Contents

Chapter 1

The World of Problem-solving

1.1 Introduction

"The greatest challenge to any thinker is stating the problem in a way that will allow a solution."

– Bertrand Russell

"A great discovery solves a great problem but there is a grain of discovery in the solution of any problem. Your problem may be modest; but if it challenges your curiosity and brings into play your inventive faculties, and if you solve it by your own means, you may experience the tension and enjoy the triumph of discovery. Such experiences at a susceptible age may create a taste for mental work and leave their imprint on mind and character for a lifetime."

– George Pólya

The digital general-purpose computer stands out as one of the most significant technological advancements of the past century, marking the beginning of a transformative era that introduced us to the Internet, smartphones, tablets, and widespread computerization. To unlock the full potential of these computers, we rely on *programming*, which involves creating a sequence of instructions, or *code*, that a computer can execute to tackle computational problems. The language used to write this code is known as a *programming language*.

Embedded within this code is the concept of an *algorithm*, which represents the abstract method for solving a problem. The objective of *algorithmic problem-solving* is to develop an effective algorithm that addresses specific computational challenges. While it is not always necessary to write code to appreciate algorithmic problem-solving, engaging in the programming process often deepens understanding and helps in discovering more efficient and straightforward solutions to complex issues.

To get an idea about what we are going to discuss in this book, i.e., algorithmic thinking (algorithmic problem solving), let us go through some common problems you encounter as a student or as a common person.

Problem-1: Suppose that you are back from college, and discover that you have lost your wallet that is of great sentimental value to you. How do you regain your lost wallet?

In our daily life, we encounter problems that are big and small. Some are easy to solve and some are really tough to solve. Some call for a structured solution, whereas some require an unstructured approach. Some are interesting and some are not.

If we consider the nature of problems, they range from mathematical to philosophical. In computing, we deal with *algorithmic problems* whose solutions are expressed as algorithms. An algorithm is a set of well-defined steps and instructions that can be translated into a form, usually known as the code or program, that is executable by a computing device. You will learn more about algorithmic problem-solving in Section 1.3.

For the time being, we shall study problem-solving in general. However, we still want to confine our domain to the logical applications of scientific and mathematical methods. Therefore, we will exclude problems such as “How to become a millionaire in 50 days?”, “How to score full A+’s in examinations?”, and “How many years do computer scientists take to build a ”human-like” robot?”

Returning to the wallet problem, have you found a solution? If you have, then something is really missing. First of all, the problem statement is incomplete. Where did you actually drop the wallet? Did you assume that it was dropped at the college? Could it have happened on the way back home? If the wallet was indeed dropped at the school, where in the school? In the classroom or playground or the library or somewhere else? These questions must be answered before you can reach a complete understanding of the problem. Assumptions should not be made without basis. Irrelevant information (such as the sentimental value of the wallet) should not get in to the way. Incomplete information must be sought out. You could only fully understand the problem after all ambiguities are removed. Then only you should try to solve the problem. Otherwise, you might end up in a situation where you get stuck.

Now, suppose your wallet was lost in the library. How do you recover the wallet?

Problem-2: In a circle, a square with side length $2a$ is inscribed. What is the area of the circle?

This is a well-stated problem in geometry. After you have fully understood the problem, you need to devise a plan for solving it. You need to determine the input (the length of each side of the square) and the output (the area of the circle). You use suitable notation to represent the data, and you examine the relationship between the data and the unknown, which will often lead you to a solution. This may involve the calculation of some intermediate result, like the length of the diagonal of the square. In this case, you need to make use of the domain knowledge of basic geometry.

$n = 0$:					1				
$n = 1$:				1		1			
$n = 2$:			1		2		1		
$n = 3$:		1		3		3		1	
$n = 4$:	1		4		6		4		1

Table 1.1: Pascal's triangle

Problem-3: Suppose that you are to ship a wolf, a sheep, and a cabbage across the river. The boat can only hold the weight of two: you plus another item. You are the only one who can row the boat. The wolf must not be left with the sheep alone, or the wolf will eat the sheep. Neither should the sheep be left alone with the cabbage. How do you get them over to the other side of the river?

This type of problem involves logic. The required solution is not a computed value as in Problem 2, but a series of moves that represent the transition from the initial state (in which all are on one side of the bank) to the final state (where all are on the opposite side).

Problem-4: Let us consider the well-known Pascal's triangle shown in Table 1.1.

There are 1's on the boundaries. For the rest, each value is the sum of the nearest two numbers above it, one on its right and the other on its left. How do you compute the combination $\binom{n}{k}$ using Pascal's triangle? How are the values related to the coefficients in the expansion of $(x + y)^n$ for non-negative n?

Problem-5: Suppose that you are given a rectangular 3×5 grid. You are allowed to move from one intersection to another, governed by the rule that you can travel only upwards or to the right, along the grid lines. How many paths are there from each of the intersections to the top-right corner of the grid?

Are the above two problems related? If so, how is this problem related to the Pascal's triangle? Knowing that Problems 4 and 5 are related, how would you use Pascal's triangle to solve this problem?

This illustrates the often-encountered situation that calls for relating a problem at hand to one that we have solved before. Figuring out the similarity among problems, and performing problem transformation, help us to sharpen our problem-solving ability.

The problems that we discussed offer a quick look into the craft of problem-solving. The Hungarian mathematician George Pólya (1887-1985) conceptualized the process and captured it in the famous book "How To Solve It". In his book, Pólya enumerates four phases a problem solver has to go through. First, we need to understand the problem and clarify any doubts about the problem statement if necessary, as we have discussed in Problem 1. Second, we need to find the connection between the data and the unknown, to make out a plan for the solution. Auxiliary problems might be created in the process. Third, we are to carry out our plan, checking the validity of each step. Finally, we have to review the solution.

The four phases are outlined below.

- **Phase 1: Understanding the problem.**
  - *What is the unknown? What are the data?*
  - What is the condition? Is it possible to satisfy the condition? Is the condition sufficient to determine the unknown? Or is it insufficient? Or redundant? Or contradictory?
  - Draw a figure. Introduce suitable notation.
  - Separate the various parts of the condition. Can you write them down?
- **Phase 2: Devising a plan.**
  - Have you seen it before? Or have you seen the *same problem in a slightly different form?*
  - Do you know a related problem?
  - Look at the unknown! Try to think of a familiar problem having the same or similar unknown.
  - Split the problem into smaller, simpler sub-problems.
  - If you cannot solve the proposed problem try to solve first some related problem. Or solve a more general problem. Or a special case of the problem. Or solve a part of the problem.
- **Phase 3: Carrying out the plan.**
  - Carrying out your plan of the solution, check each step.
  - Can you see clearly that the step is correct?
  - Can you prove that it is correct?
- **Phase 4: Looking back.**
  - Can you check the result?
  - Can you derive the result differently?
  - Can you use the result, or the method, for some other problem?

You should note that asking questions is a running theme in all phases of the process. In fact, asking questions is the building block of problem-solving. So, you should start developing the art of asking questions from now on, if you have not done so.

1.2 Problem-Solving Strategies

Problem-solving strategies are essential tools that enable you to effectively tackle a wide range of challenges by providing structured methods to analyze, understand, and resolve problems. These strategies include systematic approaches such as Trial and Error, Heuristics, Means-Ends Analysis, and Backtracking, each offering unique benefits and applications. Understanding multiple problem-solving strategies is crucial as it allows for adaptability and flexibility, ensuring that one can choose the most efficient method for any given situation. This versatility enhances problem-solving efficiency and improves outcomes by offering diverse perspectives and potential solutions. Additionally, employing various strategies enriches cognitive skill development, critical thinking, and creativity. By integrating these strategies into everyday problem-solving, you can approach challenges with confidence and resilience, ultimately achieving more successful and innovative solutions

1.2.1 Importance of Understanding Multiple Problem-Solving Strategies

Understanding multiple problem-solving strategies is crucial because it equips individuals with a diverse toolkit to tackle a variety of challenges. Different problems often require different approaches, and being familiar with multiple strategies allows for greater flexibility and adaptability. For example, some problems might be best solved through a systematic trial and error method, while others might benefit from a more analytical approach like means-ends analysis. By knowing several strategies, one can quickly switch tactics when one method does not work, increasing the chances of finding a successful solution.

Additionally, having a collection of problem-solving strategies enhances critical thinking and creativity. It encourages thinking outside the box and considering multiple perspectives, which can lead to more innovative and effective solutions. This broad understanding also helps in recognizing patterns and similarities between different problems, making it easier to apply previous knowledge to new situations. In both academic and real-world scenarios, this versatility not only improves problem-solving efficiency but also boosts confidence and competence in facing various challenges. Some benefits are as follows

- **Adaptability:** Different problems require different approaches. Understanding multiple strategies allows for flexibility and adaptability in problem-solving.
- **Efficiency:** Some strategies are more effective for specific types of problems. Having a repertoire of strategies can save time and resources.
- **Improved Outcomes:** Diverse strategies offer multiple perspectives and potential solutions, increasing the likelihood of finding optimal solutions.
- **Skill Development:** Exposure to various strategies enhances cognitive skills, critical thinking, and creativity.

Let us look at some commonly used problem-solving strategies

1.2.2 Trial And Error Problem-Solving Strategy

The trial-and-error problem-solving strategy involves attempting different solutions and learning from mistakes until a successful outcome is achieved. It is a fundamental method that relies on experimentation and iteration, rather than systematic or analytical approaches.

Consider the situation where you have forgotten the password to your online account, and there is no password recovery option available. You decide to use trial and error to regain access:

1. **Initial Attempts**: You start by trying passwords you commonly use. For instance, you might first try ”password123,” ”Qwerty2024,” or ”MyDog-Tommy.”
2. **Learning from Mistakes**: None of these initial attempts work. You then recall that you sometimes use a combination of personal information. You try variations incorporating your birthdate, pet’s name, or favorite sports team.
3. **Refinement**: After several failed attempts, you remember you recently started using a new format for your passwords, combining a favorite quote with special characters. You attempted various combinations, such as”ToBeOrNotToBe!”, ”NeeNeeyaayirikkuka#,” and other combinations.
4. **Success**: Eventually, through persistent trial and error, you hit upon the correct password: ”NeeNeeyaayirikkuka#2024.”

In this scenario, trial and error involved systematically trying different potential passwords, learning from each failed attempt, and refining the approach based on what you remember about your password habits. This method is practical when there is no clear pathway to the solution and allows for discovering the correct answer through persistence and adaptability.

1.2.3 Algorithmic Problem-Solving Strategy

An algorithm is a step-by-step, logical procedure that guarantees a solution to a problem. It is systematic and follows a defined sequence of operations, ensuring consistency and accuracy in finding the correct solution.

When baking a cake, you follow a precise recipe, which acts as an algorithm:

1. **Gather Ingredients**: Measure out 2 cups of flour, 1 cup of sugar, 2 eggs, $\frac{1}{2}$ cup of butter, 1 teaspoon of baking powder, and 1 cup of milk.
2. **Preheat Oven**: Set the oven to 175°C.
3. **Mix Ingredients**: In a bowl, combine the flour, baking powder, and sugar. In another bowl, beat the eggs and then mix in the butter and milk. Gradually combine the wet and dry ingredients, stirring until smooth.

4. **Prepare Baking Pan**: Grease a baking pan with butter or cooking spray.
5. **Pour Batter**: Pour the batter into the prepared pan.
6. **Bake**: Place the pan in the preheated oven and bake for 30-35 minutes.
7. **Check for Doneness**: Insert a toothpick into the center of the cake. If it comes out clean, the cake is done.
8. **Cool and Serve**: Let the cake cool before serving.

By following this algorithm (the recipe), you systematically achieve the desired result — a perfectly baked cake.

1.2.4 Heuristic Problem-Solving Strategy

A heuristic is a practical approach to problem-solving based on experience and intuition. It does not guarantee a perfect solution but provides a good enough solution quickly, often through rules of thumb or educated guesses. When driving in a city with frequent traffic congestion, you might use a heuristic approach to find the fastest route to your destination:

1. **Rule of Thumb**: You know from experience that certain streets are typically less congested during rush hour.
2. **Current Conditions**: You use a traffic app to check current traffic conditions, looking for red or yellow indicators on major roads.
3. **Alternative Routes**: You consider side streets and shortcuts you have used before that tend to be less busy.
4. **Decision**: Based on the app and your knowledge, you decide to avoid the main highway (which shows heavy congestion) and take a series of back roads that usually have lighter traffic.

While this heuristic approach does not guarantee that you will find the absolute fastest route, it combines your experience and real-time data to make an informed, efficient decision, likely saving you time compared to blindly following the main routes.

1.2.5 Means-Ends Analysis Problem-Solving Strategy

Means-ends analysis is a strategy that involves breaking down a problem into smaller, manageable parts (means) and addressing each part to achieve the final goal (ends). It involves identifying the current state, the desired end state, and the steps needed to bridge the gap between the two.

Imagine you want to plan a road trip from Trivandrum to Kashmir. Here is how you might use means-ends analysis:

1. **Define the Goal**: Your ultimate goal is to drive from Trivandrum to Kashmir.
2. **Analyze the Current State**: You start in Trivandrum with your car ready to go.
3. **Identify the Differences**: The primary difference is the distance between Trivandrum and Kashmir, which is approximately 3,700 kilometers.
4. **Set Sub-Goals (Means)**:
    - **Fuel and Rest Stops**: Determine where you will need to stop for fuel and rest.
    - **Daily Driving Targets**: Break the trip into daily segments, such as driving 500-600 kilometers per day.
    - **Route Planning**: Choose the most efficient and scenic route, considering highways, weather conditions, and places you want to visit.
5. **Implement the Plan**:
    - **Day 1**: Drive from Trivandrum to Bangalore, Karnataka (approx. 720 km). Refuel in Madurai, Tamil Nadu. Overnight stay in Bangalore.
    - **Day 2**: Drive from Bangalore to Hyderabad, Telangana (approx. 570 km). Refuel in Anantapur, Andhra Pradesh. Overnight stay in Hyderabad.
    - **Day 3**: Drive from Hyderabad to Nagpur, Maharashtra (approx. 500 km). Refuel in Adilabad, Telangana. Overnight stay in Nagpur.
    - **Day 4**: Drive from Nagpur to Jhansi, Uttar Pradesh (approx. 580 km). Refuel in Sagar, Madhya Pradesh. Overnight stay in Jhansi.
    - **Day 5**: Drive from Jhansi to Agra, Uttar Pradesh (approx. 290 km). Refuel in Gwalior, Madhya Pradesh. Overnight stay in Agra. Visit the Taj Mahal.
    - **Day 6**: Drive from Agra to Chandigarh (approx. 450 km). Refuel in Karnal, Haryana. Overnight stay in Chandigarh.
    - **Day 7**: Drive from Chandigarh to Jammu (approx. 350 km). Refuel in Pathankot, Punjab. Overnight stay in Jammu.
    - **Day 8**: Drive from Jammu to Srinagar, Kashmir (approx. 270 km).
6. **Adjust as Needed**: Throughout the trip, you may need to make adjustments based on traffic, road conditions, or personal preferences.

By breaking down the long journey into smaller, achievable segments and addressing each part systematically, you can effectively plan and complete the road trip. This method ensures that you stay on track and make steady progress toward your final destination, despite the complexity and distance of the trip.

1.2.6 Problem decomposition

In the previous example of traveling from Trivandrum to Kashmir, breaking down the journey into smaller segments helped you come up with an effective plan for the completion of the trip. This is the key to solving complex problems. When the problem to be solved is too complex to manage, break it into manageable parts known as sub-problems. This process is known as **problem decomposition**. Here are the steps in solving a problem using the decomposition approach:

1. **Understand the problem**: Develop a thorough understanding of the problem.
2. **Identify the sub-problems**: Decompose the problems into smaller parts.
3. **Solving the sub-problems**: Once decomposition is done, you proceed to solve the individual sub-problems. You may have to decide upon the order in which the various sub-problems are to be solved.
4. **Combine the solution**: Once all the sub-problems have been solved, you should combine all those solutions to form the solution for the original problem.
5. **Test the combined solution**: Finally you ensure that the combined solution indeed solves the problem effectively.

1.2.7 Other Problem-solving Strategies

Here are some examples of problem-solving strategies that may equally be adopted to see which works best for you in different situations:

i. **Brainstorming**

   Brainstorming involves generating a wide range of ideas and solutions to a problem without immediately judging or analyzing them. The goal is to encourage creative thinking and explore various possibilities.

   In a team meeting to improve customer satisfaction, everyone contributes different ideas, such as enhancing product quality, improving customer service training, offering loyalty programs, and using customer feedback to make improvements. These ideas are later evaluated and the best ones are implemented.

ii. **Lateral Thinking**

   Lateral thinking is about looking at problems from new and unconventional angles. It involves thinking outside the box and challenging established patterns and assumptions.

   A company facing declining sales of a product might use lateral thinking to identify new uses for the product or new markets to target, rather than just trying to improve the existing product or marketing strategy.

iii. **Root Cause Analysis**

Root cause analysis involves identifying the fundamental cause of a problem rather than just addressing its symptoms. The goal is to prevent the problem from recurring by solving its underlying issues.

If a factory's production line frequently breaks down, rather than just repairing the machinery each time, the team conducts a root cause analysis and discovers that poor maintenance scheduling is the underlying issue. By implementing a better maintenance plan, they reduce the breakdowns.

iv. **Mind Mapping**

Mind mapping is a visual tool for organizing information. It helps in brainstorming, understanding, and solving problems by visually connecting ideas and concepts.

When planning a large event, an organizer creates a mind map with the event at the center, branching out into categories like venue, catering, entertainment, invitations, and logistics. Each category further branches into specific tasks and considerations.

v. **SWOT Analysis**

SWOT analysis involves evaluating the **S**trengths, **W**eaknesses, **O**pportunities, and **T**hreats related to a particular problem or decision. It helps in understanding both internal and external factors that impact the situation.

A business firm considering a new product launch performs a SWOT analysis. They identify their strengths (strong brand, good distribution network), weaknesses (limited R&D budget), opportunities (market demand, potential partnerships), and threats (competition, economic downturn). This analysis guides their decision-making process.

vi. **Decision Matrix**

A decision matrix, also known as a decision grid or Pugh matrix, helps in evaluating and prioritizing a list of options. It involves listing options and criteria, assigning weights to each criterion, and scoring each option based on the criteria.

A family deciding on a new car creates a decision matrix with criteria such as cost, fuel efficiency, safety features, and brand reputation. They rate each car option against these criteria and calculate a total score to make an informed choice.

vii. **Simulation**

Simulation involves creating a model of a real-world system and experimenting with it to understand how the system behaves under different conditions. It helps in predicting outcomes and identifying the best course of action.

Urban planners use traffic simulation software to model the impact of new road constructions on traffic flow. By testing different scenarios, they can design the most effective road network to reduce congestion.

viii. **Use Experience**

The use of experience as a problem-solving strategy involves drawing on previous knowledge and experiences to address current challenges. This strategy relies on the idea that similar problems often have similar solutions, and leveraging past experiences can lead to efficient and effective outcomes.

A company trying to market a new clothing line may consider marketing tactics they have previously used, such as magazine advertisements, influencer campaigns, or social media advertisements. By analyzing what tactics have worked in the past, they can create a successful marketing campaign again.

These strategies provide various approaches to problem-solving, each suitable for different types of challenges and contexts.

1.3 Algorithmic problem solving with computers

In today's digital era, the computer has become an indispensable part of our life. Down from performing simple arithmetic right up to accurately determining the position for a soft lunar landing, we rely heavily on computers and allied digital devices. Computers are potent tools for solving problems across diverse disciplines. Problem-solving is a systematic way to arrive at solutions for a given problem.

1.3.1 The Problem solving process

Let us now explore how computers can be put to solving problems:

1. **Understand the problem**: Effective problem-solving demands a thorough knowledge of the problem domain. Once you have identified the problem, its exact nature must be sought and defined. The problem context, objectives, and constraints if any are to be understood properly. Several techniques can be used to gather information about a problem. Some of these include conducting interviews and sending questionnaires to the stakeholders (people who are concerned with the problem). Segmenting a big problem into simple manageable ones often helps you to develop a clear picture of the problem.

2. **Formulate a model for the solution**: After the problem is thoroughly understood, the next step is to devise a solution. You should now identify

the various ways to solve the problem. Brainstorming is one of the most commonly used techniques for generating a large number of ideas within a short time. Brainwriting and Mind mapping are two alternative techniques that you can employ here. The generated ideas are then transformed into a conceptual model that can be easily converted to a solution. Mathematical modeling and simulation modeling are two popular modeling techniques that you could adopt. Whatever the modeling technique is, ensure the defined model accurately reflects the conceived ideas.

3. **Develop an algorithm**: Once a list of possible solutions is determined, they have to be translated into formal representations – *algorithms*. Obviously, you do not implement all the solutions. So the next step is to assess the pros and cons of each algorithm to select the best one for the problem. The assessment is based on considering various factors such as memory, time, and lines of code.

4. **Code the algorithm**: The interesting part of the process! Coding! After the *best algorithm* is determined, you implement it as an executable program. The program or the code is a set of instructions that is more or less, a concrete representation of the algorithm in some programming language. While coding, always follow the incremental paradigm – start with the essential functionalities and gradually add more and more to it.

5. **Test the program**: Nobody is perfect! Once you are done with the coding, you have to inspect your code to verify its correctness. This is formally called *testing*. During testing, the program is evaluated as to whether it produces the desired output. Any unexpected output is an *error*. The program should be executed with different sets of inputs to detect errors. It is impossible to test the program with all possible inputs. Instead, a smaller set of representative inputs called *test suite* is identified and if the program runs correctly on the test suite, then it is concluded that the program will probably be correct for all inputs. You can get the help of automated testing tools to generate a test suite for your code. Closely associated with testing is the process of *debugging* which involves fixing or resolving the errors (technically called bugs) identified during testing. Testing and debugging should be repeated until all errors are fixed.

6. **Evaluate the solution**: This final step is crucial to ensure that the program effectively addresses the problem and attains the desired objectives. You have to first define the evaluation criteria. These could include metrics like efficiency, feasibility, and scalability, a few to mention. The potential risks that could arise with the program's deployment are also to be assessed. Collect quantitative and qualitative feedback from the stakeholders. Based on the feedback, you have to work on making necessary improvements to the program. Nevertheless, the refined code should also be subject to rigorous testing.

1.3.2 A case study - The Discriminant calculator

Tell me and I forget. Teach me and I remember. Involve me and I learn. - Benjamin Franklin

How about going around the problem-solving process a second time? But this time, with a real problem at hand – determining the discriminant of a quadratic equation. Roll up your sleeves!

1. **Understand the problem**: Here we formally define the problem by specifying the inputs and output.

   ***Input*:** The three coefficients a, b and c of the quadratic equation

   ***Output*:** The discriminant value D for the quadratic equation

2. **Formulate a model for the solution**: Develop a mathematical model for the solution, that is identify the mathematical expression for the quadratic equation discriminant D:

$$D = b^2 - 4ac$$

3. **Develop an algorithm**: A possible algorithm (actually, a pseudocode) for our discriminant problem is given below:

```
1 Start
2 READ($a, b, c$)
3 $d = b * b - 4 * a * c$
4 PRINT($d$)
5 Stop
```

   You will learn more about pseudocodes in Chapter 2.

4. **Code the algorithm**: The Python program to calculate the discriminant is as follows:

```
#Input the coefficients
a = int(input("Enter the value of first coefficient"))
b = int(input("Enter the value of second coefficient"))
c = int(input("Enter the value of third coefficient"))
#Find the discriminant
d = (b**2) - (4*a*c)
#Print the discriminant
print(d)
```

   Completely puzzled about the code? Don't worry! We will start with Python programming in Chapter 4.

5. **Test the program**: You create a test suite similar to the one shown in Table 1.2. Each row denotes a set of inputs (a, b, and c) and the expected output (d) with which the actual output is to be compared.

Table 1.2: A test suite for the discriminant calculator

Sl. No.	a	b	c	d
1	10	2	5	-196
2	5	7	1	29
3	2	4	2	0
4	1	1	1	-3
5	3	2	5	-56
6	2	8	2	48

1.4 Conclusion

In this chapter, we've explored the diverse landscape of problem-solving, highlighting the importance of understanding and applying various strategies. By delving into methods like trial and error, heuristics, and means-ends analysis, we've gained insight into how different approaches can be leveraged depending on the nature of the problem at hand. The discussion on algorithmic problem-solving has further illustrated how structured processes can be applied, particularly when using computers to tackle complex issues.

The case study on the Discriminant calculator effectively bridges theory with practice, showing that these problem-solving strategies are not just abstract concepts but have real-world applications. This chapter provides a foundational toolkit for tackling a variety of challenges and enhancing your confidence, creativity, and precision in both computational and practical scenarios.

1.5 Exercises

1. A bear, starting from the point P, walked one mile due south. Then he changed direction and walked one mile due east. Then he turned again to the left and walked one mile due north, and arrived at the point P he started from. What was the color of the bear?

2. Two towns A and B are 3 kilometers apart. It is proposed to build a new school serving 100 students in town A and 50 students in town B. How far from town A should the school be built if the total distance travelled by all 150 students is to be as small as possible?

3. A traveller arrives at an inn. He has no money but only a silver chain consisting of 6 links. He uses one link to pay for each day spent at the inn, but the innkeeper agrees to accept no more than one broken link.

How should the traveller cut up the chain in order to settle accounts with the innkeeper on a daily basis?

4. What is the least number of links that have to be cut if the traveller stays 100 days at the inn and has a chain consisting of 100 links? What is the answer in the general case (n days and n links)?

5. The minute and hour hands of a clock coincide exactly at 12 o'clock. At what time later do they first coincide again?

6. Six glasses are in a row, the first three full of juice, the second three empty. By moving only one glass, can you arrange them so that empty and full glasses alternate?

7. You throw away the outside and cook the inside. Then you eat the outside and throw away the inside. What did you eat?

8. Rearrange the letters in the words new door to make one word.

9. A mad scientist wishes to make a chain out of plutonium and lead pieces. There is a problem, however. If the scientist places two pieces of plutonium next to each other, BOOM! The question is, in how many ways can the scientist safely construct a chain of length n?

10. Among 12 ball bearings, one is defective, but it is not known if it is heavier or lighter than the rest. Using a traditional balance (with two pans hanging down the opposite ends of a lever supported in the middle), how do you determine which is the defective ball bearing, and whether it is heavier or lighter than the others, within three attempts?

Bibliography

[1] George Pólya and John Conway, How to Solve It: A New Aspect of Mathematical Method, Princeton University Press, 2014.

[2] Maureen Sprankle and Jim Hubbard, Problem Solving and Programming Concepts, Pearson, 2011.

[3] Donald E. Knuth, The Art of Computer Programming (Volume 1), Addison-Wesley, 1997.

[4] Nell Dale and John Lewis, Computer Science Illuminated, Jones and Bartlett Publishers, 2019.

[5] R G Dromey, How To Solve It By Computer, Pearson, 2008.

Chapter 2

Algorithms, pseudocodes, and flowcharts

"When you choose an algorithm, you choose a point of view"
– Anonymous

"An Algorithm is a storytellers' script while Pseudocode is the builders' blueprint"
– Anonymous

2.1 Algorithms and pseudocodes

An **algorithm** describes a systematic way of solving a problem. It is a step-by-step procedure that produces an output when given the necessary inputs. An algorithm uses pure English phrases or sentences to describe the solution to a problem. A **Pseudocode** is a high-level representation of an algorithm that uses a mixture of natural language and programming language-like syntax. It is more structured than an algorithm in that it uses mathematical expressions with English phrases to capture the essence of a solution concisely. You can also use programming constructs (See Section 2.1.2 below) in a pseudocode which are not permitted in an algorithm in a strict sense.

As the name indicates, pseudocode is not a true program and thus is independent of any programming language. It is not executable rather helps you understand the flow of an algorithm.

Confused between algorithms and pseudocodes? Let us take an example. We will now write an algorithm and a pseudocode to evaluate an expression, say $d = a + b * c$. Here $a, b, c,$ and d are known as *variables.* Simply put, a variable is a name given to a memory location that stores some data value. First, let us look at the algorithm for the expression evaluation.

```
EVALUATE-ALGO
1  Start
2  Read the values of $a, b$ and $c$.
3  Find the product of $b$ and $c$.
4  Store the product in a temporary variable temp.
5  Find the sum of $a$ and temp.
6  Store the sum in $d$.
7  Print the value of $d$.
8  Stop.
```

The following is the pseudocode to evaluate the same expression.

```
EVALUATE-PSEUDO
1  Start
2  READ($a, b, c$)
3  $d = a + b * c$
4  PRINT($d$)
5  Stop
```

In a pseudocode, READ is used to read input values. PRINT is used to print a message. The message to be printed should be enclosed in a pair of double quotes. For example,

PRINT("Hello folks!!")

prints

```
Hello folks!!
```

To print the value of a variable, just use the variable name (without quotes). In the above example, PRINT(d) displays the value of the variable d.

Although pseudocode and algorithm are technically different, these words are interchangeably used for convenience.

2.1.1 Why pseudocodes?

Wondering why pseudocodes are important? Here are a few motivating reasons:

1. **Ease of understanding**: Since the pseudocode is programming language independent, novice developers can also understand it very easily.
2. **Focus on logic**: A pseudocode allows you to focus on the algorithm's logic without bothering about the syntax of a specific programming language.
3. **More legible**: Combining programming constructs with English phrases makes pseudocode more legible and conveys the logic precisely.
4. **Consistent**: As the constructs used in pseudocode are standardized, it is useful in sharing ideas among developers from various domains.

Table 2.1: Relational operators

Operator	**Meaning**
$>$	greater than
$<$	less than
$==$	equal to
$>=$	greater than or equal to
$<=$	less than or equal to
$!=$	not equal to

5. **Easy translation to a program**: Using programming constructs makes mapping the pseudocode to a program straightforward.

6. **Identification of flaws**: A pseudocode helps identify flaws in the solution logic before implementation.

2.1.2 Constructs of a pseudocode

A good pseudocode should follow the structured programming approach. Structured coding aims to improve the readability of pseudocode by ensuring that the execution sequence follows the order in which the code is written. Such a code is said to have a *linear flow of control*. Sequencing, selection, and repetition (loop) are three programming constructs that allow for linear control flow. These are also known as *single entry – single exit* constructs.

☞ When it is said that "the pseudocode is executed", it just means that the pseudocode instructions are interpreted. It doesn't denote the actual execution on a computer.

In the sequence structure, all instructions in the pseudocode are executed (exactly) once without skipping any. On the other hand, with selection and loop structures, it is possible to execute certain instructions repeatedly or even skip some. In such structures, the decision as to which statements are to be executed or whether the execution should repeat will be determined based on the outcome of testing a condition. We use special symbols called *relational operators* to write such conditions. The various relational operators are listed in Table 2.1. It is also possible to combine two or more conditions using *logical operators* like "**AND**" (&&), "**OR**" (||). Eg: $a > b$ **AND** $a > c$.

2.1.2.1 Sequence

This is the most elementary construct where the instructions of the algorithm are executed in the order listed. It is the logical equivalent of a straight line. Consider the code below.

```
S1
S2
S3
.
.
Sn
```

The statement `S1` is executed first, which is then followed by statement `S2`, so on and so forth, `Sn` until all the instructions are executed. No instruction is skipped and every instruction is executed only once.

2.1.2.2 Decision or Selection

A selection structure consists of a test condition together with one or more blocks of statements. The result of the test determines which of these blocks is executed. There are mainly two types of selection structures, as discussed below:

A if structure

There are three variations of the if-structure:

A.1 if structure

The general form of this structure is:

```
if (condition)
    TRUE__INSTRUCTIONS
endif
```

If the test condition is evaluated to TRUE, the statements denoted by TRUE__INSTRUCTIONS are executed. Otherwise, those statements are skipped.

Example 2.1. The pseudocode CHECKPOSITIVE(x) checks if an input value x is positive.

```
CHECKPOSITIVE(x)
1  if ($x > 0$)
2      PRINT(x,“ is positive”)
3  endif
```

A.2 if else structure

The general form is given below:

```
if (condition)
    TRUE__INSTRUCTIONS
else
    FALSE__INSTRUCTIONS
endif
```

This structure contains two blocks of statements. If the test condition is met, the first block (denoted by TRUE__INSTRUCTIONS) is executed and the algorithm skips over the second block (denoted by FALSE__INSTRUCTIONS). If the test condition is not met, the first block is skipped and only the second block is executed.

Example 2.2. The pseudocode PersonType(*age*) checks if a person is a major or not.

```
PersonType(age)
1  if (age >= 18)
2      Print("You are a major")
3  else
4      Print("You are a minor")
5  endif
```

A.3 if else if else structure

When a selection is to be made out of a set of more than two possibilities, you need to use the *if else if else structure*, whose general form is given below:

```
if (condition_1)
    TRUE__INSTRUCTIONS_1
else if (condition_2)
    TRUE__INSTRUCTIONS_2
else
    FALSE__INSTRUCTIONS
endif
```

Here, if *condition*$_1$ is met, TRUE__INSTRUCTIONS$_1$ will be executed. Else *condition*$_2$ is checked. If it evaluates to True, TRUE__INSTRUCTIONS$_2$ will be selected. Otherwise FALSE__INSTRUCTIONS will be executed.

Example 2.3. The pseudocode CompareVars(x, y) compares two variables x and y and prints the relation between them.

```
CompareVars($x, y$)
if ($x > y$)
    Print($x$,“is greater than”,$y$)
else if ($x < y$)
    Print($y$,“is greater than”,$x$)
else
    Print(“The two values are equal”)
endif
```

There is no limit to the number of **else if** statements, but in the end, there has to be an **else** statement. The conditions are tested one by one starting from the top, proceeding downwards. Once a condition is evaluated to be True, the corresponding block is executed, and the rest of the structure is skipped. If none of the conditions are met, the final **else** part is executed.

B Case Structure

The **case** structure is a refined alternative to **if else if else** structure. The pseudocode representation of the **case** structure is given below.

The general form of this structure is:

```
caseof ($expression$)
case 1 $value_1$:
    BLOCK$_1$
case 2 $value_2$:
    BLOCK$_2$
    ⋮
default :
    DEFAULT__BLOCK
endcase
```

The case structure works like this: First, the value of *expression* (you can also have a single variable in the place of *expression*) is compared with $value_1$. If there is a match, the first block of statements denoted as BLOCK$_1$ will be executed. Typically, each block will have a `break` at the end which causes the case structure to be exited.

If there is no match, the value of the expression (or of the variable) is compared with $value_2$. If there is a match here, BLOCK$_2$ is executed and the structure is exited at the corresponding `break` statement. This process continues until either a match for the expression value is found or until the end of the cases is encountered. The DEFAULT__BLOCK will be executed when the expression does not match any of the cases.

If the `break` statement is omitted from the block for the matching case, then the execution continues into subsequent blocks even if there is no match in the subsequent blocks, until either a `break` is encountered or the end of the case structure is reached.

Example 2.4. The pseudocode PRINTDIRECTION(*dir*) prints the direction name based on the value of a character called *dir*.

```
PRINTDIRECTION(dir)
caseof (dir)
    case 'N':
        PRINT("North")
        break
    case 'S':
        PRINT("South")
        break
    case 'E':
        PRINT("East")
        break
    case 'W':
        PRINT("West")
        break
    default :
        PRINT("Invalid direction code")
endcase
```

2.1.2.3 Repetition or loop

When a certain block of instructions is to be repeatedly executed, we use the repetition or loop construct. Each execution of the block is called an **iteration** or a **pass**. If the number of iterations (how many times the block is to be executed) is known in advance, it is called **definite iteration**. Otherwise, it is called **indefinite** or **conditional iteration**. The block that is repeatedly executed is called the *loop body*. There are three types of loop constructs as discussed below:

A while loop

A **while** loop is generally used to implement indefinite iteration. The general form of the **while** loop is as follows:

```
while (condition)
    TRUE_INSTRUCTIONS
endwhile
```

Here, the loop body (TRUE_INSTRUCTIONS) is executed repeatedly as long as *condition* evaluates to TRUE. When the condition is evaluated as FALSE, the loop body is bypassed.

B repeat-until loop

The second type of loop structure is the **repeat-until** structure. This type of loop is also used for indefinite iteration. Here the set of instructions constituting the loop body is repeated as long as *condition* evaluates to FALSE. When the condition evaluates to TRUE, the loop is exited. The pseudocode form of **repeat-until** loop is shown below.

```
repeat
    FALSE__INSTRUCTIONS
until (condition)
```

There are two major differences between **while** and **repeat-until** loop constructs:

1. In the **while** loop, the pseudocode continues to execute as long as the resultant of the condition is TRUE; in the **repeat-until** loop, the looping process stops when the resultant of the condition becomes TRUE.

2. In the **while** loop, the condition is tested at the beginning; in the **repeat-until** loop, the condition is tested at the end. For this reason, the while loop is known as an *entry controlled loop* and the repeat-until loop is known as an *exit controlled loop.*

You should note that when the condition is tested at the end, the instructions in the loop are executed at least once.

C for loop

The **for** loop implements definite iteration. There are three variants of the **for** loop. All three **for** loop constructs use a variable (call it the *loop variable*) as a counter that starts counting from a specific value called *begin* and updates the loop variable after each iteration. The loop body repeats execution until the loop variable value reaches *end*. The first **for** loop variant can be written in pseudocode notation as follows:

```
for var = begin to end
    LOOP__INSTRUCTIONS
endfor
```

Here, the loop variable (*var*) is first assigned (initialized with) the value *begin*. Then the condition *var* <= *end* is tested. If the outcome is TRUE, the loop body is executed. After the first iteration, the loop variable is incremented (increased by 1). The condition *var* <= *end* is again tested with the updated value of *var* and the loop is entered (loop body is executed), if the condition evaluates to TRUE. This process of updating the loop variable after an iteration and proceeding with the execution if the condition (tested with the updated value of the loop variable) evaluates to TRUE continues until the counter value becomes greater than *end*. At that time, the condition evaluates to FALSE and the loop execution stops.

In the second **for** loop variant, whose pseudocode syntax is given below, the loop variable is decremented (decreased by 1) after every iteration. And the condition being tested is *var* >= *end*. Here, *begin* should be greater than or equal to *end*, and the loop exits when this condition is violated.

```
for var = begin downto end
    LOOP__INSTRUCTIONS
endfor
```

It is also possible to update the loop variable by an amount other than 1 after every iteration. The value by which the loop variable is increased or decreased is known as *step*. In the pseudocode shown below, the *step* value is specified using the keyword **by** .

```
for var = begin to end by step
    LOOP__INSTRUCTIONS
endfor
```

Table 2.2 lists some examples of **for** loops. In these examples, *var* is the loop variable.

Table 2.2: **for** loop examples

Loop construct	**Description**	**Values taken by** *var*
for *var* = 1 **to** 10	*var* gets incremented by 1 till it reaches 10	$1, 2, \cdots 9, 10$
for *var* = 10 **downto** 1	*var* gets decremented by 1 till it reaches 1	$10, 9, \cdots 2, 1$
for *var* = 2 **to** 20 **by** 2	*var* gets increased by 2 till it reaches 20	$2, 4, \cdots 18, 20$
for *var* = 20 **downto** 2 **by** 2	*var* gets decreased by 2 till it reaches 2	$20, 18, \cdots 4, 2$

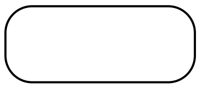
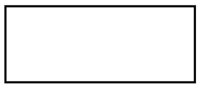
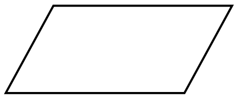
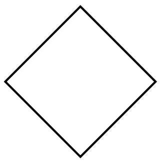
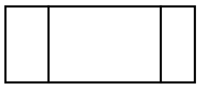
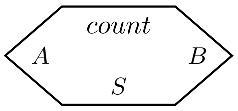
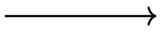
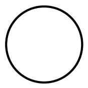
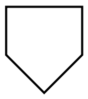
2.2 Flowcharts

A flowchart is a diagrammatic representation of an algorithm that depicts how control flows in it. Flowcharts are composed of various *blocks* interconnected by *flow-lines*. Each block in a flowchart represents some stage of processing in the algorithm. Different types of blocks are defined to represent the various programming constructs of the algorithm.

Flow lines indicate the order in which the algorithm steps are executed. The flow lines entering a block denote data (or control) flow into the block and the flow lines emerging from a block denote data (control) outflow. Most blocks have only single incoming and outgoing flow lines. The exception is for blocks representing selection and loop constructs. Such blocks have multiple exits, one for each possible outcome of the condition being tested and each such outcome is called a *branch*.

Table 2.3 lists some commonly used flowchart symbols and their descriptions.

Table 2.3: The different flowchart symbols

Flowchart symbol	**Description**
	Flattened ellipse indicates the start and end of a module.
	Rectangle is used to show arithmetic calculations.
	Parallelogram denotes an input/output operation.
	Diamond indicates a decision box with a condition to test. It has two exits. One exit leads to a block specifying the actions to be taken when the tested condition is TRUE and the other exit leads to a second block specifying the actions for FALSE case.
	Rectangle with vertical side-lines denotes a module. A module is a collection of statements written to achieve a task. It is known by the name *function* in the programming domain.
	Hexagon denotes a **for** loop. The symbol shown here is the representation of the loop: **for** *count* = *A* **to** *B* **by** *S*.
	Flowlines are indicated by arrows to show the direction of data flow. Each flowline connects two blocks.
	This indicates an on-page connector. This is used when one part of a long flowchart is drawn on one column of a page and the other part in the other column of the same page.
	This indicates an off-page connector. This is used when the flowchart is very long and spans multiple pages.

2.3 Solved problems - Algorithms and Flowcharts

Problem 2.1 To find simple interest.

Solution:

See Figure 2.1 for the algorithm and flowchart.

```
SimpleInterest
1  Start
2  Read($principal, rate, years$)
3  $SI = (principal * rate * years)/100$
4  Print($SI$)
5  Stop.
```

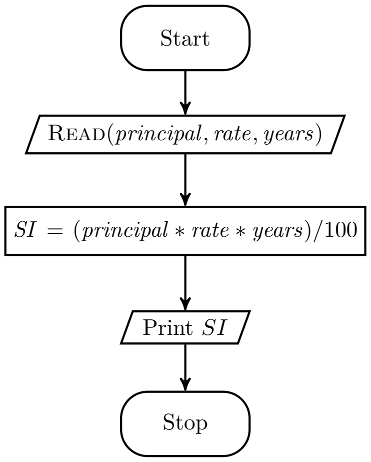

Start
Read(principal, rate, years)
SI = (principal ∗ rate ∗ years)/100
Print SI
Stop

Figure 2.1: To find simple interest

- Read the principal amount, interest rate, and period values. Store them as three variables: *principal*, *rate*, and *years* respectively.
- Multiply *principal*, *rate* and *years* and divide the result by 100 to obtain the simple interest, which is stored in the variable *SI*.
  - Division by 100 is necessary since *rate* is input as a percentage(7% instead of 0.07).
- Finally, the value of *SI* is displayed on the terminal.

Problem 2.2 To determine the larger of two numbers.

Solution:

See Figure 2.2 for the algorithm and flowchart.

```
LargerTwo
1 Start
2 Read$(a, b)$
3 if $(a > b)$
4         $large = a$
5 else
6         $large = b$
7 endif
8 Print$(large)$
9 Stop.
```

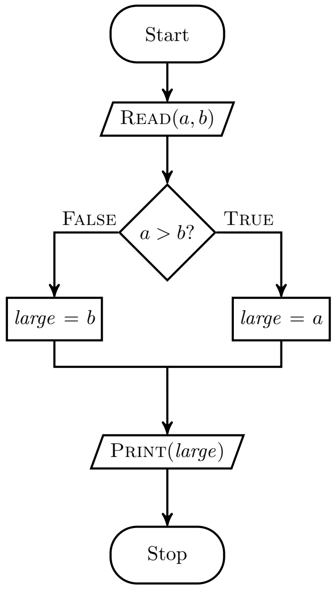


Figure 2.2: To find the larger of two numbers

- First, read two numbers from the user and store them in the variables a and b respectively.
- Next, compare these two numbers using an if-else statement.
- Check whether a is greater than b.
    - If this condition is True, assign the value of a to the variable *large*.
- Otherwise, the control moves to the else part, where the value of b is assigned to *large*.
- Finally, the value of *large* is printed, which is the largest of the two numbers.

Problem 2.3 To determine the smallest of three numbers.

Solution:

See Figure 2.3 for the algorithm and flowchart.

```
SMALLESTTHREE
1  Start
2  READ($a, b, c$)
3  if ($a < b$)
4       $small = a$
5  else
6       $small = b$
7  endif
8  if ($c < small$)
9       $small = c$
10 endif
11 PRINT($small$)
12 Stop.
```

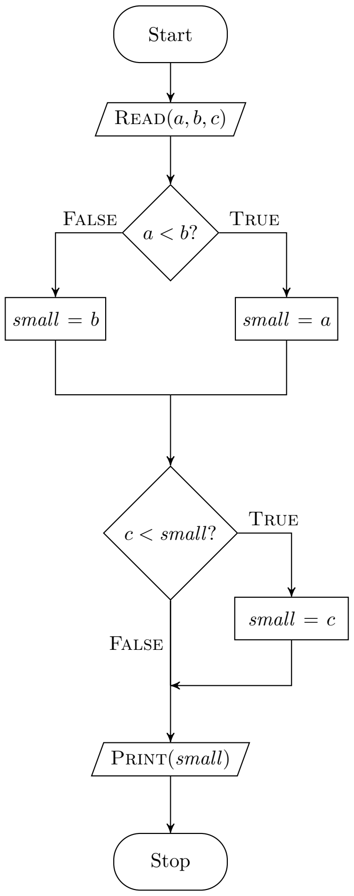


Figure 2.3: To find the smallest of three numbers

- Similar to the previous problem, input three numbers and store them in variables a, b, and c respectively.
- To solve this problem, first find the smaller of the two numbers a and b and then compare that smaller number with the third variable c.
- The first if statement determines the smaller between a and b and keeps it in *small*.
- Using a second if statement, check whether c is less than *small*.
  - If so, the value of c is assigned to *small*.
- Finally, *small* is printed.

