Step 3: Calculate
- $SI = (p * n * r) / 100$
- Step 4: Display S
- Step 5: Stop

# Importance of Efficiency:

- *Efficient algorithms save time and resources.*
- *Time Complexity: Measure of how long an algorithm takes to run.*
- *Space Complexity: Measure of how much memory an algorithm uses.*
- *Example: Searching for a Name in a Phone Book*
  - *Linear search (inefficient): Go through every name until finding the desired one.*
  - *Binary search (efficient): Divide the book in half and eliminate half of the remaining options with each step.*

# Algorithmic Problem Solving, and Simple Strategies for Developing Algorithms

- *Understand the Problem:*
  - *Before you start designing an algorithm, it's crucial to thoroughly understand the problem you're trying to solve. Break down the problem into smaller, manageable sub-problems if necessary.*
- *Define the Input and Output:*
  - *Clearly specify the input parameters and the expected output of the algorithm. Knowing what data you're working with and what result you need helps in designing a suitable algorithm.*
- *Plan Your Approach:*
  - *Consider various strategies and approaches for solving the problem. Think about what data structures and operations might be useful. Brainstorm and outline your initial thoughts.*

# Algorithmic Problem Solving, and Simple Strategies for Developing Algorithms

- **Divide and Conquer:**
  - For complex problems, consider using a divide-and-conquer approach. Break the problem into smaller, more manageable sub-problems and solve them separately.
- **Recursion:**
  - Recursion is a powerful technique for solving problems that can be divided into smaller instances of the same problem. Write recursive functions when appropriate.
- **Iteration:**
  - Use loops and iterative constructs when a problem can be solved through repetitive operations. Iteration can be more efficient than recursion for certain problems.
- **Test and Debug:**
  - Implement the algorithm in your chosen programming language and test it with various inputs, including edge cases. Debug and refine the algorithm as needed.

# Factorial of a number using recursion

- *# Factorial of a number using recursion*
- *def recur\_factorial(n):*
  - *if n == 1:*
  - *return n*
  - *else:*
  - *return n\*recur\_factorial(n-1)*
- *num = 7*
- *# check if the number is negative*
- *if num < 0:*
  - *print("Sorry, factorial does not exist for negative numbers")*
- *elif num == 0:*
  - *print("The factorial of 0 is 1")*
- *else:*
  - *print("The factorial of", num, "is", recur\_factorial(num))*

# Algorithmic Problem Solving, and Simple Strategies for Developing Algorithms

- *Analyze Time and Space Complexity:*
  - *Assess the algorithm's efficiency by analyzing its time and space complexity. Strive for algorithms that run efficiently in terms of both time and memory usage.*
- *Optimize:*
  - *Once you have a working algorithm, look for opportunities to optimize it. This might involve improving the algorithm's time complexity or reducing memory usage.*
- *Document Your Algorithm:*
  - *Finally, document your algorithm, including its purpose, inputs, outputs, and any assumptions. Clear documentation helps others understand and use your algorithm.*



# Analysis of Algorithms (Computational Complexity - Time and Space)

- Analysis of algorithms, specifically in terms of computational complexity, involves evaluating how an algorithm's runtime and memory usage grow as the size of the input data increases.
- This analysis helps us understand the efficiency of algorithms and make informed choices when selecting the best algorithm for a given problem.

# Time Complexity:

- Time complexity measures the computational time required by an algorithm as a function of the input size.
- It helps answer questions like, "How does the runtime of the algorithm change when the input size grows?"
- Time complexity is typically expressed using Big O notation ( $O(f(n))$ ), where  $f(n)$  represents an upper bound on the number of operations an algorithm performs concerning the input size ( $n$ ).