Problem 2.4 To determine the entry-ticket fare in a zoo based on age as follows:

Age	Fare
< 10	7
$>= 10$ and < 60	10
$>= 60$	5

Solution:

See Figure 2.4 for the pseudocode and flowchart.

```
TICKETFARE
1   Start
2   READ(age)
3   if (age < 10)
4           fare = 7
5   else if (age < 60)
6           fare = 10
7   else
8           fare = 5
9   endif
10  PRINT(fare)
11  Stop
```

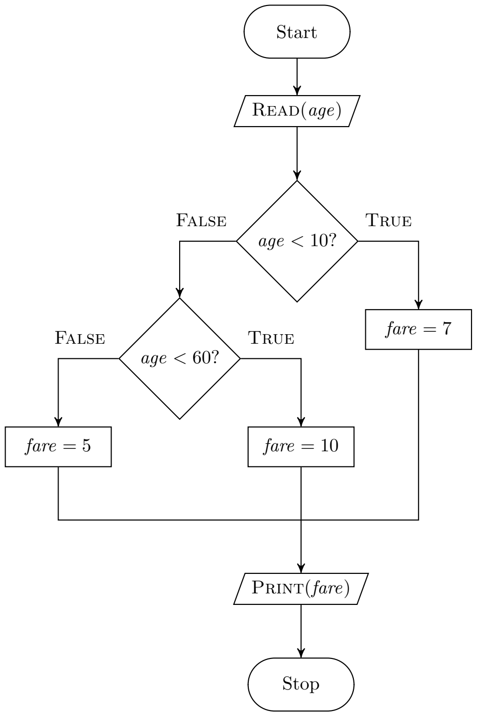


Figure 2.4: To determine the entry fare in a zoo

- Accept *age* from the user.
- First check whether *age* is less than 10. If so, *fare* is assigned the value 7.
- If the condition is FALSE, the next condition is checked. If *age* $>=$ 10 and *age* $<$ 60, *fare* gets the value 10.
- If the above condition also turns out to be FALSE (i.e. *age* $>=$ 60), else statement is executed, and *fare* gets the value 5.
- Finally, the *fare* value is printed.

Problem 2.5 To print the colour based on a code value as follows:

Grade	Message
R	Red
G	Green
B	Blue
Any other value	Wrong code

Solution:

Figure 2.5 for the pseudocode and flowchart.

```
PrintColour
Start
Read(code)
caseof (code)
    case 'R':
        Print("Red")
        break
    case 'G':
        Print("Green")
        break
    case 'B':
        Print("Blue")
        break
    default :
        Print("Wrong code")
endcase
Stop
```

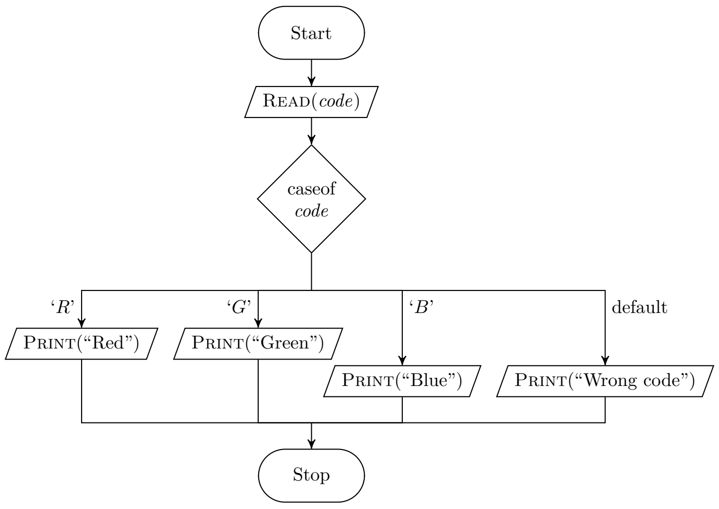


Figure 2.5: To print colors based on a code value

- Read the *code* (a character constant) from the user.
- The value of *code* is matched against a number of case constants (*R*, *G*, *B* as per the question).
- If the *code* is '*R*', the statements associated with it are executed until the **break** statement is encountered (i.e. "Red" is printed here). A **break** statement moves the control out of the case structure.
- A similar execution happens if the value of *code* is *G* or *B*. If *code* is '*G*', "Green" is printed, and "Blue" is printed if *code* is '*B*'.
- If the user inputs any other character other than '*R*', '*G*', or '*B*', the default statement is executed.

Problem 2.6 To print the numbers from 1 to 50 in descending order.

Solution:

See Figure 2.6 for pseudocode and flowchart.

```
PRINTDOWN
1  Start
2  for count = 50 downto 1
3        PRINT(count)
4  endfor
5  Stop
```

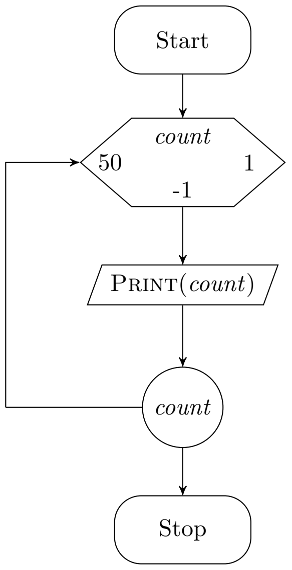


Figure 2.6: To print numbers in descending order

- The *count* variable is initially assigned 50 and the condition, *count* >= 1 (i.e. 50 >= 1) is checked.
- Since it evaluates to TRUE, print statement is executed (Body of for loop in this question).
- '**downto**' decrements the value of *count* by 1 i.e. *count* now becomes 49.

- The condition, *count* >= 1 is checked, and since it evaluates to TRUE, the body of the loop is executed (49 is printed), and again *count* is decremented.
- The above step repeats until *count* becomes 0 (in which case, the condition evaluates to FALSE) and you stop. The sequence thus printed is 50 49 48 ⋯ 1.

Problem 2.7 To find the factorial of a number.

Solution: The factorial of a number n is defined as $n! = n \times n-1 \times \cdots\cdots \times 2 \times 1$.

See Figure 2.7 for the pseudocode and flowchart.

```
FACTORIAL
1  Start
2  READ(n)
3  fact = 1
4  for var = n downto 1
5        fact = fact * var
6  endfor
7  PRINT(fact)
8  Stop
```

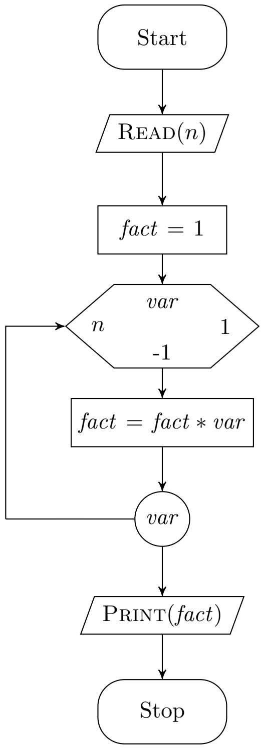


Figure 2.7: To find the factorial of a number

- Read n from the user, whose factorial is to be calculated.
- Initialize *fact* to 1
- Write a loop, with the loop control variable *var* initialized to n.
  - For each iteration of the loop, multiply the value inside *fact* variable with *var* and store it in*fact*.

- This is continued until the value of *var* becomes greater than or equal to 1, with *var* being decremented by 1 after every iteration.
- At the end of all iterations, *fact* is printed.

Problem 2.8 To determine the largest of n numbers.

Solution: See Figure 2.8 for pseudocode and flowchart.

```
LargeN
Start
Read($n$, $num$)
$large = num$
for $count = 1$ to $n - 1$
    Read($num$)
    if ($num > large$)
        $large = num$
    endif
endfor
Print($large$)
Stop
```

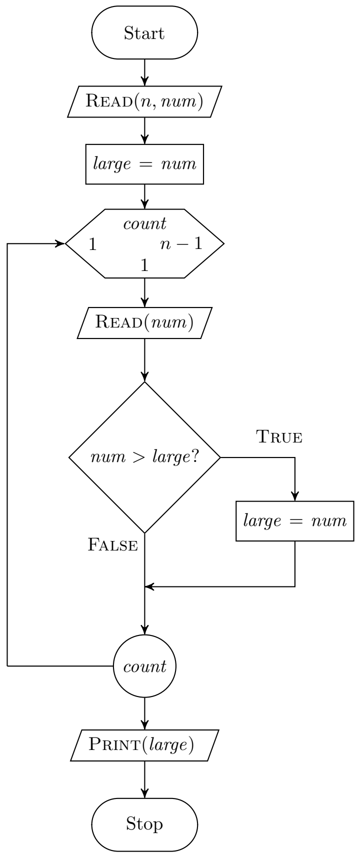


Figure 2.8: To find the largest of n numbers

- Read n from the user.
- Along with that, read a single number *num* from the user and assign a variable *large* with *num*. That is the first one among the set of input integers is assumed to be the largest.
- Initialize a loop, with the loop control variable *count* being assigned the value of 1.

- During every iteration of the loop,
  - obtain a number from the user and keep it in *num* variable
  - *num* is checked against *large*
  - If $num > large$, update *large*to *num*
- This is repeated as long as the value of *count* is less than or equal to *n* - 1.
  - You need to iterate the loop only *n* - 1 times since you received the first integer before entering the loop.
- After all iterations, the largest value is displayed.

Problem 2.9 To determine the average age of students in a class. The user will stop giving the input by giving the age as 0.

Solution: See Figure 2.9 for pseudocode and flowchart.

```
AVERAGEAGEV1
1   Start
2   $sum = 0$
3   $count = 0$
4   READ($age$)
5   while ($age$!=0)
6           $sum = sum + age$
7           $count = count + 1$
8           READ($age$)
9   endwhile
10  $average = sum/count$
11  PRINT($average$)
12  Stop
```

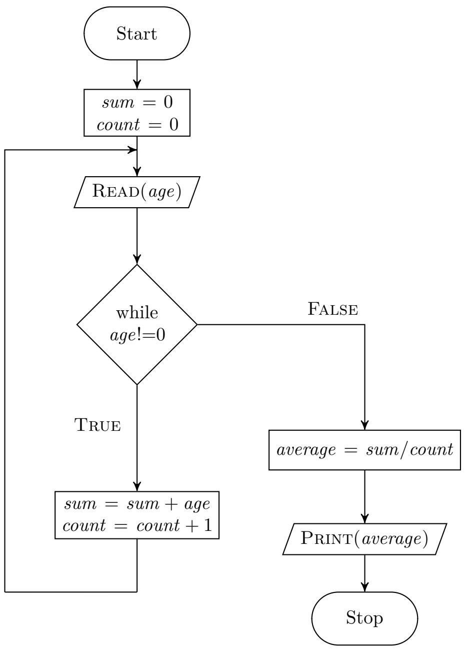


Figure 2.9: To determine the average age using **while** loop

- Initialize the variables *sum* and *count* to 0.
- Read the age of the first student and keep it in *age* variable.

- Write a while loop that will run until *age* is not equal to zero.
- In every iteration,
  - add the *age* value to *sum* and increment *count* by 1
  - read the next value of *age*
- After getting out of the loop, determine the average age by dividing *sum* by *count*.
- Print the average value.

Problem 2.10 Redo Problem 2.9 using repeat-until loop construct.

Solution: See Figure 2.10 for flowchart and pseudocode.

```
AVERAGEAGEV2
1   Start
2   $sum = 0$
3   $count = 0$
4   READ($age$)
5   repeat
6       $sum = sum + age$
7       $count = count + 1$
8       READ($age$)
9   until ($age == 0$)
10  $average = sum/count$
11  PRINT($average$)
12  Stop
```

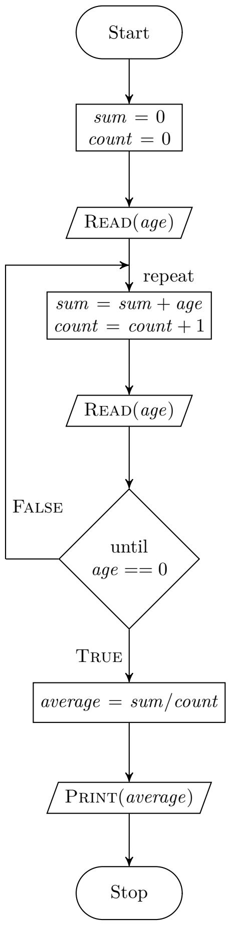


Figure 2.10: To determine the average age using **repeat until** loop

- Initialize *sum, count* to 0.
- Read *age* from the user.
- Write the repeat-until loop:
- Inside the loop:
  - add *age* value to *sum*
  - increment *Count*
  - read the next *age* value
- Continue the loop until the user inputs a 0 for *age*.
- After getting out of the loop, determine the average age by dividing *sum* by *count*.
- Print the average value.

Problem 2.11 To find the average height of boys and average height of girls in a class of n students.

Solution: See Figure 2.11.

```
AVERAGEHEIGHT
Start
READ($n$)
$btotal = 0$
$bcount = 0$
$gtotal = 0$
$gcount = 0$
for $var = 1$ to $n$
    READ($gender, height$)
    if ($gender ==$ 'M')
        $btotal = btotal + height$
        $bcount = bcount + 1$
    else
        $gtotal = gtotal + height$
        $gcount = gcount + 1$
    endif
endfor
$bavg = btotal/bcount$
$gavg = gtotal/gcount$
PRINT($bavg, gavg$)
Stop
```

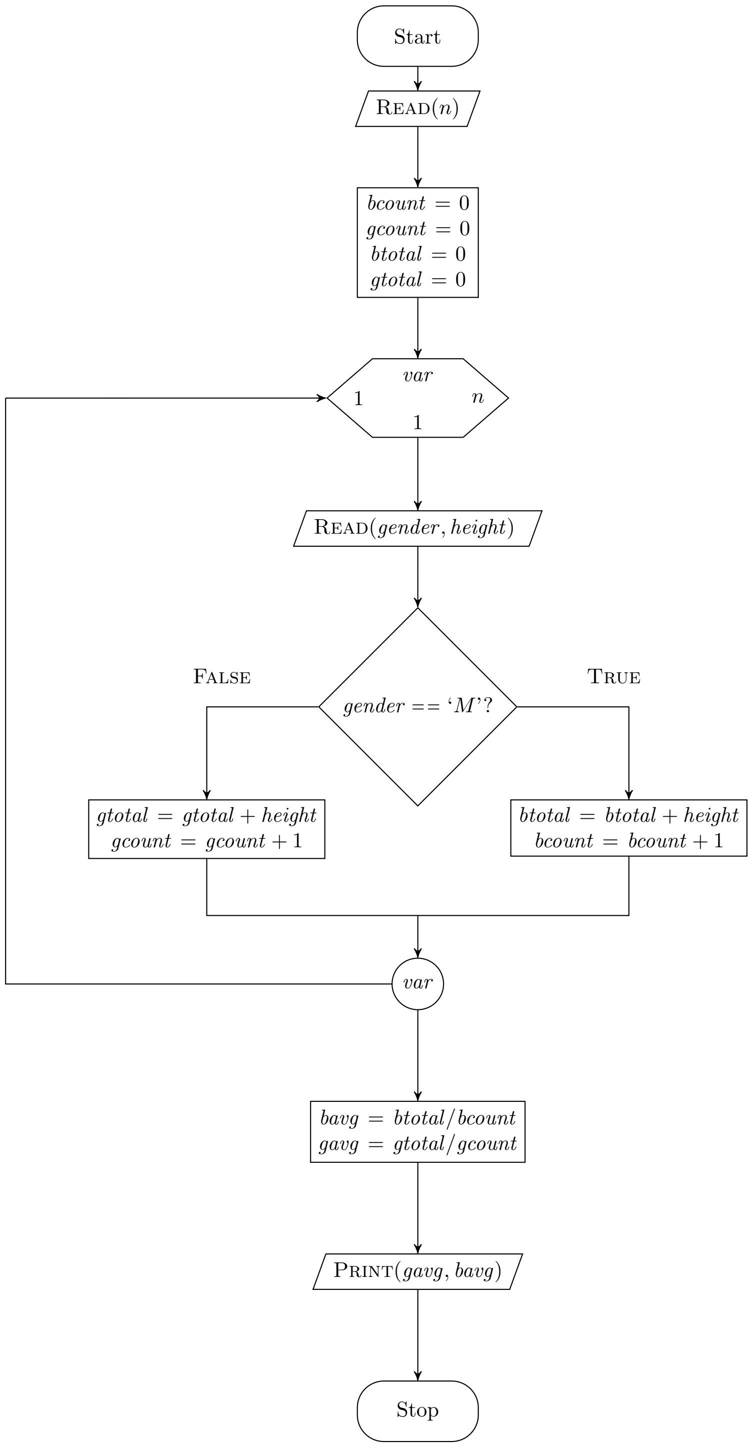


Figure 2.11: To find the average height of boys and girls

- Input *n* from the user.
- Define variables *btotal*, *bcount*, *gtotal*, and *gcount* to store the total height of boys, number of boys, total height of girls and number of girls respectively. Initialize all the four variables to 0
- Write a loop that prompts the user for the relevant inputs for all the *n* students.
- During each iteration:
  - Read *gender* and *height*
  - If the gender is '*M*', add the input height to *btotal* and increment *bcount*
  - Otherwise, add the input height to *gtotal* and increment *gcount*
- After getting out of the loop, determine the average height of the boys, *bavg* by dividing *btotal* with *bcount*. In a similar way, calculate the average height of the girls, *gavg* by dividing *gtotal* with *gcount*.
- Finally print the average values.

2.4 Conclusion

In this chapter, we've explored the foundational elements of algorithms, flowcharts, and pseudocode — the essential tools in the development of efficient and effective programs. Algorithms provide a clear, step-by-step approach to problem-solving, while flowcharts offer a visual representation that aids in understanding the logical flow. Pseudocode serves as an intermediary step, bridging the gap between the abstract algorithm and the actual code, making it easier to translate ideas into executable code. Together, these tools help in breaking down complex problems into manageable parts, ensuring clarity and precision in programming. Mastering these concepts is crucial for anyone aiming to develop robust software solutions. As you move forward, these skills will serve as a solid foundation for tackling more complex problems.

2.5 Exercises

1. Write algorithms for the following:
   1. to find the area and circumference of a circle.
   2. to find the area of a triangle given its three sides.
   3. to find the area and perimeter of a rectangle.
   4. to find the area of a triangle given its length and breadth.

2. If the three sides of a triangle are input, write an algorithm to check whether the triangle is isosceles, equilateral, or scalene.

3. Write a switch statement that will examine the value of `flag` and print one of the following messages, based on the value assigned to the flag.

Flag value	Message
1	HOT
2	LUKE WARM
3	COLD
Any other value	OUT OF RANGE

4. Write algorithms for the following:

   (a) to display all odd numbers between 1 and 500 in descending order.

   (b) to compute and display the sum of all integers that are divisible by 6 but not by 4 and that lie between 0 and 100.

   (c) to read a value, and do the following: If the number is even, halve it; if it's odd, multiply by 3 and add 1. Repeat this process until the value is 1, printing out each value.

5. Write an algorithm that inputs two values a and b and that finds a^b. Use the fact that a^b is multiplying a with itself b times.

6. You visit a shop to buy a new mobile. In connection with the festive season, the shop offers a 10% discount on all mobiles. In addition, the shop also gives a flat exchange price of ₹ 1000 for old mobiles. Draw a flowchart to input the original price of the mobile and print its selling price. Note that all customers may not have an old mobile for exchange.

7. Draw flowcharts for the following:

   (a) to find the volume of a hemisphere by inputting the radius.

   (b) to find the profit or loss incurred by getting the cost price and selling price of an item. Note that you are not asked to determine whether profit or loss is incurred but rather the value of profit or loss. Assume cost price $\neq$ selling price.

   (c) to find the average of a list of numbers entered by the user. The user will stop the input by giving the value -999.

Bibliography

[1] Maureen Sprankle and Jim Hubbard, Problem Solving and Programming Concepts, Pearson, 2011.

[2] Thomas H. Cormen and Charles E. Leiserson and Ronald L. Rivest and Clifford Stein, Introduction to Algorithms, The MIT Press, 2009.

[3] Michael T. Goodrich and Roberto Tamassia, Algorithm Design and Applications, Wiley, 2015.

[4] Richard F. Gilberg and Behrouz A. Forouzan, Data Structures: A Pseudocode Approach With C, Cengage Learning, 2004.

[5] Donald E. Knuth, The Art of Computer Programming (Volume 1), Addison-Wesley, 1997.

Chapter 3

Foundations of computing

"The problem of computing is not about speed, it's about understanding."
– Donald E Knuth

3.1 Introduction

The computer is an advanced electronic device that takes raw data as *input* from the user and processes this data under the control of a set of instructions (called *program*) and gives the result (output) and saves the output for future use. It can perform numerical (arithmetic) and non-numerical (logical) calculations.

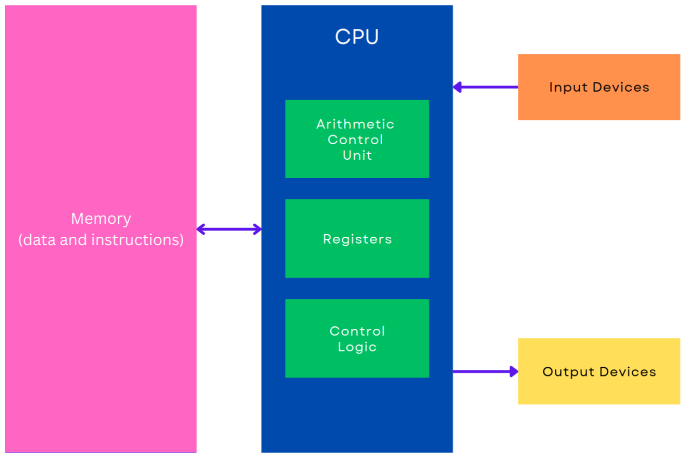


Figure 3.1: A typical computer architecture

3.2 Architecture of a computer

A typical computer architecture comprises of three main components:

1. Input/Output (I/O) Unit,
2. Central Processing Unit (CPU), and
3. Memory Unit.

The I/O unit consists of the input unit and output devices. The input devices accept data from the user, which the CPU processes. The output devices transfer the processed data (information) to the user. The memory unit is used to store the input data, the instructions required to process the input, and also the output information. Figure 3.1 illustrates a typical architecture of a computer.

3.2.1 Input/output unit

The user interacts with the computer via the I/O unit.

3.2.1.1 Input devices

Input devices allow users to input data into the computer for processing. They are necessary to convert the input data into a form that can be understood by the computer. The data input to a computer can be in the form of text, audio, video, etc. Input devices are classified into two categories:

1. Human data entry devices – the user enters data into the computer by typing or pointing a device to a particular location. Some examples are
   (a) Keyboard – the most common typing device.
   (b) Mouse – the most common pointing device. You move the mouse around on a *mouse pad* and a small pointer called *cursor* follows your movements on the computer screen.
   (c) Track ball – an alternative to a mouse which has a ball on the top. The cursor on the computer screen moves in the direction in which the ball is moved.
   (d) Joystick – has a stick whose movement determines the cursor position.
   (e) Graphics tablet – converts hand-drawn images into a format suitable for computer processing.
   (f) Light pen – can be used to "draw" on the screen or to select options from menus presented on the screen by directly pointing the pen on the screen.
2. Source data entry devices –They use special equipment to collect data at the source, create machine-readable data, and feed them directly into the computer. This category comprises:

(a) Audio input devices – It uses human voice or speech to give input. Microphone is an example.

(b) Video input devices – They accept input in the form of video or images. Video cameras, webcams are examples.

(c) Optical input devices – They use optical technology (light source) to input the data into computers. Some common optical input devices are scanners and bar code readers.

☞ Other examples include Optical Character Recogniser (OCR), Magnetic Ink Character Recogniser (MICR), and Optical Mark Recogniser (OMR).

3.2.1.2 Output devices

An output device takes processed data from the computer and converts them into information that can be understood by humans. The output could be on paper or a film in a tangible form or an intangible form like audio, video, etc. Output devices are classified as follows:

1. Hard copy devices – The output obtained in a tangible form on paper or any surface is called hard copy output. The hard copy can be stored permanently and is portable. The hard copy output can be read or used without a computer. Examples in this category include:

   (a) Printer – prints information on paper. The information could be textual or even images. Drum printers, laser printers, and inkjet printers are some commonly used printer types.

   (b) Plotter – used to produce very large drawings on paper sizes up to A0 (16 times as big as A4). A plotter draws onto the paper using very fine pens. Flatbed plotters and drum plotters are two types of plotters.

   (c) COM (Computer Output on Microfilm) – stores the output (mostly as images) on a microfilm.

2. Soft Copy Devices – Generates a soft copy of the output (output obtained in an intangible form) on a visual display, audio unit, or video unit. The soft copy can be stored and sent via e-mail to other users. It also allows corrections to be made. The soft copy output requires a computer to be read or used. This category comprises:

   (a) Monitor – the primary output device of a computer. Monitors can be of two types: monochrome (black and white) or color. It forms images from tiny dots, called *pixels*, that are arranged in a rectangular form.

   (b) Video output devices – produce output in the form of video or images. An example is a screen image projector or data projector that displays information. from the computer onto a large white screen.

(c) Audio output devices – speakers, headsets, or headphone, are used for audio output (in the form of sound) from a computer. The signals are sent to the speakers via a *sound card* that translates the digital sound back into analog signals.

3.2.2 Central Processing Unit

The Central Processing Unit (CPU) or the processor, is often known as the *brain* of a computer. It consists of an Arithmetic Logic Unit (ALU) and a Control Unit (CU). In addition, the CPU also has a set of *registers* which are temporary storage areas for holding data and instructions.

3.2.2.1 Arithmetic Logic Unit

This unit consists of two sub-units, namely arithmetic and logic units.

1. **Arithmetic unit** – performs arithmetic operations like addition, subtraction, multiplication, and division, on the data.

2. **Logic unit** – performs logical operations like comparisons of data values.

3.2.2.2 Registers

Registers are high-speed storage areas within the CPU but have the least storage capacity. They store data, instructions, addresses, and intermediate results of processing. Some of the commonly used registers are:

- Accumulator (ACC) – stores the result of arithmetic and logic operations.

- Instruction Register (IR) – holds the instruction that is currently being executed.

- Program Counter (PC) – holds the address of the next instruction to be processed.

- Memory Address Register (MAR) – contains the address of the data to be fetched.

- Memory Buffer Register (MBR) – temporarily stores data fetched from memory or the data to be sent to memory.

  ☞MBR is also called a Memory Data Register (MDR).

- Data Register (DR) stores the operands and any other data.

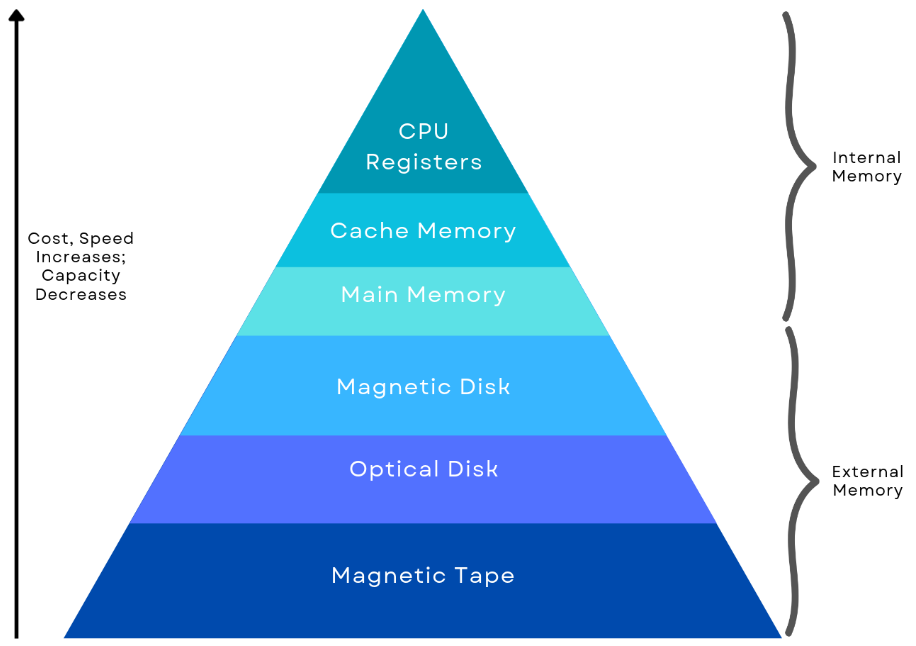


Figure 3.2: Memory hierarchy

3.2.2.3 Control Unit

This unit manages and coordinates the operations of all parts of the computer but does not carry out any actual data processing operations. The functions of this unit are:

1. generate *control signals* that controls various operations of the computer.
2. obtain instructions from the memory, interpret them, and then direct the ALU to execute those instructions.
3. communicate with I/O devices for transfer of data or results from/to memory.
4. decides when to fetch the data and instructions, what operation to perform, where to store the results, the ordering of various events during processing etc.

3.2.3 Memory unit

Memory is the storage space in a computer where data to be processed and instructions required for processing are stored. The various memories can be organized hierarchically called *memory hierarchy* as shown in Figure 3.2. As we ascend the memory hierarchy,

- Capacity in terms of storage decreases.
- Cost per bit of storage increases.
- Frequency of access of the memory by the CPU increases.
- Access time decreases.

Memory is primarily of two types :

1. Internal Memory: memories that reside on the motherboard.
2. External memory: memories that are outside the motherboard.

3.2.3.1 Internal Memory

Internal memory includes:

1. Registers – high-speed storage areas within the CPU.
2. Primary memory – main memory of the computer. It is categorized into two:

   (a) Random Access Memory (RAM) – used for storing data and instructions during the operation of a computer. Data to be processed are brought to RAM from input devices or secondary memory. After processing, the results are stored in RAM before being sent to the output device.

   ☞RAM is often referred to as *volatile memory*, since the data stored in RAM are lost when you switch off the computer or if there is a power failure.

   ☞Dynamic RAM (DRAM) and Static RAM (SRAM) are the two types of RAM

   .

   (b) Read Only Memory (ROM) – a non-volatile primary memory. It does not lose its content when the power is switched off. ROM, as the name implies, has only read capability and no write capability. After the information is stored in ROM, it is permanent and cannot be modified. Therefore, ROM is used to store the data that do not require a change, for example, the *boot information* (information required while starting the computer when it is switched on).

☞PROM (Programmable Read Only Memory), EPROM (Erasable and Programmable Read Only Memory), EEPROM (Electrically Erasable and Programmable Read Only Memory), and UVEPROM (Ultra-Violet Erasable and Programmable Read Only Memory) are the various ROM types.

3. Cache memory – placed between RAM and CPU and stores the data and instructions that are frequently used . During data processing, the CPU first checks the cache for the required data or instruction. If found in the cache, the data or instructions are retrieved from the cache itself. Otherwise, they are then retrieved from RAM.

3.2.3.2 External Memory

External memory includes:

1. Magnetic tape – a plastic tape with magnetic coating mounted on a reel or in a cassette. It is a sequential access device, meaning that the data can be read only in the order in which they are stored.
2. Magnetic disk – a thin plastic or metallic circular plate coated with magnetic oxide and encased in a protective cover. Hard disk is an example.

   ☞A magnetic disk is a direct-access secondary storage device, ie., you can directly access the location you want, without accessing the previous locations.

3. Optical disk – a flat circular disk coated with reflective plastic material that can be altered by laser light. The data bits 1 and 0 are stored as spots that are relatively bright and light, respectively. CDs and DVDs are examples.

3.3 Von Neumann architecture

If you want to perform some processing on data, they must be stored in computer memory. Similarly, the instructions that process the data must also be stored in the memory. This concept of storing programs in computer memory is known as **stored program concept**. A computer based on *Von Neumann architecture* stores data and instructions in the same memory.

☞As an alternative to *Von Neumann architecture*, you have the *Harvard architecture*, wherein there are separate memories for storing programs and data (code memory for programs and data memory for data). It also utilises *stored program concept.*

3.4 Instruction execution

An instruction 'instructs' the processor to perform an elementary operation. Curious to know what an instruction looks like and how it is executed? Read on!

3.4.1 Instruction format

An instruction format defines the layout of an instruction, in terms of its constituent parts. The set of instructions that a computer processor can understand and execute is called its *Instruction Set Architecture* or ISA in short. Each processor has its own ISA. Irrespective of the ISA, an instruction can be thought to be divided into various parts called *fields.* The most common fields of instruction are *Operation code (opcode)* and *Operand code* as shown below.

opcode	operand code

The remainder of the instruction fields differ from one ISA to another. The operation code represents the action the processor must perform. The operand code defines the data (operands) on which the operations are to be performed. It directly specifies the value of the operands or tells the locations where to fetch the operands from.

3.4.2 Instruction execution cycle

CPU executes an instruction in a series of steps called *instruction execution cycle.* An instruction cycle (sometimes called a fetch–decode–execute cycle) is the process by which a computer retrieves a program instruction from its memory, determines what actions to perform, and carries out those actions to produce meaningful output. The instruction cycle which starts with fetching the instruction itself is shown in Figure 3.3.

The cycle comprises the following steps:

1. Fetch the instruction

   1.1 Fetch the instruction from memory whose address is currently held in the program counter.

   1.2 Store the fetched instruction in the instruction register.

   1.3 Increment the program counter so that it has the address of the next instruction to be fetched.

2. Decode the instruction
   - 2.1 Based on the instruction set architecture, the instruction is broken into opcode and operand codes.
   - 2.2 The operation to be performed is thus identified.
3. Execute the instruction
   - 3.1 The operation identified in the decode step is now performed by the CPU.
4. Store the result
   - 4.1 The result generated by the operation is stored in the main memory, or sent to an output device.

This cycle is then repeated for the next instruction.

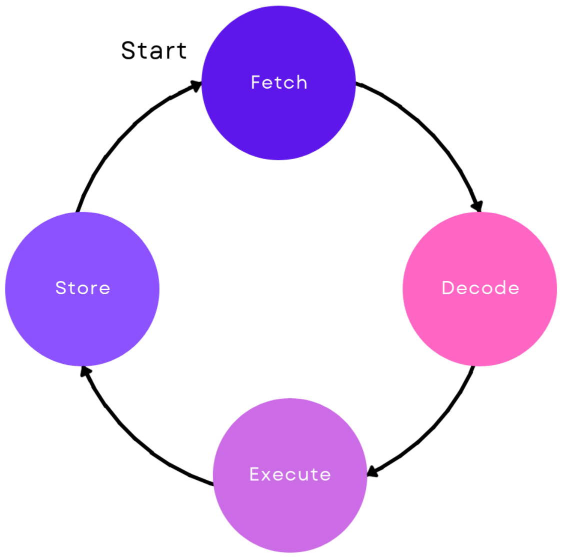


Figure 3.3: Instruction execution cycle[1]

3.5 Programming languages

Programming languages are used to write programs that are precise representations of algorithms and control the behaviour of a computer. Each language has a unique set of keywords (words that it understands) and syntax (set of rules) to organize the program instructions.

Programming languages fall into three categories:

[1]Image courtesy: Slightly adapted from *Computer Fundamentals, Anita Goel* and redrawn.

1. Machine language:- is what the computer can understand, but it is difficult for the programmer to understand.

   ☞Machine languages consist of binary numbers only.

2. High-level language:- is easier to understand and use for the programmer but difficult for the computer.

   ☞The programs written in high-level languages contain English-like statements as well as programming constructs specific to the language.

3. Assembly language:- falls in between machine language and high-level language. It is similar to machine language, but easier to write code because it allows the programmer to use symbolic names (like `ADD`, `SUB`) for operations.

   ☞Machine languages and assembly languages are also called **low-level languages**.

3.5.1 Machine Language

A program written in machine language is a collection of binary digits or bits (strings of 0's and 1's) that the computer reads and interprets. It is also referred to as *machine code* or *object code*. Some features of a program written in machine language are:

1. The computer can understand the programs written in machine language directly. No translation of the program is needed.

2. Program written in machine language can be executed very fast (Since no translation is required).

3. Machine language is defined by the hardware of a computer. It depends on the underlying processor, memory arrangement, operating systems, and peripheral devices; and is thus machine-dependent. A machine-level program written for one type of computer may not work on another type.

4. Each object code instruction has an **opcode** field that specifies the actual operation (such as add or compare) and some other fields for the operands.

   Example 3.1. Assume that the opcode and operands occupy 4 bits each and that the opcode for addition is 1010, then to add 3 and 6, the binary code would be $\underbrace{1010}_{add}\underbrace{0011}_{3}\underbrace{0110}_{6}$

3.5.2 Assembly Language

A program written in assembly language uses a symbolic representation of machine codes. The opcodes are replaced by symbolic names called **mnemonic codes** that are much easier to remember. The mnemonics for various operations are decided by the processor manufacturer and cannot be changed by the programmer. Some of the features of assembly code (program written in assembly language) are:

1. Assembly language programs are easier to write than machine language programs since assembly language programs use short, English-like representations of machine code.

   Example 3.2. If the mnemonic code for addition is `ADD`, then to add 3 and 6, the assembly level code will be `ADD 3,6`.

2. Assembly language programs are also machine-dependent.

3. Although assembly language programs use symbolic representation, they are still difficult to write. They are generally employed when efficiency matters.

3.5.3 High-level Language

A high-level language is a programming language such as C, FORTRAN, or Pascal that enables a programmer to write programs that are more or less independent of a particular type of hardware. Such languages are considered high-level because they stand closer to humans but farther from machine languages. Some of the features of programs written in high-level language are as follows:

1. Programs are easier to write, read, and understand in high-level languages than in machine language or assembly language.

2. Most of the operations like arithmetic and logical operations are denoted by symbols called *operators*.

   Example 3.3. To add 3 and 6, the high level language code will be `3+6`. Here '`+`' is the operator for addition.

3. The programs written in high-level languages are easily portable from one computer to another, since they are not machine-dependent

3.6 Translator software

The computer can understand only machine code (strings of 0's and 1's). Thus when the program is written in a language other than machine language (assembly or high-level languages), the program is to be converted to machine code. This conversion is called **translation** and is performed by the translator software.

☞In the translation process, the original program is called *source code*, and the translated code (object code) is the *target code.*

There are three types of translator software as discussed below.

3.6.1 Assembler

The assembler converts a program written in assembly language into machine code. There is usually a one-to-one correspondence between the assembly statements and the machine language instructions.

☞The machine language is dependent on the processor architecture. Thus, the converted assembly language programs also differ for different computer architectures.

3.6.2 Compiler

The compiler is a software that translates programs written in high-level language to object code, which can be then executed independently. Each programming language has its compiler. The compilation process generally involves breaking down the source code into small pieces creating an intermediate representation, and then constructing the object code from the intermediate representation.

3.6.3 Interpreter

The interpreter also converts the high-level language program into machine code. However, the interpreter functions in a different way than a compiler. An interpreter reads the source code line-by-line, converts it into machine-understandable form, executes the line, and then proceeds to the next line. This is unlike a compiler that takes the entire source code and converts it to object code. The key differences between a compiler and an interpreter are shown in Table 3.1.

Table 3.1: Comparison of compiler and interpreter

Compiler	**Interpreter**
1. converts the entire source code into object code which is then executed by the user.	1. translates the source code line-by-line – takes a line of source code, converts it into machine executable form, executes it, and proceeds with the next line.
2. Once the object code is created, it can be executed multiple times without the need to compile during each execution.	2. During each execution, the source code is first interpreted and then executed.
3. During execution of an object code, neither the source nor the compiler is required	3. Both interpreter and source code are required during execution.
4. faster	4. slower
5. Languages like C, C++ and Java are compiled.	5. Languages like BASIC, Pascal and Python are interpreted.

3.7 Conclusion

Through the exploration of computer architecture and computation, we've unraveled the essential components that form the backbone of modern computing. By examining the architecture of a computer, including the input/output units, central processing unit (CPU), and memory unit, we've gained insights into data flows and how data are processed within a computing system. The Von Neumann architecture, a foundational concept in computer science, was discussed as the blueprint that guides the organization of these components, highlighting the interaction among them.

The journey continued with a detailed look at how instructions are formatted and executed, providing clarity on how computers systematically perform tasks. We explored the various types of programming languages, from machine language to assembly and high-level languages, illustrating the evolution of languages that facilitate human-computer interaction. Additionally, the role of translator software viz. assemblers, compilers, and interpreters was emphasized as crucial for bridging the gap between human-readable code and machine-executable instructions.

This comprehensive overview of computer architecture and computation lays a strong foundation for understanding how computers function as powerful tools for solving complex problems. By grasping these concepts, you're better equipped to appreciate the intricacies of programming and the computational processes that drive the digital world.

Bibliography

[1] Anita Goel, Computer Fundamentals, Pearson, 2010.

[2] Behrouz A. Forouzan, Foundations of Computer Science, Cengage Learning, 2017.

[3] V. Rajaraman and Neeharika Adabala, Fundamentals of Computers, PHI Learning, 2014.

Chapter 4

Python fundamentals

"The joy of coding Python should be in seeing short, concise, readable classes that express a lot of action in a small amount of clear code – not in reams of trivial code that bores the reader to death."

– Guido van Rossum

4.1 Introduction

Python is a high-level, interpreted language. Its features are:

- **Python is interpreted**: Python is processed at runtime by the interpreter. You need not compile your program before executing it.
- **Python has simple, conventional syntax**: Python statements are very close to those of pseudocodes, and Python expressions use the conventional notation found in algebra.
- **Python is highly interactive**: Expressions and statements can be entered at an interpreter's terminal to allow the programmer to try out the code and receive immediate output.
- **Python is object-oriented**: Python supports Object-Oriented Programming (OOP) principles. OOP closely models real-world objects and their interactions, leading to more natural and understandable program structures.
- **Python scales well**: Python is the most apt language in which a novice programmer can start coding. At the same time, it is so powerful to cater to the research community's needs.

☞ Python comes in two flavours: Python 2 and Python 3. Throughout this book, we use version 3.

4.1.1 How to run your Python code?

There are two ways you can run your Python code:

1. using the Python shell
2. running as a standalone script

4.1.1.1 Using Python shell

The **Python shell** is an interactive terminal-based environment wherein you can directly communicate with the Python interpreter. First, you open a terminal and type `python3`. Now the shell turns up with a welcome message as shown below.

```
user@Ubuntu2204LTS:~$ python3
Python 3.10.12 (main, Jul 29 2024, 16:56:48) [GCC 11.4.0] on
    linux
Type "help", "copyright", "credits" or "license" for more
    information.
>>>
```

The symbol `>>>` is called the **shell prompt**. This symbol prompts you for Python statements. When you enter a statement in the shell, the Python interpreter processes it and displays the result, if any, then followed by a new prompt as shown below:

```
>>> "Welcome to the world of Python programming"
'Welcome to the world of Python programming'
>>>
```

The shell-based execution of Python code will become cumbersome if you want to do some complex processing. You may have to type in a lot of Python statements and you can give only one command at a time to the interpreter. Undoubtedly, this is frustrating for most of us. The workaround is to run your code as a standalone script.

4.1.1.2 Running as a standalone script

Here are the steps:

1. Combine all the statements that you wish to execute into a Python program. program is known as **script** and should be saved with "py" extension, for example `sample.py`.

2. You may use some text editor to create your script. `gedit`, `vim` are some of the editors that you can probably use.

3. Then open a terminal in the directory where the script is stored. To run your script (assuming the name is `sample.py`), just give

   ```
   python3 sample.py
   ```

   Now the script will be executed and you get the desired output.

Feel to have an example? Here you go! Suppose you have written a script `sample.py` to display the message "My first Python program!". This is how you run it (The second line below shows the output):

```
user@Ubuntu2204LTS:~$ python3 sample.py
My first Python program!
```

4.1.2 The Python interpreter

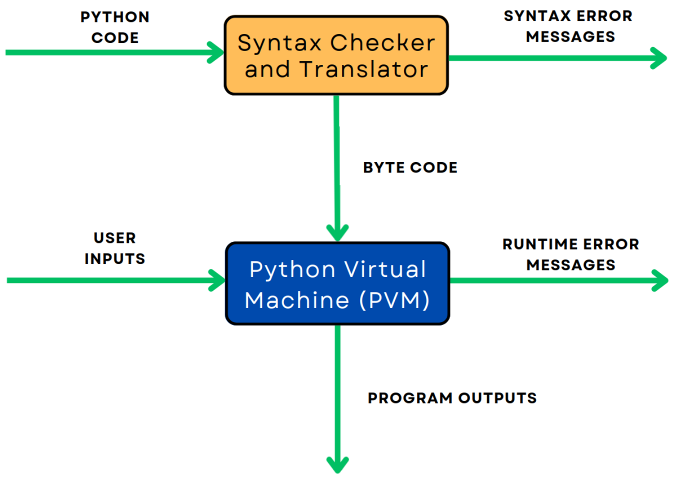


Figure 4.1: Steps in interpreting a Python program[1]

As shown in Figure 4.1, the interpreter performs a sequence of steps to execute your script:

1. The interpreter first reads a Python expression or a statement and verifies that it is valid with regard to the rules, or *syntax*, of the language. Any

[1]Image courtesy: Slightly adapted from *Fundamentals of Python: First Programs, Kenneth A Lambert* and redrawn.

invalid statement is called a *syntax error*. When the interpreter encounters such an error, it halts with an error message.

2. If the expression is well formed, the interpreter then translates it to an executable form called **byte code**. The *Syntax checker and Translator* component of the interpreter is responsible for verifying the syntax of the statements and translating valid statements to byte code.
3. Then, the byte code is sent to another interpreter component called the *Python Virtual Machine (PVM)* for execution. If any error occurs during execution, the process is halted with a runtime error message. Otherwise, the execution runs to completion and the output is produced.

4.2 “On your marks. Get set. Go!”

Let us now start our Python journey. Learning Python (or any programming language) goes hand in hand with learning English. Remember! How did you master this vernacular language? You started with learning the alphabet and then forming words out of them. Later you combined words to create meaningful sentences, then progressed to form paragraphs out of sentences. Finally, an article or an essay consists of one or more paragraphs. This is depicted in Figure 4.2.

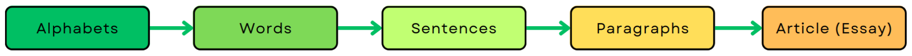


Figure 4.2: Steps in learning English[2]

You follow the same procedure to learn Python. As Figure 4.3 illustrates, you start with the basic building bricks: alphabets, numbers, and special symbols. Then, you combine them to build the foundation: constants, variables, and keywords. Over the foundation, you create rooms viz. expressions or instructions. A group of rooms (instructions) constitute a building (Python function). The Python city (program) is carved out of multiple buildings (functions).

Alphabets, Digits, Special Symbols
Constants, Variables, Keywords
Expressions, Instructions
Functions
Program

Figure 4.3: Steps in learning Python[2]

4.3 Character set

The set of characters supported by a programming language is called **character set**. A character can be an alphabet, a digit, or a special symbol. Python

[2]Image courtesy: Slightly adapted from *Let us Python, Yashavant Kanetkar* and redrawn.

supports the following characters:

- upper case alphabets (*A*–*Z*)
- lower case alphabets (*a*–*z*)
- digits (0–9)
- special symbols like @,#,%,$ etc.

Python maps each valid character to an integer value called **ASCII value**. (ASCII is the abbreviation of American Standard Code for Information Interchange.) For example, the ASCII value of character `'A'` is 65.

4.4 Constants, variables, and keywords

When properly combined, the characters in the character set form constants, variables, and keywords. A 'constant' is an entity whose value doesn't change. 3, 100, etc. are all constants.

The data on which programs operate are stored in various memory locations. To simplify the retrieval and use of data values, these memory locations are given names. Since the value stored in each location may change occasionally, the names given to these locations are called 'variable names' or simply 'variables'. Thus, a variable is a name that refers to a value.

You should choose meaningful names for your variables. The names can include both letters and digits; however, there are restrictions:

1. Variable names must start with a letter or the underscore '`_`' and can be followed by any number of letters, digits, or underscores.
2. Variable names are case sensitive; thus, the variable `COUNT` is a different name from the variable `count`.
3. Variable names cannot be a keyword. **Keywords**(also called **reserved words**) are special words, reserved for other purposes; thus, they cannot be used as variable names. Python has thirty-three keywords as listed in Figure 4.4. All the keywords except `True, False,` and `None` are in lowercase, and they must be written as is.

☞ '`_`' is a valid variable name.

4.5 Data types

Any data item stored in memory has an associated *data type*. For example, your roll number is stored as a number whereas, your name is stored as a string.

`and`	`elif`	`from`	`None`	`True`
`assert`	`else`	`global`	`not`	`try`
`break`	`except`	`if`	`or`	`while`
`class`	`exec`	`import`	`pass`	`with`
`continue`	`False`	`in`	`print`	`yield`
`def`	`finally`	`is`	`raise`	
`del`	`for`	`lambda`	`return`	

Figure 4.4: Reserved words in Python

The data type of an item defines the operations that can be performed on it, and how the values are stored in memory. Python supports the following data types:

- Number
- String
- List
- Tuple
- Set
- Dictionary

Here, Only the first two are discussed. The remaining data types will be explored in later chapters.

4.5.1 Numbers

The number or numeric data type is used to store numeric values. There are three distinct numeric types:

Type	**Description**	**Examples**
`int`	integers	`700,198005`
`float`	numbers with decimal point	`3.14,6.023`
`complex`	complex numbers	`3+4j, 10j`

The integers include numbers that do not have decimal point. The `int` data type supports integers ranging from -2^{31} to $2^{31} - 1$.

☞ Python 2 defined two integer data types viz. `int` and `long`. The difference is that the `long` type supports a much wider range of numbers than what `int` does. On the other hand, Python 3 has just one integer type, namely `int` which is more or less like the `long` type from Python 2.

Python uses `float` type to represent real numbers (with decimal points). The values of `float` type range approximately from -10^{308} to 10^{308} and have 16 digits of precision (number of digits after the decimal point). A floating-point number can be written using either ordinary decimal notation or scientific notation. Scientific notation is often useful for denoting numbers with very large or very small magnitudes. See the examples below:

Decimal notation	**Scientific notation**	**Meaning**
3.146	3.146e0	3.146×10^{0}
314.6	3.146e2	3.146×10^{2}
0.3146	3.146e-1	3.146×10^{-1}
0.003146	3.146e-3	3.146×10^{-3}

`complex` numbers are written in the form `x+yj`, where `x` is the real part and `y` is the imaginary part.

`int` type has a subtype `bool`. Any variable of type `bool` can take one of the two possible boolean values, `True` and `False` (internally represented as 1 and 0, respectively).

4.5.2 Strings

A string literal or a string is a sequence of characters enclosed in a pair of single quotes or double quotes. `"Hi", "8.5", `hello'` are all strings. Multi-line strings are written within a pair of triple quotes, ``` or `"""`. The following is an example of a multi-line string:

```
""" this is a multiline
string """
```

The strings `' '` and `" "` are called *empty strings*.

4.6 Statements

A statement is an instruction that the Python interpreter can execute. A statement can be an expression statement or a control statement. An expression statement contains an arithmetic expression that the interpreter evaluates. A control statement is used to represent advanced features of a language like decision-making and looping.

4.7 Conclusion

This chapter has introduced essential concepts that form the foundation of programming in Python. Starting with the character set, which defines the symbols and letters the language recognizes, the chapter progresses to cover constants, variables, and keywords, which are the building blocks of Python code. Understanding these elements is crucial as they allow you to store and manipulate data effectively.

The chapter also delves into data types, specifically numbers, and strings, highlighting how Python categorizes and processes different kinds of data. Finally, the exploration of Python statements provides insight into how instructions are structured and executed. With these fundamentals in place, you are now equipped to write basic Python programs and ready to explore more advanced concepts.

4.8 Exercises

1. Suggest the most appropriate data type for the following data items:
   (a) your name
   (b) your branch of study
   (c) your year of birth
   (d) your age
   (e) duration of B.Tech. programme
   (f) the constant π
2. Which of the following are valid variable names in Python?
   (a) `myvariable`
   (b) `my-variable`
   (c) `my_variable`
   (d) `MYVARIABLE`
   (e) `1stvariable`
   (f) `variable2`
3. Represent the following in scientific notation:
   (a) 3.14
   (b) speed of light
   (c) current Indian population
   (d) 0.000000257
   (e) 2
   (f) Avogadro's constant
   (g) Gravitational constant

Bibliography

[1] Kenneth A. Lambert, Fundamentals of Python: First Programs, Cengage Learning, 2019.

[2] Yashavant Kanetkar, Let Us Python, BPB Publication, 2022.

[3] John V Guttag, Introduction to Computation and Programming Using Python, MIT Press, 2013.

[4] Allen Downey, How to Think Like a Computer Scientist: Learning with Python, O'Reilly Media, 2015.

Chapter 5

Python expressions

"In the arithmetic of life, we add our joys, subtract our sorrows, multiply our love, and divide our time."

– Anonymous

5.1 Introduction

An arithmetic expression comprises operands and operators. Operands represent data items on which various operations are performed. The operations are denoted by operators. The operands can be constants or variables. When a variable name appears in the place of an operand, it is replaced with its value before the operation is performed. The operands acted upon by arithmetic operators, must represent numeric values. Thus, the operands can be integer quantities, floating-point quantities, or even characters (recall that every character has an equivalent ASCII value).

5.2 Operators in Python

Python language supports the following types of operators.

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

5.2.1 Arithmetic Operators

The various arithmetic operators in Python are tabulated in Table 5.1. The remainder operator (%) requires both operands to be integers, and the second operand is non-zero. Similarly, the division operator (/) requires that the second operand be non-zero. The floor division operator (//) returns the floor value of the quotient of a division operation. See the example below:

```
>>> 7.0//2
3.0
>>> 7//2
3
```

☞Floor division is also called integer division.

Table 5.1: Arithmetic operators

Operation	Operator
Negation	-
Addition	+
Subtraction	--
Multiplication	*
Division	/
Floor division	//
Remainder	%
Exponentiation	**

Table 5.2: Comparison operators

Operation	Operator
Equal to	==
Not equal to	!=
Greater than	>
Greater than or equal to	>=
Less than	<
Less than or equal to	<=

5.2.2 Assignment operator

'=' is the assignment operator. The **assignment statement** creates new variables and gives them values that can be used in subsequent arithmetic expressions. See the example:

```
>>> a=10
>>> b=5
>>> a+b
15
```

Python allows you to assign a single value to several variables simultaneously. An example follows:

```
>>> a=b=5
>>> a
```

```
5
>>> b
5
```

You can also assign different values to multiple variables. See below

```
>>> a,b,c=1,2.5,"ram"
>>> a
1
>>> b
2.5
>>> c
'ram'
```

Python supports the following six additional assignment operators (called *compound assignment* operators): `+=, -=, *=, /=, //=` and `%=`. These are described below:

Expression	**Equivalent to**
`a+=b`	`a=a+b`
`a-=b`	`a=a-b`
`a*=b`	`a=a*b`
`a/=b`	`a=a/b`
`a//=b`	`a=a//b`
`a%=b`	`a=a%b`

5.2.3 Comparison Operators

Table 5.2 shows the various comparison operators. Comparison operators are also called relational operators. The result of a comparison is either `True` or `False`. `==` and `!=` are also known as *equality operators*.

The use of comparison operators is illustrated below:

```
>>> i,j,k=3,4,7
>>> i>j
False
>>> (j+k)>(i+5)
True
```

Comparison operators support chaining. For example, `x < y <= z` is equivalent to `x < y and y <= z`.

5.2.4 Logical Operators

Python includes three Boolean (logical) operators viz. `and` , `or` , and `not` . The `and` operator and `or` operator expect two operands,which are hence called binary operators. The `and` operator returns `True` if and only if both of its operands are `True`, and returns `False` otherwise. The `or` operator returns `False` if and only if both of its operands are `False`, and returns `True` otherwise. The `not` operator expects a single operand and is hence called a unary operator. It returns the logical negation of the operand, that is, `True`, if the operand is `False`, and `False` if the operand is `True`.

The behaviour of each operator can be specified in a **truth table** for that operator. The first row in the truth table contains labels for the operands and computed expressions. Each row below the first row contains a possible combination of values for the operands and the value resulting from applying the operator to them. Tables 5.3 shows the truth tables for `or` and `and` operators. Table 5.4 shows the truth tables for `not`.

Table 5.3: Truth tables for logical OR and logical AND operators

a	b	a or b	a and b
False	False	False	False
False	True	True	False
True	False	True	False
True	True	True	True

Table 5.4: Truth table for logical NOT

a	not a
False	True
True	False

In the context of logical operators, Python interprets all non-zero values as `True` and zero as `False`. See examples below:

```
>>> 7 and 1
1
>>> -2 or 0
-2
>>> -100 and 0
0
```

5.2.5 Bitwise operators

Bitwise operators take the binary representation of the operands and work on their bits, one bit at a time. The bits of the operand(s) are compared starting with the rightmost bit - the least significant bit, then moving towards the left and ending with the leftmost (most significant) bit. The result of the comparison will depend on the compared bits and the operation being performed. These bitwise operators can be divided into three general categories as discussed below:

5.2.5.1 One's complement operator

One's complement is denoted by the symbol ~. It operates by changing all zeroes to ones and ones to zeroes in the binary representation of the operand. The operand must be an integer-type quantity.

Example 5.1. This example illustrates the one's complement operator. See below:

```
>>> ~98
-99
>>> ~102
-103
```

Let us understand the results obtained. Take 102. Its binary is `01100110`. Flipping the bits, yields `10011001` = `-103`. This is shown below:

$$\begin{aligned} \texttt{102} &= \underline{\texttt{0110 0110}} \\ \sim \texttt{102} &= \texttt{1001 1001} \\ &= \texttt{-103} \end{aligned}$$

5.2.5.2 Logical bitwise operators

There are three logical bitwise operators: bitwise and (&), bitwise exclusive or (^), and bitwise or (|). Each of these operators require two integer-type operands. The operations are performed on each pair of corresponding bits of the operands based on the following rules:

- A **bitwise and** expression will return `1` if both the operand bits are `1`. Otherwise, it will return `0`.
- A **bitwise or** expression will return `1` if at least one of the operand bits is `1`. Otherwise, it will return `0`.
- A **bitwise exclusive or** expression will return `1` if the bits are not alike (one bit is 0 and the other is 1). Otherwise, it will return `0`.

These results are summarized in Table 5.5. In this table, b_1 and b_2 represent the corresponding bits within the first and second operands, respectively.

Table 5.5: Logical bitwise operators

b_1	b_2	$b_1 \& b_2$	$b_1 \mid b_2$	b_1 ^ b_2
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Example 5.2. This example illustrates the operation of the three logical bitwise operators as shown below:

```
>>> a=20
>>> b=108
>>> a&b
4
>>> a|b
124
>>> a^b
120
```

The following justifies these results.

$$\begin{aligned}
\texttt{a} &= \texttt{0001 0100}\\
\texttt{b} &= \underline{\texttt{0110 1100}}\\
\texttt{a \& b} &= \texttt{0000 0100}\\
&= \texttt{4}
\end{aligned}$$

$$\begin{aligned}
\texttt{a} &= \texttt{0001 0100}\\
\texttt{b} &= \underline{\texttt{0110 1100}}\\
\texttt{a} \mid \texttt{b} &= \texttt{0111 1100}\\
&= \texttt{124}
\end{aligned}$$

$$\begin{aligned}
\texttt{a} &= \texttt{0001 0100}\\
\texttt{b} &= \underline{\texttt{0110 1100}}\\
\texttt{a} \;^\wedge\; \texttt{b} &= \texttt{0111 1000}\\
&= \texttt{120}
\end{aligned}$$

5.2.5.3 Bitwise shift operators

The two bitwise shift operators are shift left (`<<`) and shift right (`>>`). The expression `x << n` shifts each bit of the binary representation of `x` to the left, `n` times. Each time we shift the bits left, the vacant bit position at the right end is filled with a zero.

The expression `x >> n` shifts each bit of the binary representation of `x` to the right, `n` times. Each time we shift the bits right, the vacant bit position at the left end is filled with a zero.

Example 5.3. This example illustrates the bitwise shift operators as shown below:

```
>>> 120>>2
30
>>> 10<<3
80
```

Let us now see the operations in detail.

$$\begin{aligned} \texttt{120} &= \underline{\texttt{0111 1000}} \\ \text{Right shifting once} &- \texttt{0011 1100} \\ \text{Right shifting twice} &- \underline{\texttt{0001 1110}} \\ \texttt{120>>2} &= \texttt{30} \end{aligned}$$

$$\begin{aligned} \texttt{10} &= \underline{\texttt{0000 1010}} \\ \text{Left shifting once} &- \texttt{0001 0100} \\ \text{Left shifting twice} &- \texttt{0010 1000} \\ \text{Left shifting thrice} &- \underline{\texttt{0101 0000}} \\ \texttt{10<<3} &= \texttt{80} \end{aligned}$$

5.2.6 Membership Operators

These operators test for the membership of a data item in a sequence, such as a string. Two membership operators are used in Python.

- **in** – Evaluates to `True` if it finds the item in the specified sequence and `False` otherwise.
- **not in** – Evaluates to `True` if it does not find the item in the specified sequence and `False` otherwise.

See the examples below:

```
>>> 'A' in 'ASCII'
True
>>> 'a' in 'ASCII'
False
>>> 'a' not in 'ASCII'
True
```

5.2.7 Identity Operators

is and **is not** are the identity operators in Python. They are used to check if two values (or variables) are located in the same part of the memory. `x is y` evaluates to `true` if and only if `x` and `y` are the same object. `x is not y` yields the inverse truth value.

Table 5.6: Precedence rules in Python

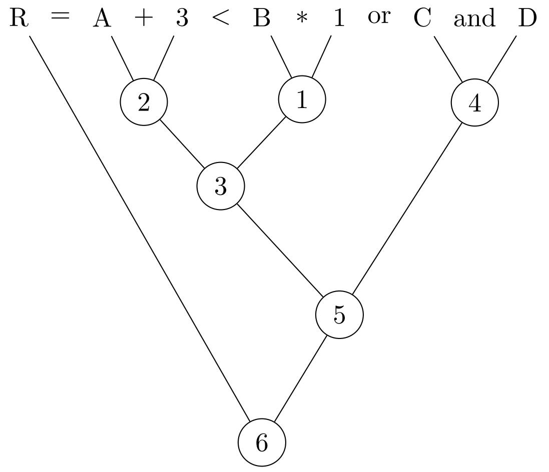
Precedence group	**Operators**	**Associativity**
Parenthesis	`()`	L → R
Exponentiation	`**`	R → L
Unary plus, Unary minus, One's complement	`+`, `-`, ∼	R → L
Multiplication, Division, Floor division, Modulus	`*, /, //, %`	L → R
Addition, Subtraction	`+, -`	L → R
Bitwise shift operators	`<<`, `>>`	L → R
Bitwise AND	`&`	L → R
Bitwise XOR	∧	L → R
Bitwise OR	|	L → R
Comparisons, Identity and Membership operators	`==, !=, <, <=, >=, > is, is not, in, not in`	L → R
Logical NOT	`not`	R → L
Logical AND	`and`	L → R
Logical OR	`or`	L → R
Assignment operators	`=, +=, -=, *= /=, //=, %=`	R → L

5.2.8 Precedence and associativity of operators

When an expression contains more than one operator, in what order will the operations be performed? To answer this question satisfactorily, one has to know the *precedence* of operators. The order in which operators in an arithmetic expression are applied to their respective operands is called the **precedence** of operators. It is also known by other names, such as **priority** or **hierarchy**. Operators with a higher precedence are applied before operators having a lower precedence. For example, the multiplication operator has precedence over the addition operator. This means that if an expression has both + and $*$, then addition will be performed only after multiplication. The precedence of operators is shown in Table 5.6. The operators are listed in groups in descending order – the upper group has higher precedence than the lower ones.

We find that in Table 5.6, more than one operator exists in the same group. These operators have the same precedence. If an expression has multiple operators with the same precedence, the tie is resolved using **associativity rules**. Associativity is of two types – Left to Right (L$\rightarrow$ R) and Right to Left (R $\rightarrow$L). The third column of Table 5.6 lists the associativity of the operators. L$\rightarrow$ R means that when there are two operators with the same precedence, the operator that comes first on a left-to-right scan of the expression will be given priority. Similarly with R $\rightarrow$L, the right operator has higher priority.

Consider the expression `a` $-$ `b` $+$ `c`. Since $+$ and $-$ are of the same precedence, so look for associativity. The associativity is L$\rightarrow$R. Thus $-$ will be evaluated first as it comes to the left.



Order	Operation	Resultant
1	$B * 1$	5
2	$A + 3$	4
3	$A + 3 < B * 1$	1
4	$C\&\&D$	1
5	$A + 3 < B * 1 \vert\vert C\&\&D$	1
6	$R = A + 3 < B * 1 \vert\vert C\&\&D$	1

Figure 5.1: Evaluation of an arithmetic expression

Example 5.4. Consider the assignment statement

$$R = A + 3 < B * 1 \texttt{ or } C \texttt{ and } D$$

Let the values of the variables be $A = 1, B = 5, C = -1, and D =$ `True`. Figure 5.1 shows the structure of the evaluation. The numbers shown in the circle denote the order in which the various operators are applied. The final result is 1, which is assigned to R.

Table 5.7 gives more examples for expression evaluations.

Table 5.7: More examples of expression evaluations

Expression	**Evaluation**	**Value**
`3 + 4 * 2`	`3 + 8`	`11`
`(3 + 4 ) * 2`	`7 * 2`	`14`
`2 ** 3 ** 2`	`2 ** 9`	`512`
`(2 ** 3) ** 2`	`8 ** 2`	`64`
`-3 ** 2`	`-(3 ** 2)`	`-9`
`-(3) ** 2`	`(-3) ** 2`	`9`
`not True and False or True`	`(False and False) or True`	`True`

5.2.9 Mixed-Mode Arithmetic

Table 5.9: Type coercion rules

Operands type	**Result type**
`int` and `int`	`int`
`float` and `float`	`float`
`int` and `float`	`float`

Performing calculations involving operands of different data types is called *mixed-mode arithmetic*. Consider computing the area of a circle having 3 unit radius:

```
>>> 3.14 * 3 ** 2
28.26
```

Python supports mixed-mode arithmetic through **type coercion**, wherein the resultant of an expression will have the most general data type among all operand data types involved. The operand of a less general type will be temporarily and automatically converted to the more general type before the operation is performed. The various conversion rules are summarized in Table 5.9.

In the above example, the value 9 (result of `3 ** 2`) is converted to 9.0 before the multiplication.

5.3 Conclusion

This chapter on Python expressions has provided a detailed exploration of the various operators that are essential for performing operations within Python programs. We've covered arithmetic operators, which form the basis of mathematical calculations, and assignment operators, which link values to variables. Comparison operators were discussed as tools for evaluating relationships between values, while logical operators allow for the combination of multiple conditions to control the flow of a program.

Bitwise operators introduced a deeper level of manipulation by enabling operations at the binary level, and membership and identity operators offered ways to test for presence within sequences and compare objects, respectively. Additionally, we delved into the concepts of operator precedence and associativity, which dictate the order in which operations are executed, ensuring accurate expression evaluation. Finally, mixed-mode arithmetic was introduced, demonstrating how Python handles expressions involving different data types.

Understanding these operators and their interactions is crucial for writing efficient and accurate Python code. This knowledge forms the bedrock of more complex programming tasks, equipping you with the skills to manipulate data, control program logic, and ultimately, build more sophisticated applications.

5.4 Exercises

1. Perform the following operations:
   (a) 15 `and` -8
   (b) 127 `or` 0
   (c) 0 `and` 1048
   (d) `not` 10787
2. Perform the following operations:
   (a) 12 & 10
   (b) 55 & 24
   (c) 45 | 50

   (d) 13 | 99
   (e) 18 ^ 20
   (f) 118 ^ 65
   (g) ~ 50
   (h) 14 << 3
   (i) 25 << 2
   (j) 15 >> 4
   (k) 45 >> 3

3. Evaluate the following expressions:
   (a) 34 ** 2 * 8 + -8
   (b) 2 ** 3 ** 4
   (c) (3 * 22) + - 8 ** 2 + 25
   (d) `True and not False or not True and True`
   (e) 45 * 2 + - 75 `and` 0 ** 4 `or` 3

Bibliography

[1] Kenneth A. Lambert, Fundamentals of Python: First Programs, Cengage Learning, 2019.

[2] Yashavant Kanetkar, Let Us Python, BPB Publication, 2022.

[3] John V Guttag, Introduction to Computation and Programming Using Python, MIT Press, 2013.

[4] Allen Downey, How to Think Like a Computer Scientist: Learning with Python, O'Reilly Media, 2015.

Chapter 6

Data input and output

"Your input determines your outlook. Your outlook determines your output, and your output determines your future."

– Zig Ziglar

6.1 Introduction

Python provides various *functions* or *methods* for performing I/O operations. You are motivated to read the box **Functions – the gist** before proceeding further.

To display something on the screen, Python uses the method `print`. It just prints whatever is given inside the quotes. See an example below:

```
>>> print("Hello World!")
Hello World!
```

Functions – the gist

A function is a named block of statements that carries out some specific, well-defined task. A program can be seen as a collection of functions, each of which serves a unique purpose. Functions are of two types: *built-in* and *user-defined* functions. Built-in functions are those that are provided by the interpreter itself. These include I/O functions, mathematical functions, etc. User-defined functions are those that are created by the programmers. Optionally a function can take input values called *parameters* or *arguments* to process and could also give back an output value, which is known as *return value.*

See another example now: Consider the statements

```
>>> a = 7
>>> print("The value of a is ",a)
The value of a is  7
```

Python prints as such, whatever is enclosed in `" "`. In the above example, the first occurrence of `a` is within double quotes. So it is printed as such. The second `a` occurs outside the quotes. Thus, its value `7` gets printed. See another example:

```
>>> x=5
>>> y=3
>>> print("The value of",x,"and",y,"is",x+y)
The value of 5 and 3 is 8
```

Python **f-strings** offer a prettier way to achieve the same message output. See below:

```
>>> print(f"The value of {x} and {y} is {x+y}")
The value of 5 and 3 is 8
```

As shown above, the string to be printed should be prefixed with a '`f`' indicating an f-string. The variables to be printed are to be enclosed in a pair of braces.

To receive input from the user, Python provides the `input()` method which accepts the input as a string. See an example:

```
>>> myName = input("What is your name?")
What is your name?Python language
>>> print("I am ",myName)
I am Python language
```

`input()` can accept numbers too but are accepted as strings. So if any arithmetic operation is to be performed on the received input, the programmer must convert them from strings to the appropriate numeric types. Python uses **type conversion functions** to convert values from one data type to another. The type conversion functions have the same name as the data type to which it converts. Table 6.1 lists the various type conversion functions.

Note that the `int` function converts a `float` to an `int` by truncation, not by rounding to the nearest whole number. Truncation simply chops off the number's fractional part.

Python provides a function `type()` to know the data type of a variable. See below:

```
>>> i=2
>>> b=True
```

Table 6.1: Type conversion functions

Conversion Function	**Sample usage**	**Value Returned**
`int`(a number or a string)	`int(3.7)`	`3`
	`int('347')`	`347`
`float`(a number or a string)	`float(33)`	`33.0`
	`float('33.8')`	`33.8`
	`float('33')`	`33.0`
`str`(any value)	`str(37)`	`'37'`
	`str(37.6)`	`'37.6'`

```
>>> s="Hello"
>>> f=7.18
>>> print(type(i))
<class 'int'>
>>> print(type(b))
<class 'bool'>
>>> print(type(s))
<class 'str'>
>>> print(type(f))
<class 'float'>
```

6.1.1 Escape sequences

With the `print` method, whatever you enclose within a pair of double quotes, gets printed as such. But there are a few exceptions, like quotation marks, commas, etc. For this reason, Python provides special character constants referred to as *escape sequences*. An escape sequence refers to a combination of characters beginning with a backslash (\) followed by letters. Some of the escape sequences in Python are shown in Table 6.2.

See some illustrative examples below:

```
>>> print("Hello world")
Hello world
>>> print("Hello\tworld")
Hello   world
>>> print("Hello\bworld")
Hellworld
>>> print("Hello \bworld")
Helloworld
```

```
>>> print("Hello\nworld")
Hello
world
>>> print("Hello\vworld")
Hello
     world
>>> print("The teacher said, \"It\'s very easy to program with
    Python\"")
The teacher said, "It's very easy to program with Python"
```

Table 6.2: Escape sequences in Python

Escape Sequence	Meaning
\b	Backspace
\n	Newline
\t	Horizontal tab
\v	Vertical tab
\\	The \ character
\'	Single quotation mark
\"	Double quotation mark

6.2 Program Comments and Docstrings

As programs get bigger and more complicated, they get more difficult to read and understand. For this reason, it is beneficial to add notes to your programs to explain the purpose of the statements. These notes are called **comments**. A comment is a piece of program text that the computer ignores but provides useful documentation to programmers. These comments begin with the # symbol and extend to the end of a line. Everything from the # to the end of the line is ignored by the interpreter while execution – it does not affect the program.

An end-of-line comment might explain the purpose of a variable. Here is an example:

```
>>> sum = 5 + 7 # the variable sum contains the sum of 5 and 7
```

You can also put comments on a separate line:

```
>>> # Let us now print Hello
>>> print("Hello")
Hello
```

Python also supports comments that extend multiple lines, one way of doing it is to use # in the beginning of each line. Here is an example:

```
>>> # This is a long comment
>>> # and it extends
>>> # to multiple line
```

Just as comments are attached to individual statements, you can also include details about the program's purpose at the beginning of the program file. This type of comment called a **docstring**, is a multi-line string. Here is an example:

```
"""
Program name: areaRect.py
Version: 1.1

This program finds the area of a rectangle.
The inputs are two integers representing the length
and breadth of a rectangle, and the output is an
integer named area that represents the area of
the rectangle
"""
```

6.3 The `math` module

Python provides many functions ranging from input/output to performing complex calculations. Python groups together functions providing similar functionalities into a **module** for easy access to them. `math, sys, os` are some commonly encountered modules. Let us have a look at the `math` module. This module includes several functions that perform basic mathematical operations - `sqrt(), sin(), exp()`, a few to mention. In addition to functions, the `math` module also has the values of the constants π and e. To use a function or a constant of the `math` module, you need to do two things:

1. import the module
2. access the function or the constant by prefixing its name with "`math.`" (`math` followed by a dot)

See the examples below:

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.e
```

```
2.718281828459045
>>> math.sqrt(3)
1.7320508075688772
```

6.4 Errors in Python program

Errors in a Python program are divided into two categories:

1. Syntax errors
2. Semantic errors

6.4.1 Syntax errors

Programming beginners mostly make typographical errors in their programs. Such errors are called *syntax errors*. Syntax is the set of rules for constructing well-formed expressions or statements in a language. A computer generates a syntax error when an expression or sentence is not well formed. When Python encounters a syntax error in a program, it halts execution with an error message indicating the reason for the error. Following are some of the syntax errors:

Name error

This error occurs when a referenced variable is not found. See an example:

```
>>> print(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

The error occurs because you try to print the value of the variable `x` without assigning any value to it.

Syntax error

A very common mistake while writing programs is to omit the required parentheses, as shown below:

```
>>> print x
  File "<stdin>", line 1
    print x
    ^^^^^^^
SyntaxError: Missing parentheses in call to 'print'. Did you
   mean print(...)?
```

Indentation error

Indentation is very significant in Python code. Each line of code must begin in the leftmost column, with no leading spaces. The only exception to this rule occurs in control statements and definitions, where nested statements must be indented one or more spaces. An indentation error is raised when there is an incorrect indentation. See below:

```
>>>  print(x)
  File "<stdin>", line 1
    print(x)
IndentationError: unexpected indent
```

Note the extra space between the prompt `>>>` and `print(x)`.

Type error

Python is a **strongly typed** programming language. This means that the interpreter checks the data types of all operands before performing any operation. If the type of an operand is not appropriate, the interpreter halts execution with an error message. This error checking prevents a program from attempting to do something that it cannot do. See an example:

```
>>> 5+'3'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

6.4.2 Semantic errors

Semantics is the set of rules that allows the computer to interpret the meaning of expressions or statements. A semantic error is detected when the action that an expression describes cannot be carried out, even though that expression is syntactically correct. Division by zero is the most common semantic error.

```
>>> 7/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

6.5 Programming examples

Program 6.1. To print “My first Python program”.

```
print("My first Python program")
```

Program 6.2. To input the user's name and print a greeting message.

```
name=input("Enter your name")
print("Hello ",name)
```

Program 6.3. To input a number and display it.

```
num=int(input("Enter a number"))
print("The number you entered is",num)
```

Program 6.4. To add and subtract two input numbers.

```
a=int(input("Enter the first number"))
b=int(input("Enter the second number"))
sum=a+b
print("The sum of the two numbers is",sum)
difference=a-b
print("The difference between the two numbers is",difference)
```

Program 6.5. To input the sides of a rectangle and find its perimeter.

```
length=int(input("Enter the length of the rectangle"))
breadth=int(input("Enter the breadth of the rectangle"))
perimeter=2*(length+breadth)
print("Perimeter of the rectangle is",perimeter)
```

Program 6.6. To input the side of a square and find its area.

```
side=int(input("Enter the side of the square"))
area=side**2
print("Area of the square is",area)
```

Program 6.7. To input the radius of a circle and find its circumference.

```
import math
radius=int(input("Enter the radius"))
c=2*math.pi*radius
print("Circumference of the circle is",c)
```

Program 6.8. To input two values a and b and then find a^b.

```
import math
a=int(input("Enter the base"))
b=int(input("Enter the exponent"))
c=math.pow(a,b)
print(a,"to the power",b,"is",c)
```

☞ `pow(a,b)` finds a^b.

Program 6.9. To input the base and altitude of a right triangle and find its hypotenuse using Pythagoras theorem.

```
import math
b=int(input("Enter the base"))
a=int(input("Enter the altitude"))
temp=a**2+b**2
h=math.sqrt(temp)
print("The hypotenuse is",h)
```

Program 6.10. To input two values a and b and then swap them.

```
a=int(input("Enter a number"))
b=int(input("Enter another number"))
print("The numbers before swapping are a =",a,"and b =",b)
temp=a
a=b
b=temp
print("The numbers after swapping are a =",a,"and b =",b)
```

Program 6.11. To input two values a and b and then swap them without using a third temporary variable

```
a=int(input("Enter a number"))
b=int(input("Enter another number"))
print("The numbers before swapping are a =",a,"and b =",b)
a=a+b
b=a-b
a=a-b
print("The numbers after swapping are a =",a,"and b =",b)
```

6.6 Conclusion

This chapter laid the foundation of Python programming by covering essential topics such as user input, data output, escape sequences, comments, and the math module. You've also gained insight into the different types of errors that can occur in Python and how to handle them. With this knowledge, you are now equipped to write more robust and interactive Python programs. As you progress, these concepts will serve as the building blocks for more complex programming tasks, allowing you to develop a deeper understanding of how Python operates.

6.7 Exercises

1. Write a Python program to calculate simple interest by inputting the value of the Principal amount, period, and interest rate from the user.

2. Write a Python program to convert the time input in minutes to hours and minutes. For example, 85 minutes is 1 hour 25 minutes.

3. Write a Python program that inputs the cost of an item and the number of items and displays the total cost.

4. Write a Python program that takes an amount in dollars and converts it to Indian rupees.

5. Write a Python program to reverse a three-digit number.

6. You are given the task of calculating the electricity bill of a house. Each house has the following components: fan, light, washing machine, and computer. Each fan consumes 1 unit per day, and each light consumes 0.5 units per day, the washing machine consumes 2 units per day and each computer consumes 3 units per day. Let the cost of 1 unit be 50 rupees. Input the number of fans, lights, washing machines, and computers for a particular house and find the total electricity bill for that house for 2 months. Assume a 30-day month.

Bibliography

[1] Kenneth A. Lambert, Fundamentals of Python: First Programs, Cengage Learning, 2019.

[2] Yashavant Kanetkar, Let Us Python, BPB Publication, 2022.

[3] John V Guttag, Introduction to Computation and Programming Using Python, MIT Press, 2013.

[4] Allen Downey, How to Think Like a Computer Scientist: Learning with Python, O'Reilly Media, 2015.

Chapter 7

Control structures

"It is not our abilities that show who we truly are, it is our choices."
– Albus Dumbledore

*"**If** you always do what you've always done, expect the same results. **Else**, be prepared for change"*
– Anonymous

"It's not what we do once in a while that shapes our lives. It's what we do consistently."
- Anthony Robbins

7.1 Adventure game

Text-based adventure games are a genre of interactive fiction where players navigate through a narrative by making choices and issuing commands, all conveyed through text. Unlike graphical games, these games rely on the player's imagination and the game's text-based descriptions to create an immersive experience. The concepts you will learn in this chapter will enable you to create your own adventure game.

7.2 Introduction

The Python programs encountered so far are very simple. These follow the sequencing construct described in Chapter 2. Python also supports the other two constructs, viz. selection, and looping. These constructs enable Python to solve real-life problems effectively. Python control structures form the topic of discussion in this chapter.

7.3 Selection statements

This section explores several types of selection statements that allow the computer to make choices.

7.3.1 One-way selection statement

First, we discuss the `if` statement, also known as *conditional execution*. The following example tests whether a variable `x` is positive or not.

```
>>> x=10
>>> if x>0:
...     print(x,"is positive")
...
10 is positive
```

It is possible to combine multiple conditions using logical operators. The following code snippet tests whether a number `x` is a single-digit number or not.

```
>>> x=int(input("Enter a number"))
Enter a number7
>>> if x > 0 and x < 10:
...     print(x,"is a positive single digit.")
...
7 is a positive single digit.
```

Python provides an alternative syntax for writing the condition in the above code that is similar to mathematical notation:

```
>>> x=int(input("Enter a number"))
Enter a number8
>>> if 0<x<10:
...     print(x,"is a positive single digit.")
...
8 is a positive single digit.
```

7.3.2 Two-way selection statement

The `if-else` statement, also known as *alternative execution*, is the most common type of selection statement in Python. See the following example to check if a person is major or minor.

```
>>> age=12
>>> if age>=18:
...     print("Major")
... else:
```

```
...     print("Minor")
...
Minor
```

7.3.3 Multi-way selection statement

Multi-way selection is achieved through the `if-elif-else` statement. `elif` is the abbreviation of "else if". The following code compares two variables and prints the relation between them.

```
>>> x=10
>>> y=5
>>> if x < y:
...     print(x, "is less than", y)
... elif x > y:
...     print(x, "is greater than", y)
... else:
...     print(x, "and", y, "are equal")
...
10 is greater than 5
```

The multi-way `if` statement is called chained conditional execution.

7.4 Repetition statements or loops

Python supports two types of loops – those that repeat an action a fixed number of times (**definite iteration**) and those that act until a condition becomes false (**conditional iteration**).

7.4.1 Definite iteration: The `for` loop

We now examine Python's `for` loop, the control statement supporting definite iteration. We use `for` in association with the `range()` function that dictates the number of iterations. To be precise, `range(k)` when used with `for` causes the loop to iterate k times. See the example below that prints "`Hello`" 5 times.

```
>>> for i in range(5):
...     print("Hello")
...
Hello
Hello
Hello
Hello
Hello
```

`range(k)` starts counting from 0, incrementing after each iteration, and stops on reaching $k-1$. Thus to print numbers from 0 to 4, you write:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

You are reminded that `range(5)` counts from 0 to 4, not 5. To get the numbers on the same line you need to write `print(count,end=" ")`. By using `end = " "`, the Python interpreter will append whitespace following the count value, instead of the default newline character `('\n')` See below:

```
>>> for i in range(5):
...     print(i,end=" ")
...
0 1 2 3 4
```

☞Loops that count through a range of numbers are called count-controlled loops.

By default, the `for` loop starts counting from 0. To count from an explicit lower bound, you need to include it as part of the `range` function. See the example below that prints the first 10 natural numbers:

```
>>> for count in range(1,11):
...     print(count,end=" ")
...
1 2 3 4 5 6 7 8 9 10
```

The `for` loops we have seen till now count through consecutive numbers in each iteration. Python provides a third variant of the `for` loop that allows to count in a non-consecutive fashion. For this, you need to explicitly mention the *step size* in the `range` function. The following code prints the even numbers between 4 and 20 (both inclusive).

```
>>> for i in range(4,21,2):
...     print(i,end=" ")
...
4 6 8 10 12 14 16 18 20
```

When the step size is negative, the loop variable is decremented by that amount in each iteration. The following code displays numbers from 10 down to 1.

```
>>> for count in range(10,0,-1):
...     print(count,end=" ")
...
10 9 8 7 6 5 4 3 2 1
```

Notice above that, to count from 10 down to 1, we write `for count in range(10,0,-1)` and not `for count in range(10,1,-1)`. Thus, to count from `a` down to `b`, we write `for count in range(a,b-1,s)` with `a > b` and `s < 0`.

7.4.2 Conditional Iteration: The `while` loop

Conditional iteration requires that a condition be tested within the loop to determine whether the loop should continue. Python's `while` loop is tailor-made for this type of control logic. Here is the code to print the first 10 natural numbers, but this time with `while`.

```
>>> i=1
>>> while i<=10:
...     print(i,end=" ")
...     i=i+1
...
1 2 3 4 5 6 7 8 9 10
```

☞There is no *exit-controlled loop* in Python, but you can modify the `while` loop to achieve the same functionality.

7.4.3 Nested loops

It is possible to have a loop inside another loop. We can have a `for` loop inside a `while` loop or inside another `for` loop. The same is possible for `while` loops too. The enclosing loop is called the **outer loop**, and the other loop is called the **inner loop**. The inner loop will be executed once for each iteration of the outer loop: Consider the following code:

```
>>> for i in range(1,5):    #This is the outer loop
...     for j in range(1,5):   #This is the inner loop
...             print(j, end=" ")
...     print()
...
1 2 3 4
1 2 3 4
```

```
1 2 3 4
1 2 3 4
```

The loop variable `i` varies from 1 to 4 (not 5). For each value of `i`, the variable `j` varies from 1 to 4. The statement `print(j, end=" ")` prints the `j` values on a single line, and the statement `print()` takes the control to the next line after every iteration of the outer loop.

7.4.4 Loop control statements

Python provides three loop control statements that control the flow of execution in a loop. These are discussed below:

7.4.4.1 `break` statement

The `break` statement is used to terminate loops. Any statements inside the loop following the `break` will be neglected, and the control goes out of the loop. `break` stops the current iteration and skips the succeeding iterations (if any) and passes the control to the first statement following (outside) the loop. This is illustrated in the following code:

```
for num in range(1,5):
    if num%2==0:
        print(num,"is even")
        break
    print("The number is",num)
print("Outside the loop")
```

This code produces the output:

```
The number is 1
2 is even
Outside the loop
```

7.4.4.2 `continue` statement

The `continue` statement is used to bypass the remainder of the current iteration through a loop. The loop does not terminate when a `continue` statement is encountered. Rather, the remaining loop statements are skipped and the control proceeds directly to the next pass through the loop. See the code below:

```
for num in range(1,5):
    if num%2==0:
        print(num,"is even")
        continue
    print("The number is",num)
print("Outside the loop")
```

The code above produces the output:

```
The number is 1
2 is even
The number is 3
4 is even
Outside the loop
```

7.4.4.3 pass statement

The `pass` statement is equivalent to the statement "Do nothing". It can be used when a statement is required syntactically but the program requires no action. Nothing happens when `pass` is executed. It results in a NOP (No OPeration). After the `pass` is executed, control proceeds as usual to the next statement in the loop. The code below illustrates this:

```
for num in range(1,5):
    if num%2==0:
        print(num,"is even")
        pass
    print("The number is",num)
print("Outside the loop")
```

This code produces the output:

```
The number is 1
2 is even
The number is 2
The number is 3
4 is even
The number is 4
Outside the loop
```

`pass` statement is useful when you want to insert empty code (empty lines) in a program, where real code statements can be added later. Empty code is not allowed in loops; if included, Python will raise errors. In such situations, `pass` statement acts as a temporary placeholder.

7.5 Short-Circuit Evaluation

Python sometimes knows the value of a Boolean expression before it has evaluated all of its operands. For instance, in the expression `A and B`, if `A` is `False`, then so is the expression, and there is no need to evaluate `B`.

Similarly in the expression `A or B`, if `A` is `True`, then so is the expression, and again, there is no need to evaluate `B`. This approach, in which evaluation stops as early as possible, is called *short-circuit evaluation*. This is especially advantageous if `B` itself is a complex expression whose evaluation can be avoided.

Short circuit evaluation can be useful in avoiding errors. For example, consider the following code causing zero division error:

```
>>> a=10
>>> b=0
>>> print(a/b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Now let us modify the code to exploit the short circuit evaluation to avoid the zero division error. The modified code follows:

```
>>> a=10
>>> b=0
>>> if b>0 and a/b>0:
...     print(a/b)
...
>>>
```

Since `b > 0` evaluates to `false`, short-circuit evaluation helps skip determining `a/b` in the latter part of the condition within the `if` statement. Had short-circuit evaluation not been in place, `a/b>0` would have raised an error.

7.6 Programming examples

Program 7.12. To print the absolute value of an integer.

```
num=int(input("Enter a number"))
if num>=0:
   abs=num
else:
   abs=-num
print("The absolute value of",num,"is",abs)
```

Program 7.13. To check if the input number is odd or even.

```
num=int(input("Enter a number"))
if num%2==0:
    print(num,"is even")
else:
    print(num,"is odd")
```

Program 7.14. To check if the input number is positive or not.

```
num=int(input("Enter a number"))
if num>0:
    print(num,"is positive")
elif num<0:
    print(num,"is negative")
else:
    print(num,"is neither positive nor negative")
```

Program 7.15. To determine the rate of entry-ticket in a trade fair based on age as follows:

Age	Rate
< 10	7
≥ 10 and < 60	10
≥ 60	5

```
age=int(input("Enter the age"))
if age<10:
    fare=7
elif age>=10 and age<60:
    fare=10
else:
    fare=5
print("The ticket fare is",fare)
```

Program 7.16. To determine the maximum of two numbers.

```
a=int(input("Enter the first number"))
b=int(input("Enter the second number"))
if a>b:
    large=a
else:
    large=b
print("The larger of",a,"and",b,"is",large)
```

Program 7.17. To find the smallest of three numbers.

```
a=int(input("Enter the first number"))
b=int(input("Enter the second number"))
c=int(input("Enter the third number"))
```

```
if a<b:
    small=a
else:
    small=b
if c<small:
    small=c
print("The smallest of",a,",",b,"and",c,"is",small)
```

Program 7.18. To determine whether a year is a leap year or not.

A year is a leap year if any of the following two conditions are true:

1. the year is divisible by 4 but not by 100
2. if the year is divisible by 100, then it should be divisible by 400 too

The second condition is equivalent to writing

2. the year is divisible by 400

as a number divisible by 400 is also divisible by 100. This logic results in the following code:

```
year = int(input("Type a year: "))
if (year%4==0 and year%100!=0) or year%400==0:
    print(year,"is a leap-year")
else:
    print(year,"is not a leap-year")
```

Program 7.19. To solve a quadratic equation.