# Common time complexities:

- $O(1)$  (constant time): The algorithm's runtime is independent of the input size.
- $O(\log n)$  (logarithmic time): The runtime grows slowly as the input size increases, common in efficient search algorithms.
- $O(n)$  (linear time): The runtime increases linearly with the input size, common in linear searches and linear traversals.
- $O(n \log n)$  (linearithmic time): Common in efficient sorting algorithms like merge sort and quicksort.
- $O(n^2)$  (quadratic time): The runtime grows with the square of the input size, common in nested loops.
- $O(2^n)$  (exponential time): Highly inefficient algorithms that grow exponentially with the input size.
- It's crucial to consider the worst-case, average-case, and best-case time complexities

# Space Complexity:

- Space complexity measures the amount of memory an algorithm uses concerning the input size.
- It helps answer questions like, "How much memory does the algorithm require as the input size increases?"
- Space complexity is also expressed using Big O notation and provides an upper bound on the memory usage.

# Common space complexities:

- $O(1)$  (constant space): The algorithm uses a fixed amount of memory regardless of the input size.
- $O(n)$  (linear space): The memory usage grows linearly with the input size, often associated with data structures that store all input data.
- $O(n^2)$  (quadratic space): Memory usage grows with the square of the input size, often associated with nested data structures.
- $O(\log n)$  (logarithmic space): Memory usage grows slowly as the input size increases, common in algorithms that use recursion with a small stack space.
- Like time complexity, it's essential to consider different cases of space complexity, especially in scenarios with limited memory resources.

# Basic Programming Fundamentals

- Basic programming fundamentals are essential for anyone learning to code.
- These fundamentals encompass various aspects of programming, including programming languages, syntax and coding environments, object-oriented programming (OOP) concepts, efficient code building practices, and debugging techniques.

# Programming Language:

- **Choose a Language:** Start by selecting a programming language that suits your goals and interests. Common choices include Python, Java, C++, JavaScript, and more.
- **Learn the Basics:** Master the fundamentals of your chosen language, including variables, data types, operators, and control structures (if statements, loops).
- **Practice:** The best way to learn a programming language is through practice. Write simple programs to reinforce your understanding.

# Syntax and Coding Environment:

- **Understand Syntax:** Each programming language has its own syntax rules. Pay attention to proper indentation, punctuation, and naming conventions.
- **Code Editor/IDE:** Use a code editor or integrated development environment (IDE) for writing and running code. Popular choices include Visual Studio Code, PyCharm, and Eclipse.



# Object-Oriented Programming (OOP) Concepts

- **Class:** Blueprint or template for creating objects, defining their structure and behavior.
- **Object:** Instance of a class representing real-world entities with attributes and behaviors.
- **Inheritance:** Mechanism to create new classes based on existing ones, promoting code reuse.
- **Encapsulation:** It refers to the bundling of data (attributes or properties) and the methods (functions) that operate on that data into a single unit called a class.
- **Polymorphism:** Polymorphism means "many shapes" and is a key concept in OOP. It allows objects of different classes to be treated as objects of a common base class.

# Efficient Code Building Practices:

- **Algorithm Design:** Focus on designing efficient algorithms by considering time and space complexity.
- **Code Modularity:** Break code into smaller functions or modules to improve readability and reusability.
- **Documentation:** Write clear comments and documentation to explain code logic and usage.
- **Testing:** Develop a habit of testing your code thoroughly to catch and fix errors early.

# *Practice and Projects:*

- *Regularly work on coding exercises and small projects to reinforce your skills.*
- *Gradually take on more complex projects as you become more comfortable with programming.*

# Online Resources and Communities:

- *Utilize online resources like tutorials, documentation, and forums to learn and seek help when you encounter challenges.*
- *Join coding communities and forums to interact with other learners and experienced developers.*

# Stay Updated:

- *Programming languages and technologies evolve. Stay updated with the latest developments and best practices in your chosen language or field.*

# Assessment

1. Give two important differences between the flowchart and the pseudocode.
2. Give two examples of selection control structures. Use flowchart.
3. Draw the different types of symbols used in the flowchart. Explain the role of each.

## Summary:

- Computational thinking breaks down complex problems.
- Algorithms provide step-by-step solutions.
- Better Logic ,Better programmer
- Algorithm should be efficient
- Pseudo codes, flowcharts are the initial steps of designing a program for any given task.