```
import math
import cmath
a = int(input("Enter the coefficient a: "))
b = int(input("Enter the coefficient b: "))
c = int(input("Enter the coefficient c: "))
d = (b**2) - (4*a*c)
if d > 0:
    print("The equation has two distinct real roots")
    sol1 = (-b+math.sqrt(d))/(2*a)
    sol2 = (-b-math.sqrt(d))/(2*a)
    print("The roots are: ", sol1, "and", sol2)
elif d == 0:
    print("The equation has only one distinct root")
    sol = (-b)/2*a
    print("The root is: ", sol)
else:
    print("The equation has two distinct complex roots")
    sol1 = (-b-cmath.sqrt(d))/(2*a)
    sol2 = (-b+cmath.sqrt(d))/(2*a)
    print("The roots are: ", sol1, "and", sol2)
```

☞ `math.sqrt(d)` requires `d`≥`0`. When `d<0`, you should use `cmath.sqrt(d)` instead. `cmath` means *complex math.*

Program 7.20. To determine the sum of the first n even numbers.

```
n=int(input("Enter the value of n"))
sum=0
for i in range(2,(2*n)+1,2):
    sum+=i
print("The sum of first",n,"even numbers is",sum)
```

Program 7.21. To determine the average of n numbers entered by the user.

```
n=int(input("Enter the value of n"))
sum=0
for i in range(n):
    num=int(input("Enter the number"))
    sum+=num
avg=sum/n
print("The average of the entered numbers is",avg)
```

Program 7.22. To find the sum of all odd numbers in a list of n numbers entered by the user.

```
n=int(input("Enter the value of n"))
sum=0
for i in range(n):
    num=int(input("Enter the number"))
    if num%2!=0: #if the number is odd
        sum+=num  #add to sum
print("The sum of the entered numbers is",sum)
```

Program 7.23. To find the sum of all multiples of 3 in a list of n numbers entered by the user.

```
n=int(input("Enter the value of n"))
sum=0
for i in range(n):
    num=int(input("Enter the number"))
    if num%3==0:
        sum+=num
print("The sum of the entered numbers is",sum)
```

Program 7.24. To find the sum of a list of numbers entered by the user. The user will stop the input by giving the value 999.

```
sum=0
num=int(input("Enter the number"))
while num!=999:
    sum+=num
    num=int(input("Enter the number"))
print("The sum of the entered numbers is",sum)
```

Program 7.25. To determine the largest and smallest in a list of n numbers entered by the user.

```
n=int(input("Enter the count of numbers"))
for count in range(n):
    num=int(input("Enter the number"))
    if count==0:
      large=small=num
    if num>large:
        large=num
    if num<small:
        small=num
print("The smallest number in the entered list of numbers
    is",small)
print("The largest number in the entered list of numbers
    is",large)
```

As was done with pseudocode, we assume the first number input by the user to be the largest as well as the smallest and then compare other numbers with it. This is achieved in the code by initializing the `small` and `large` variables to the first number (when `count` is 0) and then proceeding to compare other numbers with these variables.

Program 7.26. To print the factorial of a number.

```
n=int(input("Enter a number"))
fact=1
for f in range(1,n+1):
    fact=fact*f
print("The factorial of",num,"is",fact)
```

Program 7.27. To check if the input number is a prime.

```
num=int(input("Enter a number"))
if num==1:
    print(num,"is neither prime nor composite")
else:
    flag=0
    for i in range(2,(num//2)+1):
        if num%i==0:
            flag=1
            break
    if flag==0:
        print(num,"is a prime number")
    else:
        print(num,"is not a prime number")
```

Program 7.28. To check if a number is a perfect number or not. A perfect number is one whose value is equal to the sum of its factors. For example, 6 is a perfect number as
$6 = 1 + 2 + 3$

```
num=int(input("Enter a number"))
sum=0
for i in range(1,(num//2)+1):
    if num%i==0:
        sum+=i
if num==sum:
    print(num,"is a pefect number")
else:
   print(num,"is not a pefect number")
```

Program 7.29. To print the reverse of a number.

```
num=int(input("Enter a number"))
n=num
rev=0
while num>0:
    d=num%10
    rev=rev*10+d
    num=num//10
print("The reverse of",n,"is",rev)
```

Program 7.30. To check if a number is an Armstrong number or not. An Armstrong number is one whose value is equal to the sum of the cubes of its digits. For example, 153 is a perfect number as
$153 = 1^3 + 5^3 + 3^3$

```
import math
```

```
num=int(input("Enter a number"))
n=num
s=0
d=0
while n>0:
   n=n//10
   d=d+1
n=num
while n>0:
   r=n%10
   s+=math.pow(r,d)
   n=n//10
if num==s:
   print(num,"is an Armstrong number")
else:
   print(num,"is not an Armstrong number")
```

Program 7.31. To print the Fibonacci series whose terms are less than or equal to a user input limit.

```
limit=int(input("Enter the limit"))
if limit==0:
    print("0")
else:
    prev1=0
    prev2=1
    print(prev1)
    print(prev2)
    next=prev1+prev2
    while next<=limit:
        print(next)
        prev1=prev2
        prev2=next
        next=prev1+prev2
```

Program 7.32. To print the first n terms of Fibonacci series.

```
count=int(input("Enter the number of terms"))
if count==1:
    print("0")
elif count==2:
    print("0")
    print("1")
else:
    prev1=0
    prev2=1
    print(prev1)
    print(prev2)
    for i in range(count-2):
        next=prev1+prev2
        print(next)
        prev1=prev2
        prev2=next
        next=prev1+prev2
```

Program 7.33. To print the prime numbers in a given range.

```
low=int(input("Enter the lower bound"))
high=int(input("Enter the upper bound"))
for num in range(low,high+1):
    flag=True
    for i in range(2,(num//2)+1):
        if num%i==0:
            flag=False
            break
    if flag==True:
        print(num)
```

Program 7.34. To print the Armstrong numbers in a given range.

```
import math
low=int(input("Enter the lower limit"))
up=int(input("Enter the upper limit"))
print("The Armstrong numbers in the input range is")
for num in range(low,up+1):
    n=num
    s=0
    while num>0:
        r=num%10
        s+=math.pow(r,3)
        num=num//10
    if n==s:
        print(n)
```

Program 7.35. To compute the sum of the following series up to n terms:

$$1+\frac{x^2}{2}+\frac{x^4}{4}+\frac{x^6}{6}+\cdots\cdots$$

```
import math
n=int(input("Enter the number of terms"))
x=int(input("Enter the value of x"))
```

```
sum=1
for i in range(1,n):
    sum+=(math.pow(x,2*i))/(2*i)
print("The sum of the series is",sum)
```

Program 7.36. To compute the sum of the following series up to n terms:

$$1 - \frac{x^2}{2} + \frac{x^4}{4} - \frac{x^6}{6} + \cdots\cdots$$

```
import math
n=int(input("Enter the number of terms"))
x=int(input("Enter the value of x"))
sum=1
for i in range(1,n):
    sum=sum+((-1)**i)*((math.pow(x,2*i))/(2*i))
print("The sum of the series is",sum)
```

Program 7.37. To compute the sum of the following series up to n terms:

$$1 + (1+2) + (1+2+3) + \cdots\cdots + (1+2+3+\cdots\cdot+n)$$

```
n=int(input("Enter the number of terms in the series"))
sum=0
for i in range(1,n+1):
    for j in range(1,i+1):
        sum+=j
print("The sum of the series with",n,"terms is",sum)
```

7.7 Conclusion

In this chapter, we explored the fundamental control structures of Python: conditional statements (if, elif, else) and iterative constructs (for loop, while loop, and range). These tools are essential for managing program flow, enabling decision-making, and performing repetitive tasks. Conditional statements allow programs to execute different code blocks based on evaluated conditions, whereas loops enable repeated execution based on specified criteria. We covered various examples demonstrating how these structures can be combined to solve complex problems like factorial calculation and primality testing.

Equipped with control structures, now try to develop a role-playing text-based adventure game. The game should proceed based on the choices made by the user. Use **if-elif-else** to create the basic structure. Also, use the **for loop** to randomize the game incorporating replayability.

7.8 Exercises

1. Recall from your Chemistry classes that aqueous solutions with a pH value less than 7 are acidic, solutions with a pH value greater than 7 are basic and those with pH equal to 7 are neutral. Write an algorithm to determine whether a solution is acidic, basic, or neutral given its pH value.

2. The Population of a town today is 1,00,000. The population has increased steadily at the rate of 10 percent per year for the last 10 years. Write a program to determine the population at the end of each year in the last decade.

3. Write a program to display alternate prime numbers till N (obtain N from the user).

4. Write a program for a number guessing game. The user must have only limited attempts at guessing the number, and for every guess, a hint can be provided to the user. For example, if the user guesses 40, the program can give a hint to make the next guess higher or lower based on the correct answer.

5. Write a program to compute and display the sum of all integers that are divisible by 6 but not by 4, and that lie below a user-given upper limit.

6. Calculate the sum of the digits of each number within a specified range (from 1 to a user-defined upper limit). Print the sum only if it is prime.

7. Write a program to check whether a particular date (dd-mm-yyyy format) is valid. Those dates that lie in the range 01-01-1900 to 31-12-2050 are considered valid.

8. A number is input through the keyboard. Write a program to determine if it's palindromic.

Bibliography

[1] Kenneth A. Lambert, Fundamentals of Python: First Programs, Cengage Learning, 2019.

[2] Yashavant Kanetkar, Let Us Python, BPB Publication, 2022.

[3] John V Guttag, Introduction to Computation and Programming Using Python, MIT Press, 2013.

[4] Allen Downey, How to Think Like a Computer Scientist: Learning with Python, O'Reilly Media, 2015.

Chapter 8

Functions

"What is this brief mortal life?If not the pursuit of legacy"
– Corlys Velaryon

8.1 Email Builder

In a bustling company, the marketing team was drowning in a sea of weekly reports. Each client demanded a personalized email, packed with specific data. Crafting these emails manually was a time-consuming ordeal, prone to errors and leaving little room for strategic thinking. By breaking down the task into manageable chunks - fetching data, building email templates, and sending them off - the team transformed the dreaded chore into an automated process.
Functions can be used to have the same effect on any repetitive task.

8.2 Introduction

Solutions for complex real-world problems are nevertheless going to be simple. Essentially, you should divide your proposed solution into subtasks. This is the idea of decomposition. **Modularization** takes it one step further by keeping the subtasks as independent as possible. These subtasks are known as **modules**. A module is a named, self-contained block of statements that carries out some specific, well-defined task. The modules should have minimal interaction among them so that one module does not affect another.

8.2.1 Motivations for modularization

The modular approach offers several advantages to program development:

1. **Promotes re-use**: If the system functionality is divided into modules, other systems can reuse them. Thus, a programmer can build on what

others have already done, instead of starting from scratch. This also eliminates redundancy (duplication of code) to a great extent.

2. **Hides complexity**: Since a module is self-contained, it does whatever it is intended to. If you want the functionality provided by a module, you access the module and get it done, without worrying about how the functionality is achieved. This mechanism is called *abstraction*.

3. **Supports division of labour**: Ideally, each module achieves a single goal. A complex task could be divided into simpler sub-tasks, each of which can be performed by individual modules. This promotes parallel operation with different teams working on separate modules, speeding up the entire process with improved collaboration among the teams.

4. **Improves debugging and testing**: Since modules have minimal interaction among themselves, you can isolate a module and investigate it separately to pinpoint errors, if any. Each of the modules can be individually tested for correctness and efficiency. This reduces the risk when the modules are integrated to build the big picture.

5. **Contributes to scalability**: New modules can be added to an existing system to enhance its functionality. This makes the system scalable without having to redesign the entire thing.

6. **Reduces development costs**: The reuse of existing modules contributes to cost savings in development. Also, the self-containment aspect of modules makes the system maintenance cost-effective and productive.

In the world of Python, modules are known as **functions**. A Python program can be seen as a collection of functions, each of which serves a unique purpose. It is beneficial to review the box **Functions – the gist** on page 88 before moving on. The portion of the function code that implements its functionality is known as *function definition*. For built-in functions, the definitions are provided by the interpreter itself. But, with user-defined functions, it is the duty of the programmer to *define* them. When we want to use the functionality provided by a function, we access it (technically known as a *function call*) by its name.

8.3 The anatomy of functions

A function should be "defined" before its first use. To define a function, you specify its name and then write the logic for implementing its functionality. A function definition looks like this:

```
def function-name(parameter-list):
    statement-1
    statement-2
    .
    .
    .
    statement-n
    return-statement
```

- Function definition begins with the keyword `def` followed by the function name. Then you have the parameter list – a comma-separated list of arguments enclosed in a pair of parentheses. This line of the function definition is called **header**.
- The block of statements that constitute the function logic (`statement-1` to `return-statement`) is called **function body**.
- The final statement `return` exits a function.
- The header has to end with a colon, and the body has to be indented.
- The parameters and the return statement are optional, i.e., you could write a function without these.

Defining a function is not the end of the game. To use its functionality, you need to *call* it. The function will not get executed unless it is called. You call a function by its name, passing (supplying) the value for each parameter separated by commas and the entire list enclosed in parentheses. If the function call does not require any arguments, an empty pair of parentheses must follow the name of the function. The following example illustrates how a Python function is defined and called:

```
>>> def printName(): #This is how you define a function
...     name=input("Enter your name")
...     print("Hi",name,"! Welcome to the world of functions!")
...
>>> printName() #This is how you call a function
Enter your nameSam
Hi Sam ! Welcome to the world of functions!
```

Having seen the basics of functions, let us now discuss how a function-based program works. The way the statements in a program get executed is called *flow of execution* or *control flow.* Execution always begins at the first statement of the program. Statements are executed one at a time, in order from top to bottom. Function definitions do not alter the flow of execution – statements inside the function body are not executed until the function is called. A function call changes the flow of execution. When a function call is encountered, instead of going to the next statement, the flow jumps to the body of the function, executes all the statements there, and then comes back to pick up where it left off. When the end of the program is reached, the execution terminates.

8.4 Arguments and return value

Arguments are the way you supply data to a function for processing. The function performs the operations on the received data and produces the result which is given back through the `return` statement. The value that is given back is known as the **return value**. Technically, you say that a function "receives" one or more arguments and "returns" a result. The parameters in the function definitions are called **formal parameters**, and those in the function call are called **actual parameters**.

The arguments and the return value are both optional. Thus, you can have four categories of functions as discussed below:

8.4.1 Function with no arguments and no return value

```
>>> def sum():
...     a=int(input("Enter an integer"))
...     b=int(input("Enter another integer"))
...     s=a+b
...     print("The sum of the entered integers is",s)
...
>>> sum()
Enter an integer6
Enter another integer8
The sum of the entered integers is 14
```

8.4.2 Function with arguments but no return value

```
>>> def sum(c,d):
...     s=c+d
...     print("The sum of the entered integers is",s)
...
>>> a=int(input("Enter an integer"))
Enter an integer3
>>> b=int(input("Enter another integer"))
Enter another integer5
>>> sum(a,b)
The sum of the entered integers is 8
```

8.4.3 Function with return value but no arguments

```
>>> def sum():
```

```
...     a=int(input("Enter an integer"))
...     b=int(input("Enter another integer"))
...     s=a+b
...     return s
...
>>> print("The sum of the entered integers is",sum())
Enter an integer4
Enter another integer7
The sum of the entered integers is 11
```

8.4.4 Function with arguments and return value

```
>>> def sum(c,d):
...     s=c+d
...     return s
...
>>> a=int(input("Enter an integer"))
Enter an integer10
>>> b=int(input("Enter another integer"))
Enter another integer15
>>> print("The sum of the entered integers is",sum(a,b))
The sum of the entered integers is 25
```

8.5 Algorithm and flowchart for modules

How do you write pseudocode for a problem solution with multiple modules? For this, you just list out the pseudocode for all the modules. See Figure 8.1 for an example that inputs two numbers, adds them, and prints the sum:

```
AddIntegers
1  Read($a$,$b$)
2  Sum($a$,$b$)

Sum($a$,$b$)
1  $s = a + b$
2  Print($s$)
```

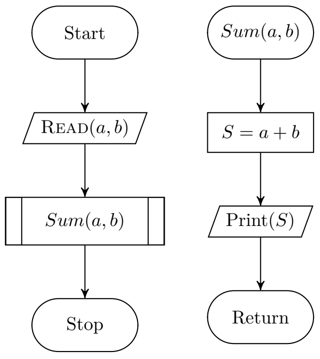


Figure 8.1: Adding two numbers using functions

What about the flowchart? For each module, we draw flowcharts separately. Usually, there will be a main module that takes the inputs and makes the function calls (AddIntegers in the example above). For all modules, the start block will be replaced by the name of the module itself. For all modules (except for the main module) that are being called, the stop block will be replaced by `return`. See below:

8.6 Variable scope and parameter passing

The region of the program text where a variable's value can be accessed is called its **scope**. Thus, the scope of a variable defines the active regions of the variable. The variables defined inside a function are said to have **local scope**. This means if you try to access the value of a variable outside the function where it is defined, you are bound to get an error:

```
>>> def defineVar():
...     var=7
...
>>> defineVar()
>>> print(var)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'var' is not defined.
```

Function calls obey scope rules. When a function is called, the values of the actual parameters are copied to the formal parameters. The changes made, if any, to the formal parameters, are not reflected in the calling function. The changes are made only to the local copy of the formal parameters. See this:

```
>>> def updateVar(var):
...     var+=9
...
>>> var=7
>>> print("The original variable value is ",var)
The original variable value is  7
>>> updateVar(var)
>>> print("The modified variable value is ",var)
The modified variable value is  7
```

8.7 Keyword arguments and default arguments

The arguments for a function can be specified in two ways:

1. Positional arguments

2. Named arguments (aka keyword arguments)

Positional arguments are the default way in which arguments are supplied to a function. Here the values are passed to arguments in the same order in which they occur in the function definition. When the positions of the arguments are changed, the values assigned also change. See an example:

```
>>> def myfun(a,b):
...     print("The values of a and b are respectively",a,b)
...
>>> myfun(3,4)
The values of a and b are respectively 3 4
>>> myfun(4,3)
The values of a and b are respectively 4 3
```

In order not to bother with the order of the parameters, Python has the mechanism of **named arguments** or **keyword arguments**. This allows you to specify the argument names in the function call along with their values. Since the argument names are specified, the order of the arguments doesn't matter. This is illustrated below:

```
>>> def myfun(a,b):
...     print("The values of a and b are respectively",a,b)
...
>>> myfun(a=3,b=4)
The values of a and b are respectively 3 4
>>> myfun(b=4,a=3)
The values of a and b are respectively 3 4
```

Default arguments or **optional arguments** allow you to assign default values to named arguments. The default values for the arguments are included while defining the function. When the function is called without passing values for the default arguments, their default values will be taken. On the other hand, if the function call includes values for default arguments as well, the passed values will override the default values. See below:

```
>>> def sample(a,b=7):
...     print("The values of a and b are respectively",a,b)
...
>>> sample(3)
The values of a and b are respectively 3 7
>>> sample(3,4)
The values of a and b are respectively 3 4
```

You can pass values to default arguments in two ways:

1. **by position**: Here the values are passed in the order in which the argu-

ments occur in the function header.

2. **by name**: Here the values are passed using the argument names in the function call.

This is illustrated below:

```
>>> def sample(a,b=7,c=9):
...     print("The values of a, b and c are respectively",a,b,c)
...
>>> sample(3)
The values of a, b and c are respectively 3 7 9
>>> sample(3,4) # default values taken by position
The values of a, b and c are respectively 3 4 9
>>> sample(3,c=5) # default values taken by name
The values of a, b and c are respectively 3 7 5
```

A function can have both default and mandatory (non-default) arguments. In such cases, the parameter list should start with the mandatory arguments, then followed by the default arguments. Otherwise, you will get an error as shown below:

```
>>> def sample(a=7,b):
  File "<stdin>", line 1
    def sample(a=7,b):
                   ^
SyntaxError: non-default argument follows default argument
```

8.8 Programming examples

Program 8.38. To print the absolute value of an integer.

```
def absoluteValue(x):
   if x < 0:
      return -x
   else:
      return x
num=int(input("Enter a number"))
abs=absoluteValue(num)
print("The absolute value of",num,"is",abs)
```

Program 8.39. To check whether a number is even or not.

```
def isEven(n):
   if n%2==0:
        return True
    else:
        return False
num=int(input("Enter a number"))
r=isEven(num)
if r==True:
    print(num,"is even")
else:
    print(num,"is odd")
```

Program 8.40. To print n lines of the following pattern.

```
1
1 2
1 2 3
1 2 3 4
```

```
def printPattern(n):
    for i in range(1,n+1):     #each value of i corresponds to
    one line
        for j in range(1,i+1): #for printing numbers on a
    particular row
            print(j,end=" ")
        print()                  #to go to next line after
    printing one row
n=int(input("Enter the number of lines"))
printPattern(n)
```

Program 8.41. To print n lines of the following pattern.

```
1
1 3
1 3 5
1 3 5 7
```

```
def printPattern(n):
    for i in range(1,n+1):     #each value of i corresponds to
    one line
        for j in range(1,2*i,2): #for printing numbers on a
    particular row
            print(j,end=" ")
        print()                  #to go to next line after
    printing one row
n=int(input("Enter the number of lines"))
```

```
printPattern(n)
```

Program 8.42. To print n lines of the following pattern.

```
2
2 4
2 4 6
2 4 6 8
```

```
def printPattern(n):
    for i in range(1,n+1):     #each value of i corresponds to
    one line
        for j in range(2,2*i+1,2): #for printing numbers on a
    particular row
            print(j,end=" ")
        print()                   #to go to next line after
    printing one row
n=int(input("Enter the number of lines"))
printPattern(n)
```

Program 8.43. To implement a menu-driven calculator. Use separate functions for the different operations

```
def add(x, y):
    return x + y
def subtract(x, y):
    return x - y
def multiply(x, y):
    return x * y
def divide(x, y):
    return x / y
num1 = int(input("Enter first number: "))
num2 = int(input("Enter second number: "))
print("Select operation.")
print("1.Add")
print("2.Subtract")
print("3.Multiply")
print("4.Divide")
choice = int(input("Enter your choice"))
if choice==1:
    result=add(num1,num2)
    print(num1,"+",num2,"=",result)
elif choice==2:
    result=subtract(num1,num2)
    print(num1,"-",num2,"=",result)
```

```
elif choice==3:
    result=multiply(num1,num2)
    print(num1,"*",num2,"=",result)
elif choice==4:
    result=divide(num1,num2)
    print(num1,"/",num2,"=",result)
else:
    print("Invalid choice")
```

Program 8.44. To find the sum of a range of numbers.

```
def sum(l,h):
    s=0
    for i in range(l,h+1):
        s+=i
    return s
low=int(input("Enter the lower limit"))
high=int(input("Enter the upper limit"))
a=sum(low,high)
print("The sum of the numbers in the range is",a)
```

Program 8.45. To print the n^{th} prime number.

```
def is_prime(num):
    for i in range(2,int((num/2))+1):
        if (num % i) == 0:
            return False
    return True
count=0
num=2
n=int(input("Enter the value of n"))
while count<n:
    if is_prime(num)==True:
        count+=1
        if count==n:
            print("The",n,"th prime number is",num)
    num+=1
```

8.9 Conclusion

Functions are essential Python constructs for creating reusable code blocks. They improve code organization, readability, and efficiency. By defining functions with parameters and return values, you can modularize complex tasks. Understanding function scope is crucial for managing variables within and outside functions. Default, positional, and keyword arguments provide flexibility in function calls. In a nutshell, functions are indispensable for building well-structured and maintainable Python programs.

8.10 Exercises

1. Answer the following questions based on the given code snippet:

```
x = 10
def outer_function(modify=True):
  global x
  y = 20

  def inner_function():
    global x
    x = x + y
    return x

  if modify:
    inner_function()
  else:
    x = inner_function() + 10

  return x

print(outer_function())
```

   (a) What is the output of the above code?
      i. 30
      ii. 40
      iii. Gives a Error Message
      iv. 50
   (b) What is the output of the above code if the function call is edited as `outer_function(False)`?
      i. 30
      ii. 40
      iii. Gives a Error Message
      iv. 50

2. Write a program to input a number and find if it's a twisted prime or not. (A number is said to be **twisted prime** if it is a prime number and its reverse is also prime. Eg: 97 is a twisted prime number)

3. Write a program to print the prime factors of a number. (Use a method isPrime() to check primality).

4. Write a function called `determineParkingFare()` that calculates the parking fare for customers who park their vehicles in a parking lot. The function takes the following arguments-

   - An integer denoting the type of vehicle: 0 for car, 1 for bus, and 2 for truck.
   - An integer between 0 and 24 denoting the hour the vehicle entered the lot.
   - An integer between 0 and 60 denoting the minute the vehicle entered the lot.
   - An integer between 0 and 24 denoting the hour the vehicle left the lot.
   - An integer between 0 and 60 denoting the minute the vehicle left the lot.

   The fare is determined based on the following table:-

Vehicle	Duration	Charges per hour
Car	For first 3 hours	Rs. 2
Car	Remaining duration	Rs. 4
Truck	For first 3 hours	Rs. 4
Truck	Remaining duration	Rs. 6
Bus	For first 3 hours	Rs. 5
Bus	Remaining duration	Rs. 7

   Since there are no fractional charges, round the total duration to the next highest hour for finding the fare. You should now write a program to determine the parking fare of n customers.

   The fare calculation for a single customer is shown below. (Assume that the vehicle entered at 13:25 hours and left at 18:20 hours)
   Type of vehicle 1
   Time In(Hours) 13
   Time In(Minutes) 25
   Time Out(Hours) 18
   Time Out(Minutes) 20
   Duration=18.33 (18 $\frac{20}{60}$ hours) - 13.42 (13 $\frac{25}{60}$ hours) = 4.91 hours
   Rounded duration = 5 hours
   Parking fare = 3×4 + 2×6 = 24 rupees

5. Write a program to implement these formulae for permutations and combinations.
   The Number of permutations of n objects taken r at a time:
   $p(n,r) = n!/(n-r)!$.
   The Number of combinations of n objects taken r at a time is:
   $c(n,r) = n!/(r! * (n-r)!)$

6. Generate and print Pascal's Triangle up to a specified number of rows. Pascal's Triangle is a triangular array where each number is the sum of the two numbers directly above it.
   HINT: Use the concept of permutation combination to obtain the sequence of the rows

```
    1
   1 1  -->     c(1,0)  c(1,1)
  1 2 1  --> c(2,0)   c(2,1)   c(2,2)
 1 3 3 1
1 4 6 4 1
```

Bibliography

[1] Kenneth A. Lambert, Fundamentals of Python: First Programs, Cengage Learning, 2019.

[2] Yashavant Kanetkar, Let Us Python, BPB Publication, 2022.

[3] John V Guttag, Introduction to Computation and Programming Using Python, MIT Press, 2013.

[4] Allen Downey, How to Think Like a Computer Scientist: Learning with Python, O'Reilly Media, 2015.

Chapter 9

Strings

"Just as words are the foundation of writing, strings are the foundation of text processing in programming."

- Anonymous

9.1 Event Invitation Generator

Imagine an event invitation generator that eliminates the hassle of planning parties and gatherings. You simply input the details—like the event name, date, and guest list—and the tool automatically creates personalized invitations for each guest. It even customizes messages based on the recipient's relationship with the host, making every invite feel unique and thoughtful. Behind the scenes, this program uses string manipulation to merge your input with beautifully crafted text templates, ensuring that each invitation is both professional and personal, all done with just a few clicks.

9.2 Introduction

A string is a **sequence** of zero or more characters. Each character in the string occupies one byte of memory, and the last character is always '`\0`' which is called the *null character*. The terminating null character is important because it is the way by which, the Python interpreter knows where the string ends.

Figure 9.1 shows how a string "`Hello World`" is stored in memory. Notice the null character at the end. The figure also shows how the character positions are numbered starting with 0 from the left end or starting with -1 from the right end. Each character position is known as an *index*. The null character position is not numbered. It is not considered a part of the string; its sole purpose is to know the string boundary.

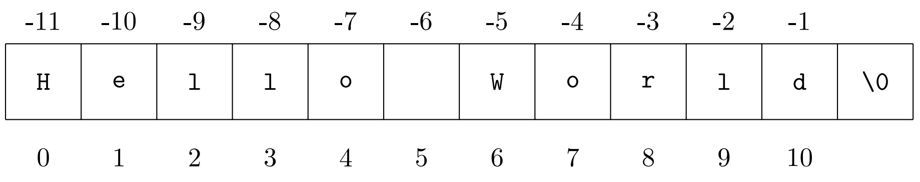


Figure 9.1: A string stored in memory

To know the length of a string, you use the `len()` method as shown below:

```
>>> str="Hello World"
>>> print(len(str))
11
```

As expected, the null character is not counted while finding the string length.

9.3 The subscript operator

To print a single character out of the string, you should use the **subscript operator**. The subscript operator (also called bracket operator) is denoted by `[ ]`. You just want to mention the index whose character you want to print. Here are some illustrations:

```
>>> str="Algorithmic thinking"
>>> print(str[0])
A
>>> print(str[5])
i
>>> print(str[11])

>>> print(str[18])
n
>>> print(str[20])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> print(str[19])
g
```

In the example above, the string length is 20; this means that the last index is 19. Thus, if you try to print the character in the 20^{th} position, you get an error.

Python also allows negative subscript values. In this case, you need to count backward from -1 to access characters from the right end of the string. See below:

```
>>> str="Try out!"
>>> print(str[-1])
!
>>> print(str[-3])
u
>>> print(str[-9])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

It is invalid to use the `[ ]` operator on the left side of an assignment, with the intention of changing a character in a string. See the example below:

```
>>> pet="cat"
>>> pet[0]='r'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

As expected, the code results in an error (Thank God! Who wants a rat as a pet?).
The “object” mentioned in the error is the string and the “item” is the character we tried to assign. The reason for the error is that strings are **immutable**, which means you can’t change an existing string.

9.4 Traversing a string

Accessing the string characters one at a time, starting from the beginning and moving towards the end, is called **traversal**. The `for` loop comes in handy for this:

```
>>> str="Hi there"
>>> for c in str:
...     print(c)
...
H
i

t
h
e
r
e
```

9.5 Concatenation and repetition operators

The + operator performs **concatenation**, which means joining the strings by linking them end-to-end. See the example:

```
>>> str1="Python"
>>> str2="Program"
>>> print(str1+str2)
PythonProgram
```

Note that the concatenation operator does not insert any white space between the joined strings.

The * operator performs repetition. It repeats a given string some fixed number of times. The left operand for * is the string to be repeated, and the right operand is an integer denoting the number of times the string is to be repeated. See the code:

```
>>> str="Order!"
>>> print(str*3)
Order!Order!Order!
```

9.6 String slicing

To extract a substring out of a given string, you should use subscript operator `[:]`. The operator `[n:m]` returns the part of the string starting at position `n` and ending at position `m-1`. See the example below:

```
>>> languages="Python, Java and C++"
>>> print(languages[0:6])
Python
>>> print(languages[8:12])
Java
>>> print(languages[17:20])
C++
```

If n≥m, the operator `[n:m]` results in an empty string. See the code:

```
>>> languages="Python, Java and C++"
>>> print(languages[4:4])

>>>
```

The indices `n` and `m` in the operator `[n:m]` are optional. If you omit the first index `n`, the slice starts at the beginning of the string. If you omit the second index `m`, the slice goes to the end of the string. An example follows:

```
>>> languages="Python, Java and C++"
>>> print(languages[:6])
Python
>>> print(languages[17:])
C++
```

When `m` is omitted, supplying a negative value to the index `n` prints the last few characters of the string. To be specific, `str[-k:]` prints the last `k` characters of `str`. See the example below:

```
>>> languages="Python, Java and C++"
>>> print(languages[-3:])
C++
```

Omitting both indices `n` and `m` prints the full string. See below:

```
>>> languages="Python, Java and C++"
>>> print(languages[:])
Python, Java and C++
```

In addition to mentioning start and end indices in the slicing operator, you can also specify the step that denotes the distance between the characters you want to include in the resultant string. Suppose you want to print every alternate character in a string `str`, you just use the operator `[::2]`. See an example:

```
>>> languages="Python, Java and C++"
>>> print(languages[::2])
Pto,Jv n +
```

Since the start and end indices are omitted, the slice starts from the beginning of the string and goes to its end. Since the step is mentioned as 2, every alternate character will be selected.

9.7 Comparison operators on strings

The comparison operators work on strings as well. To see if two strings are equal, we simply use the equality operator:

```
>>> str1="python"
>>> str2="pythen"
>>> print(str1==str2)
False
>>> str2="python"
>>> print(str1==str2)
```

```
True
```

Other comparison operations are useful for determining the lexicographic (alphabetical) order of strings. The comparison operator when used with strings compares the ASCII values of the corresponding characters in the strings. Since the ASCII values are assigned in increasing order, a character that has a lower ASCII value precedes (comes before alphabetically) another character with a higher ASCII value. See the code below:

```
>>> str1="C"
>>> str2="Java"
>>> if str1 < str2:
...     print(str1,"comes before",str2)
... elif str2 < str1:
...     print(str2,"comes before",str1)
... else:
...     print("The two strings are the same")
...
C comes before Java
```

9.8 The `in` operator

`in` is a boolean operator that takes two strings and returns `True` if the first appears as a substring in the second and `False` otherwise. See the example below:

```
>>> str1="C"
>>> str2="C++"
>>> str3="Python"
>>> print(str1 in str2)
True
>>> print(str1 in str3)
False
>>> print(str2 in str1)
False
```

The `in` operator can be used to print the characters that are present in (common to) two given strings as follows:

```
>>> str1="algorithms"
>>> str2="problem solving"
>>> for c in str1:
...     if c in str2:
...             print(c,end=" ")
```

```
...
l g o r i m s
```

9.9 The `string` module

The `string` module contains **string methods** that can be used to manipulate strings. Table 9.1 lists some of the string methods. To use them, you have to first import the module:

```
import string
```

Table 9.1: Some useful string methods, `s` represents to a string

String method	**Purpose**
`s.count(sub, start, end)`	Returns the number of non-overlapping occurrences of substring `sub` in the portion of `s` starting from `start` and ending at `end-1`.
`s.find(sub, start, end)`	Returns the index of the first occurrence of the substring `sub` in the portion of `s` starting from `start` and ending at `end-1`.
`s.replace(old, new, count)`	Returns a copy of `s` with all occurrences of substring `old` replaced by `new`. If the optional argument `count` is given, only the first `count` occurrences are replaced.
`s.isalpha()`	Returns `True` if `s` contains only letters or `False` otherwise.
`s.isdigit()`	Returns `True` if `s` contains only digits or `False` otherwise.
`s.lower()`	Returns a copy of `s` converted to lowercase.
`s.upper()`	Returns a copy of `s` converted to uppercase.

☞The parameters `start` and `end` are optional in the methods `find()` and `count()`. If not present, the entire string is considered.

☞`find()` returns `-1` if the substring to be searched is not present in the given string.

The following example illustrates the use of some of the string methods.

```
>>> import string
>>> s="malayalam"
>>> str1="string"
>>> str2="STR2"
>>> print(s.find("al"))
1
>>> print(s.find("al",2))
5
>>> print(s.find("al",2,5))
-1
>>> print(s.count("al"))
2
>>> print(s.count("al",2))
1
>>> print(s.count("al",2,5))
0
>>> print(s.replace("la","pq"))
mapqyapqm
>>> print(s.replace("la","pq",1))
mapqyalam
>>> print(str1.isalpha())
True
>>> print(str2.isalpha())
False
>>> print(str1.upper())
STRING
>>> print(str2.lower())
str2
```

In addition to string methods, the `string` module also provides several string constants; some of which are illustrated below:

```
>>> import string
>>> print(string.ascii_lowercase)
abcdefghijklmnopqrstuvwxyz
```

```
>>> print(string.ascii_uppercase)
ABCDEFGHIJKLMNOPQRSTUVWXYZ
>>> print(string.ascii_letters)
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
>>> print(string.digits)
0123456789
```

You can use these constants to test whether a character is upper-case or lower-case, or whether it is a digit. See below:

```
>>> import string
>>> ch='p'
>>> print(ch in string.ascii_lowercase)
True
```

9.10 Programming examples

Program 9.46. To count the occurrences of a character in a string without using the `count` method.

```
str=input("Enter a string")
ch=input("Enter the character to search for")
count=0
for char in str:
    if char==ch:
        count+=1
print(ch,"occurs",count,"times in",str)
```

Program 9.47. To count the occurrences of lower-case letters, upper-case letters, vowels, digits, and blank spaces in a string.

```
import string
def is_lower(s):
    return s in string.ascii_lowercase
def is_upper(s):
    return s in string.ascii_uppercase
def is_digit(s):
    return s in string.digits
def is_vowel(s):
    return s in "aeiouAEIOU"
def is_consonant(s):
    return s in "bcdfghjklmnpqrstvwxyzBCDFGHJKLMNPQRSTVWXYZ"
def is_space(s):
    return s in string.whitespace
```

```
str=input("Enter a string")
countL=countU=countD=countV=countC=countS=0
length=len(str)
for char in str:
    if is_lower(char)==True:
        countL+=1
    if is_upper(char)==True:
        countU+=1
    if is_digit(char)==True:
        countD+=1
    if is_vowel(char)==True:
        countV+=1
    if is_consonant(char)==True:
        countC+=1
    if is_space(char)==True:
        countS+=1
if is_space(str[0])==False and is_space(str[length-1])==False:
    countW=countS+1
elif is_space(str[0])==True and is_space(str[length-1])==True:
    countW=countS-1
else:
    countW=countS
print("The string",str,"has")
print(countL,"lower case letters")
print(countU,"upper case letters")
print(countD,"digits")
print(countV,"vowels")
print(countC,"consonants")
print(countS,"blank spaces")
print(countW,"words")
```

Program 9.48. To print all the prefixes and suffixes of a string.

```
str=input("Enter a string")
length=len(str)
print("The prefixes of",str,"are")
for index in range(1,length+1):
    print(str[0:index])
print("The suffixes of",str,"are")
for index in range(length):
    print(str[index:length])
```

This code will produce the following output when the string `python` is input:

```
The prefixes of python are
p
```

```
py
pyt
pyth
pytho
python
The suffixes of python are
python
ython
thon
hon
on
n
```

Program 9.49. To reverse a string.

```
def reverse(str):
    r = ""
    index = len(str)-1
    while index>=0:
        r += str[index]
        index -= 1
    return r
s=input("Enter a string")
rev=reverse(s)
print("The reverse of the string is",rev)
```

Program 9.50. To check whether a string is a palindrome or not. Use functions.

```
def is_palindrome(str):
    length = len(str)
    if length==1:
        return True
    else:
        for i in range(length//2):
            if str[i] != str[length-1-i]:
                return False
    return True
s=input("Enter a string")
flag=is_palindrome(s)
if flag==True:
    print(s,"is a palindrome")
else:
    print(s,"is not a palindrome")
```

Program 9.51. To check whether a string is a palindrome or not. Use recursion.

```
def is_palindrome(str):
    length=len(str)
    if length == 1:
        return True
    if str[0] != str[length-1]:
        return False
    else:
        return is_palindrome(str[1:length-1])
s=input("Enter a string")
flag=is_palindrome(s)
if flag==True:
    print(s,"is a palindrome")
else:
    print(s,"is not a palindrome")
```

Program 9.52. To convert a binary number to decimal.

```
bstring = input("Enter a binary number")
decimal = 0
exponent = len(bstring) - 1
for digit in bstring:
    decimal = decimal + int(digit)*2**exponent
    exponent = exponent - 1
print("The decimal equivalent of",bstring,"is", decimal)
```

Program 9.53. To convert a decimal number to binary.

```
decimal = int(input("Enter a decimal integer"))
if decimal == 0 or decimal == 1:
    binary=decimal
else:
    d=decimal
    binary = " "
    while decimal > 0:
        remainder = decimal % 2
        decimal = decimal // 2
        binary = str(remainder) + binary
print("The binary equivalent of",d,"is",binary)
```

9.11 Conclusion

This chapter provided a comprehensive introduction to strings in Python, starting from the basics and gradually covering more advanced topics. We explored how strings are represented and stored in memory, and learned various techniques to manipulate them, such as traversing, concatenating, and slicing. By understanding the use of string operators and various methods from the string module, you now have the foundational knowledge needed to work efficiently with strings in Python. With these tools at your disposal, you can handle text data more effectively in your future programming endeavors.

9.12 Exercises

1. Predict the output:

```
str="bcfhmprs"
for char in str:
    print(char+"at",end=" ")
```

2. Write a program to input a character and check if it's a special character or not.

3. Write a program to count the number of words in a sentence.

4. Write a program that takes a binary number as string input and prints whether the number is odd or even.

5. Write a program to input a word and replace all alternate characters with $.

6. Write a program to read a sentence. Print all capital letters first, then small letters, then digits, and at last special characters. Print it as a new word.

7. Write a program to input a sentence. Frame a new sentence with the last word first and then the first word. Print it in capital letters.
   Eg: Input : Exam is Easy Output : EASY EXAM.

8. Write a program to input a word and display it in PIGLATIN form.
   (The first vowel occurring in the input word is placed at the start of the new word along with the remaining alphabets of it. The alphabets present before the first vowel ia shifted at the end of the new word, followed by "ay".)
   Eg: Input = PARIS Output = ARISPay
   Input = amazon Output = amazonay

9. Write a program to input a word and print it in the pattern as shown below.
   eg: For **PIZZA**, print like below:
   ```
   A
   ZZ
   ZZZ
   IIII
   PPPPP
   ```

Bibliography

[1] Kenneth A. Lambert, Fundamentals of Python: First Programs, Cengage Learning, 2019.

[2] Yashavant Kanetkar, Let Us Python, BPB Publication, 2022.

[3] John V Guttag, Introduction to Computation and Programming Using Python, MIT Press, 2013.

[4] Allen Downey, How to Think Like a Computer Scientist: Learning with Python, O'Reilly Media, 2015.

Chapter 10

Lists and tuples

"The art of simplicity is a puzzle of complexity."

- Douglas Horton

"Immutability is a powerful concept; it ensures that data remains reliable and predictable."

- Robert C. Martin

10.1 Restaurant Order Management System

In a busy restaurant, waiters struggled to track multiple orders, especially during peak hours. The process of writing down orders, calculating bills manually, and updating the kitchen staff was time-consuming and prone to errors. This often led to mix-ups and frustrated customers.

This can be automated using lists and tuples where we store each order as a list of items, with each item represented as a tuple containing the dish name, quantity, and price. This allows the system to easily calculate totals, apply discounts, and send correct orders to the kitchen. The result is a streamlined order management system that reduces errors, improves service speed, and allows the staff to focus more on customer satisfaction.

Lists and tuples help to manage and process complex orders efficiently, ensuring smooth operation during busy periods.

10.2 Lists

A list is an ordered set of data values. The values that make up a list are called its **elements** or **items**. The logical structure of a list is similar to that of a string. Each item in the list has a unique index that specifies its position and the items are ordered by their positions. Unlike strings, where each element is a character, the items in a list can be of different data types.

10.2.1 Creating lists

To create a new list; you simply enclose the elements in a pair of square brackets (`[ ]`). Following are some examples:

```
>>> newlist=[10,"hi",3.14]
>>> print(newlist)
[10, 'hi', 3.14]
>>> mynest=[2,4,[1,3]]
>>> print(mynest)
[2, 4, [1, 3]]
```

As in `mynest` above, a list can have another list as its element. Such a list is called **nested list**. The list `[1,3]` which is an element of `mynest` is called a *member list*. A list that contains no elements is called an *empty list*, denoted as `[ ]`.

Lists of integers can also be built using the `range()` and `list()` functions. See some examples now:

```
>>> digits=list(range(10))
>>> print(digits)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> naturals=list(range(1,11))
>>> print(naturals)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> mul5=list(range(10,50,5))
>>> print(mul5)
[10, 15, 20, 25, 30, 35, 40, 45]
```

Once a list is created, you can determine its length (number of elements in the list) using the `len()` method. In the case of a nested list, the member list will be counted as a single element. See examples below:

```
>>> languages=["C","Java","Python","C++"]
>>> print(len(languages))
4
>>> breakfast=["dosa","chapathi",["bread","butter","jam"]]
>>> print(len(breakfast))
3
```

10.2.2 Accessing list elements and traversal

List indices work the same way as string indices. Thus you can use the subscript operator to access the list elements. See some examples:

```
>>> newnest=[10, 20, [25, 30], 40]
>>> print(newnest[0])
10
>>> print(newnest[2])
[25, 30]
>>> print(newnest[-1])
40
>>> print(newnest[4])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> print(newnest[5-4])
20
>>> print(newnest[1.0])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list indices must be integers or slices, not float
>>> print(newnest[-5])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

You could also use an expression as an index in the subscript operator, as in `newnest[5-4]` in the example above. The interpreter will evaluate the expression to determine the index.

Printing the list by specifying its name in the `print` method displays the list elements enclosed in a pair of brackets. To display the entire list without the brackets themselves, you should traverse the list printing the elements one by one. See below:

```
>>> prime=[2,3,5,7,11]
>>> for p in prime:
...     print(p,end=" ")
...
2 3 5 7 11
```

10.2.3 List Comprehension

Creating a new list from an existing list is called **list comprehension**. Assume you have a list of numbers from which you want to create another list consisting only of odd numbers in the original list. This can be done by using a `for` loop to traverse the original list and copying the odd numbers to a second list. But with list comprehension, life is far more easy! See below:

```
>>> numbers = [1, 28, 73, 4, 100, 358, 75, 208]
>>> odd = [num for num in numbers if num%2 != 0]
>>> print(odd)
[1, 73, 75]
```

The statement

`odd = [num for num in numbers if num%2 != 0]`

essentially directs the interpreter to iterate through the list `numbers` and copy all odd numbers to another list `odd`.

Suppose, now you need to create a list of two-digit even numbers, you could use comprehension:

```
>>> even2dig = [num for num in range(10,100) if num%2 == 0]
>>> print(even2dig)
[10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38,
    40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68,
    70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98]
```

Here, `range(10,100)` generates two digit numbers and the condition `if num%2 == 0` makes sure that only the even numbers are included in the constructed list.
Finally, assume you want to replace all multiples of 5 (less than 30) with -1, you just resort to comprehension:

```
>>> nomul5 = [num if num%5 != 0 else -1 for num in range(20)]
>>> print(nomul5)
[-1, 1, 2, 3, 4, -1, 6, 7, 8, 9, -1, 11, 12, 13, 14, -1, 16,
    17, 18, 19]
```

The interpreter generates two-digit numbers less than 20 and then puts the generated number into the constructed list if it is not a multiple of 5, otherwise puts a -1 into the list.

10.2.4 List operations

The `+` operator concatenates lists:

```
>>> prime=[2,3,5,7]
>>> composite=[4,6,8,10]
>>> numbers=prime+composite
>>> print(numbers)
```

```
[2, 3, 5, 7, 4, 6, 8, 10]
```

The * operator repeats a list a given number of times:

```
>>> binary=[0,1]
>>> bytesequence=binary*4
>>> print(bytesequence)
[0, 1, 0, 1, 0, 1, 0, 1]
```

Equality operator works well on lists. It checks if two lists have the same elements. See an example:

```
>>> even=[2,4,6,8]
>>> mul2=[2,4,6,8]
>>> print(even==mul2)
True
>>> composite=[4,6,8]
>>> print(even==composite)
False
```

Other relational operators also work with lists. Consider:

```
>>> prime=[2,3,5]
>>> even=[2,4,6]
>>> print(prime>even)
False
```

Python starts by comparing the first element from each list. If they are equal, it goes on to the next element, and so on, until it finds the first pair of elements that are different and determines the relation between them. In the above example, `prime[0]` == `even[0]`. Next, `prime[1]` and `even[1]` are compared. Thus the resulting relation is '<' and `prime` > `even` is thus False. Note that once the result is determined, the subsequent elements are skipped. See below:

```
>>> prime=[2,3,7]
>>> even=[2,4,6]
>>> print(prime>even)
False
```

Membership operators can be applied to a list as well. See below:

```
>>> even=[2,4,6,8]
>>> composite=[4,6,8]
>>> print(2 in even)
True
>>> print(2 in composite)
False
```

```
>>> print(3 not in composite)
True
```

10.2.5 List slices

The slice operator on a list follows the same rules as applied to strings. See some illustrations:

```
>>>sequence=["strings","lists","tuples","bytearrays",
"bytesequences","range objects"]
>>> print(sequence[:])
['strings', 'lists', 'tuples', 'bytearrays', 'bytesequences',
    'range objects']
>>> print(sequence[1:4])
['lists', 'tuples', 'bytearrays']
>>> print(sequence[:3])
['strings', 'lists', 'tuples']
>>> print(sequence[3:])
['bytearrays', 'bytesequences', 'range objects']
```

10.2.6 List mutations

Unlike strings, lists are **mutable**. In other words, a list is updatable – elements can be inserted, removed, or replaced. You use the subscript operator to replace an element at a given position:

```
>>> even=[2,4,5,8]
>>> even[2]=6
>>> print(even)
[2, 4, 6, 8]
```

You can also replace a single list item with a new list. See below:

```
>>> even=[2,4,6,8]
>>> even[3]=[8,10]
>>> print(even)
[2, 4, 6, [8, 10]]
```

10.2.6.1 Slice operator and mutations

The slice operator is a nice tool to mutate lists in terms of replacing, inserting, or removing list elements.

Replacing elements

You can use the slice operator to replace a single element or multiple elements in a list. See an example below:

```
>>> composite=[13,17,19,23,25,27,37,41]
>>> composite[4:6]=[29,31]
>>> print(composite)
[13, 17, 19, 23, 29, 31, 37, 41]
>>> odd=[1,3,5,8]
>>> odd[3:4]=[7]
>>> print(odd)
[1, 3, 5, 7]
```

Inserting elements

The slice operator is useful for inserting new elements into a list at the desired location:

```
>>> prime=[2,3,5,7,13,17,19]
>>> prime[4:4]=[11]
>>> print(prime)
[2, 3, 5, 7, 11, 13, 17, 19]
>>> composite=[2,10,12]
>>> composite[1:1]=[4,6,8]
>>> print(composite)
[2, 4, 6, 8, 10, 12]
```

Removing elements

The slice operator can also be used to remove elements from a list by assigning the empty list to them. Examples follow:

```
>>> composite=[3,5,7,9,11]
>>> composite[3:4]=[]
>>> print(composite)
[3, 5, 7, 11]
>>> composite=[29,31,33,35,37]
>>> composite[2:4]=[]
>>> print(composite)
[29, 31, 37]
```

10.2.6.2 Using `del` keyword for deletion

Assigning a list element to an empty list for deletion is cumbersome. As an alternative, Python provides the `del` keyword exclusively for deletion. See below:

```
>>> composite=[3,5,7,9,11]
>>> del composite[3]
>>> print(composite)
[3, 5, 7, 11]
>>> composite=[29,31,33,35,37]
>>> del composite[2:4]
>>> print(composite)
[29, 31, 37]
>>> del composite[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

As illustrated above, `del` causes a runtime error if the index is out of range.

10.2.7 List methods

The `list` type includes several methods for inserting and removing elements. These methods are summarized in Table 10.1.

Table 10.1: Some useful list methods. `L` denotes a list

List method	**Purpose**
`L.index(element)`	Returns the position of `element` in the list `L`. It produces an error if there is no such element.
`L.insert(position, element)`	Inserts `element` at `position` if `position` is less than the length of `L` . Otherwise, inserts `element` at the end of `L` .
`L.append(element)`	Inserts `element` at the end of `L`.
`L.extend(aList)`	Inserts the elements of the list `aList` to the end of `L`.
`L.remove(element)`	Removes `element` from the list `L`. It throws an error if there is no such element.
`L.pop()`	Removes and returns the element at the end of `L` .
`L.pop(position)`	Removes and returns the element at `position` in the list `L`.
`L.count(element)`	Returns the number of times `element` appears in the list `L`.
`L.sort(element)`	Sorts (arrange in ascending order) the elements of the list `L`.
`L.reverse()`	Reverses the elements of the list `L`.

The following illustrates the use of the various list methods:

```
>>> a = [66.25, 333, 333, 7, 1, 1234.5]
>>> print("Index of 1 is",a.index(1))
Index of 1 is 4
>>> print("Index of 333 is",a.index(333))
Index of 333 is 1
>>> a.insert(2,4587)
>>> a.insert(8,-55)
>>> print("After inserting 4587 and -55, the list is",a)
After inserting 4587 and -55, the list is [66.25, 333, 4587,
    333, 7, 1, 1234.5, -55]
>>> a.append(200)
>>> print("The list on appending 200 is",a)
The list on appending 200 is [66.25, 333, 4587, 333, 7, 1,
    1234.5, -55, 200]
>>> a.sort()
>>> print("The list after sorting is",a)
The list after sorting is [-55, 1, 7, 66.25, 200, 333, 333,
    1234.5, 4587]
>>> b=['a',25.47,'c',7,18]
>>> a.extend(b)
>>> print("The list a after extending with elements of b is",a)
The list a after extending with elements of b is [-55, 1, 7,
    66.25, 200, 333, 333, 1234.5, 4587, 'a', 25.47, 'c', 7, 18]
>>> a.remove(1)
>>> print("The list a after removing 1 is",a)
The list a after removing 1 is [-55, 7, 66.25, 200, 333, 333,
    1234.5, 4587, 'a', 25.47, 'c', 7, 18]
>>> a.remove(333)
>>> print("The list a after removing 333 is",a)
The list a after removing 333 is [-55, 7, 66.25, 200, 333,
    1234.5, 4587, 'a', 25.47, 'c', 7, 18]
>>> x=a.pop()
>>> print("The element popped from the list a is",x)
The element popped from the list a is 18
>>> a.pop(6)
4587
>>> print("The list a after popping element at position 6 is",a)
The list a after popping element at position 6 is [-55, 7,
    66.25, 200, 333, 1234.5, 'a', 25.47, 'c', 7]
>>> print("The list a has",a.count(7),"occurrences of 7")
The list a has 2 occurrences of 7
>>> a.reverse()
>>> print("The list a in reverse is ",a)
The list a in reverse is  [7, 'c', 25.47, 'a', 1234.5, 333,
    200, 66.25, 7, -55]
```

10.2.8 Lists and functions

A function can be made to return multiple values using lists. This is done by defining the function to return a list of values instead of a single value as in the normal case. The following code illustrates this:

```
>>> def printTop3(list):
...     list.sort()
...     list.reverse()
...     top3=[list[i] for i in range(3)]
...     return(top3)
...
>>> mylist=[23,1,78,50,100,-5]
>>> printTop3(mylist)
[100, 78, 50]
```

10.2.9 Strings and lists

To construct a list out of the characters from a string, you can use `list()` method as follows:

```
>>> str="two words"
>>> strlist=list(str)
>>> print(strlist)
['t', 'w', 'o', ' ', 'w', 'o', 'r', 'd', 's']
```

If you want to split a multi-word string into its constituent words and construct a list out of those words, you should use the `split()` method:

```
>>> str="two words"
>>> strwords=str.split()
>>> print(strwords)
['two', 'words']
```

In the above example, the white space where you split the string is known as the **delimiter**. If you want to specify a different delimiter, you can pass that character (or even a string) as an argument to the `split()` method. The following example illustrates this:

```
>>> str="two words"
>>> wsplit=str.split('w')
>>> print(wsplit)
['t', 'o ', 'ords']
>>> wosplit=str.split('wo')
>>> print(wosplit)
['t', ' ', 'rds']
```

Notice that the delimiter doesn't appear in the list.

The `join()` method works in the opposite sense of `split()`. It concatenates a list of strings by inserting a "separator" between them. The following code uses a blank space as the separator.

```
>>> sep=" "
>>> wordlist=["three", "word", "string"]
>>> str=sep.join(wordlist)
>>> print(str)
three word string
```

The following code uses a hyphen as a separator.

```
>>> sep="-"
>>> wordlist=["three", "word", "string"]
>>> str=sep.join(wordlist)
>>> print(str)
three-word-string
```

10.2.10 List of lists

All the elements in a list may be again lists. Such a list is called **list of lists**. Here is one example:

```
lol=[[1], [15, 9], [108, 778]]
```

Since the constituent elements themselves are lists, to access the items in these constituents, you need to use the subscript operator twice as illustrated below:

```
>>> lol=[[1], [15, 9], [108, 778]]
>>> celt=lol[1]
>>> elt=celt[1]
>>> print(elt)
9
```

You can even combine the above two steps:

```
>>> lol=[[1], [15, 9], [108, 778]]
>>> elt=lol[1][1]
>>> print(elt)
9
```

A list of lists is often used to represent matrices. For example, the matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

might be represented as:

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

To print a row, you use the subscript operator:

```
>>> print(matrix[2])
[7, 8, 9]
```

To extract an individual element from the matrix, you should use the double-index form:

```
>>> print(matrix[2][1])
8
```

10.2.11 List aliasing and cloning

If you assign a list variable to another, both variables refer to the same list. Because the same list has two different names now, we say that the list is **aliased**. Changes made with one list affect the other. See below:

```
>>> list = [1, 2, 3]
>>> alias=list
>>> print(alias)
[1, 2, 3]
>>> alias[1]=4
>>> print(list)
[1, 4, 3]
```

If you want to modify a list and also keep a copy of the original, you need to use a technique called **cloning**. The easiest way to clone a list is to use the slice operator. See the example below:

```
>>> list = [1, 2, 3]
>>> clone=list[:]
>>> print(clone)
[1, 2, 3]
>>> clone[1]=4
>>> print(list)
[1, 2, 3]
```

```
>>> print(clone)
[1, 4, 3]
>>> list[1]=5
>>> print(clone)
[1, 4, 3]
>>> print(list)
[1, 5, 3]
```

Taking a slice of any list creates a new list. Thus you are free to make changes to the new list without modifying the original or vice versa.

10.2.12 Equality of lists

A list and its alias refer to the same list. On the other hand, a list and its clone refer to two different lists although both have the same elements. The equality between a list and its alias is **object identity** and the equality between a list and its clone is known as **structural equivalence**.
The `is` operator can be used to test for object identity. It returns `True` if the two operands refer to the same list, and it returns `False` if the operands refer to distinct lists (even if they are structurally equivalent). The following code illustrates this: Consider the following code:

```
>>> mylist=[1,2,3]
>>> alias=mylist
>>> clone=mylist[:]
>>> print(alias is list)
True
>>> print(clone is list)
False
```

10.3 Tuples

A tuple is a sequence of values that resembles a list, except that a tuple is **immutable**. You create a tuple by enclosing its elements in parentheses instead of square brackets. (Strictly speaking, the enclosing parentheses are optional, but are included to improve readability.) The elements are to be separated by commas. Following are some examples:

```
t = ('a', 'b', 'c', 'd', 'e')
s = ('g', 2, 7, 8.978)
```

To create a tuple with a single element, we have to include a comma at the end, even though there is only one value. Without the comma, Python treats the element as a string in parentheses. The following code illustrates this:

```
>>> t1=('a',)
>>> t2=('a')
>>> print(type(t1))
<class 'tuple'>
>>> print(type(t2))
<class 'str'>
```

10.3.1 Tuple creation

As discussed above, a tuple can be created by enclosing its elements in parentheses. Another way to create a tuple is to use the `tuple()` function. See below:

```
>>> t1=tuple("string")
>>> print(t1)
('s', 't', 'r', 'i', 'n', 'g')
>>> t2=tuple([1,2,3])
>>> print(t2)
(1, 2, 3)
>>> t3=tuple()
>>> print(t3)
()
>>> t4=tuple([4])
>>> print(t4)
(4,)
```

In the above example, `t3` is an empty tuple. Also, notice how `t4` is printed with commma at the end.

10.3.2 Tuple operations

Most of the operators and functions used with lists can be used similarly with tuples. A few examples follow:

```
>>> lan=tuple("python")
>>> print(lan[0])
p
>>> print(lan[1:4])
('y', 't', 'h')
```

But, unlike lists, tuples are immutable – you are bound to get error when you try to modify the tuple contents. See below:

```
>>> lan=tuple("python")
>>> lan[0]='P'
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Python provides a quick way to swap two variables `a` and `b` without using any third temporary variable:

```
>>> a,b=5,10
>>> print(a,b)
5 10
>>> a,b=b,a
>>> print(a,b)
10 5
```

This is known as **tuple assignment**.

10.4 The `numpy` package and arrays

NumPy stands for Numerical Python. It is an excellent package for advanced computing in Python like linear algebra, Fourier transforms, and random simulation. This package also provides extensive support for arrays. An array is a homogeneous collection of data items, unlike lists and tuples that are heterogeneous.

As always, if you want to use the components in NumPy, you should first import the package. NumPy is usually imported with the `np` alias name:

```
import numpy as np
```

Now the NumPy package can be referred to as `np` instead of `numpy`.

10.4.1 Creating arrays

To create an array, you use the `array()` method of `numpy` package as shown below:

```
>>> import numpy as np
>>> arr = np.array([1, 2, 3, 4, 5, 6])
>>> print(arr)
[1 2 3 4 5 6]
```

You can also create matrices (two-dimensional arrays) with `array()` method:

```
>>> mat = np.array([[1, 2, 3], [4, 5, 6]])
>>> print(mat)
[[1 2 3]
 [4 5 6]]
```

To know the dimensions of an array, you use the `ndim` attribute in NumPy. See below:

```
>>> arr = np.array([1, 2, 3, 4, 5])
>>> print(arr.ndim)
1
>>> mat = np.array([[1, 2, 3], [4, 5, 6]])
>>> print(mat.ndim)
2
```

To know the size across each dimension, you use the `shape` attribute:

```
>>> arr = np.array([1, 2, 3, 4, 5])
>>> mat = np.array([[1, 2, 3], [4, 5, 6]])
>>> print(mat.shape)
(2, 3)
>>> print(arr.shape)
(6,)
```

The first output `(2, 3)` means that the array `mat` has 2 elements in the first dimension (2 rows) and 3 elements in the second dimension (3 columns). The second output `(6,)` means that the array `arr` has 6 elements in the first dimension and no elements in the second dimension).

10.4.2 Accessing array elements

Indexing and slicing work the same way with arrays as with other sequences like lists, and tuples. Some examples follow:

```
>>> arr = np.array([1,2,3,4,5,6])
>>> mat = np.array([[1, 2, 3], [4, 5, 6]])
>>> print(arr[3])
4
>>> print(arr[2:5])
[3 4 5]
>>> print(mat[1])
[4 5 6]
>>> print(mat[0,2])
3
>>> print(mat[0][2])
3
>>> print(arr[-2])
5
>>> print(mat[-1,2])
6
>>> print(mat[-2,-3])
```

```
1
>>> print(arr[-2:])
[5 6]
>>> print(arr[1:5:2])
[2 4]
>>> print(arr[:5:2])
[1 3 5]
>>> print(mat[:2:2])
[[1 2 3]]
```

`for` loop is used to traverse the array:

```
>>> arr = np.array([1,2,3,4,5,6])
>>> mat = np.array([[1, 2, 3], [4, 5, 6]])
>>> for elt in arr:
...     print(elt,end=" ")
...
1 2 3 4 5 6
>>> for row in mat:
...     print(row)
...
[1 2 3]
[4 5 6]
>>> for row in mat:
...     for elt in row:
...             print(elt,end=" ")
...
1 2 3 4 5 6
>>> for row in mat:
...     for elt in row:
...             print(elt,end=" ")
...     print("")
...
1 2 3
4 5 6
```

10.4.3 Changing array dimensions

Changing the array dimensions means to increase or decrease the number of dimensions. For this, you use the `reshape()` method. See some illustrations:

```
>>> arr = np.array([1,2,3,4,5,6])
>>> mat = np.array([[1, 2, 3], [4, 5, 6]])
>>> mat.reshape(1,6)
array([[1, 2, 3, 4, 5, 6]])
```

```
>>> arr.reshape(3,2)
array([[1, 2],
       [3, 4],
       [5, 6]])
```

When you convert a one-dimensional array into a two-dimensional array, there should be enough number of elements to fill all the rows. Otherwise, you will end up with an error. See below:

```
>>> arr=np.array([1,2,3,4,5])
>>> arr.reshape(2,3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot reshape array of size 5 into shape (2,3)
```

As the error itself mentions, you cannot convert a one-dimensional array with 5 elements to a two-dimensional array of 6 elements (2 rows and 3 columns).

10.4.4 Matrix operations

Table 10.2: Matrix operations with `numpy`

Operation	**Implementation**
Add matrices `x` and `y`	`numpy.add(x,y)`
Subtract matrices `x` and `y`	`numpy.subtract(x,y)`
Multiply matrices `x` and `y`	`numpy.matmul(x,y)`
Transpose matrix `x`	`x.T`

NumPy has excellent support for doing most of the matrix operations. Some of the matrix operations and their Python implementation are listed in Table 10.2. See some illustrations now:

```
>>> mat1=np.array([[1,2],[3,4]])
>>> mat2=np.array([[5,6],[7,8]])
>>> print(np.add(mat1,mat2))
[[ 6  8]
 [10 12]]
>>> print(np.subtract(mat1,mat2))
[[-4 -4]
 [-4 -4]]
>>> print(np.matmul(mat1,mat2))
[[19 22]
 [43 50]]
```

```
>>> print(mat1.T)
[[1 3]
 [2 4]]
>>> print(mat2.T)
[[5 7]
 [6 8]]
```

10.5 Programming examples

Program 10.54. To find the median of a set of numbers.

Solution:

The **median** of a set of numbers is the value that is less than half the numbers in the set and greater than the other half. If the number of values in the list is odd, the median is the value at the middle when the set of numbers is sorted; otherwise, you will have two middle values and the median is the average of these two values.

For example, the median of the list `[1, 3, 3, 5, 7]` is 3, and the median of the list `[1, 8, 6, 4]` is 7 ($\frac{8+6}{2} = 7$).

```
numbers=[]
n=int(input("Enter the size of list"))
for i in range(n):
    num=int(input("Enter a number"))
    numbers.append(num)
numbers.sort()
mid=n//2
if n%2!=0:
    median=numbers[mid]
else:
    median=(numbers[mid]+numbers[mid-1])/2
print("The median is",median)
```

Program 10.55. To input a matrix and print its transpose.

```
import numpy as np
r=int(input("Enter the number of rows"))
c=int(input("Enter the number of columns"))
list=[]
print("Input the matrix")
for i in range(r*c):
    elmt=int(input())
    list.append(elmt)
```

```
arr=np.array(list)
mat=arr.reshape(r, c)

print("The matrix A is")
for row in range(r):
    for col in range(c):
        print(mat[row][col],end=" ")
    print("")

trans=mat.T
print("The transpose matrix is")
for row in range(c):
    for col in range(r):
        print(trans[row][col],end=" ")
    print("")
```

10.6 Conclusion

Lists and tuples are very important data structures in Python and they both serve different purposes. The list is mutable and lets you add, remove, or modify elements in the list. Therefore lists are handy when it comes to collections that need frequent addition/deletion/modifications. On the other hand, tuples are immutable and provide a good way to store a fixed list of data elements.

In this chapter we went over several built-in functions for manipulating lists, such as method calls to add and remove elements or sort them and query the existence of an element. These will help developers to work with complex data more efficiently. On the other hand, we can not modify tuples but this immutability property of them could be a benefit when it comes to using some data that should never change.

10.7 Exercises

1. Given a list L and an integer *target*, you have to find a pair of integers whose sum is equal to a given integer, *target*
   sample input: L = [1,2,3,4,5] ; *target* = 9
   sample output: (4,5)

2. Given a list L, rotate the list k times in the clockwise direction
   sample input: L = [1,2,3,4,5] ; k = 2
   sample output: [4,5,1,2,3]

Explanation: list after first rotation: [5,1,2,3,4]
list after second rotation: [4,5,1,2,3]

3. Write a Python program that inputs a list of numbers and then doubles the odd numbers and halves the even numbers.
4. Given a list *nums* of numbers and an integer *val*, remove all occurrences of *val* in the list.
   sample input: *nums* = [1,2,2,3,4,5,6,6,2,2,2,8] ; *val* = 2
   sample output: [1,3,4,5,6,6,8]
5. Given a list of strings, find out the longest word in that list. Display the longest word, then replace the longest word with the word "found" and print the list.
6. Write a Python program to input any two tuples and swap them.
   Sample input: tuple1 = (1,2,3,4,5) ; tuple2 = (10,20,30,40,50)
   Sample output: tuple1 = (10,20,30,40,50) ; tuple2 = (1,2,3,4,5)
7. Write a Python program to accept values from the user. Add it to a tuple and display the elements one by one. Also, display the maximum and minimum values of the tuple.
8. Write a program to print the Fibonacci sequence using lists.
9. Sarah is a data analyst working for a marketing agency. She has been given a list of customer ages from a recent survey conducted by her company. The list contains a mix of integers representing ages and some strings due to data entry errors. Sarah needs to clean up this data by removing the erroneous entries (non-integer values) and then analyze the data to find the following:
   (a) The youngest and oldest customers.
   (b) The average age of the customers.
   (c) The most common age in the list.
10. Write a menu-driven program to input two matrices and do the following:
    (a) Find the sum of the two matrices
    (b) Find the difference of the two matrices
    (c) Find the product of the two matrices
11. Input a tuple `tup` from the user and an integer value `k`. You should replace `k` present in the tuple with the square of that number.
    Sample input: `tup` = (1,2,3,4,5) ; k = 4
    Sample output: `tup` = (1,2,3,16,5)

12. Imagine waking up on a deserted railway station, where trains keep arriving and departing, yet no one is present to manage the hustle. As you wander about the station, a note catches your eye: ”The person who reads this becomes the master of the station.”

    Manage this special railway station, ensuring trains operate smoothly and efficiently. Determine whether each approaching train will halt or pass through, assigning platforms to those that stop. If all platforms are occupied, prioritize, departing the oldest train to accommodate the newcomer. For trains passing through, an empty platform must be available; if not, the oldest train must depart to clear the way.

    Create a menu-driven program that allows you to input the number of platforms (n). Simulate the arrival of trains, deciding whether they will halt or pass through. Assign platforms to halting trains, ensuring efficient management of the station. At the end of the simulation, display the total number of trains that have stopped or passed through the station.

Bibliography

[1] Kenneth A. Lambert, Fundamentals of Python: First Programs, Cengage Learning, 2019.

[2] Yashavant Kanetkar, Let Us Python, BPB Publication, 2022.

[3] John V Guttag, Introduction to Computation and Programming Using Python, MIT Press, 2013.

[4] Allen Downey, How to Think Like a Computer Scientist: Learning with Python, O'Reilly Media, 2015.

Chapter 11

Sets and Dictionaries

"If two people always agree, one of them is redundant."

- Anonymous

"I hold this to be the highest task for a bond between two people: that each protects the solitude of the other."

- Rainer Maria Rilke

"Every key tells a story, and every value is its plot—dictionaries bring data to life."

- Anonymous

11.1 Membership Manager

Think about running a membership website wherein you must find a user's details given the unique user ID or verify whether the user is a member. Using dictionaries, the website can store detailed information about each member, such as their name, email address, membership status, and any other relevant data. This allows for quick retrieval of a user's details simply by referencing their unique ID, making operations like profile updates, access control, and personalized content delivery seamless. Additionally, sets can be used to store collections of these unique IDs, ensuring that no duplicates exist and enabling quick checks to verify membership. This approach not only keeps the data organized but also enhances the website's efficiency in managing and accessing member information.

11.2 Introduction

Dictionaries store data in key-value pairs, making it easy to retrieve or update information using a unique identifier, like a user ID. They are highly efficient

for tasks that require quick lookups or modifications.
Sets are collections of unique elements, ideal for managing groups of items where duplicates are not allowed. They are handy for checking membership, performing set operations like unions and intersections, and ensuring that each element is distinct.

11.3 Sets

A set is a collection of items just as tuples or lists but unlike the other two, a set is unordered; the items in a set do not have a defined order.

11.3.1 Creating a set and accessing its elements

A set is created by enclosing the items in a pair of braces. Following are some examples:

```
>>> directions={'east','west','south','north'}
>>> print(directions)
{'east', 'south', 'north', 'west'}
```

Notice that since the items are unordered in a set, each time you print the set, the output may be different concerning the order of the elements.
You can also use the `set` method to create a set from a list or a tuple as follows:

```
>>> numbers=set([1,2,3,4])
>>> print(numbers)
{1, 2, 3, 4}
>>> vowels=set(('a','e','i','o,','u'))
>>> print(vowels)
{'o', 'u', 'a', 'e', 'i'}
```

The items in a set aren't associated with index or position. This means it is impossible to print an arbitrary element of the set.

```
>>> vowels=set(('a','e','i','o,','u'))
>>> vowels[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object is not subscriptable
```

To traverse the set, use a `for` loop: The code

```
>>> for char in vowels:
...     print(char,end=" ")
...
o u a e i
```

Sets do not allow duplicate elements. The duplicates are automatically removed even if you supply non-unique values. See below:

```
>>> fibonacci={0,1,1,2,3,5}
>>> print(fibonacci)
{0, 1, 2, 3, 5}
```

11.3.2 Adding and removing set elements

To add a single element to a set, you use the `add()` method:

```
>>> even={2,4,8,10}
>>> even.add(6)
>>> print(even)
{2, 4, 6, 8, 10}
```

To add multiple items to a set, the `update()` method is to be used:

```
>>> odd={1,3,7,9,13,15}
>>> odd.update({5,11})
>>> print(odd)
{1, 3, 5, 7, 9, 11, 13, 15}
```

To remove an item from a set, you should use the `remove()` method:

```
>>> primes={3,5,7,9,11}
>>> primes.remove(9)
>>> print(primes)
{3, 5, 7,11}
```

If the item to be removed is not present in the set, `remove()` will raise an error:

```
>>> primes={3,5,7,11}
>>> primes.remove(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 2
```

A second way to remove an item from a set is to use the `discard()` method which works the same way as `remove`. The difference between these two methods is that `discard()` will not raise an error if the item to be removed doesn't exist in the set unlike `remove()`. See an example:

```
>>> primes={3,5,7,9,11}
>>> primes.discard(9)
>>> print(primes)
```

```
{3, 5, 7, 11}
>>> primes.discard(2)
>>> print(primes)
{3, 5, 7, 11}
```

To remove all the set items at once, you can use the `clear()` method as illustrated below:

```
>>> primes={3,5,7,9,11}
>>> primes.clear()
>>> print(primes)
set()
```

The keyword `del` can be used to delete the set itself, that is, after the `del` operation, the set will not exist anymore. Any subsequent access to the set will throw an error. See below:

```
>>> primes={3,5,7,9,11}
>>> del primes
>>> print(primes)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'primes' is not defined
```

11.3.3 Sets and relational operators

The relational operators can be used to check the relationship (subset, superset, etc) between two sets. The meaning of the relational operators in the context of sets is summarized in Table 11.1

Table 11.1: Relational operators with sets

Operator	**Interpretation**
`<`	$\subset$
`>`	$\supset$
`<=`	$\subseteq$
`>=`	$\supseteq$
`==`	checks if two sets are same
`!=`	checks if two sets are not same

See some illustrations:

```
>>> digits={0,1,2,3,4,5,6,7,8,9}
>>> natural={1,2,3,4,5,6,7,8,9,10}
>>> even={0,2,4,6,8}
>>> odd={1,3,5,7,9}
>>> prime={3,5,7}
>>> composite={4,6,8}
>>> print(even<natural)
False
>>> print(odd<digits)
True
>>> print(prime<=odd)
True
>>> print(digits>=even)
True
>>> print(composite==even)
False
>>> print(prime!=odd)
True
```

11.3.4 Mathematical set operations on Python sets

Python supports all the mathematical set operations (union, intersection, etc.). These are summarized in Table 11.2. See some illustrative examples below:

```
>>> positive={1,2,3,4,5}
>>> negative={-5,-4,-3,-2,-1}
>>> numbers=positive.union(negative)
>>> print(numbers)
{1, 2, 3, 4, 5, -2, -5, -4, -3, -1}
>>> even={2,4,6,8,10}
>>> mul3={3,6,9,12}
>>> even={2,4,6,8,10,12}
>>> mul6=even.intersection(mul3)
>>> print(mul6)
{12, 6}
>>> mul15={0,15,30,45}
>>> mul5={0,5,10,15,20,25,30,25,40,45}
>>> mul5.intersection_update(mul15)
>>> print(mul5)
{0, 45, 30, 15}
>>> mul4={4,8,12,16,-4,-12,-20}
>>> mul8={8,16,-40,-120}
>>> mul4only=mul4.difference(mul8)
>>> print(mul4only)
{4, 12, -20, -12, -4}
```

```
>>> mul4.difference_update(mul8)
>>> print(mul4)
{-4, 4, -12, -20, 12}
>>> numbers={-3,-2,-1,0,1,2,3}
>>> natural={1,2,3,4,5}
>>> integers=numbers.symmetric_difference(natural)
>>> print(integers)
{0, 4, 5, -1, -3, -2}
>>> numbers.symmetric_difference(natural)
>>> print(numbers)
{0, 4, 5, -1, -3, -2}
```

Table 11.2: Set operations on two sets `A` and `B`

Set method	**Meaning**
`A.union(B)`	returns a new set $S = A \cup B$
`A.intersection(B)`	returns a new set $S = A \cap B$
`A.intersection_update(B)`	changes the set `A` to $A \cap B$
`A.difference(B)`	returns a new set $S = A - B$
`A.difference_update(B)`	changes the set `A` to $A - B$
`A.symmetric_difference(B)`	returns a new set $S = (A - B) \cup (B - A)$
`A.symmetric_difference_update(B)`	changes the set `A` to $(A - B) \cup (B - A)$

11.4 Dictionaries

A dictionary associates a set of **keys** with data **values**. For example, the words in a standard English Dictionary comprise a set of keys, whereas their associated meanings are the data values. Thus a dictionary is a mapping between a set of keys and a set of values. Each key maps to a value. The association of a key and a value is called a **key-value pair**, sometimes called an **item** or an **entry**.

A Python dictionary is written as a sequence of key/value pairs separated by commas and the entire sequence is enclosed in a pair of braces. Each key is separated from its value by a colon (:). Such a list of key-value pairs enclosed in a pair of braces is known as a **dictionary literal**. Following is an example dictionary:

```
>>> stud = {'Name': 'Ram', 'Age': 18, 'Class': 'S1'}
>>> print(stud)
{'Name': 'Ram', 'Age': 18, 'Class': 'S1'}
```

Here `Name`, `Age` and `Class` are keys whereas `Ram`, `18` and `S1` are the corresponding values. An empty dictionary without any items is written as `{}`.

Keys are unique within a dictionary although values need not be. Although the entries may appear to be ordered in a dictionary, this ordering is not the same, each time we print the entries. In general, the order of items in a dictionary is unpredictable.

11.4.1 Dictionary operations

We now move on to the discussion of some common operations on dictionaries.

11.4.1.1 Accessing values

The subscript operator is used to obtain the value associated with a key. See below:

```
>>> stud = {'Name': 'Ram', 'Age': 18, 'Class': 'S1'}
>>> print("I am",stud['Name'],"aged",stud['Age'],"and I am
    studying in",stud['Class'])
I am Ram aged 18 and I am studying in S1
```

However, if the key is not present in the dictionary, Python raises an error:

```
>>> stud = {'Name': 'Ram', 'Age': 18, 'Class': 'S1'}
>>> print(stud['College'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'College'
```

11.4.1.2 Traversing a dictionary

A simple `for` loop can be used to traverse a dictionary as follows:

```
>>> stud = {'Name': 'Ram', 'Age': 18, 'Class': 'S1'}
>>> for key in stud:
...     print(key,"-",stud[key])
...
Name - Ram
Age - 18
Class - S1
```

The key-value pairs need not be always printed in the order you gave.

11.4.1.3 Inserting keys and updating key values

You should use the subscript operator to add a new key/value pair to the dictionary. The following code illustrates this:

```
>>> stud['College']="Engineering college"
>>> print(stud)
{'Name': 'Ram', 'Age': 18, 'Class': 'S1', 'College':
    'Engineering college'}
```

The subscript is also used to replace the value of an existing key:

```
stud = {'Name': 'Ram', 'Age': 18, 'Class': 'S1'}
>>> stud['Age']=19
>>> print(stud)
{'Name': 'Ram', 'Age': 19, 'Class': 'S1'}
```

11.4.1.4 Removing keys

To remove a key from a dictionary, use the `del` operator. The corresponding key-value pair will be removed. This is illustrated below:

```
stud = {'Name': 'Ram', 'Age': 18, 'Class': 'S1'}
>>> del stud['Age']
>>> print(stud)
{'Name': 'Ram', 'Class': 'S1'}
```

11.4.1.5 Miscellaneous operations

The `len()` function returns the number of key-value pairs in a dictionary:

```
>>> stud = {'Name': 'Ram', 'Age': 18, 'Class': 'S1'}
>>> print(len(stud))
3
```

The `in` operator can be used to know if a key exists in the dictionary. See below:

```
>>> stud = {'Name': 'Ram', 'Age': 18, 'Class': 'S1'}
>>> print('Age' in stud)
True
>>> print('College' in stud)
False
```

11.4.2 Dictionary methods

Python provides several methods to manipulate the dictionary elements. Table 11.3 summarizes the commonly used dictionary methods, where `dict` refers to a dictionary.

Table 11.3: Some dictionary methods, with **`dict`** denoting the dictionary

Dictionary method	**Purpose**
`dict.get(key,default)`	Returns the value if the key exists or returns the `default` if the key does not exist. Displays "`None`" if the default is omitted and the key does not exist.
`dict.pop(key,default)`	Removes the key and returns the value if the key exists or returns the text `default` if the key does not exist. Raises an error if the default is omitted and the key does not exist.
`dict.keys()`	Returns a list of the keys.
`dict.values()`	Returns a list of values.
`dict.items()`	Returns a list of tuples containing the key and value pairs.
`dict.clear()`	Removes all the items from the dictionary.

Some illustrations follow now:

```
>>> stud = {'Name': 'Ram', 'Age': 18, 'Class': 'S1'}
>>> print(stud.get('Name'))
Ram
>>> print(stud.get('College'))
None
>>> print(stud.get('College',"Oops!!No such key!"))
Oops!!No such key!
>>> stud.pop('College')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'College'
>>> stud.pop('Age')
18
>>> print(stud)
{'Name': 'Ram', 'Class': 'S1'}
>>> print(stud.keys())
dict_keys(['Name', 'Class'])
>>> print(stud.values())
dict_values(['Ram', 'S1'])
>>> print(stud.items())
dict_items([('Name', 'Ram'), ('Class', 'S1')])
>>> stud.clear()
>>> print(stud)
```

```
{}
```

11.4.3 Dictionary aliasing and copying

Assigning a dictionary literal to another creates an alias of the original dictionary. In this case, the changes made to one will affect the other. If you want to modify a dictionary and keep a copy of the original, you need to use the copy() method. With a copy, you can make changes to it without affecting the original. The following illustrates this:

```
>>> directions={'N':'North','E':'East','S':'South','W':'West'}
>>> alias=directions
>>> copy=directions.copy()
>>> copy['S']='Sit down students!'
>>> print(copy)
{'N': 'North', 'E': 'East', 'S': 'Sit down students!', 'W':
    'West'}
>>> print(directions)
{'N': 'North', 'E': 'East', 'S': 'South', 'W': 'West'}
>>> alias['S']='Sit down students!'
>>> print(directions)
{'N': 'North', 'E': 'East', 'S': 'Sit down students!', 'W':
    'West'}
>>> print(alias)
{'N': 'North', 'E': 'East', 'S': 'Sit down students!', 'W':
    'West'}
```

11.5 Programming examples

Program 11.56. To create a histogram for a string.

Solution:

Histogram for a string shows how many times each letter appears in the string. A dictionary is very apt for representing a histogram. The keys represent the letters in the string and values represent the corresponding counts.

```
histogram = {}
str=input("Enter a string")
for letter in str:
    histogram[letter] = histogram.get(letter, 0) + 1
print(histogram)
```

If the letter does not exist in the dictionary as a key, you just let the `get()` method return `0` and insert the item into the dictionary. Otherwise, you increase the count of that letter key in the dictionary.

Program 11.57. To create a dictionary out of the keys and values input from the user.

```
dict={}
keys=[]
values=[]
n=int(input("How many entries you want?"))
print("Input the keys")
for i in range(n):
   keys.append(int(input()))
print("Input the corresponding values")
for i in range(n):
   values.append(int(input()))
dict={keys[i]:values[i] for i in range(n)}
print(dict)
```

See how comprehension is put to use here.

Program 11.58. To initialize a dictionary and then print its contents in the ascending order of keys.

```
dict={'Name':'XYZ','Class':'S1','Age':18,'College':'Engineering
    college'}
keys=list(dict.keys())
keys.sort()
for key in keys:
    print(key,"-",dict[key])
```

11.6 Conclusion

In this chapter, we explored two fundamental data structures – sets and dictionaries, learning how they streamline data management. We discovered how sets can handle collections of unique items efficiently, while dictionaries allow us to map keys to values for quick lookups. We also covered various functions and methods for manipulating these structures, which empower us to work on them more effectively and flexibly. By mastering these concepts and functions, you now have the skills to handle complex data tasks easily, enhancing your programming and problem-solving skills.

11.7 Exercise

1. You are organizing a series of meetings and need to find which time slots are free for two participants in the meeting. Given three sets: one with all possible time slots and other two with time slots already booked by the two individual participants, write a Python function to determine the time slots that are still available. Also, determine the time slots during which only one of the participants is available.

   Example Input:

   ```
   all_time_slots = {"9:00-10:00", "10:00-11:00", "11:00-12:00",
   "12:00-1:00", "1:00-2:00", "2:00-3:00"}
   booked_slots_1 = {"10:00-11:00", "1:00-2:00"}
   booked_slots_2 = {"12:00-1:00", "2:00-3:00"}
   ```

   Expected Output:

   ```
   Available Slots: {'9:00-10:00', '11:00-12:00'}
   Slots Where Only One Participant is Available: {'2:00-3:00',
   '10:00-11:00','12:00-1:00', '1:00-2:00',}
   ```

2. Write a program to read N words and group the words by their length in a dictionary. (The dictionary should have the length of the words as keys and sets of words of that length as values.)

   Example Input:

   ```
   N = 6
   Words = ["Hello", "World", "This", "Apple", "Banana", "Program"]
   ```

   Expected Output:

   ```
   {4: {"This"}, 5: {"Hello", "World", "Apple"}, 6: {"Banana"},
   7: {"Program"}}
   ```

3. Design a menu-based program that uses a dictionary to track the stock of products in a store. The program should allow the user to:

   (a) Add a new product.
   (b) Update the stock of an existing product.
   (c) Check the stock of a given product.
   (d) Display all products and their current stock levels.
   (e) Exit the program.

   Error cases to be handled:

- The user tries to add a product that already exists.
- The user attempts to update or check stock for a product that does not exist.

4. You are given data on user interactions on three social media posts: "Post1", "Post2", and "Post3". Each post has a list of users who interacted with it in some manner (liked, commented, or shared). Write a program to find the following:

   (a) Common Interactors: Find the users who have interacted with all three posts.

   (b) Exclusive Interactors: Find users who interacted with only one post.

   (c) Most Popular Post: Determine which post has the most unique user interactions.

   (d) Post Comparison: Compare two posts to see which users interacted with both (Post1 and Post2).

   Example Input:

   ```
   post1 = {'Alice': 'like', 'Bob': 'comment', 'Charlie': 'share',
   'David': 'like', 'Eve': 'like'}
   post2 = {'Alice': 'comment', 'Charlie': 'like', 'David': 'comment',
   'Frank': 'share'}
   post3 = {'Bob': 'like', 'Charlie': 'comment', 'Eve': 'share',
   'Grace': 'like'}
   ```

   Expected Output:

   ```
   Common Interactors: Charlie
   Exclusive Interactors: Eve, Frank, Grace
   Most Popular Post: post1
   Common users for post1 and post2: Alice, David
   ```

5. Write a program to find anagrams from a set of n words. An anagram of a word is another word consisting of the same letters but rearranged in a different order. (e.g., stop and tops are both anagrams of pots.) The input consists of n followed by the actual words. The output consists of anagram sets with each set on different lines. The anagram sets should be displayed in decreasing order of anagram word lengths. On each line, the anagrams should be displayed in alphabetical order.

   Example Input:

   ```
   8
   stop
   tops
   pots
   opts
   ```

```
dog
god
from
form
```

Expected Output:

```
form from
dog god
opts pots stop tops
```

6. Create a telephone directory using a dictionary. The name of the individual and the telephone number will be key and value, respectively. Write a Python program that allows the user to perform the following operations:

   (a) Add a Contact: Add a new contact with a name and phone number to the directory.

   (b) Update a Contact: Update the phone number of an existing contact.

   (c) Delete a Contact: Remove a contact from the directory.

   (d) Search for a Contact: Look up the phone number of a contact by their name.

   (e) Display All Contacts: Print all contacts in the directory.

   (f) Exit the program.

   Use a menu-driven approach.

Bibliography

[1] Kenneth A. Lambert, Fundamentals of Python: First Programs, Cengage Learning, 2019.

[2] Yashavant Kanetkar, Let Us Python, BPB Publication, 2022.

[3] John V Guttag, Introduction to Computation and Programming Using Python, MIT Press, 2013.

[4] Allen Downey, How to Think Like a Computer Scientist: Learning with Python, O'Reilly Media, 2015.

Chapter 12

Recursive Thinking

"There are two kinds of people in the world, those who divide the world into two kinds of people and those who do not."

– Anonymous

"To understand recursion, one must first understand recursion."

– Stephen Hawking

Recursive thinking is a cornerstone of computer science and mathematical problem-solving, characterized by breaking down a problem into smaller, more manageable subproblems that are similar to the original problem. This approach is particularly effective for solving complex problems that can be defined as simpler versions of themselves. Recursion is a natural and elegant method for tackling issues in various domains, including algorithms, data structures, and mathematical computations. This chapter explores the principles of recursive thinking, its applications, and how it can be leveraged to develop elegant and efficient solutions to complex problems.

12.1 What is Recursion?

Recursion involves a function calling itself to solve smaller or simpler instances of the same problem. The recursive approach is built on the idea of solving a problem by solving instances of the same problem. Imagine searching for a name in a phone book by first opening it to the middle. If the name is on that page, you have found it. If not, you decide where to search next: if the name should be in the earlier part of the book, you focus on the first half; if it should be in the later part, you focus on the second half. You continue this process, repeatedly narrowing down your search to progressively smaller sections of the phone book until you locate the name.

The concept of recursion can be beautifully illustrated using the metaphor of Russian dolls, also known as Matryoshka dolls. Here's how it works:

A Russian doll set consists of multiple dolls that can be nested within one another (see Figure12.1). Each doll, starting from the largest, can be opened to reveal a smaller doll inside it, except for the smallest doll, which is indivisible. To determine how many dolls are nested within the largest one, you must open each doll and separate its pieces. This process continues until you reach the smallest, indivisible doll. Once you have reached this smallest doll, you can begin reassembling the dolls, counting each one as you fit it back into the next larger doll until you finally nest all the dolls within the outermost one.

This nesting process is analogous to how a recursive function operates. Just as each Russian doll contains a smaller one until reaching the smallest, a recursive algorithm solves a problem by breaking it down into smaller instances of the same problem. It continues solving these progressively smaller problems until it reaches a point where the problem is simple enough to solve directly. The algorithm then works backward, solving each larger sub-problem step by step until it has solved the entire problem.

Figure 12.1: Russian nesting dolls

[1]

Russian Doll Metaphor for Recursion

1. **Base Case (Smallest Doll)**:
    - The smallest Russian doll represents the base case in recursion. It is the simplest, indivisible case of the problem that does not need further breaking down. In recursion, this is where the function terminates.
2. **Recursive Case (Nested Dolls)**:
    - Larger Russian dolls contain smaller dolls inside them. This represents the recursive case. Each larger doll contains a smaller doll, just as each recursive call contains a simpler subproblem that leads toward the base case.

[1]Source: https://www.pngall.com/matryoshka-doll-png/download/42036

3. **Unwinding (Assembling the Dolls)**:
   - Once the smallest doll (base case) is reached and processed, the dolls are reassembled in the opposite order they were opened. This is analogous to how the recursive function's calls are completed and results are combined as the recursion unwinds.

Each recursive call represents a larger doll containing a smaller one until the smallest doll is reached. The results from the smallest to the largest are combined as the recursion unwinds.

☞ Just as Russian dolls are nested within one another, recursive algorithms solve a problem by solving smaller instances of the same problem.

This metaphor helps illustrate how recursion involves breaking down a problem into smaller instances, solving them, and then combining the solutions.

12.1.1 Key Concepts

At its core, recursion involves a function calling itself with modified arguments. A recursive function typically has two key components:

- **Base Case**: The simplest version of the problem that can be solved directly without further recursion.
- **Recursive Case**: The part of the problem that involves calling the function itself to handle a smaller or simpler instance.

Once we determine the two essential components, implementing a recursive function involves calling the function again based on the recursive relation until the base case is reached.

To understand recursion, let us consider a classic example: calculating the factorial of a number.

12.2 Example: Calculating Factorials

The factorial function is a classic example of recursion. The factorial of a non-negative integer n, denoted $n!$, is defined as:

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 1$$

with the special case $0! = 1$.

12.2.1 Recursive Definition

The factorial function can be defined recursively as follows:

- **Base Case**: $0! = 1$
- **Recursive Case**: $n! = n \times (n-1)!$ for $n > 0$

12.2.2 Recursive Function Implementation

Here is a Python function that computes the factorial of a number using recursion:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

12.2.3 Explanation of the Recursive Function

- **Base Case**: When $n = 0$, the function returns 1. This is the simplest instance of the problem and terminates the recursion.
  The base case is crucial for terminating the recursion. Without a base case, the function would call itself indefinitely, leading to a stack overflow error. A well-defined base case ensures that the recursion stops once the simplest form of the problem is reached.

  ☞ Always define a base case that provides a straightforward solution to the smallest instance of the problem. This ensures the recursion terminates and prevents infinite loops.

- **Recursive Case**: For $n > 0$, the function computes $n \times factorial(n-1)$. This involves calling ‘factorial‘ with $n - 1$, which continues until the base case is reached.

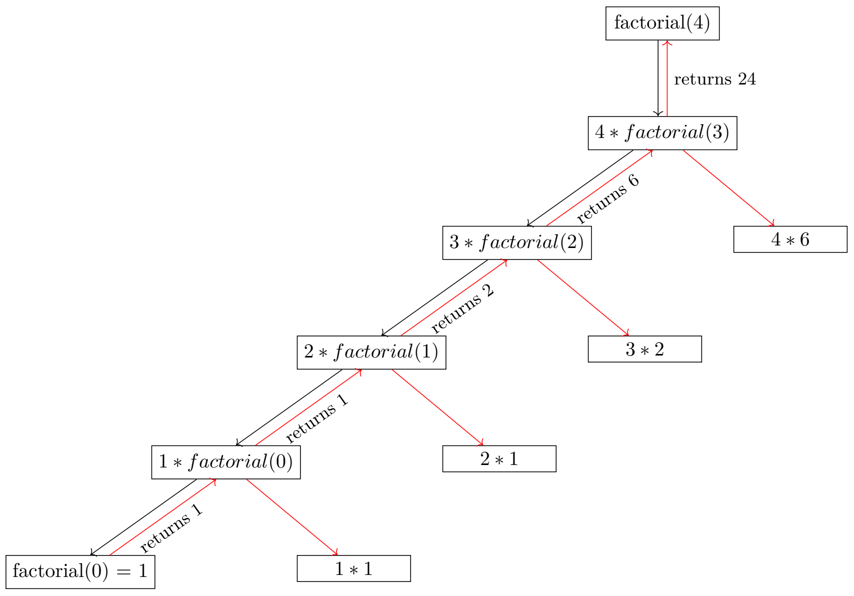
  ☞ Break down the problem into smaller instances of itself. Each recursive call should move towards the base case, simplifying the problem incrementally.

12.2.4 How It Works

Suppose the user passes the value 4 to the `factorial`. What happens now is

```
factorial(4) executes 4 * factorial(3)
factorial(3) executes 3 * factorial(2)
factorial(2) executes 2 * factorial(1)
factorial(1) executes 1 * factorial(0)
factorial(0) returns 1 to its calling function factorial(1)
factorial(1) returns 1 * 1 = 1 to its calling function
   factorial(2)
factorial(2) returns 2 * 1 = 2 to its calling function
    factorial(3)
factorial(3) returns 3 * 2 = 6 to its calling function
    factorial(4)
factorial(4) returns 4 * 6 = 24
```

Visualizing the execution of a recursive function, such as the factorial function, can help understand how recursion works. To illustrate how the recursive calls are made, we will visualize the recursion tree for 4!.



In the tree diagram:

- Each node represents a call to the factorial function.
- The children of each node represent the recursive calls made by that function.
- The return values are shown next to each node to illustrate the result of the recursive call.
- The multiplication operations are indicated to show how each result is computed.

This visualization helps to understand how the function calls itself with a decremented value until it reaches the base case, and then multiplies the results as it returns from each recursive call.

12.3 The Call Stack

The call stack is a crucial data structure used in programming to manage function calls and control flow. It operates on a last-in, first-out (LIFO) principle, meaning that the most recent function call is processed first when returning control to previous calls. Each time a function is invoked, a stack frame is created, storing information such as the function's parameters, local variables, and the return address. As functions call other functions, new frames are pushed onto the stack, and as functions return, their frames are popped off. This mechanism ensures that each function executes in the correct context and allows for the orderly execution and return of function calls, especially important in recursive programming where functions call themselves. This section gives an overview of what the call stack is, how it works, and its significance.

12.3.1 How the Call Stack Works

1. **Function Call**: When a function is called, an activation record (also known as a stack frame) is created and pushed onto the top of the call stack. This frame contains information such as:
    - The return address (where to return control after the function execution completes)
    - The parameters of the function
    - Local variables of the function
    - Saved registers
2. **Function Execution**: The CPU executes the function. If the function calls another function, a new frame is pushed onto the stack.
3. **Function Return**: When a function finishes executing, its frame is popped from the stack. Control is then transferred back to the return address stored in the popped frame, and execution resumes from there.

12.3.2 Importance of the Call Stack

- **Function Management**: The call stack manages function calls and returns in a structured manner, ensuring that each function's local variables and return address are properly maintained.
- **Recursion**: The call stack is crucial for handling recursive functions, where a function calls itself. Each recursive call adds a new frame to the stack.

- **Error Handling**: The call stack helps in debugging and error handling. A stack trace (a report of the active stack frames at a certain point in time) is often used to diagnose the sequence of function calls leading to an error or exception.

☞ Be mindful of how recursive calls use the call stack. Each function call adds a new layer to the stack, which can lead to stack overflow if not managed correctly.

Let us look at an example. Consider the following simple example of a recursive function:

```python
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

#Calling the function
print(factorial(5))
```

Here is how the call stack would look during the execution of **factorial(5)**:

- **Initial Call**: **factorial(5)**
  - A frame for **factorial(5)** is pushed onto the stack.
- **First Recursive Call**: **factorial(4)**
  - A frame for **factorial(4)** is pushed onto the stack.
- **Second Recursive Call**: **factorial(3)**
  - A frame for **factorial(3)** is pushed onto the stack.
- **Third Recursive Call**: **factorial(2)**
  - A frame for **factorial(2)** is pushed onto the stack.
- **Fourth Recursive Call**: **factorial(1)**
  - A frame for factorial(1) is pushed onto the stack.
- **Base Case**: **factorial(0)**
  - A frame for **factorial(0)** is pushed onto the stack. Since $n == 0$, it returns 1 and this frame is popped.

As the base case returns, each function call completes and returns control to its caller, popping frames off the stack in the reverse order of their addition.

12.3.3 Call Stack Visualization

Visualizing the call stack for a recursive function like **factorial(5)** can help understand how each function call is managed. Here is a step-by-step illustration of how the call stack evolves as **factorial(5)** executes:

Call Stack Visualization for factorial(5)

1. **Initial Call: factorial(5)**
    - Stack Frame: **factorial(5)**
    - Parameters: $n = 5$
    - Local Variables: None yet
    - Return Address: To the location where **factorial(5)** was called
    - **factorial(5)** calls **factorial(4)**
2. **Stack Frame: factorial(4)**
    - Parameters: $n = 4$
    - Local Variables: None yet
    - Return Address: To the location after the call to **factorial(4)** in **factorial(5)**
    - **factorial(4)** calls **factorial(3)**
3. **Stack Frame: factorial(3)**
    - Parameters: $n = 3$
    - Local Variables: None yet
    - Return Address: To the location after the call to **factorial(3)** in **factorial(4)**
    - **factorial(3)** calls **factorial(2)**
4. **Stack Frame: factorial(2)**
    - Parameters: $n = 2$
    - Local Variables: None yet
    - Return Address: To the location after the call to **factorial(2)** in **factorial(3)**
    - **factorial(2)** calls **factorial(1)**
5. **Stack Frame: factorial(1)**
    - Parameters: $n = 1$
    - Local Variables: None yet

- Return Address: To the location after the call to **factorial(1)** in **factorial(2)**
- **factorial(1)** calls **factorial(0)**

6. **Stack Frame: factorial(0)** *(Base Case)*
   - Parameters: $n = 0$
   - Local Variables: None yet
   - Return Address: To the location after the call to **factorial(0)** in **factorial(1)**
   - **factorial(0)** returns 1

7. **Returning from factorial(0)**
   - **factorial(1)** receives the result 1
   - **factorial(1)** calculates: $1 * 1 = 1$
   - **factorial(1)** returns 1

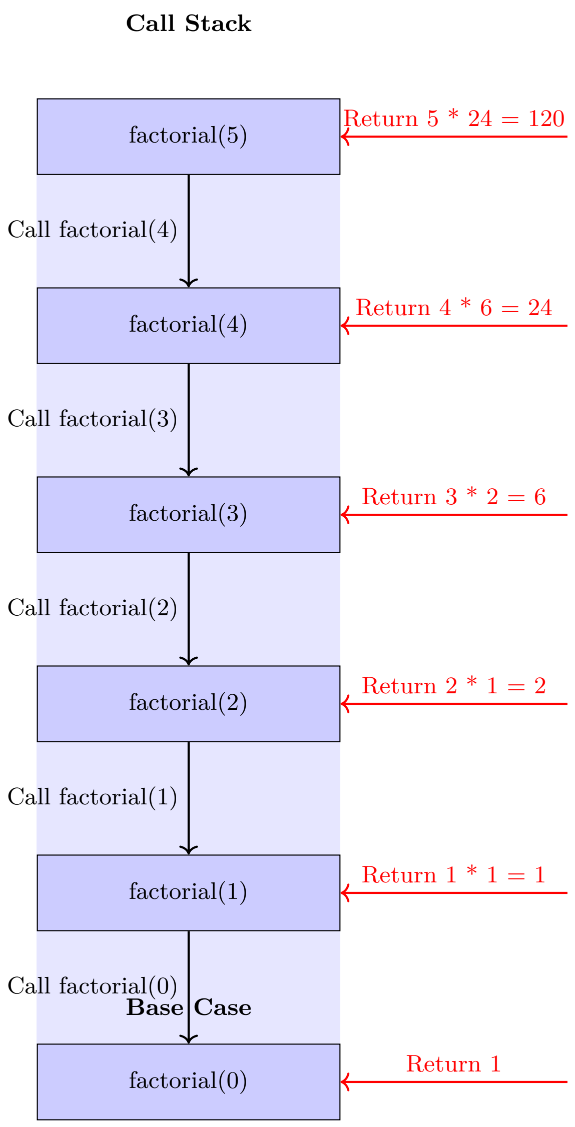
8. **Returning from factorial(1)**
   - **factorial(2)** receives the result 1
   - **factorial(2)** calculates: $2 * 1 = 2$
   - **factorial(2)** returns 2

9. **Returning from factorial(2)**
   - **factorial(3)** receives the result 2
   - **factorial(3)** calculates: $3 * 2 = 6$
   - **factorial(3)** returns 6

10. **Returning from factorial(3)**
    - **factorial(4)** receives the result 6
    - **factorial(4)** calculates: $4 * 6 = 24$
    - **factorial(4)** returns 24

11. **Returning from factorial(4)**
    - **factorial(5)** receives the result 24
    - **factorial(5)** calculates: $5 * 24 = 120$
    - **factorial(5)** returns 120

Final Result: factorial(5) returns 120.
This process can be pictorially depicted as follows.



Explanation

- **factorial(5)** calls **factorial(4)** and waits for the result.
- **factorial(4)** calls **factorial(3)** and waits for the result.
- **factorial(3)** calls **factorial(2)** and waits for the result.
- **factorial(2)** calls **factorial(1)** and waits for the result.
- **factorial(1)** calls **factorial(0)** and waits for the result.

- **factorial(0)** returns `1` (base case).
- **factorial(1)** receives the result `1` and returns `1 * 1 = 1`.
- **factorial(2)** receives the result `1` and returns `2 * 1 = 2`.
- **factorial(3)** receives the result `2` and returns `3 * 2 = 6`.
- **factorial(4)** receives the result `6` and returns `4 * 6 = 24`.
- **factorial(5)** receives the result `24` and returns `5 * 24 = 120`.

In this visualization, each function call adds a new frame to the stack, and each return removes a frame, reflecting the LIFO nature of the call stack.

12.3.4 Implications of Recursion and the Call Stack

- **Memory Usage**: Each recursive call adds a new frame to the stack. Deep recursion can lead to high memory usage and even stack overflow if the recursion depth is too large.
- **Stack Overflow**: This occurs when the call stack exceeds its limit due to too many recursive calls. It's often a sign of either too deep recursion or an infinite recursion due to missing base cases.
- **Efficiency**: Recursion can be elegant and easy to understand but might be less efficient than iterative solutions in terms of both time and space. Tail recursion optimization can help mitigate some inefficiencies in languages that support it.
- **Debugging**: Debugging recursive functions involves tracking the state of each frame on the call stack, which can be complex. Stack traces are often used to trace the function calls leading up to an error.

Understanding the call stack is essential for debugging, optimizing, and writing efficient and error-free code, especially in languages that support recursion and have complex function call structures.

12.4 Why Use Recursion?

☞ If you organize your thoughts recursively, your algorithm becomes a three-line program, and all the details are behind the scenes in the recursive stack that your program does for you.

Recursion is a powerful and often elegant approach to problem-solving. It offers several advantages over iterative methods, particularly for certain types of problems. This section explores why recursion can be beneficial, illustrated with examples.

12.4.1 Simplification

Recursion can simplify the solution to complex problems by breaking them down into smaller, more manageable subproblems. This reduction can make the problem easier to understand and solve.

Example: Finding the Greatest Common Divisor (GCD) Using Euclidean Algorithm

The Euclidean algorithm is a classic example of how recursion can simplify a problem. It provides an elegant and concise method for finding the greatest common divisor (GCD) of two integers.

Problem Definition

The **GCD** of two integers a and b is the largest integer that divides both a and b without leaving a remainder. The Euclidean algorithm relies on the principle that:

$$\mathbf{gcd}(a, b) = \mathbf{gcd}(b, a \bmod b)$$

where 'mod' denotes the modulo operation. The algorithm continues until one of the numbers becomes zero, at which point the other number is the **GCD**.

This recursive property is the foundation of the algorithm, reducing the problem size in each step.

Recursive Approach

The recursive solution directly reflects the Euclidean algorithm's steps, making the implementation both elegant and easy to understand:

```python
def gcd(a, b):
    if b == 0:
       return a
    else:
       return gcd(b, a % b)

# Example usage
a = 48
b = 18
print(f"The GCD of {a} and {b} is {gcd(a, b)}")
```

Explanation

1. **Base Case**: When $b = 0$, the **GCD** is a. This is because any number is divisible by itself, and the remainder is zero.

2. **Recursive Case**: Compute **gcd**$(b, a \mod b)$. The modulo operation reduces the size of the problem, and the recursion continues with the new values.

Why this approach is elegant and concise?

- **Direct Mapping**: The recursive solution closely follows the mathematical definition of the Euclidean algorithm, making it straightforward and intuitive.
- **Compact Code**: The recursive implementation is concise and easy to read, requiring only a few lines of code to solve the problem.
- **Simplified Logic and Readability**: The recursive approach simplifies the problem-solving process by reducing the problem size in each step and directly applying the algorithm's principle.

Iterative Approach for Comparison

The iterative approach to finding the GCD involves using a loop to simulate the recursive steps:

```python
def gcd_iterative(a, b):
    while b != 0:
       a, b = b, a % b
    return a

# Example usage
a = 48
b = 18
print(f"The GCD of {a} and {b} is {gcd_iterative(a, b)}")
```

While the iterative approach is also efficient, it involves additional logic for managing the loop and updating variables. The recursive approach provides a cleaner and more direct implementation of the Euclidean algorithm.

The recursive approach to finding the GCD using the Euclidean algorithm highlights the power of recursion in simplifying complex problems. By directly implementing the algorithm's principle, the recursive solution is both elegant and concise, showcasing how recursion can streamline problem-solving and reduce code complexity.

12.4.2 Natural Fit for Certain Problems

Recursion is particularly well-suited for problems with a recursive structure, where the solution involves solving smaller instances of the same problem.

Example: Fibonacci Sequence

The sequence $0, 1, 1, 2, 3, 5, 8, 13, 21 \cdots$ known as the Fibonacci sequence is a classic example of a problem where recursion is a natural fit. The problem's recursive nature simplifies the implementation and makes it easy to understand. The Fibonacci sequence is defined as follows:

$$\begin{aligned} &F(0) = 0 \\ &F(1) = 1 \\ &F(n) = F(n-1) + F(n-2) && \text{for } n \geq 2 \end{aligned}$$

In this sequence, each number is the sum of the two preceding ones. The recursive definition of the Fibonacci sequence lends itself naturally to a recursive implementation.

Recursive Approach

The recursive solution directly follows the mathematical definition of the Fibonacci sequence, making it both intuitive and easy to understand.

Recursive Solution:

```
def fibonacci_recursive(n):
    if n <= 1:
       return n
    else:
       return fibonacci_recursive(n - 1) +
    fibonacci_recursive(n - 2)

# Example usage
n = 5
print(f"Fibonacci number at position {n} is
    {fibonacci_recursive(n)}")   #outputs 5
```

Explanation:

- **Base Cases**: If n is 0 or 1, return n. These are the initial conditions of the Fibonacci sequence.

- **Recursive Case**: Compute $F(n)$ as $F(n-1) + F(n-2)$. The function calls itself to calculate the two preceding Fibonacci numbers, breaking the problem into smaller subproblems.

Iterative Approach for Comparison

While recursion provides an elegant solution, the iterative approach can be more efficient, especially for larger values of n :

```
def fibonacci_iterative(n):
    if n <= 1:
       return n
    a, b = 0, 1
    for _ in range(2, n + 1):
       a, b = b, a + b
    return b

# Example usage
n = 7
print(f"Fibonacci number at position {n} is
    {fibonacci_iterative(n)}") #outputs 13
```

The iterative approach uses a loop to compute the Fibonacci numbers. It keeps track of the last two numbers and updates them iteratively. The iterative approach is more efficient in terms of time and space complexity compared to the recursive approach, especially for large n, as it avoids redundant calculations and stack overhead.

The recursive solution to the Fibonacci sequence problem demonstrates how recursion is a natural fit for problems with recursive definitions. The recursive approach provides a clear and elegant implementation that mirrors the problem's mathematical structure. While iterative solutions can be more efficient, recursion remains a valuable technique for its simplicity and alignment with the problem's inherent structure.

In summary, recursion is a valuable concept to learn and understand. It is good programming practice when used appropriately and can provide elegant solutions to problems. However, it's also important to know when iterative solutions might be more efficient or simpler to implement.

12.5 Steps for Solving Computational Problems using Recursion

As already said, the recursive problem-solving technique involves solving a problem by breaking it down into smaller, more manageable subproblems of the same type. This method relies on a function calling itself to handle these subproblems, which can lead to elegant and efficient solutions. To effectively use recursion for computational problem-solving, one must follow a structured approach. This includes understanding the problem thoroughly, identifying the base case to prevent infinite recursion, defining the recursive case to break the problem into simpler instances, implementing the recursive function by combining the base and recursive cases, and rigorously testing the function to ensure its correctness across various inputs. By adhering to these steps, one can leverage recursion to tackle a wide range of computational challenges, from simple arithmetic operations to complex data structure manipulations.

☞ The recursive approach is like peeling an onion; you peel off layers until you reach the core, then you rebuild the layers to reach the final result.

To illustrate these steps, let us apply recursion to compute the sum of elements in a list.

1. **Understand the Problem**

   Problem: Given a list of numbers, calculate the sum of all its elements.

   Example Input: $[1, 2, 3, 4, 5]$

   Example Output: 15

   To solve this, we need to add up all the elements in the list. Recursion will help us simplify this task by handling one element at a time and summing it with the result of the sum of the remaining elements.

2. **Identify the Base Case**

   Base Case: The simplest instance of the problem that can be solved directly without recursion.

   For summing elements in a list, the base case is when the list is empty, and so the sum is 0.

   Base Case Condition: $\mathbf{sum([]) = 0}$

3. **Define the Recursive Case**

   Recursive Case: Break down the problem into smaller subproblems and express the solution in terms of these subproblems.

   For a non-empty list, the sum can be calculated by adding the first element of the list to the sum of the remaining elements. This reduces the problem size by removing the first element and applying the same sum operation to the rest of the list.

   Recursive Case Formula: $\mathbf{sum(lst) = lst[0] + sum(lst[1{:}])}$

4. **Implement the Recursive Function**

   Combine the base case and recursive case into a single function.

Implementation in Python:

```
def sum_list(lst):
    # Base case: if the list is empty, return 0
    if len(lst) == 0:
          return 0

    # Recursive case: add the first element to the sum of
    # the rest of the list

    else:
          return lst[0] + sum_list(lst[1:])
```

Explanation:

- If the list is empty (**len**(**lst**) == **0**), return 0 (base case).
- Otherwise, return the first element (**lst**[**0**]) plus the result of calling **sum_list** on the rest of the list (**lst**[**1** :]).

5. **Test the Recursive Function**

   Testing ensures that the function handles different scenarios correctly.

   Test Cases:

```
#Test case 1: Empty list
print(sum_list([]))          # Expected output: 0

# Test case 2: List with one element
print(sum_list([5]))         # Expected output: 5

# Test case 3: List with multiple elements
print(sum_list([1, 2, 3, 4, 5])) # Expected output: 15

# Test case 4: List with negative and positive elements
print(sum_list([-1, 2, -3, 4]))  # Expected output: 2
```

   Explanation of Tests:

   - The function should correctly return 0 for an empty list.
   - It should return the element itself if the list has only one element.
   - For a typical list with multiple elements, it should compute the sum of all elements.
   - For a list with negative and positive numbers, the function should still correctly compute the sum.

The example of summing elements in a list demonstrates how recursion can simplify the problem-solving process and lead to clean, concise code.

12.6 Example Problems

To effectively illustrate the utility of recursion, we consider a range of example problems that showcase its versatility. These include finding the maximum value within a list and reversing a string. Each of these problems exemplifies how recursion can simplify and streamline solutions by leveraging base and recursive cases. By understanding and implementing these examples, one can grasp the fundamental principles of recursion and apply them to a wide array of computational challenges.

12.6.1 Finding the Maximum Element in a List

The objective is to find the maximum element in a list of numbers.

Example Input: $[1, 3, 5, 2, 4]$

Example Output: 5

Solution Steps

1. **Understand the Problem**: Identify the largest number in the list.

2. **Identify the Base Case**: If the list has only one element, that element is the maximum.

3. **Define the Recursive Case**: The maximum of a list is the greater of the first element and the maximum of the rest of the list.

4. **Implement the Recursive Function**:

```
def max_list(lst):
      if len(lst) == 1:  # Base case: single element
          return lst[0]
      else:
          max_of_rest = max_list(lst[1:])
          return lst[0] if lst[0] > max_of_rest else
    max_of_rest
```

5. **Test the Recursive Function**:

```
print(max_list([1]))            # Expected output: 1
print(max_list([1, 3, 5, 2, 4])) # Expected output: 5
print(max_list([-1, -2, -3]))    # Expected output: -1
```

12.6.2 Reversing a String

The objective is to reverse a given string.

Example Input: ”hello”

Example Output: ”olleh”

Solution Steps

1. **Understand the Problem**: Reverse the order of characters in the string.
2. **Identify the Base Case**: An empty string or a single character string is its own reverse.
3. **Define the Recursive Case**: The reverse of a string is the reverse of the substring excluding the first character, plus the first character.
4. **Implement the Recursive Function**:

```
def reverse_string(s):
    """ Base case: empty string or single character """
    if len(s) <= 1:
        return s
    else:
        return reverse_string(s[1:]) + s[0]

print(reverse_string(""))      # Expected output: ""
print(reverse_string("a"))     # Expected output: "a"
print(reverse_string("hello")) # Expected output: "olleh"
```

5. **Test the Recursive Function**:

```
print(reverse_string(""))      # Expected output: ""
print(reverse_string("a"))     # Expected output: "a"
print(reverse_string("hello")) # Expected output: "olleh"
```

We will go through some additional examples with minimal detailed explanation.

12.6.3 Counting the Occurrences of a Value in a List

Count how many times a specific value appears in a list.

Example Input: [1, 2, 3, 2, 4, 2], **value** = 2

Example Output: 3

Solution Idea:

1. If the list is empty, return 0.
2. Compare the first element with the target value.
3. Add 1 to the result if they are equal; otherwise, add 0.
4. Recursively count the occurrences in the rest of the list.

Recursive Solution:

- **Base Case**: If the list is empty, the count is 0.
- **Recursive Case**: The count of **value** in the list is 1 if the first element matches **value**, otherwise, it is 0 plus the count in the rest of the list.

Python Code:

```
def count_occurrences(lst, value):
    if not lst:  # Base case: empty list
       return 0
    count = 1 if lst[0] == value else 0
    return count + count_occurrences(lst[1:], value)
```

12.6.4 Flattening a Nested List

Flatten a nested list (a list that may contain other lists) into a single list of values.

Example Input: [1, [2, [3, 4], 5], 6]

Example Output: [1, 2, 3, 4, 5, 6]

Solution Idea:

1. Initialize an empty list to store the result.
2. Iterate over each element in the nested list.
3. If the element is a list, recursively flatten it and extend the result list.
4. If the element is not a list, append it directly to the result list.

Recursive Solution:

- Base Case: If the element is not a list, return it as a single-element list.
- **Recursive Case**: If the element is a list, recursively flatten each item in the list and concatenate the results.
    - Process each element in the list.

  - If an element is a list, recursively flatten it.
  - Otherwise, add the element to the result.

Python Code:

```
def flatten(nested_list):
    flat_list = []
    for element in nested_list:
       if isinstance(element, list):
           flat_list.extend(flatten(element))
       else:
           flat_list.append(element)
    return flat_list
```

12.6.5 Calculating the Power of a Number

Compute the power of a number x^n, where x is the base and n is the exponent.

Example Input: $x = 2, n = 4$

Example Output: 16

Solution Idea:

1. If n is 0, return 1.
2. Multiply the base x by the result of **power**$(x, n - 1)$.
3. If n is negative, handle it by computing the reciprocal of the positive power.

Recursive Solution:

- **Base Case**: If the exponent n is 0, the result is 1 (since any number raised to the power of 0 is 1).
- **Recursive Case**: To compute x^n, multiply x by $x^{(n-1)}$
  - If $n > 0$, return $x \times \text{power}(x, n - 1)$.
  - For $n < 0$, use the reciprocal of the base for the negative exponent.

Python Code:

```
def power(x, n):
    if n == 0:  # Base case: x^0 = 1
       return 1
    else:
       return x * power(x, n - 1)
```

12.6.6 Generating All Subsets of a Set

Generate all possible subsets of a given set.

Example Input: $[1, 2, 3]$

Example Output: $[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]$

Solution Idea:

1. If the set is empty, return a list containing only the empty subset.
2. Recursively generate subsets of the remaining elements.
3. Combine the subsets with and without the first element.

Recursive Solution:

- **Base Case**: The subsets of an empty set are just the empty set itself.
- **Recursive Case**: To generate subsets of a set with elements, generate subsets including and excluding the first element.
  - Compute subsets of the remaining elements.
  - Include each subset of the remaining elements with and without the first element.

Python Code:

```
def generate_subsets(s):
    if len(s) == 0:  # Base case: empty set
       return [[]]
    first = s[0]
    rest_subsets = generate_subsets(s[1:])
    return rest_subsets + [subset + [first] for subset in
    rest_subsets]

print(generate_subsets([1,2,3]))

# Error in INPUT CASE
```

12.6.7 Solving a Maze (Backtracking)

Find a path from the start to the end of a maze represented as a 2D grid, where 0 indicates a free cell and 1 indicates a wall.

Example Input:

```
[
[0, 0, 1, 0],
[0, 0, 1, 0],
[0, 0, 0, 0],
[0, 1, 0, 0]
]
```

Example Output: Path from start (0, 0) to end (3, 3).

Solution Idea:

1. Check if the current cell is out of bounds or blocked, or if it has been visited.
2. If the current cell is the destination, return **True**.
3. Mark the current cell as visited.
4. Recursively attempt to move to adjacent cells.
5. If moving to an adjacent cell leads to a solution, return **True**.
6. If none of the moves lead to a solution, backtrack by unmarking the current cell.

Recursive Solution:

- **Base Case**: If the current position is the end, return **True**.
- **Recursive Case**: Move to adjacent cells (up, down, left, right), and recursively check if moving to these cells leads to a solution.
  - If moving to an adjacent cell is valid (i.e., within bounds and not a wall), mark it as part of the path.
  - Recursively try to solve the maze from the new position.
  - If a solution is found, return **True**; otherwise, backtrack.

Python Code:

```python
def solve_maze(maze, x, y, path):
    if x < 0 or x >= len(maze) or y < 0 or y >= len(maze[0]) or
    maze[x][y] == 1 or (x, y) in path:
        return False

    # Base case: reached the end of the maze
    if (x, y) == (len(maze) - 1, len(maze[0]) - 1):
        path.append((x, y))
        return True
    path.append((x, y))

    if (solve_maze(maze, x + 1, y, path) or
        solve_maze(maze, x - 1, y, path) or
        solve_maze(maze, x, y + 1, path) or
        solve_maze(maze, x, y - 1, path)):
        return True

    path.pop()
    return False

# Example usage:
maze = [[0, 0, 0, 0],
        [1, 1, 0, 1],
        [0, 0, 0, 0],
        [0, 1, 1, 0]]
path = []
if solve_maze(maze, 0, 0, path):
    print("Path found:", path)
else:
    print("No path found")

# Output is given below:
# Path found: [(0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (2, 3),
    (3, 3)]
```

12.7 Avoiding Circularity in Recursion

Avoiding circularity in recursion is crucial to ensure that your recursive functions terminate correctly and do not fall into infinite loops. Circularity in recursion can occur if a function keeps calling itself without a proper termination condition or if the termination conditions are not properly defined. Here are some strategies to avoid circularity in recursion:

1. **Define a Clear Base Case**

   The base case is a condition that stops further recursive calls. Every recursive function must have a base case to prevent infinite recursion.

   Python Example: Sum of Natural Numbers

```
def sum_natural_numbers(n):
    """ Base case: if n is 0 or negative, return 0 """
    if n <= 0:
        return 0
    else: # Recursive call
        return n + sum_natural_numbers(n - 1)

# Test
print(sum_natural_numbers(5))  # Output: 15
# (5 + 4 + 3 + 2 + 1 = 15)
```

2. **Ensure Progress Towards the Base Case**

   Each recursive call should make progress toward the base case. This means modifying parameters so that the function eventually reaches the base case.

   Python Example: Counting Down

```
def countdown(seconds):
    """ Base case: If seconds is less than or equal to 0,
    print 'Time's up!' and end recursion. """
    if seconds <= 0:
        print("Time's up!")
    else:
        print(seconds)
        # Ensure progress by decrementing the number of
    seconds.
        # Recursive call with the updated number of seconds.
        countdown(seconds - 1)

# Test
countdown(5)  # Counts down from 5 to 0
```

3. **Avoid Unchanging Parameters**

   The parameters passed in each recursive call should be updated to ensure that they eventually lead to the base case. Avoid passing parameters that would lead to redundant or unchanged recursion.

   Python Example: Computing Exponentiation

```
def power(base, exponent):
    """ Base case: any number to the power 0 is 1 """
    if exponent == 0:
        return 1
    else:
        # Recursive call with decremented exponent
        return base * power(base, exponent - 1)

# Test
print(power(2, 3))  # Output: 8 (2^3)
```

4. **Handle Edge Cases Appropriately**

   Ensure that edge cases are handled properly so the function behaves correctly for all possible inputs, including special or boundary instances.

   Python Example: Printing Elements in a List

```python
def print_list(lst, index=0):
    """ Base case: index out of range of the list """
    if index >= len(lst):
        return
    print(lst[index])
    # Recursive call with incremented index
    print_list(lst, index + 1)

# Test
print_list([10, 20, 30])  # Output: 10 20 30
```

5. **Use Debugging Tools**

   If you suspect your recursive function might have issues with termination, use debugging techniques to trace function calls and verify that recursion progresses toward the base case.

   Python Example: Fibonacci Sequence with Debugging

```python
def fibonacci(n):
    print(f"Calling fibonacci({n})")  # Debugging statement
    if n <= 1:  # Base case: fibonacci of 0 or 1
        return n
    else:  # Recursive calls
        return fibonacci(n - 1) + fibonacci(n - 2)

# Test
print(fibonacci(4))
# Output:
""" Calling fibonacci(4)
Calling fibonacci(3)
Calling fibonacci(2)
Calling fibonacci(1)
Calling fibonacci(0)
Calling fibonacci(1)
Calling fibonacci(2)
Calling fibonacci(1)
Calling fibonacci(0)
3 """
```

Following these strategies and examples, you can create recursive functions that are effective and safe from circularity issues.

12.8 Iteration vs. Recursion

In programming, two fundamental techniques for solving problems are iteration and recursion. Each approach has its own strengths and weaknesses, and understanding these differences can help you choose the most appropriate method for a given problem.

12.8.1 Iteration

Iteration involves using looping constructs such as `for`, `while`, or `do-while` loops to repeatedly execute a block of code until a specific condition is met.

Key Characteristics

- **Control Flow:** Iteration explicitly manages the flow of execution through loop constructs. The loop continues as long as the loop condition evaluates to `true`.
- **State Management:** State is maintained using loop variables and accumulators that are updated during each iteration.
- **Memory Usage:** Iteration typically requires a fixed amount of additional memory for loop variables, making it more memory-efficient compared to recursion.
- **Performance:** Generally faster due to reduced overhead associated with function calls and stack management.
- **Termination:** The loop terminates when the loop condition becomes `false`.

Example

To compute the sum of the first n natural numbers using iteration:

```python
def sum_iterative(n):
    total = 0
    for i in range(1, n + 1):
       total += i
    return total
```

Advantages

- **Efficiency:** More efficient in terms of both time and space for simple repetitive tasks.
- **Simplicity:** Easier to understand and implement for straightforward problems.

12.8.2 Recursion

Recursion involves a function calling itself to solve smaller instances of the same problem. The function continues to call itself with modified arguments until a base case is reached.

Key Characteristics

- **Control Flow:** Recursion implicitly manages execution flow through recursive function calls. Each call reduces the problem size, and the base case provides a stopping point.
- **State Management:** Each recursive call creates a new stack frame, leading to potentially higher memory usage.
- **Memory Usage:** Can be less efficient due to the overhead of managing multiple stack frames.
- **Performance:** May be slower due to the overhead associated with function calls and stack management.
- **Termination:** Recursion terminates when the base case is reached, and the function begins returning and resolving each recursive call.

Example

To compute the factorial of a number using recursion:

```
def factorial_recursive(n):
    if n == 0:
       return 1
    else:
       return n * factorial_recursive(n - 1)
```

Advantages

- **Elegance:** Provides a more natural and concise solution for problems with a recursive structure.
- **Readability:** Simplifies complex problems, such as those involving hierarchical or divide-and-conquer structures.

12.8.3 Comparing Iteration and Recursion

Memory Efficiency

- **Iteration:** Generally uses a constant amount of memory for loop variables.
- **Recursion:** May be less efficient due to the overhead of maintaining multiple stack frames.

Control Flow

- **Iteration:** Explicitly controlled with loop constructs and conditions.
- **Recursion:** Controlled through recursive function calls and base cases.

Performance

- **Iteration:** Typically more performance due to reduced overhead.
- **Recursion:** May be slower due to the cost of managing the call stack.

Use Cases

- **Iteration:** Best suited for problems involving straightforward repetitive operations.
- **Recursion:** Ideal for problems with a hierarchical structure or that can be divided into similar sub-problems.

☞ Recognize that recursion can sometimes be less efficient due to the overhead of function calls and memory usage. Evaluate whether iteration might be a more efficient alternative for some problems.

The choice between iteration and recursion depends on the nature of the problem, performance considerations, and readability. Iteration is generally preferred for simple, repetitive tasks due to its efficiency, while recursion is useful for problems with a recursive structure or hierarchical nature.

12.9 Conclusion

Recursive thinking is a fundamental technique in problem-solving, offering a clear and elegant way to tackle complex problems by breaking them down into simpler subproblems. While recursion may introduce challenges related to performance and memory usage, understanding its principles and applications is essential for mastering various computational problems. Through practice and visualization, recursive thinking becomes an invaluable tool in a problem solver's arsenal, providing a powerful approach to designing and implementing efficient algorithms. This chapter has introduced the basics of recursion, illustrated through many examples, and highlighted the benefits and challenges of recursive thinking. In the following chapters, we will explore more complex examples and applications of recursion in different problem domains.

Exploring recursive thinking can deepen your understanding of problem-solving and algorithm design. Here are some highly recommended books that provide a solid foundation in recursive thinking and related concepts. [1] Often referred to as CLRS, this book is a comprehensive resource on algorithms, including recursion. It covers various algorithmic techniques and provides detailed explanations and examples. This book focuses on recursive algorithms, dynamic programming, and divide-and-conquer techniques. [2] Knuth's classic work covers a wide range of algorithms and includes deep dives into recursive algorithms and their analysis. This book focuses on recursion, mathematical foundations, and algorithm analysis. [3] This book offers a modern introduction to algorithms, with a focus on both practical applications and theoretical foundations. It includes discussions on recursion and provides numerous examples. [4] Bentley's book provides practical advice on programming and problem-solving, including recursive techniques. It's known for its insightful and clear explanations. [5] This book offers a collection of programming problems and solutions, including those involving recursion. It is designed to help with coding interviews and practical problem-solving. [6] A focused resource on recursion and backtracking techniques. It provides explanations and examples to help understand and apply recursive thinking. [7] This book provides an in-depth look at data structures and algorithms, with a focus on recursive algorithms in C++. It includes practical examples and analysis. [8] This book focuses on implementing algorithms in Python, including recursive algorithms. It provides practical examples and exercises.

12.10 Exercises

1. Determine if a string is a palindrome by checking if the first and last characters match and recursively checking the substring between them.

2. Find the length of a string by counting one character at a time and recursively reducing the problem to the rest of the string.

3. Compute the sum of the digits of a number by recursively summing the last digit and the sum of the digits of the remaining number.

4. Reverse a string by considering the first character and recursively reversing the rest of the string.

5. Generate all combinations of k items from a set of n items by including or excluding each item recursively.

6. Check if a list is sorted in ascending order by comparing adjacent elements and recursively checking the rest of the list.

7. Find the median of two sorted arrays of possibly different sizes by dividing them into halves recursively.

8. Determine the number of unique paths from the top-left to the bottom-right corner of a grid with obstacles, where you can only move right or down.

9. Given a set of integers, find all subsets whose product equals a given target value.

10. Implement a recursive function to generate all valid balanced parentheses expressions for n pairs of parentheses.

11. Find the number of ways to make change for a given amount using a limited supply of coins.

12. Given an $m \times n$ matrix, recursively transpose the matrix (swap rows and columns).

13. Calculate the number of ways to climb a staircase where you can take 1, 2, or up to k steps at a time, given n stairs.

14. Count the number of ways to tile a $2 \times n$ board using 1×2.

15. Count the number of binary strings of length n that do not contain two consecutive zeros.

16. Given a chessboard, print all sequences of moves of a knight on a chessboard such that the knight visits every square only once.

17. Implement a recursive function to solve the N-Queens problem and place n queens on an $n \times n$ chessboard such that no two queens threaten each other.

Bibliography

[1] Thomas H. Cormen and Charles E. Leiserson and Ronald L. Rivest and Clifford Stein, Introduction to Algorithms, The MIT Press, 2009.

[2] Donald E. Knuth, The Art of Computer Programming (Volume 1), Addison-Wesley, 1997.

[3] Robert Sedgewick and Kevin Wayne, Algorithms, Addison-Wesley, 2011.

[4] programming Jon Bentley, Programming Pearls, Addison-Wesley, 1986.

[5] Adnan Aziz and Tsung-Hsien Lee and Prakash Rao, Elements of Programming Interviews, EPI, 2012.

[6] Eric Roberts, Recursion and Backtracking, Prentice Hall, 2011.

[7] Mark Allen Weiss, Data Structures and Algorithm Analysis in C++, Pearson, 2013.

[8] Magnus Lie Hetland, Python Algorithms: Mastering Basic Algorithms in the Python Language, Apress, 2013.

[9] Ronald Graham, Donald Knuth, and Oren Patashnik, Concrete Mathematics, Addison-Wesley, 1989.

Chapter 13

Computational Approaches To Problem-Solving

"The problem of computing is not about speed, it's about understanding."
– Donald E Knuth

"We build our machines in such a way that they carry out our instructions, and we in turn must carry out instructions given to us by our machines."
– Edsger W. Dijkstra

Computational approaches to problem-solving encompass a diverse range of methodologies that leverage computational power to address complex challenges across various domains. This chapter explores several fundamental strategies—brute force, divide-and-conquer, dynamic programming, greedy algorithms, and randomized approaches—each offering unique insights and techniques to tackle different classes of problems effectively.

The brute-force approach represents simplicity and exhaustive computation. It involves systematically checking every possible solution to find the optimal one, making it ideal for problems like cracking padlocks or guessing passwords. Despite its simplicity, brute-force methods can be computationally expensive, especially for problems with large solution spaces, leading to impractical execution times in real-world applications.

In contrast, the divide-and-conquer approach breaks down problems into smaller, more manageable sub-problems until they become simple enough to solve directly. The merge sort algorithm exemplifies this strategy by recursively dividing an array into halves, sorting them, and then merging them back together. This approach benefits from improved efficiency over brute-force methods by reducing the time complexity through systematic decomposition. However, it may incur additional overhead due to recursive function calls and memory requirements for storing sub-problems.

Dynamic programming focuses on solving problems by breaking them down into overlapping sub-problems and storing the results to avoid redundant computations. Unlike divide-and-conquer, dynamic programming optimizes efficiency by memoizing intermediate results, significantly reducing computational complexity for problems with overlapping sub-problems. This approach highlights the trade-off between space and time complexity, making it particularly effective for optimization problems where solutions depend on prior computed results.

Greedy algorithms, such as maximizing the number of tasks completed within a limited time frame, make locally optimal choices at each step with the aim of reaching a global optimum. This approach is motivated by its simplicity and efficiency in finding quick solutions. However, greedy algorithms may overlook globally optimal solutions due to their myopic decision-making process, emphasizing immediate gains over long-term strategy.

Lastly, randomized approaches introduce randomness into problem-solving, offering probabilistic solutions to otherwise deterministic problems. Examples include scenarios like coupon collecting or hat-checking at a party, where outcomes depend on random chance rather than deterministic algorithms. Motivated by their ability to explore solution spaces unpredictably, randomized approaches provide insights into stochastic phenomena and offer innovative solutions in scenarios where exact solutions are impractical or unavailable.

This chapter will delve into each computational approach in detail, exploring their theoretical underpinnings, practical applications, advantages, and limitations. By understanding these methodologies, you will gain insights into selecting appropriate strategies for solving diverse computational problems effectively across various disciplines.

13.1 Brute-Force Approach to Problem Solving

"To solve any problem, you need to start with a clear definition of the problem and then look at all possible solutions."

– John McCarthy

Many problems are addressed by exploring a vast number of possibilities. For instance, chess engines evaluate numerous move variations to determine the "best" positions. This method is known as brute force. Brute force algorithms take advantage of a computer's speed, allowing us to rely less on sophisticated techniques. However, even with brute force, some level of creativity is often required. For example, a brute force solution might involve evaluating 2^{40} options, but a more refined approach could potentially reduce this to 2^{20}. Such a reduction can significantly decrease the computational time. The effectiveness of a brute force approach often hinges on how cleverly the problem is analyzed and optimized, making it crucial to explore different strategies to improve efficiency.

The brute-force approach is a fundamental method in problem-solving that involves systematically trying every possible solution to find the optimal one. This section explores the concept, applications, advantages, and limitations of the brute-force approach through various examples across different domains.

The brute-force approach, also known as exhaustive search, operates by checking all possible solutions systematically, without employing any sophisticated strategies to narrow down the search space. It ensures finding a solution if it exists within the predefined constraints but can be computationally intensive and impractical for problems with large solution spaces.

Examples of Brute-Force Approach

1. **Padlock**

   Imagine you encounter a padlock with a four-digit numeric code. The brute-force approach would involve sequentially trying every possible combination from ”0000” to ”9999” until the correct code unlocks the padlock. Despite its simplicity and guaranteed success in finding the correct combination eventually, this method can be time-consuming, especially for longer or more complex codes.

2. **Password Guessing** In the realm of cybersecurity, brute-force attacks are used to crack passwords by systematically guessing every possible combination of characters until the correct password is identified. This approach is effective against weak passwords that are short or lack complexity. For instance, attacking a six-character password consisting of letters and digits would involve testing all 2.18 billion (36^6) possible combinations until the correct one is identified.

3. **Cryptography: Cracking Codes**

   In cryptography, brute-force attacks are used to crack codes or encryption keys by systematically testing every possible combination until the correct one is found. For example, breaking a simple substitution cipher involves trying every possible shift in the alphabet until the plain-text message is deciphered.

4. **Sudoku Solving**

   Brute-force methods can be applied to solve puzzles like Sudoku by systematically filling in each cell with possible values and backtracking when contradictions arise. This method guarantees finding a solution but may require significant computational resources, especially for complex puzzles.

13.1.1 Characteristics of Brute-Force Solutions

1. **Exhaustive Search**: Every possible solution is examined without any optimization.

2. **Simplicity**: Easy to understand and implement.

3. **Inefficiency**: Often slow and resource-intensive due to the large number of possibilities.

4. **Guaranteed Solution**: If a solution exists, the brute-force method will eventually find it.

13.1.2 Solving Computational Problems Using Brute-force Approach Approach

To solve computational problems using the brute-force approach, one must systematically explore and evaluate all possible solutions to identify the correct answer. This involves defining the problem clearly, generating every potential candidate solution, and checking each one against the problem's criteria to determine its validity. While this method ensures that all possible solutions are considered, it often results in high computational costs and inefficiencies, especially for large or complex problems. Despite these limitations, the brute-force approach provides a straightforward and reliable way to solve problems by exhaustively searching the solution space, offering a foundation for understanding and improving more advanced algorithms.

☞ The brute-force approach is like searching for a needle in a haystack by sifting through each strand one by one; it is simple but can be overwhelmingly inefficient.

Let us look at some examples and see how we can apply the brute-force approach to solve a computational problem.

13.1.2.1 Problem-1 (String Matching)

The brute-force string matching algorithm is a simple method for finding all occurrences of a pattern within a text. The idea is to slide the pattern over the text one character at a time and check if the pattern matches the substring of the text starting at the current position. Here is a step-by-step explanation:

1. **Start at the beginning of the text**: Begin by aligning the pattern with the first character of the text.
2. **Check for a match**: Compare the pattern with the substring of the text starting at the current position. If the substring matches the pattern, record the position.
3. **Move to the next position**: Shift the pattern one character to the right and repeat the comparison until you reach the end of the text.
4. **Finish**: Continue until all possible positions in the text have been checked.

This approach ensures that all possible starting positions in the text are considered, but it can be slow for large texts due to its time complexity.

Here is how you can implement the brute-force string-matching algorithm in Python:

```
def brute_force_string_match(text, pattern):
    n = len(text)       # Length of the text
    m = len(pattern)    # Length of the pattern

    for i in range(n - m + 1):
        substring = text[i:i + m]
        """ Loop over each possible starting index in the text,
    Extracting the substring of the text from the current
    position """


        # Compare the substring with the pattern
        if substring == pattern:
           print(f"Pattern found at index {i}")




# Example usage
text = "ABABDABACDABABCABAB"
pattern = "ABABCABAB"
brute_force_string_match(text, pattern)
```

Explanation:

1. **Function Definition**: The function **brute__force__string__match** takes two arguments: **text** and **pattern**.

2. **Length Calculation**: It calculates the lengths of both the **text** and **pattern** to determine how many possible starting positions there are.

3. **Loop Over Positions**: It uses a for loop to slide the pattern across the text. The loop runs from **0** to **n - m + 1**, where **n** is the length of the **text** and **m** is the length of the **pattern**.

4. **Substring Extraction**: At each position **i**, the code extracts a substring from the text that has the same length as the pattern (**text[i:i + m]**).

5. **Comparison**: It then compares this substring with the pattern. If they match, it prints the starting index where the pattern was found.

6. **Example Usage**: The example shows how to call the function with a sample text and pattern. In this case, the function prints the indices where the pattern occurs within the text.

This implementation is straightforward and guarantees finding all occurrences of the pattern, but it may not be efficient for large texts or patterns due to its **((n - m + 1) * m)** time complexity.

Given the example usage in our **brute__force__string__match** function:

```
text = "ABABDABACDABABCABAB"
pattern = "ABABCABAB"
brute_force_string_match(text, pattern)
```

Here is what the output of the function will be:

```
Pattern found at index 10
```

The function searches through the **text** and finds the **pattern** starting at index **10**. In this case, **"ABABCABAB"** begins at position **10** in the **text**, so that is where the function prints the message indicating the pattern's location.

The function does not find the pattern at any other starting position in the provided text, so only this single index is printed.

13.1.2.2 Problem-2 (Subset Sum Problem)

The Subset Sum Problem involves determining if there exists a subset of a given set of numbers that sums up to a specified target value. The brute-force approach to solve this problem involves generating all possible subsets of the set and checking if the sum of any subset equals the target value.

Here is how the brute-force approach works:

1. **Generate subsets:** Iterate over all possible subsets of the given set of numbers.
2. **Calculate sums:** For each subset, calculate the sum of its elements.
3. **Check target:** Compare the sum of each subset with the target value.
4. **Return result:** If a subset's sum matches the target, return that subset. Otherwise, conclude that no such subset exists.

This method guarantees finding a solution if one exists but can be inefficient for large sets due to its exponential time complexity.

Here is a Python code to implement the brute-force approach for the Subset Sum Problem:

```
def subset_sum_brute_force(nums, target):
    n = len(nums)

    # Loop over all possible subsets

    for i in range(1 << n):  # There are 2^n subsets
       subset = [nums[j] for j in range(n) if (i & (1 << j))]
       if sum(subset) == target:
           return subset

    return None
```

```
# Example usage
nums = [3, 34, 4, 12, 5, 2]
target = 9
result = subset_sum_brute_force(nums, target)
if result:
   print(f"Subset with target sum {target} found: {result}")
else:
   print("No subset with the target sum found.")
```

Explanation:

1. **Function Definition**:**subset__sum__brute__force** takes a list of numbers (**nums**) and a target sum (**target**).
2. **Subset Generation**: The loop **for i in range(1 « n)** iterates over all possible subsets. Here, **1 « n equals** 2^n, which is the total number of subsets for **n** elements. Each subset is generated using a bitmask approach: for each bit in the integer **i**, if it is set, the corresponding element is included in the subset.
3. **Subset Construction**: The subset is constructed by including elements where the corresponding bit in **i** is set ((**i & (1 « j)**)).
4. **Sum Calculation**: For each subset, the sum of its elements is calculated using **sum(subset)**.
5. **Target Check**: If the sum of the subset equals the target, the subset is returned.
6. **Return Result**: If no subset matches the target sum, the function returns **None**.
7. **Example Usage**: The example demonstrates finding a subset that sums up to **9** in the list [**3, 34, 4, 12, 5, 2**]. The output will either show the subset that matches the target or indicate that no such subset was found.

This brute-force approach is straightforward and guarantees finding a solution if one exists, but may not be efficient for large sets due to its exponential time complexity.

Given the example usage in our **subset__sum__brute__force** function:

```
nums = [3, 34, 4, 12, 5, 2]
target = 9
result = subset_sum_brute_force(nums, target)
if result:
   print(f"Subset with target sum {target} found: {result}")
else:
   print("No subset with the target sum found.")
```

Here is what the output of the function will be:

```
Subset with target sum 9 found: [3, 4, 2]
```

The function generates all possible subsets of the list **nums** and checks if any of them sum up to the target value **9**.

- **Subset Generation**: The function iterates through all possible subsets. For each subset, it calculates the sum and checks if it matches the target.
- **Subset Found**: In this case, the subset **[3, 4, 2]** sums up to **9**, which matches the target value. Therefore, this subset is returned and printed.

If no such subset were found, the function would print "No subset with the target sum found."

13.1.2.3 Problem-3 (Sudoku Solver)

The Sudoku Solver using the brute-force approach is a method to solve a Sudoku puzzle by trying every possible number in each empty cell until the puzzle is solved. The brute-force algorithm systematically fills in each cell with numbers from 1 to 9 and checks if the puzzle remains valid after each placement. If a placement leads to a valid state, the algorithm proceeds to the next empty cell. If a placement leads to a contradiction, the algorithm backtracks and tries the next number.

Here is a step-by-step explanation:

1. **Find an Empty Cell:** Locate the first empty cell in the Sudoku grid.
2. **Try Numbers:** Attempt to place each number from 1 to 9 in the empty cell.
3. **Check Validity:** Verify that placing the number does not violate Sudoku rules:
   - No repeated numbers in the same row.
   - No repeated numbers in the same column.
   - No repeated numbers in the same 3x3 sub-grid.
4. **Move to Next Cell:** If the placement is valid, proceed to the next empty cell.
5. **Backtrack if Necessary:** If a placement leads to an invalid state later, undo the placement (backtrack) and try the next number.
6. **Complete:** Continue until the Sudoku puzzle is fully solved or all possibilities are exhausted.

The brute-force approach guarantees finding a solution if one exists, but it can be inefficient for larger puzzles due to its exhaustive nature.

Here is the Python code for solving a Sudoku puzzle using the brute-force approach with recursive backtracking::

```
def is_valid(board, row, col, num):
    # Check if num is not repeated in the row

    if num in board[row]:
       return False

    # Check if num is not repeated in the column

    if num in (board[i][col] for i in range(9)):
       return False

    # Check if num is not repeated in the 3x3 sub-grid

    start_row, start_col = 3 * (row // 3), 3 * (col // 3)
    for i in range(start_row, start_row + 3):
       for j in range(start_col, start_col + 3):
           if board[i][j] == num:
               return False

    return True

def solve_sudoku(board):
    for row in range(9):
       for col in range(9):
           if board[row][col] == 0:  # Find an empty cell
               for num in range(1, 10):  # Try all numbers
                                         # from 1 to 9
                   if is_valid(board, row, col, num):
                       board[row][col] = num
                       # Place the number
                       if solve_sudoku(board):
                           return True
                       board[row][col] = 0  # Backtrack if
    needed
                   return False  # Trigger backtracking

    return True  # Puzzle solved

# Example usage
sudoku_board = [
[5, 3, 0, 0, 7, 0, 0, 0, 0],
[6, 0, 0, 1, 9, 5, 0, 0, 0],
[0, 9, 8, 0, 0, 0, 0, 6, 0],
[8, 0, 0, 0, 6, 0, 0, 0, 3],
[4, 0, 0, 8, 0, 3, 0, 0, 1],
[7, 0, 0, 0, 2, 0, 0, 0, 6],
[0, 6, 0, 0, 0, 0, 2, 8, 0],
[0, 0, 0, 4, 1, 9, 0, 0, 5],
[0, 0, 0, 0, 8, 0, 0, 7, 9]
]

if solve_sudoku(sudoku_board):
   for row in sudoku_board:
       print(row)
else:
    print("No solution exists.")
```

Explanation:

1. **is_valid Function**: This function checks if placing a number in a specific cell is valid according to Sudoku rules:
   - **Row Check**: Ensures the number is not already present in the same row.
   - **Column Check**: Ensures the number is not already present in the same column.
   - **Sub-grid Check**: Ensures the number is not already present in the 3x3 sub-grid.
2. **solve_sudoku Function**:
   - **Find Empty Cell**: Iterates through the grid to locate an empty cell (**0**).
   - **Try Numbers**: For each empty cell, try placing numbers from **1** to **9**.
   - **Check Validity**: Uses **is_valid** to check if the placement is valid.
   - **Recursive Call**: Recursively attempts to solve the rest of the board with the current placement.
   - **Backtrack**: If no valid number can be placed, reset the cell and try the next number.
   - **Completion**: If all cells are filled validly, returns **True**. If no cells are left to fill, returns **True** indicating the board is solved.
3. **Example Usage**: Demonstrates solving a Sudoku puzzle with a given **sudoku_board**. If the puzzle is solved, the board is printed row by row. If no solution exists, a message is displayed.

This brute-force algorithm ensures a solution if one exists but can be slow due to its exhaustive search approach.

Here is what the output would look like if the provided **solve_sudoku** function successfully solves the given Sudoku puzzle:

```
[5, 3, 4, 6, 7, 8, 9, 1, 2]
[6, 7, 2, 1, 9, 5, 3, 4, 8]
[1, 9, 8, 3, 4, 2, 5, 6, 7]
[8, 5, 9, 7, 6, 1, 4, 2, 3]
[4, 2, 6, 8, 5, 3, 7, 9, 1]
[7, 1, 3, 9, 2, 4, 8, 5, 6]
[9, 6, 1, 5, 3, 7, 2, 8, 4]
[2, 8, 7, 4, 1, 9, 6, 3, 5]
[3, 4, 5, 2, 8, 6, 1, 7, 9]
```

Explanation of the Output:

1. **Completed Sudoku Grid**: Each row shows a valid configuration where all rows, columns, and 3x3 sub-grids contain the numbers 1 through 9 without repetition.

2. **Successful Solution**: The **solve_sudoku** function filled all empty cells (originally **0** values) with valid numbers, resulting in a fully completed Sudoku board.

If the **solve_sudoku** function did not find a solution, it would print:

```
No solution exists.
```

This approach guarantees to find a solution if one exists but might be slow for more complex or larger Sudoku puzzles due to its exhaustive nature.

☞ Brute-force methods offer a straightforward path to solving problems by exploring every possible solution, but they often become impractical as the problem size grows.

13.1.3 Advantages and Limitations of Brute-Force Approach

Advantages of Brute-Force Approach

1. **Guaranteed Solution**: Brute-force methods ensure finding a solution if one exists within the predefined constraints.

2. **Simplicity**: The approach is straightforward to implement and understand, requiring minimal algorithmic complexity.

3. **Versatility**: Applicable across various domains where an exhaustive search is feasible, such as puzzle solving, cryptography, and optimization problems.

Limitations of Brute-Force Approach

1. **Computational Intensity**: It can be highly resource-intensive, especially for problems with large solution spaces or complex constraints.
2. **Time Complexity**: Depending on the problem size, brute-force approaches may require impractically long execution times to find solutions.
3. **Scalability Issues**: In scenarios with exponentially growing solution spaces, brute-force methods may become impractical or infeasible to execute within reasonable time constraints.

☞ Simplicity is the hallmark of brute-force algorithms; they operate without complex strategies but may suffer from exponential growth in computational demands.

13.1.4 Optimizing Brute-force Solutions

- **Pruning**: Eliminate certain candidates early if they cannot possibly be a solution. For example, in a search tree, cutting off branches does not lead to feasible solutions.
- **Heuristics**: Use rules of thumb to guide the search and reduce the number of candidates.
- **Divide and Conquer**: Break the problem into smaller, more manageable parts, solve each part individually, and combine the results.
- **Dynamic Programming**: Store the results of subproblems to avoid redundant computations.

☞ While brute-force approaches can serve as a baseline for evaluating other algorithms, their inefficiency limits their practical use to small-scale problems or as a verification tool.

The brute-force approach is a fundamental but often inefficient problem-solving technique. While it guarantees finding a solution if one exists, its practical use is limited by computational constraints. Understanding brute-force methods is essential for developing more sophisticated algorithms and optimizing problem-solving strategies. This section provides an in-depth exploration of the brute-force approach for solving computational problems, highlighting its simplicity, applications, limitations, and potential optimizations.

13.2 Divide-and-conquer Approach to Problem Solving

The process of breaking down a complex problem into simpler sub-problems is not only a fundamental programming technique but also a powerful strategy for managing complexity."

– Donald E Knuth

To illustrate the divide-and-conquer approach, imagine a classroom scenario where students are asked to organize a large number of books in a library. The problem of categorizing and shelving thousands of books can seem difficult at first. By applying divide-and-conquer principles, the task can be simplified significantly. The students can start by dividing the books into smaller groups based on genres, such as fiction, non-fiction, and reference. Each genre can then be further subdivided into categories like science fiction, historical novels, and biographies. Finally, within each category, the books can be organized alphabetically by author. This method of dividing the problem into manageable parts, solving each part, and then combining the results effectively demonstrates how divide-and-conquer can make complex tasks more approachable.

In the realm of project management, divide-and-conquer strategies are essential for handling large projects. For example, consider the construction of a high-rise building. The project is divided into smaller tasks, such as foundation work, structural framework, electrical installations, and interior finishes. Each task is managed by different teams or contractors specialized in that area. By breaking the project into these distinct components, project managers can ensure that each part is completed efficiently and effectively. This modular approach allows for parallel progress, timely completion, and integration of the individual tasks to achieve the final goal of constructing the building.

In software development, the divide-and-conquer approach is frequently employed in designing complex systems and applications. For instance, consider developing a comprehensive e-commerce platform. The platform is divided into various functional modules, such as user authentication, product catalog, shopping cart, and payment processing. Each module is developed and tested independently, allowing developers to focus on specific aspects of the system. Once all modules are completed, they are integrated to form a cohesive application. This method ensures that the development process is manageable and that each component functions correctly before being combined into the final product.

In healthcare, divide-and-conquer strategies are applied in diagnostic processes and treatment plans. For example, when diagnosing a complex medical condition, doctors might first divide the patient's symptoms into different categories, such as neurological, cardiovascular, and respiratory. Each category is investigated separately using targeted tests and consultations with specialists. The results from these investigations are then combined to form a comprehensive diagnosis and treatment plan. This approach helps in managing the complexity of medical diagnoses and ensures a thorough and accurate evaluation of the patient's health.

Finally, consider the field of logistics and supply chain management, where divide-and-conquer techniques are used to optimize the distribution of goods. For example, a company managing the supply chain for a large retailer might divide the supply chain into regional distribution centers. Each center handles a specific geographic area and manages local inventory, transportation, and delivery. By decentralizing the management of the supply chain into smaller, regional units, the company can improve efficiency, reduce costs, and enhance service levels. The results from each distribution center are then integrated to ensure a seamless supply chain operation across all regions.

As illustrated in these examples, the divide-and-conquer approach is also a fundamental computational problem-solving technique used to solve problems by breaking them down into smaller, more manageable sub-problems similar to the original problem. The basic idea is to divide the problem into smaller sub-problems, solve each sub-problem independently, and then combine their solutions to solve the original problem. This method is particularly effective for problems that exhibit recursive structure and can be broken into similar sub-problems.

Key Steps of Divide-and-Conquer:

1. **Divide**: Split the original problem into smaller sub-problems that are easier to solve. The sub-problems should be similar to the original problem.

   Consider the task of organizing a large set of files into a well-structured directory system. The first step involves breaking down this problem into smaller, more manageable subproblems. For instance, you might divide the files by their type (e.g., documents, images, videos) or by their project affiliation. Each subset of files is then considered a subproblem, which is more straightforward to organize than the entire set of files. The key is to ensure that each subset is similar to the original problem but simpler to handle individually.

2. **Conquer**: Solve the smaller sub-problems. If the sub-problems are small enough, solve them directly. Otherwise, apply the divide-and-conquer approach recursively to these sub-problems

   Once the files are divided into smaller subsets, each subset is organized recursively. For example, you could sort documents into subcategories such as reports, presentations, and spreadsheets. Each of these categories might be further divided into subfolders based on date or project. This recursive approach allows you to systematically manage and categorize each subset. For very small subsets, such as a single folder with a few files, a direct solution is applied without further division, making the problem-solving process more manageable.

3. **Combine**: Combine the solutions of the sub-problems to form the solution to the original problem.

After each subset of files is organized, the final step is to combine these organized subsets into the overall directory system. This involves merging the categorized folders into a hierarchical structure that reflects the original file organization plan. The result is a complete and well-structured directory system that maintains the overall organization and makes it easy to locate and manage files. The combination of the organized subsets ensures that the entire file system is coherent and functional, achieving the goal of efficient file organization.

By applying these steps, the complex task of organizing a large set of files is broken down into manageable steps, leveraging recursion to handle each subset and integrating the results into a comprehensive directory system. This section will explore the fundamental principles of the divide-and-conquer strategy, its applications, and its advantages and disadvantages.

13.2.1 Principles of Divide-and-Conquer

13.2.1.1 Divide

The first step involves breaking down the problem into smaller subproblems. This division should be done so that the subproblems are similar to the original problem. The key is to ensure that each subproblem is easier to solve than the original.

13.2.1.2 Conquer

Once the problem is divided, each subproblem is solved recursively. This step leverages the power of recursion, making the problem-solving process more manageable. For very small subproblems, a direct solution is applied without further division.

13.2.1.3 Combine

The final step involves combining the solutions of the subproblems to form the solution to the original problem. This step often requires merging results in a manner that maintains the problem's overall structure and requirements.

☞ Divide-and-conquer breaks a problem into smaller, manageable subproblems, solving each independently and combining their solutions to address the original problem.

Let us walk through an example to get an idea about how the principle of divide-and-conquer is applied to solve computational problems.

Example: Merge Sort Algorithm

Merge Sort is a classic example of the divide-and-conquer strategy used for sorting an array of elements. It operates by recursively breaking down the array into progressively smaller sections. The core idea is to split the array into two halves, sort each half, and then merge them back together. This process continues until the array is divided into individual elements, which are inherently sorted.

The merging process relies on a straightforward principle: when combining two sorted halves, the smallest value of the entire array must be the smallest value from either of the two halves. By iteratively comparing the smallest elements from each half and appending the smaller one to the sorted array, we efficiently merge the halves into a fully sorted array. This approach is not only intuitive but also simplifies the coding of the recursive splits and the merging procedure. Here is how it works:

1. **Divide**: Split the array into two halves.
2. **Conquer**: Recursively sort both halves.
3. **Combine**: Merge the two sorted halves to produce the sorted array.

To get an idea of how the Merge sort works, let us visualize the working of the algorithm. Visualizing the Merge Sort algorithm helps to understand how the divide-and-conquer approach works by breaking down the array into smaller parts and then merging them back together. Here's a step-by-step visualization of Merge Sort:

Visualization Steps:

1. **Divide**: The array is recursively divided into two halves until each sub-array contains a single element.
2. **Merge**: The sub-arrays are then merged in a sorted order.

Let us use an example array: [**38, 27, 43, 3, 9, 82, 10**].

Step 1: Divide the Array

1. **Initial Array**: [**38, 27, 43, 3, 9, 82, 10**]
2. **Divide into Halves**:
   - Left Half: [**38, 27, 43**]
   - Right Half: [**3, 9, 82, 10**]
3. **Further Divide**:
   - Left Half: [**38, 27, 43**] becomes:
     * [**38**] and [**27, 43**]
     * [**27, 43**] becomes:
       · [**27**] and [**43**]

- Right Half: [**3, 9, 82, 10**] becomes:
  - [**3, 9**] and [**82, 10**]
  - [**3, 9**] becomes:
    - [**3**] and [**9**]
  - [**82, 10**] becomes:
    - [**82**] and [**10**]

Step 2: Merge the Arrays

1. **Merge Smaller Arrays**:
   - [**27**] and [**43**] are merged to form [**27, 43**]
   - [**3**] and [**9**] are merged to form [**3, 9**]
   - [**82**] and [**10**] are merged to form [**10, 82**]
2. **Merge Larger Arrays**:
   - [**38**] and [**27, 43**] are merged to form [**27, 38, 43**]
   - [**3, 9**] and [**10, 82**] are merged to form [**3, 9, 10, 82**]
3. **Final Merge**:
   - [**27, 38, 43**] and [**3, 9, 10, 82**] are merged to form the sorted array [**3, 9, 10, 27, 38, 43, 82**]

This is how the merge sort algorithm works, breaking down the problem into smaller subproblems, solving each independently, and combining the results to get the final sorted array.

To visualize this process, here is a diagram representing the Merge Sort:
Explanation:

1. **Initial Split**: The array is divided into smaller chunks recursively.
2. **Recursive Sorting**: Each chunk is sorted individually.
3. **Merging**: The sorted chunks are merged back together in sorted order.

By following this visualization, you can see how the divide-and-conquer approach efficiently breaks down the problem and then builds up the solution step-by-step. This method ensures that each element is placed in its correct position in the final sorted array.

Here is a detailed explanation in English of the merge sort algorithm along with the merge function:

1. **Function mergeSort**
   - Check if the array has one or zero elements. If true, return the array as it is already sorted.
   - Otherwise, find the middle index of the array.
   - Split the array into two halves: from the beginning to the middle and from the middle to the end.
   - Recursively apply mergeSort to the first half and the second half.

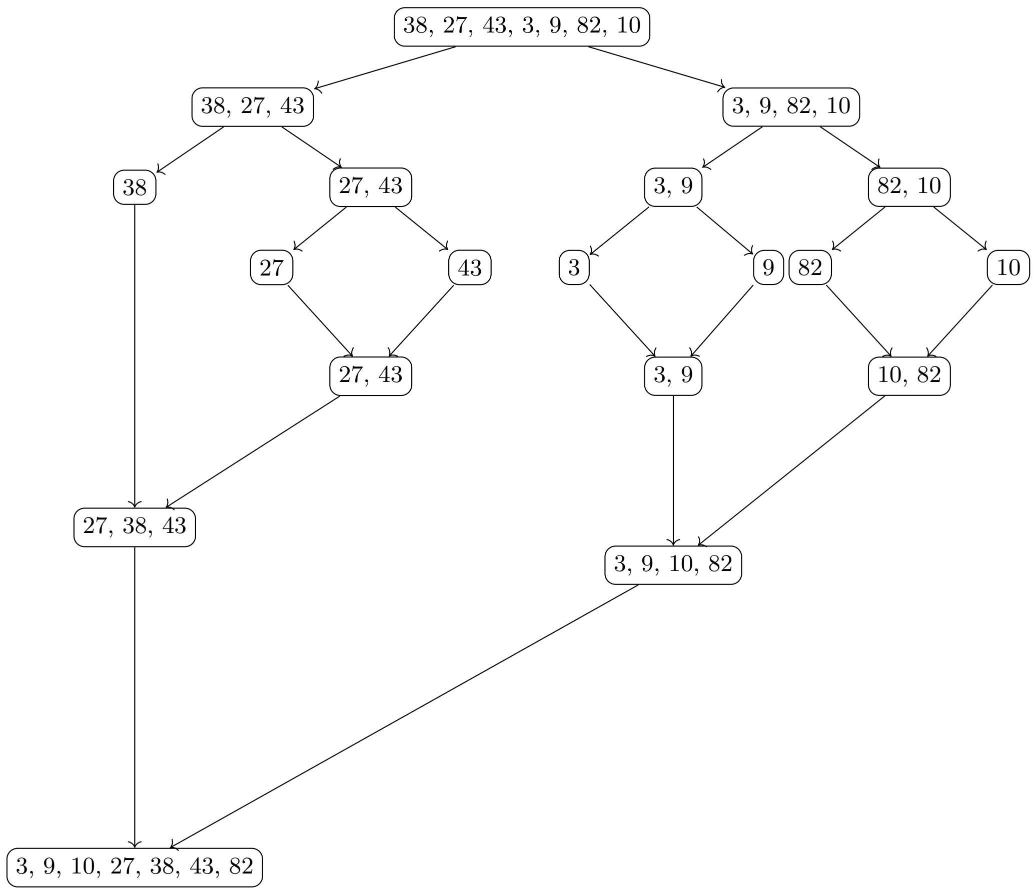


Figure 13.1: The recursion tree given when performing a recursive split of the array [**38, 27, 43, 3, 9, 82, 10**].

- Merge the two sorted halves using the merge function.
- Return the merged and sorted array.

2. **Function merge**

   - Create an empty list called **sorted_arr** to store the sorted elements.
   - While both halves have elements:
     - Compare the first element of the left half with the first element of the right half.
     - Remove the smaller element and append it to the **sorted_arr** list.
   - If the left half still has elements, append them all to the **sorted_arr** list.
   - If the right half still has elements, append them all to the **sorted_arr** list.
   - Return the **sorted_arr** list, which now contains the sorted elements from both halves.

Here is a Python implementation of Merge Sort

```python
def merge_sort(arr):
    """ Sorts an array using the merge sort algorithm."""
    if len(arr) <= 1:
        return arr

    # Divide the array into two halves
    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid])
    right_half = merge_sort(arr[mid:])

    # Combine the sorted halves
    return merge(left_half, right_half)

def merge(left, right):
    """ Merges two sorted arrays into one sorted array. """
    sorted_arr = []
    i = j = 0

    # Merge the two arrays
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            sorted_arr.append(left[i])
            i += 1
        else:
            sorted_arr.append(right[j])
            j += 1

    # Add remaining elements (if any)
    while i < len(left):
        sorted_arr.append(left[i])
        i += 1
    while j < len(right):
        sorted_arr.append(right[j])
        j += 1

    return sorted_arr

# Example usage
array = [38, 27, 43, 3, 9, 82, 10]
sorted_array = merge_sort(array)
print("Sorted array:", sorted_array)
```

Explanation:

1. **merge_sort Function**:
   - If the array has one or zero elements, it's already sorted.
   - The array is divided into two halves recursively until the base case is reached (arrays of size one).
   - The sorted halves are combined using the merge function.
2. **merge Function**:
   - Takes two sorted arrays and merges them into one sorted array.
   - Compares elements from both arrays and appends the smaller element to the result array.
   - After one array is exhausted, append any remaining elements from the other array.

13.2.2 Solving Computational Problems Using Divide and Conquer Approach

Let us see how we can apply the principles of the divide-and-conquer approach to solve computational problems.

13.2.2.1 Problem-1 (Finding the Maximum Element in an Array)

Given an array of integers, find the maximum value in the array.

Step-by-Step Solution

1. **Initial Setup:**

- Begin with the entire array and determine the range to process. Initially, this range includes the entire array from the first element to the last element.

2. **Divide**

   - If the array contains more than one element, split it into two approximately equal halves. This splitting continues recursively until each subarray has only one element.

3. **Conquer:**

   - For subarrays with only one element, that element is trivially the maximum for that subarray.
   - For larger subarrays, recursively apply the same process to each half of the subarray.

4. **Combine:**

   - After finding the maximum element in each of the smaller subarrays, combine the results by comparing the maximum values from each half. Return the largest of these values as the maximum for the original array.

Python Implementation

Here is how to implement this algorithm in Python

```
def find_max(arr, left, right):
    # Base case: If the array segment has only one element
    if left == right:
       return arr[left]

    # Divide: Find the middle point of the current segment
    mid = (left + right) // 2

    """ Conquer: Recursively find the maximum in the left and
    right halves """

    max_left = find_max(arr, left, mid)  # Maximum in the left
    half
    max_right = find_max(arr, mid + 1, right)  # Maximum in the
    right half

    # Combine: Return the maximum of the two halves

    return max(max_left, max_right)
```

```python
# Example usage
array = [3, 6, 2, 8, 7, 5, 1]
result = find_max(array, 0, len(array) - 1)
print("Maximum element:", result)  # Output: Maximum element: 8
```

Step-by-Step Explanation:

1. **Initial Setup:**
   - The **find_max** function is called with the entire array and indices that cover the whole array. For example, **find_max(array, 0, len(array) - 1)**.
2. **Divide:**
   - Calculate the middle index **mid** of the current array segment. For the array [**3, 6, 2, 8, 7, 5, 1**], **mid** would be **(0 + 6) // 2 = 3**.
   - Split the array into two halves based on this **mid** index
     - Left half: [**3, 6, 2, 8**]
     - Right half: [**7, 5, 1**]
3. **Conquer:**
   - Recursively apply the **find_max** function to the left half [**3, 6, 2, 8**]:
     - Split into [**3, 6**] and [**2, 8**]
     - Further split [**3, 6**] into [**3**] and [**6**], finding **3** and **6**, respectively.
     - Combine these to get the maximum **6**.
     - Similarly, split [**2, 8**] into [**2**] and [**8**], finding **2** and **8**, respectively.
     - Combine these to get the maximum **8**.
     - Combine **6** and **8** from the two halves to get **8**.
   - Apply the same process to the right half [**7, 5, 1**]:
     - Split into [**7**] and [**5, 1**]
     - Further split [**5, 1**] into [**5**] and [**1**], finding **5** and **1**, respectively.
     - Combine these to get the maximum **5**.
     - Combine **7** and **5** from the two halves to get **7**.
4. **Combine:**
   - Finally, compare the maximums obtained from the left half **(8)** and the right half **(7)**.
   - Return the larger value, which is **8**.

This method efficiently finds the maximum element in the array by recursively dividing the problem, solving the subproblems, and combining the results.

13.2.2.2 Problem-2 (Finding the Maximum Subarray Sum)

Given an array of integers, which include both positive and negative numbers, find the contiguous subarray that has the maximum sum.

Step-by-Step Solution

1. **Initial Setup:**
   - If the array contains only one element, that element is the maximum subarray sum.
2. **Divide:**
   - Divide the array into two approximately equal halves. This involves finding the middle index of the array and separating the array into a left half and a right half.
3. **Conquer:**
   - Recursively find the maximum subarray sum for:
     - The left half of the array.
     - The right half of the array.
     - The subarray that crosses the middle boundary between the two halves.
4. **Combine:**
   - Combine the results from the three areas:
     - The maximum subarray sum in the left half.
     - The maximum subarray sum in the right half.
     - The maximum subarray sum that crosses the middle.
   - Return the largest of these three values.

Python Implementation

Here is the Python code implementing this algorithm

```
def max_crossing_sum(arr, left, mid, right):
# Find maximum sum of subarray crossing the middle point

# Start from mid and move left

    left_sum = float('-inf')
    sum_left = 0
    for i in range(mid, left - 1, -1):
        sum_left += arr[i]
    if sum_left > left_sum:
        left_sum = sum_left

    # Start from mid+1 and move right

    right_sum = float('-inf')
    sum_right = 0
    for i in range(mid + 1, right + 1):
        sum_right += arr[i]
    if sum_right > right_sum:
        right_sum = sum_right

    """ Return the maximum sum of the subarray that crosses the
    mid """

    return left_sum + right_sum

def max_subarray_sum(arr, left, right):
        # Base case: only one element
    if left == right:
        return arr[left]

    # Divide the array into two halves
    mid = (left + right) // 2

    # Find maximum subarray sum in the left half
    left_max = max_subarray_sum(arr, left, mid)

    # Find maximum subarray sum in the right half
    right_max = max_subarray_sum(arr, mid + 1, right)

    # Find maximum subarray sum crossing the middle
    cross_max = max_crossing_sum(arr, left, mid, right)

    # Return the maximum of the three results
    return max(left_max, right_max, cross_max)

# Example usage
array = [2, 3, -4, 5, -1, 2, 3, -2, 4]
result = max_subarray_sum(array, 0, len(array) - 1)
print("Maximum subarray sum:", result)  # Output: Maximum
    subarray sum: 9
```

Step-by-Step Explanation:

1. **Initial Setup:**

    - If the array segment contains only one element (base case), return that element as the maximum subarray sum.

2. **Divide:**
   - Calculate the middle index: **mid = (left + right) // 2**.
   - Divide the array into two subarrays:
     - Left subarray from **left** to **mid**.
     - Right subarray from **mid + 1** to **right**.
3. **Conquer:**
   - **Find Maximum Subarray Sum in the Left Half:**
     - Recursively call **max_subarray_sum** on the left half.
   - **Find Maximum Subarray Sum in the Right Half:**
     - Recursively call **max_subarray_sum** on the right half.
   - **Find Maximum Subarray Sum Crossing the Middle:**
     - Call **max_crossing_sum** to find the maximum subarray sum that crosses the midpoint of the array. This involves calculating the maximum sum of the subarray that ends in the left half and starts in the right half.
4. **Combine:**
   - The result is the maximum of:
     - The maximum sum found in the left half.
     - The maximum sum found in the right half.
     - The maximum sum of the subarray crossing the middle.

Example Walkthrough:

Array: [2, 3, -4, 5, -1, 2, 3, -2, 4]

1. **Initial Setup:**
   - The array is divided into subarrays until base cases with single elements are reached.
2. **Divide:**
   - For the array [**2, 3, -4, 5, -1, 2, 3, -2, 4**], **mid = 4**, so divide into:
     - Left half: [**2, 3, -4, 5, -1**]
     - Right half: [**2, 3, -2, 4**]
3. **Conquer:**
   - **Left Half [2, 3, -4, 5, -1]:**
     - Further divide into [**2, 3**] and [**-4, 5, -1**].

  - Find maximum subarray sums for these segments and combine them.

- **Right Half [2, 3, -2, 4]:**
  - Further divide into [**2, 3**] and [**-2, 4**].
  - Find maximum subarray sums for these segments and combine them.
- **Crossing Subarray:**
  - Calculate the maximum sum of subarrays crossing the middle index for both halves.

4. **Combine:**
   - Compare the maximum subarray sums from the left half, right half, and crossing subarray to get the overall maximum.

Explanation of Crossing Subarray

Consider the array: [**2, 3, -4, 5, -1, 2, 3, -2, 4**]

Let us find the maximum subarray sum that crosses the midpoint of the array.

Step-by-Step Explanation:

1. **Divide the Array:**
   - Suppose we are working with the whole array and the midpoint is calculated to be **4**. So we divide the array into two halves:
     - Left half: [**2, 3, -4, 5, -1**]
     - Right half: [**2, 3, -2, 4**]
   - Our goal is to find the maximum sum of subarrays that might cross this midpoint between the two halves.
2. **Find the Maximum Crossing Subarray:**
   - To find the maximum subarray sum that crosses the midpoint, we need to consider two parts:
     - The part of the subarray that extends from the midpoint to the left end
     - The part of the subarray that extends from the midpoint to the right end.
   - **Calculate the Maximum Sum of the Left Part:**
     - Start from the midpoint and extend leftward.
     - Track the maximum sum while extending leftward from the midpoint.

  - In the example, the midpoint is **4**, so we start from index **4** and move left.

    Array segment: [**-1, 5, -4, 3, 2**]
  - Compute maximum subarray sum ending at the midpoint:
    * Start from **-1**, sum is **-1**.
    * Extend to include **5**: sum becomes **4**.
    * Extend to include **-4**: sum becomes **0**.
    * Extend to include **3**: sum becomes **3**.
    * Extend to include **2**: sum becomes **5**.

    The maximum subarray sum ending at the midpoint (and extending leftward) is **5**.

- **Calculate the Maximum Sum of the Right Part:**
  - Start from the midpoint + 1 and extend rightward.
  - Track the maximum sum while extending rightward from the midpoint

    Array segment: [**2, 3, -2, 4**]
  - Compute maximum subarray sum starting from the midpoint + 1:
    * Start from **2**, sum is **2**.
    * Extend to include **3**: sum becomes **5**.
    * Extend to include **-2**: sum becomes **3**.
    * Extend to include **4**: sum becomes **7**.

    The maximum subarray sum starting at midpoint + 1 (and extending rightward) is **7**.

- **Combine the Results:**
  - The maximum sum of the subarray that crosses the midpoint is the sum of the maximum sum of the left part and the maximum sum of the right part.
  - From the above calculations, the maximum crossing sum is **5 (left) + 7 (right) = 12**.

The crossing subarray that yields the maximum sum spans the midpoint and consists of elements from both the left and right parts. By combining the best sums of subarrays extending from the midpoint, we get the total maximum crossing sum.

This approach efficiently computes the maximum subarray sum using the divide and conquer strategy, which can be particularly useful for larger arrays due to its manageable number of comparisons.

☞ By recursively dividing a complex problem into simpler components, the divide-and-conquer approach transforms large-scale challenges into solvable tasks with a clear, structured strategy.

13.2.3 Advantages and Disadvantages of Divide and Conquer Approach

Advantages of Divide and Conquer Approach

1. **Simplicity in Problem Solving**: By breaking a problem into smaller subproblems, each subproblem is simpler to understand and solve, making the overall problem more manageable.

2. **Efficiency**: Many divide-and-conquer algorithms, such as merge sort and quicksort, have optimal or near-optimal time complexities. These algorithms often have lower time complexities compared to iterative approaches.

3. **Modularity**: Divide-and-conquer promotes a modular approach to problem-solving, where each subproblem can be handled by a separate function or module. This makes the code easier to maintain and extend.

4. **Reduction in Complexity**: By dividing the problem, the overall complexity is reduced, and solving smaller subproblems can lead to simpler and more efficient solutions.

5. **Parallelism**: The divide-and-conquer approach can easily be parallelized because the subproblems can be solved independently and simultaneously on different processors, leading to potential performance improvements.

6. **Better Use of Memory**: Some divide-and-conquer algorithms use memory more efficiently. For example, the merge sort algorithm works well with large data sets that do not fit into memory, as it can process subsets of data in chunks.

Disadvantages of Divide and Conquer Approach

1. **Overhead of Recursive Calls**: The recursive nature can lead to significant overhead due to function calls and maintaining the call stack. This can be a problem for algorithms with deep recursion or large subproblem sizes.

2. **Increased Memory Usage**: Divide-and-conquer algorithms often require additional memory for storing intermediate results, which can be a drawback for memory-constrained environments.

3. **Complexity of Merging Results**: The merging step can be complex and may not always be straightforward. Efficient merging often requires additional algorithms and can add to the complexity of the overall solution.

4. **Not Always the Most Efficient**: For some problems, divide-and-conquer might not be the most efficient approach compared to iterative or dynamic programming methods. The choice of strategy depends on the specific problem and context.

5. **Difficulty in Implementation**: Implementing divide-and-conquer algorithms can be more challenging, especially for beginners. The recursive nature and merging steps require careful design to ensure correctness and efficiency.

6. **Stack Overflow Risk**: Deep recursion can lead to stack overflow errors if the recursion depth exceeds the system's stack capacity, particularly with large inputs or poorly designed algorithms.

☞ The power of divide-and-conquer lies in its efficiency; it often reduces the problem's complexity by tackling smaller parts, which can lead to significant performance improvements.

The divide-and-conquer approach is a versatile and powerful problem-solving strategy that breaks down complex problems into simpler subproblems. Its applications span various fields, including sorting, searching, and computational geometry. While it offers significant advantages in terms of efficiency and simplicity, it also comes with challenges such as recursion overhead and merge step complexity. Understanding and mastering this technique is essential for tackling a wide range of algorithmic problems.

13.3 Dynamic Programming Approach to Problem Solving

"Dynamic programming is a method for solving complex problems by breaking them down into simpler subproblems. It is a way of combining solutions to overlapping subproblems to avoid redundant calculations."

– Richard Bellman

Imagine you need to travel by taxi from Thiruvananthapuram to Ernakulam. You have several possible routes through different cities, and your goal is to find the one that gets you to Ernakulam in the shortest time, which you call the optimal route. Your cousin in Alappuzha knows the best route from Alappuzha to Ernakulam. Given that any optimal route from Thiruvananthapuram must pass through Alappuzha, you can simplify your task by first finding the best route from Thiruvananthapuram to Alappuzha, which is closer and has fewer possible routes. Once you have this route, you can follow the one your cousin has suggested from Alappuzha to Ernakulam.

This approach is based on the principle of optimality, which states that any optimal route from Thiruvananthapuram to Ernakulam via Alappuzha must have optimal sub-routes. Specifically, the segment from Alappuzha to Ernakulam must be the best route between these two cities, and the segment from Thiruvananthapuram to Alappuzha must also be optimal. More generally, if you have an optimal route consisting of cities $\mathbf{C_1}, \mathbf{C_2}, \ldots, \mathbf{C_p}$, then each segment of this route (from $\mathbf{C_1}$ to $\mathbf{C_2}$, $\mathbf{C_2}$ to $\mathbf{C_3}$, etc.) must be optimal on its own. By solving the problem in smaller parts—finding the best route from Thiruvananthapuram to Alappuzha and using the known optimal route from Alappuzha to Ernakulam—you can effectively solve the larger problem. This principle, known as Bellman's principle of optimality, was developed by Richard Bellman in the late 1940s and applies to various optimization problems beyond travel routes. In this section, we will explore how dynamic programming leverages this principle to tackle complex optimization challenges.

Dynamic programming (DP) is a method for solving problems by breaking them down into smaller overlapping subproblems, solving each subproblem just once, and storing their solutions. It is particularly useful for optimization problems where the problem can be divided into simpler subproblems that are solved independently and combined to form a solution to the original problem.

Dynamic Programming was first introduced by Richard Bellman in the 1950s as part of his research in operations research and control theory. In this context, the term "programming" does not relate to coding but refers to the process of optimizing a series of decisions. Bellman chose the term "dynamic programming" to avoid confusion and political issues, as "programming" was strongly associated with computers at the time.

At its core, Dynamic Programming (DP) involves breaking down a problem into smaller, more manageable subproblems and storing the solutions to these subproblems for future use. This approach is particularly effective for problems that exhibit two key properties: optimal substructure and overlapping subproblems.

1. **Optimal Substructure**: A problem has optimal substructure if the best solution to the overall problem can be constructed from the best solutions to its smaller subproblems. This means that if you have the optimal solutions for the smaller components of the problem, you can combine them to find the best solution for the entire problem. This property allows Dynamic Programming to build solutions incrementally, using previously computed results to achieve the most efficient outcome.

   Example: Shortest Path in a Grid

   Imagine you need to find the shortest path from the top-left corner to the bottom-right corner of a grid. You can only move right or down. Each cell in the grid has a certain cost associated with entering it, and your goal is to minimize the total cost of the path.

Problem Breakdown:

(a) **Smaller Subproblems**: To find the shortest path to a particular cell (i, j), you can look at the shortest paths to the cells immediately above it $(i - 1, j)$ and to the left of it $(i, j - 1)$. The cost to reach cell (i, j) will be the minimum of the costs to reach these neighboring cells plus the cost of the current cell.

(b) **Optimal Substructure**: If you know the shortest paths to cells $(i - 1, j)$ and $(i, j - 1)$, you can use these to determine the shortest path to cell (i, j). The optimal path to cell (i, j) can be constructed from the optimal paths to its neighboring cells.

How it Works:

- You start by solving the problem for the smallest subproblems (the cells directly above and to the left).
- You then build up solutions incrementally, using the results of the smaller subproblems to solve larger parts of the grid.
- Finally, you combine the results to find the shortest path to the bottom-right corner of the grid.

This approach ensures that you are using the most efficient solutions to smaller problems to construct the best solution for the entire grid.

2. **Overlapping Subproblems**: Many problems require solving the same subproblems multiple times. Dynamic Programming improves efficiency by storing the results of these subproblems in a table to avoid redundant calculations. By caching these results, the algorithm reduces the number of computations needed, leading to significant performance improvements.

☞ Dynamic programming breaks problems down into overlapping subproblems, storing solutions to avoid redundant calculations.

Example: Fibonacci Sequence

In the Fibonacci sequence, each number is the sum of the two preceding ones. For example, to find **Fibonacci(5)**, you need the values of **Fibonacci(4)** and **Fibonacci(3)**. To compute **Fibonacci(4)**, you need **Fibonacci(3)** and **Fibonacci(2)**. Notice that **Fibonacci(3)** is computed multiple times when calculating different Fibonacci numbers.

Without Dynamic Programming:

- To compute **Fibonacci(5)**, you might end up calculating **Fibonacci(3)** twice.

- This redundancy leads to a lot of repeated work.

With Dynamic Programming:

- You compute **Fibonacci(3)** once and store its result.
- When you need **Fibonacci(3)** again, you retrieve the stored result instead of recalculating it.
- This caching of results avoids redundant calculations and speeds up the process.

How it Works:

(a) **Compute**: Calculate the Fibonacci numbers and store them in an array.

(b) **Reuse**: Whenever you need the value of a Fibonacci number that has already been computed, look it up in the array instead of recalculating.

By storing results and reusing them, Dynamic Programming reduces the number of calculations needed to solve the problem, leading to significant performance improvements.

☞ By caching intermediate results, dynamic programming transforms exponential time complexity into polynomial time.

13.3.1 Comparison with Other Problem-solving Techniques

Dynamic Programming shares similarities with other problem-solving techniques like Divide and Conquer and Greedy Algorithms, but it has unique characteristics that set it apart:

Divide and Conquer: Both techniques break problems into smaller subproblems. However, Divide and Conquer solves each subproblem independently, often without considering if the same subproblems are solved multiple times. In contrast, Dynamic Programming stores and reuses solutions to overlapping subproblems, which improves performance by avoiding redundant calculations.

Greedy Algorithms: Greedy algorithms make a series of locally optimal choices with the hope of finding the global optimum. They are typically simpler to implement but may not always yield the best overall solution. Dynamic Programming, on the other hand, guarantees an optimal solution by evaluating all possible choices and storing the best solutions for each subproblem, ensuring the most efficient overall result.

13.3.2 Fundamental Principles of Dynamic Programming

In this section, we will explore the fundamental principles that make Dynamic Programming an effective problem-solving technique, focusing on overlapping subproblems, optimal substructure, and the two primary approaches: memoization and tabulation.

Overlapping Subproblems: Dynamic Programming is particularly useful for problems with overlapping subproblems. This means that when solving a larger problem, you encounter smaller subproblems that are repeated multiple times. Instead of recomputing these subproblems each time they are encountered, Dynamic Programming saves their solutions in a data structure, such as an array or hash table. This avoids redundant calculations and significantly improves efficiency.

For example, in a recursive approach to solving a problem, the same function might be called multiple times with the same arguments. Without Dynamic Programming, this leads to wasted time as the same subproblems are recalculated repeatedly. By using Dynamic Programming, the solutions to these subproblems are stored once computed, which optimizes overall algorithm efficiency.

Optimal Substructure: Another key principle of Dynamic Programming is optimal substructure. This property means that an optimal solution to the larger problem can be constructed from the optimal solutions to its smaller subproblems. In other words, if you can determine the best solution for smaller problems, you can use these solutions to build the best solution for the entire problem.

Optimal substructure is central to Dynamic Programming's recursive nature. By solving subproblems optimally and combining their solutions, you ensure that the final solution is also optimal.

☞ Optimal substructure is the key to dynamic programming, where the global solution can be constructed from optimal solutions of smaller subproblems.

13.3.3 Approaches in Dynamic Programming

Dynamic Programming can be implemented using two main approaches: memoization (top-down) and tabulation (bottom-up).

13.3.3.1 Memoization (Top-Down Approach)

Memoization involves solving the problem recursively and storing the results of subproblems in a table (usually a dictionary or array). This way, each subproblem is solved only once, and subsequent calls to the subproblem are served from the stored results.

Steps:

1. Identify the base cases.
2. Define the recursive relation.
3. Store the results of subproblems in a table.
4. Use the stored results to solve larger subproblems.

Example: Fibonacci sequence using memoization

```
def fib(n, memo={}):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fib(n-1, memo) + fib(n-2, memo)
    return memo[n]
```

Code Explanation: Fibonacci Sequence Using Memoization:

The given code defines a function **fib** that calculates the **n**-th Fibonacci number using a technique called memoization. Memoization is a method used to optimize recursive algorithms by storing the results of expensive function calls and reusing them when the same inputs occur again.

Here is a detailed explanation of each part of the code:

```
def fib(n, memo={}):
```

- The function **fib** takes two arguments:
  - **n**: The position in the Fibonacci sequence for which we want to find the Fibonacci number.
  - **memo**: A dictionary used to store previously computed Fibonacci numbers. It defaults to an empty dictionary if not provided.

```
if n in memo:
    return memo[n]
```

  - This line checks if the Fibonacci number for the given **n** has already been computed and stored in the memo dictionary.

  - If it has, the function immediately returns the stored value, avoiding redundant calculations.

```
if n <= 1:
      return n
```

  - This line handles the base cases of the Fibonacci sequence.
    * If **n** is **0** or **1**, the function returns **n** because the Fibonacci sequence is defined as **fib(0) = 0** and **fib(1) = 1**.

```
memo[n] = fib(n-1, memo) + fib(n-2, memo)
```

  - This line computes the Fibonacci number for **n** by recursively calling fib for **n-1** and **n-2**.
  - The results of these recursive calls are added together to get the Fibonacci number for **n**.
  - The computed Fibonacci number is then stored in the **memo** dictionary to avoid redundant calculations in future calls.

```
return memo[n]
```

  - This line returns the Fibonacci number for **n** that was just computed and stored in the **memo** dictionary.

The **fib** function leverages memoization to optimize the calculation of Fibonacci numbers by storing the results of previously computed numbers in a dictionary. This approach significantly reduces the time complexity of the algorithm from exponential to linear by avoiding redundant calculations.

Memoization is often easier to implement and understand. It starts with the original problem and solves subproblems as needed. However, it may have overhead due to recursive function calls and may not be as efficient for some problems.

13.3.3.2 Tabulation (Bottom-Up Approach)

Tabulation involves solving the problem iteratively and filling up a table (usually an array) in a bottom-up manner. This approach starts with the smallest subproblems and uses their solutions to construct solutions to larger subproblems.

Steps:

1. Identify the base cases.
2. Define the table to store solutions to subproblems.
3. Fill the table iteratively using the recursive relation.
4. Extract the solution to the original problem from the table.

Example: Fibonacci sequence using tabulation

```
def fib(n):
    if n <= 1:
        return n
    dp = [0] * (n + 1)
    dp[1] = 1
    for i in range(2, n + 1):
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]
```

Code Explanation: Fibonacci Sequence Using Table:

The given code defines a function **fib** that calculates the **n**-th Fibonacci number using a tabular approach. Dynamic programming is a method for solving problems by breaking them down into simpler subproblems and storing the solutions to these subproblems in a table to avoid redundant calculations.

Here is a detailed explanation of each part of the code:

```
def fib(n):
```

- The function **fib** takes a single argument **n**, which represents the position in the Fibonacci sequence for which we want to find the Fibonacci number.

```
if n <= 1:
    return n
```

- This line handles the base cases of the Fibonacci sequence.
  * If **n** is **0** or **1**, the function returns **n** because the Fibonacci sequence is defined as $\mathbf{fib(0) = 0}$ and $\mathbf{fib(1) = 1}$.

```
dp = [0] * (n + 1)
```

- This line initializes a list **dp** of length **n+1** with all elements set to **0**.
- The list **dp** will be used to store the Fibonacci numbers for each position from **0** to **n**.

```
dp[1] = 1
```

- This line sets the second element of the list **dp** to **1**, which corresponds to **fib(1) = 1**.

```
for i in range(2, n + 1):
    dp[i] = dp[i-1] + dp[i-2]
```

- This loop iterates over the range from **2** to **n** (inclusive).
- For each **i** in this range, the Fibonacci number at position **i** is calculated by adding the Fibonacci numbers at positions **i-1** and **i-2**.
- The result is stored in **dp[i]**.

```
return dp[n]
```

- This line returns the Fibonacci number for the given **n**, which is stored in **dp[n]**.

This approach reduces the time complexity and the space complexity, making it much more efficient than the naive recursive approach.

Tabulation tends to be more memory-efficient and can be faster than memoization due to its iterative nature. However, it requires careful planning to set up the data structures and dependencies correctly.

☞ The core strength of dynamic programming lies in turning recursive problems into iterative solutions by reusing past work.

13.3.4 Solving Computational Problems Using Dynamic Programming Approach

Here is a step-by-step guide on how to solve computational problems using the dynamic programming approach:

1. **Identify the Subproblems**: Break down the problem into smaller subproblems. Determine what the subproblems are and how they can be combined to solve the original problem.

2. **Define the Recurrence Relation**: Express the solution to the problem in terms of the solutions to smaller subproblems. This usually involves finding a recursive formula that relates the solution of a problem to the solutions of its subproblems.

3. **Choose a Memoization or Tabulation Strategy**: Decide whether to use a top-down approach with memoization or a bottom-up approach with tabulation.

   - **Memoization (Top-Down)**: Solve the problem recursively and store the results of subproblems in a table (or dictionary) to avoid redundant computations.
   - **Tabulation (Bottom-Up)**: Solve the problem iteratively, starting with the smallest subproblems and building up the solution to the original problem.

4. **Implement the Solution**: Write the code to implement the dynamic programming approach, making sure to handle base cases and use the table to store and retrieve the results of subproblems.

5. **Optimize Space Complexity (if necessary)**: Sometimes, it is possible to optimize space complexity by using less memory. For example, if only a few previous states are needed to compute the current state, you can reduce the size of the table.

Let us see how we can apply the dynamic programming approach to solve a computational problem.

13.3.4.1 Problem-1 (The Knapsack Problem)

The knapsack problem is a classical example of a problem that can be solved using dynamic programming. The problem is defined as follows:

Given weights and values of **n** items, put these items in a knapsack of capacity **W** to get the maximum total value in the knapsack. Each item can only be taken once.

Consider the following example:

- Capacity of the knapsack $W = 50$
- Number of items $n = 3$
- Weights of the items: $w = [10, 20, 30]$
- Values of the items: $v = [60, 100, 120]$

We want to find the maximum value we can carry in the knapsack. For this example, the maximum value we can carry in the knapsack of capacity 50 is 220.

Now we discuss how to apply the dynamic programming approach to solve the Knapsack problem.

Step-by-Step Solution

1. **Define the Subproblems**: The subproblem in this case is finding the maximum value for a given knapsack capacity **w** using the first **i** items. Let us define **dp**[**i**][**w**] as the maximum value that can be obtained with a knapsack capacity **w** using the first **i** items.

2. **Recurrence Relation**: For each item **i**, you have two choices:

   - **Do not include the item i in the knapsack**: The maximum value is the same as without this item, which is **dp**[**i-1**][**w**].
   - **Include the item i in the knapsack**: The maximum value is the value of this item plus the maximum value of the remaining capacity, which is **values**[**i-1**] + **dp**[**i-1**][**w-weights**[**i-1**]] (only if **weights**[**i-1**] $\leq$ **w**).
     The recurrence relation is:

     $$\mathbf{dp}[\mathbf{i}][\mathbf{w}] = \max\{\mathbf{dp}[\mathbf{i}-\mathbf{1}][\mathbf{w}], \mathbf{values}[\mathbf{i}-\mathbf{1}] + \mathbf{dp}[\mathbf{i}-\mathbf{1}][\mathbf{w}-\mathbf{weights}[\mathbf{i}-\mathbf{1}]\}$$

3. **Base Case**: If there are no items or the capacity is zero, the maximum value is zero:

   $$\mathbf{dp}[\mathbf{i}][\mathbf{0}] = \mathbf{0} \quad \textbf{for all i}$$
   $$\mathbf{dp}[\mathbf{0}][\mathbf{w}] = \mathbf{0} \quad \textbf{for all w}$$

4. **Tabulation**: We use a 2D array **d**p where **dp**[**i**][**w**] represents the maximum value obtainable using the first **i** items and capacity **w**.

Here is the dynamic programming algorithm solution in Python:

```
def knapsack(W, weights, values, n):
    """ Create a 2D array to store the maximum value for each
    subproblem. """
    dp = [[0 for _ in range(W + 1)] for _ in range(n + 1)]

    # Build the table in a bottom-up manner
    for i in range(n + 1):
       for w in range(W + 1):
           if i == 0 or w == 0:
               dp[i][w] = 0
           elif weights[i-1] <= w:
               dp[i][w] = max(values[i-1] +
    dp[i-1][w-weights[i-1]], dp[i-1][w])
           else:
               dp[i][w] = dp[i-1][w]
    """ The maximum value that can be obtained with the given
    capacity is in dp[n][W] """

    return dp[n][W]

    # Example usage:
    values = [60, 100, 120]
    weights = [10, 20, 30]
    W = 50
    n = len(values)
    print(knapsack(W, weights, values, n))  # Output: 220
```

Explanation:

1. **Initialization**: We initialize a 2D list **dp** with dimensions **(n+1) x (W+1)**, where **dp[i][w]** represents the maximum value achievable with the first **i** items and a knapsack capacity **w**. Initially, all values are set to **0**.

2. **Filling the DP Table**:

   - We iterate through each item **i** (from **0** to **n**).
   - For each item, we iterate through each capacity **w** (from **0** to **W**).
   - If the current item can be included in the knapsack (**weights[i-1]** $\leq$w), we calculate the maximum value by either including or excluding the item.
   - If the current item cannot be included, the maximum value is the same as without this item.

3. **Result**: The maximum value obtainable with the given knapsack capacity is stored in **dp[n][W]**.

Let us walk through an example with **values = [60, 100, 120], weights = [10, 20, 30]**, and **W = 50**.

- Initialize the **dp** table with dimensions **4** $\times$ **51** (all values set to **0**).
- Iterate through each item and each capacity, updating the **dp** table according to the recurrence relation.
- The final **dp** table will contain the maximum values for each subproblem.
- The value in **dp[3][50]** will be the maximum value obtainable, which is 220.

The time complexity of the knapsack algorithm is ($(\mathbf{n} + \mathbf{1}) \times (\mathbf{W}+\mathbf{1})$, where **n** is the number of items and **W** is the maximum weight capacity of the knapsack.

This is because we use a 2D array **dp** with dimensions **(n+1)**× **(W+1)**, and we fill this table by iterating through each item (from **0** to **n**) and each possible weight (from **0** to **W**). The space complexity of the algorithm is also **(n + 1)** × **(W + 1)** due to the 2D array **dp** used to store the maximum value for each subproblem.

By using dynamic programming, we reduce the exponential time complexity of the naive recursive solution to a polynomial (pseudo-polynomial to be more correct - The dynamic programming solution is indeed linear in the value of **W**, but exponential in the length of **W**) time complexity, making it feasible to solve larger instances of the problem. Also, by following these steps and principles, you can effectively use the dynamic programming approach to solve the knapsack problem and other similar computational problems.

13.3.5 Examples of Dynamic Programming

In this section, we will examine some classic examples of problems that can be effectively addressed using Dynamic Programming. These examples provide foundational insights into how this powerful problem-solving technique is applied in practice.

Longest Common Subsequence (LCS): The Longest Common Subsequence (LCS) problem is a fundamental string comparison challenge that identifies the longest sequence common to two or more strings. Unlike a substring, a subsequence maintains the order of characters but does not need to be contiguous.

Brute-force solutions to the LCS problem involve examining all possible subsequences to determine the longest common one, which is computationally expensive and impractical for longer strings due to its exponential time complexity. Dynamic Programming offers a more efficient approach by dividing the problem into smaller subproblems and using memoization or tabulation to store intermediate results.

Rod Cutting Problem: The Rod Cutting problem is a classic optimization problem, relevant in fields such as manufacturing and finance. Given a rod of length n and a price table for various lengths, the goal is to determine the maximum revenue achievable by cutting the rod into pieces and selling them.

A brute-force approach involves evaluating all possible cutting combinations and calculating the revenue for each, which becomes infeasible for longer rods due to its high complexity. Dynamic Programming addresses this issue by breaking the problem into smaller subproblems and using memoization or tabulation to find the optimal solution efficiently.

☞ Dynamic programming shines in problems where brute force would be inefficient, allowing us to solve complex problems more systematically.

These classic examples showcase the effectiveness of Dynamic Programming. By leveraging the concepts of overlapping subproblems and optimal substructure, Dynamic Programming provides efficient solutions to problems that would otherwise be computationally prohibitive. Its practical applications extend across various fields, making it an invaluable technique for programmers.

13.3.6 Advantages and Disadvantages of the Dynamic Programming Approach

Advantages of the Dynamic Programming Approach

1. **Efficiency**: DP reduces the time complexity of problems with overlapping subproblems by storing solutions to subproblems and reusing them.
2. **Optimal Solutions**: DP ensures that the solution to the problem is optimal by solving each subproblem optimally and combining their solutions.
3. **Versatility**: DP can be applied to a wide range of problems across different domains.

Disadvantages of the Dynamic Programming Approach

1. **Space Complexity**: DP often requires additional memory to store the results of subproblems, which can be a limitation for problems with a large number of subproblems.
2. **Complexity of Formulation**: Developing a DP solution requires a deep understanding of the problem's structure and properties, which can be challenging.
3. **Overhead of Table Management**: Managing and maintaining the DP table or memoization structure can add overhead to the algorithm.

☞ Dynamic programming is not just a technique; it is a framework for efficiently solving problems with a recursive structure.

Dynamic programming is a powerful technique for solving problems with overlapping subproblems and optimal substructure. By breaking down problems into simpler subproblems and storing their solutions, DP achieves efficiency and guarantees optimal solutions. Despite its complexity and memory requirements, DP's versatility and effectiveness make it an essential tool in algorithm design.

13.4 Greedy Approach to Problem Solving

In the last two sections, we explored dynamic programming and divide-and-conquer methods. All these approaches aim to simplify complex problems by breaking them into smaller, more manageable subproblems. Divide-and-conquer does this by splitting the problem into independent parts and solving each separately. Dynamic programming, on the other hand, involves storing and reusing solutions to overlapping subproblems to avoid redundant work.

The greedy approach, by contrast, is often the most intuitive method in algorithm design. When faced with a problem that requires a series of decisions, a greedy algorithm makes the ”best” choice available at each step, focusing solely on the immediate situation without considering future consequences. This approach simplifies the problem by reducing it to a series of smaller subproblems, each requiring fewer decisions. For example, if you are navigating Thiruvananthapuram City and need to head northeast, moving north or east at each step will consistently reduce your distance to the destination. However, in more complex scenarios, such as driving where roads might be one-way, a purely greedy strategy might not always yield the best outcome. In such cases, planning ahead is essential to avoid obstacles and ensure a successful route.

We often deal with problems where the solution involves a sequence of decisions or steps that must be taken to reach the optimal outcome. The greedy approach is a strategy that finds a solution by making the locally optimal choice at each step, based on the best available option at that particular stage. At a fundamental level, it shares a similar philosophy with dynamic programming and divide-and-conquer, which involves breaking down a large problem into smaller, more manageable components that are easier to solve.

Whether the greedy approach is the best method depends on the problem at hand. In some cases, it might lead to an approximate but not entirely optimal solution. For these situations, dynamic programming or brute-force methods might provide a more accurate result. However, when the greedy approach is appropriate, it typically offers faster execution times compared to dynamic programming or brute-force methods.

Example: Coin Changing Problem

Given a set of coin denominations, the task is to determine the minimum number of coins needed to make up a specified amount of money. One approach to solving this problem is to use a greedy algorithm, which works by repeatedly selecting the largest denomination that does not exceed the remaining amount of money. This process continues until the entire amount is covered.

While this greedy algorithm can provide the optimal number of coins for some sets of denominations, it does not always guarantee the minimum number of coins for all cases. For instance, with coin values of 1, 2, and 5, the greedy approach yields the optimal solution for any amount. However, with denominations of 1, 3, and 4, the greedy algorithm can produce a suboptimal result. For example, to make 6 units of money, the greedy method would use coins of values 4, 1, and 1, totaling three coins. The optimal solution, however, would use only two coins of values 3 and 3.

Greedy Solution

1. **Sort the coin denominations** in descending order.
2. Start with the highest denomination and take as many coins of that denomination as possible without exceeding the amount.
3. Repeat the process with the next highest denomination until the amount is made up.

Example

Suppose you have coin denominations of 1, 5, 10, and 25 Rupees, and you need to make change for 63 Rupees.

1. Take two 25-Rupee coins (63 - 50 = 13 Rupees left).
2. Take one 10-Rupee coin (13 - 10 = 3 Rupees left).
3. Take three 1-Rupee coins (3 - 3 = 0 Rupee left).

Thus, the minimum number of coins needed is six (two 25-Rupee coins, one 10-Rupee coin, and three 1-Rupee coins).

☞ The greedy approach makes local choices at each step, aiming for immediate benefit in hopes of finding the global optimum.

Key Characteristics of the Greedy Approach

1. **Local Optimization**: At each step, the algorithm makes the best possible choice without considering the overall problem. This choice is made with the hope that these local optimal decisions will lead to a globally optimal solution.
2. **Irrevocable Decisions**: Once a choice is made, it cannot be changed. The algorithm proceeds to the next step, making another locally optimal choice.
3. **Efficiency**: Greedy algorithms are typically easy to implement and run quickly, as they make decisions based on local information and do not need to consider all possible solutions.

13.4.1 Motivations for the Greedy Approach

The Greedy Approach is motivated by several key factors that make it a desirable strategy for problem-solving. Some are as follows:

1. **Simplicity and Ease of Implementation**:
    - **Straightforward Logic**: Greedy algorithms make the most optimal choice at each step based on local information, making them easy to understand and implement.

- **Minimal Requirements**: These algorithms do not require complex data structures or extensive bookkeeping, reducing the overall implementation complexity.

2. **Efficiency in Time and Space**:

    - **Fast Execution**: Greedy algorithms typically run in linear or polynomial time, which is efficient for large input sizes.
    - **Low Memory Usage**: Since they do not need to store large intermediate results, they have low memory overhead, making them suitable for memory-constrained environments.

3. **Optimal Solutions for Specific Problems**:

    - **Greedy-Choice Property**: Problems with this property allow local optimal choices to lead to a global optimum.
    - **Optimal Substructure**: Problems where an optimal solution to the whole problem can be constructed efficiently from optimal solutions to its sub-problems.

4. **Real-World Applicability**:

    - **Practical Applications**: Greedy algorithms are useful in many real-world scenarios like scheduling, network routing, and resource allocation.
    - **Quick, Near-Optimal Solutions**: In situations where an exact solution is not necessary, greedy algorithms provide quick and reasonably good solutions.

☞ Greedy algorithms work when a problem exhibits the greedy choice property, where local optima lead to global optima.

13.4.2 Characteristics of the Greedy Algorithm

1. **Local Optimization**:

    - Greedy algorithms make the best possible choice at each step by considering only the current problem state without regard to the overall problem. This local choice is made with the hope that these local optimal choices will lead to a globally optimal solution.

2. **Irrevocable Decisions**:

    - Once a choice is made, it cannot be changed. This means that the algorithm does not backtrack or reconsider previous decisions.

3. **Problem-Specific Heuristics**:

- Greedy algorithms often rely on problem-specific heuristics to guide their decision-making process. These heuristics are designed based on the properties of the problem.

4. **Optimality**:

   - Greedy algorithms are guaranteed to produce optimal solutions for some problems (e.g., Coin change, Huffman coding, Kruskal's algorithm for Minimum Spanning Tree) but not for some other problems. The success of a greedy algorithm depends on the specific characteristics of the problem.

5. **Efficiency**:

   - Greedy algorithms are generally very efficient regarding both time and space complexity because they make decisions based on local information and do not need to explore all possible solutions.

☞ By choosing the best option at every stage, greedy algorithms often provide efficient and simple solutions to complex problems.

13.4.3 Solving Computational Problems Using Greedy Approach

To solve computational problems using the Greedy Approach, identify if the problem can be decomposed into sub-problems with an optimal substructure and ensure it possesses the greedy-choice property. Define a strategy to make the best local choice at each step, ensuring these local decisions lead to a globally optimal solution. Design the algorithm by sorting the input data if needed and iterating through it, making the optimal local choice at each iteration while keeping track of the solution being constructed. Finally, analyze the algorithm's efficiency and correctness by testing it on various cases, including edge cases.

Let us see how we can apply the greedy approach to solve a computational problem.

13.4.3.1 Problem-1 (Task Completion Problem)

Given an array of positive integers each indicating the completion time for a task, find the maximum number of tasks that can be completed in the limited amount of time that you have.

In the problem of finding the maximum number of tasks that can be completed within a limited amount of time, the optimal substructure can be identified by recognizing how smaller sub-problems relate to the overall problem. Here's how it works:

1. **Break Down the Problem**: Consider a subset of the tasks and determine the optimal solution for this subset. For example, given a certain time limit, find the maximum number of tasks that can be completed from the first k tasks in the array.
2. **Extend to Larger Sub-problems**: Extend the solution from smaller sub-problems to larger ones. If you can solve the problem for k tasks, you can then consider the $(k+1)$th task and decide if including this task leads to a better solution under the given time constraint.
3. **Recursive Nature**: The optimal solution for the first k tasks should help in finding the optimal solution for the first $(k + 1)$ tasks. This recursive approach ensures that the overall solution is built from the solutions of smaller sub-problems.
4. **Greedy Choice**: At each step, make the greedy choice of selecting the task with the shortest completion time that fits within the remaining available time. This choice reduces the problem size and leads to a solution that maximizes the number of tasks completed.

By iteratively applying this approach and making the best local choices (selecting the shortest tasks first), you can construct a globally optimal solution from optimal solutions to these smaller sub-problems, demonstrating the optimal substructure property.

☞ Greedy algorithms excel in problems with optimal substructure, where the problem can be broken down into smaller, solvable components.

To solve the problem of finding the maximum number of tasks that can be completed in a limited amount of time using a greedy algorithm, you can follow these steps:

1. **Sort the tasks by their completion times in ascending order**: This ensures that you always consider the shortest task that can fit into the remaining time, maximizing the number of tasks completed.
2. **Iterate through the sorted list of tasks and keep track of the total time and count of tasks completed**: For each task, if adding the task's completion time to the total time does not exceed the available time, add the task to the count and update the total time.

Here is the greedy algorithm solution in Python:

```
def max_tasks(completion_times, available_time):
    """ Step 1: Sort the tasks by their completion times in
    ascending order. """

    completion_times.sort()
    total_time = 0
    task_count = 0

    """ Step 2: Iterate through the sorted list of tasks for
    time in completion_times:

        If adding the task's completion time does not exceed
    the available time"""

    for time in completion_times:
        if total_time + time <= available_time:
           total_time += time
           task_count += 1
        else:
            break # No more tasks can be completed in the
                # available time

    return task_count

# Example usage
completion_times = [2, 3, 1, 4, 6]
available_time = 8
print(f"Maximum number of tasks that can be completed:
    {max_tasks(completion_times, available_time)}")
#{max_tasks(completion_times, available_time)}
```

Explanation:

1. **Sorting**: The list of completion times is sorted in ascending order. This step ensures that we always consider the shortest tasks first, which helps in maximizing the number of tasks that can be completed within the given time.

2. **Iterating through sorted tasks**: The algorithm iterates through the sorted list and maintains two variables:

   - **total_time**: The cumulative time of tasks completed so far.
   - **task_count**: The count of tasks completed.

3. **Checking time constraint**: For each task, it checks if adding the task's completion time to **total_time** exceeds **available_time**. If it does not exceed, the task is added to the count, and **total_time** is updated. If it exceeds, the loop breaks because no more tasks can be completed without exceeding the available time.

Example

Consider the example usage with **completion_times = [2, 3, 1, 4, 6]** and **available_time = 8**:

- After sorting: [1, 2, 3, 4, 6]
- Iterating:
  - Add task with time **1: total_time = 1, task_count = 1**
  - Add task with time **2: total_time = 3, task_count = 2**
  - Add task with time **3: total_time = 6, task_count = 3**
  - Next task with time **4** would exceed **available_time**, so the loop breaks.

The maximum number of tasks that can be completed in 8 units of time is 3.

13.4.4 Greedy Algorithms vs. Dynamic Programming

Greedy Algorithms:

- **Approach**: Make the best possible choice at each step based on local information, without reconsidering previous decisions.
- **Decision Process**: Makes decisions sequentially and irrevocably.
- **Optimality**: Guaranteed to produce optimal solutions only for certain problems with the greedy-choice property and optimal substructure.
- **Efficiency**: Typically faster and uses less memory due to the lack of extensive bookkeeping.
- **Example Problems**: Coin Change Problem (specific denominations), Kruskal's Algorithm for Minimum Spanning Tree, Huffman Coding.

Dynamic Programming:

- **Approach**: Breaks down a problem into overlapping sub-problems and solves each sub-problem only once, storing the results to avoid redundant computations.
- **Decision Process**: Considers all possible decisions and combines them to form an optimal solution, often using a bottom-up or top-down approach.
- **Optimality**: Always produces an optimal solution by considering all possible ways of solving sub-problems and combining them.
- **Efficiency**: Can be slower and use more memory due to storing results of all sub-problems (memoization or tabulation).
- **Example Problems**: Fibonacci Sequence, Longest Common Subsequence, Knapsack Problem.

13.4.5 Advantages and Disadvantages of the Greedy Approach

Advantages of the Greedy Approach

1. **Simplicity**: Greedy algorithms are generally easy to understand and implement.
2. **Speed**: These algorithms typically run quickly, making them suitable for large input sizes.
3. **Optimal for Certain Problems**: For some problems, like the Coin Change Problem with certain denominations, greedy algorithms provide an optimal solution.

Disadvantages of the Greedy Approach

1. **Suboptimal Solutions**: Greedy algorithms do not always produce the optimal solution for every problem. They are most effective when the problem has the greedy-choice property, meaning a global optimum can be reached by making local optimal choices.
2. **Irrevocable Decisions**: Once a choice is made, it cannot be changed, which may lead to a suboptimal solution in some cases.
3. **Lack of Backtracking**: Greedy algorithms do not explore all possible solutions or backtracks, which means they can miss better solutions.

☞ Although greedy algorithms may not always find the perfect solution, they often provide fast and close-to-optimal answers.

The greedy approach is a powerful and efficient problem-solving strategy that works well for certain types of optimization problems. Its simplicity and speed make it a valuable tool in the algorithmic toolkit. However, it is essential to analyze whether the problem at hand has the greedy-choice property to ensure that the greedy algorithm will produce an optimal solution. For problems where the greedy approach does not guarantee optimality, other methods such as dynamic programming or backtracking may be more appropriate.

13.5 Randomized Approach to Problem Solving

In the domain of computational problem solving, randomized approaches using simulations offer a dynamic and accessible method for tackling complex challenges. For example, consider the task of estimating the area of a circle inscribed within a square. One way to approach this is by randomly placing points throughout the square and determining how many of those points fall inside the circle. By calculating the ratio of points that land inside the circle to the total number of points placed, we can approximate the proportion of the circle's area relative to the square's area. This proportion, when multiplied by the area of the square, provides an estimate of the circle's area. This example illustrates how simulations can model geometric problems and provide insights that might be challenging to achieve through traditional deterministic methods.

Simulations leverage randomness to model and analyze problems, providing insights that might be difficult to achieve through traditional deterministic approaches. These methods allow us to explore the behavior of systems under uncertainty, estimate performance metrics, and derive practical solutions for a variety of problems. This section introduces the principles of simulation within the context of randomized problem-solving, using classical examples to illustrate their practical applications.

Consider a classic example known as the "birthday problem". Imagine a group of students in a classroom, and you want to determine the probability that at least two students share the same birthday. The exact calculation involves combinatorial mathematics, but a simulation offers a more intuitive way to understand the problem. By randomly assigning birthdays to students across many simulated classrooms and recording the number of times at least two students share a birthday, we can empirically estimate the probability. This approach not only simplifies the analysis but also provides a hands-on understanding of probabilistic concepts.

Another illustrative example is the "random walk on a grid", where a person starts at a fixed point on a grid and takes steps in random directions. The goal is to determine the expected time it takes for the person to return to the starting point. Simulating this random walk multiple times allows us to estimate the average return time, offering insights into the behavior of random processes. This example demonstrates how simulations can model stochastic systems and provide valuable data about their dynamics.

Another example is the "random coin flips" problem, where we want to determine the probability of getting a certain number of heads in a series of coin flips. For instance, simulating 100 coin flips repeatedly and counting the number of heads in each simulation allows us to estimate the probability distribution of getting a specific number of heads. This example highlights how simulations can be used to analyze probabilistic events and gain empirical insights into the behavior of random processes

These examples highlight the power of using random processes to address computational problems where traditional methods might be complex or impractical. By incorporating randomness, we can explore a wide range of scenarios, estimate outcomes, and gain insights into the behavior of systems with uncertain or probabilistic elements. This approach provides a flexible and accessible way to tackle problems that may be challenging to solve using deterministic methods alone.

This section will delve into various problems and applications where randomness plays a crucial role in solving computational challenges. By examining simple yet illustrative examples, we will demonstrate how random processes can enhance our problem-solving toolkit and provide valuable insights into complex systems.

☞ Randomized approach introduces randomness into the decision-making process, often yielding simple and efficient solutions to complex problems.

13.5.1 Motivations for the Randomized Approach

The randomized approach to problem-solving offers several compelling advantages that can make it a valuable tool in both theoretical and practical applications. Some of them are as follows:

1. **Complexity Reduction**: A randomized approach often simplifies complex problems by introducing probabilistic choices that lead to efficient solutions. For example, imagine you are organizing a community health screening event in a large city. You need to decide on the number of screening stations and their locations to maximize coverage and efficiency. Instead of analyzing every possible combination of locations and station numbers—which would be highly complex and time-consuming - you could randomly select several potential locations and test their effectiveness. By evaluating a sample of these random setups, you can identify patterns or clusters of locations that work well. This method simplifies the complex problem of optimizing station placement by reducing the number of scenarios you need to explore in detail.

2. **Versatility**: Applicable across diverse domains, from combinatorial optimization to stochastic simulations, where deterministic solutions may be impractical or infeasible. For example, consider a company that is developing a new app and wants to test its usability. Testing every feature with every possible user scenario could be impractical. Instead, the company could randomly select a diverse group of users and a subset of features to test. By analyzing how this sample of users interacts with the app and identifying any issues they encounter, the company can gain insights that are broadly applicable to all users. This approach allows the company to obtain useful feedback and make improvements without needing to test every possible combination of user and feature.

3. **Performance**: In certain scenarios, a randomized approach can offer significant performance improvements over deterministic counterparts, particularly when dealing with large datasets or complex systems For example, imagine a large library that wants to estimate how often books are checked out. Instead of tracking every single book's check-out frequency—which would be a massive task—the library staff could randomly sample a selection of books from different genres and record their check-out rates over a period of time. By analyzing this sample, they can estimate the

average check-out frequency for the entire collection. This approach improves performance in terms of both time and resources, allowing the library to make informed decisions about which books to keep, acquire, or remove based on practical data from the sampled books.

☞ The power of randomness lies in its ability to break symmetries and explore solution spaces that deterministic methods may overlook.

13.5.2 Characteristics of Randomized Approach

Randomized approaches, which incorporate elements of randomness into their decision-making processes, possess distinct characteristics that differentiate them from deterministic methods. Some of them are as follows:

1. **Probabilistic Choices**: A randomized approach makes decisions based on random sampling or probabilistic events, leading to variable but statistically predictable outcomes. For instance, consider a company deciding where to place new vending machines in a large office building. Instead of assessing every possible location in detail, the company could randomly select a few potential spots, test their performance, and use this data to make a final decision. Although the locations chosen may vary each time the process is conducted, the overall approach helps identify the most effective spots based on statistical analysis of the sampled data.

2. **Efficiency**: They often achieve efficiency by sacrificing deterministic guarantees for probabilistic correctness, optimizing performance in scenarios where exhaustive computation is impractical. For instance, suppose you need to determine the most popular menu items in a large restaurant chain. Instead of surveying every customer, which would be time-consuming and expensive, you might randomly select a subset of customers and analyze their preferences. Although this method does not guarantee that you will capture every preference perfectly, it provides a practical and efficient way to understand overall trends without needing to gather data from every single customer.

3. **Complexity Analysis**: Evaluating the performance of randomized approaches involves analyzing their average-case behavior or expected outcomes over multiple iterations, rather than deterministic worst-case scenarios. For example, if you are estimating the average time it takes for customers to complete a purchase at an online store, you might randomly sample customer transactions over a period of time. Instead of focusing on the longest possible wait time, you analyze how the average wait time behaves across many transactions. This approach provides a practical understanding of performance under typical conditions, rather than the extremes, offering a more balanced view of how the system performs in real-world scenarios.

Overall, the characteristics of randomized approaches—probabilistic choices, efficiency, and average-case complexity analysis—highlight their adaptability and practical advantages. These features make them a powerful tool for tackling complex problems where deterministic methods may fall short, offering a balance between performance and reliability in a wide range of computational scenarios.

☞ Randomized approaches balance simplicity and performance, trading deterministic precision for probabilistic guarantees of correctness.

13.5.3 Randomized Approach vs Deterministic Methods

Randomized approaches and deterministic methods each offer unique advantages and are suited to different types of problems. Randomized methods incorporate elements of chance, which can simplify complex issues and provide efficient solutions when dealing with large or variable datasets. For example, consider a company that wants to estimate customer satisfaction levels across a vast number of branches. Instead of surveying every customer at each branch, the company could randomly select a few branches and survey a sample of customers from those locations. This approach provides a statistically valid estimate of overall satisfaction while avoiding the need for exhaustive data collection.

In contrast, deterministic methods are based on predictable, fixed processes and deliver consistent results each time they are applied. For instance, if you need to calculate the total cost of items in a shopping cart, you would use a deterministic approach where each item's price is added together to get an exact total. This method ensures accuracy and repeatability but may be less adaptable when dealing with uncertainty or incomplete data, such as predicting future sales based on historical trends.

Randomized approaches are especially beneficial in scenarios where processing or analyzing every possible option is impractical. For instance, if a researcher wants to estimate the average time people spend exercising each week, they might use randomized surveys to gather data from a representative sample of individuals rather than interviewing everyone. This method allows for efficient data collection and analysis, offering insights into exercise habits without the need for comprehensive surveys of every individual.

On the other hand, deterministic methods are ideal for situations where precision and reliability are essential. For example, when designing a new piece of machinery, engineers use deterministic methods to perform precise calculations to ensure the machinery operates safely and efficiently. These methods provide exact and consistent results, which are crucial for meeting stringent safety and performance standards. The choice between randomized and deterministic methods depends on the nature of the problem, including the need for accuracy, efficiency, and the ability to handle variability and uncertainty.

13.5.4 Solving Computational Problems Using Randomization

Having discussed the motivation and characteristics of randomized algorithms, let us look at how to solve computational problems using randomization. To get an idea of how to solve computational problems using randomization, let us start with the problem of estimating the value of **Pi** (π).

A common randomized approach to estimate the value of **Pi** (π) is the Monte Carlo method. This method involves simulating random points in a square that contains a quarter circle and calculating the ratio of points that fall inside the quarter circle to the total number of points.

Monte Carlo Method to Estimate π

1. Generate random points within a unit square (1×1).
2. Count how many points fall inside a quarter circle of radius 1.
3. The ratio of points inside the quarter circle to the total points approximates the area of the quarter circle ($\frac{\pi}{4}$).
4. Multiply this ratio by 4 to estimate π.

Here is a Python code to estimate π using the Monte Carlo method:

```
import random

def estimate_pi(num_samples):
    inside_circle = 0

    for _ in range(num_samples):
       x = random.random()
       y = random.random()
       if x**2 + y**2 <= 1:
           inside_circle += 1

    pi_estimate = (inside_circle / num_samples) * 4
    return pi_estimate

# Example usage:
num_samples = 1000000  # Number of random points to generate
pi_estimate = estimate_pi(num_samples)
print(f"Estimated value of pi with {num_samples} samples:
    {pi_estimate}")
```

Explanation:

1. **Random Points Generation**:

- **random.random()**: Generates a random floating-point number between 0 and 1. These numbers represent the x and y coordinates of points in a unit square.

2. **Check Points Inside the Circle**:

   - **x**2 + y**2 <= 1**: Checks if the point (x, y) lies within the quarter circle of radius 1.

3. **Calculate π**:

   - **inside_circle/num_samples** gives the fraction of points that lie inside the quarter circle.
   - Multiplying this fraction by 4 gives an estimate of π because the area of the quarter circle is $\frac{\pi}{4}$.

Running the above code with a large number of samples will provide an estimate of π. Here is an example output:

```
Estimated value of pi with 1000000 samples: 3.141592
```

As you increase the number of samples, the estimate becomes more accurate. This demonstrates the power of the Monte Carlo method in approximating mathematical constants through randomized simulations.

☞ By leveraging randomness, we can simplify the analysis of algorithms, often focusing on expected performance rather than worst-case scenarios.

We look at two more problems - **The coupon problem and The Hat problem** - in depth that will reinforce the randomized approach to problem-solving.

13.5.4.1 Problem-1 (Coupon Problem)

A company selling jeans gives a coupon for each pair of jeans. There are n different coupons. Collecting n different coupons would give you free jeans. How many jeans do you expect to buy before getting a free one?

Let us start with an algorithmic solution in plain English to determine how many pairs of jeans you might need to buy before collecting all n different coupons and getting a free pair of jeans:

Algorithmic Solution:

1. **Initialize Variables**:

   - **Total Jeans Bought**: Start with a counter set to zero to track how many pairs of jeans you have bought.

- **Coupons Collected**: Use a set to keep track of the different types of coupons you have received.
- **Number of Coupons**: The total number of different coupon types is n.

2. **Buying Process**:
   - **Loop Until All Coupons Are Collected**: Continue buying jeans until you have one of each type of coupon in your set.
   - Each time you buy a pair of jeans, increase the counter for the total jeans bought by one.
   - When you buy a pair of jeans, you get a coupon. Add this coupon to your set of collected coupons.
   - Check if you have collected all n different types of coupons by comparing the size of your set to n.
3. **Repeat for Accuracy**:
   - To get a reliable estimate, repeat the entire buying process many times (e.g., 100,000 times).
   - Keep a running total of the number of jeans bought across all these repetitions.
4. **Calculate the Average**:
   - After completing all repetitions, calculate the average number of jeans bought by dividing the total number of jeans bought by the number of repetitions.
5. **Output the Result**:
   - The average number of jeans bought from the repeated simulations gives you a good estimate of how many pairs of jeans you would typically need to buy before collecting all n coupons and getting a free pair.

Let us walk through an example

1. **Setup and Initialization Step**:
   - Imagine there are 10 different types of coupons.
   - Start with **total_jeans = 0** and an empty set **coupons_collected**.
2. **Buying Jeans**:
   - You buy a pair of jeans and get a coupon. Add the coupon to your set.
   - Increase **total_jeans** by 1.

- Check if your set now contains all 10 different coupons.

3. **Continue Until Complete**:
   - Repeat the buying process, each time adding the coupon to your set and increasing the total jeans count.
   - Once you have all 10 types in your set, note the total number of jeans bought for this repetition.
4. **Repeat Many Times**:
   - To ensure accuracy, repeat this entire process (buying jeans, collecting coupons) 100,000 times.
   - Sum the total number of jeans bought over all repetitions.
5. **Calculate Average**:
   - Divide the sum of all jeans bought by 100,000 to get the average number of jeans you need to buy to collect all coupons.

By following these steps, you can estimate how many pairs of jeans you need to buy before getting a free pair by collecting all the different coupons. This method uses repetition and averaging to account for the randomness in coupon distribution, providing a reliable estimate.

To implement a programmatic solution to calculate the expected number of jeans you need to buy before getting a free one when there are n different coupons, we can simulate the process and compute the average number of purchases. Here is a Python implementation using simulation:

```
import random

def expected_jeans(n, num_simulations=100000):
    total_jeans = 0

    for _ in range(num_simulations):
        coupons_collected = set()
        jeans_bought = 0

        while len(coupons_collected) < n:
            jeans_bought += 1
            coupon = random.randint(1, n)  # simulate getting a
    random coupon
            coupons_collected.add(coupon)  # add the coupon to
    the set

        total_jeans += jeans_bought

    expected_jeans = total_jeans / num_simulations
```

```python
    return expected_jeans

# Example usage:
n = 10  # number of different coupons
expected_num_jeans = expected_jeans(n)
print(f"Expected number of jeans before getting a free one with
    {n} coupons: {expected_num_jeans}")
```

Explanation:

1. **Function expected_jeans(n, num_simulations)**:
   - **n**: Number of different coupons.
   - **num_simulations**: Number of simulations to run to estimate the expected value. More simulations lead to a more accurate estimation.
2. **Simulation Process**:
   - For each simulation, initialize an empty set **coupons_collected** to keep track of collected coupons and a counter **jeans_bought** to count the number of jeans purchased.
   - Loop until all **n** different coupons are collected:
     - Simulate buying a pair of jeans (**jeans_bought** increments).
     - Simulate getting a random coupon (from 1 to **n**).
     - Add the coupon to **coupons_collected** if it has not been collected before.
   - After collecting all **n** coupons, record the total number of jeans bought in **total_jeans**.
3. **Calculate Expected Value**:
   - Compute the average number of jeans bought across all simulations (**total_jeans / num_simulations**).
   - Return the estimated expected number of jeans.
4. **Example Usage**:
   - Set **n** to the number of different coupons (e.g., 10).
   - Call **expected_jeans(n)** to get the estimated expected number of jeans before getting a free one.

The code will print the expected number of jeans you need to buy before collecting all 10 coupons for each of the three runs. Each run provides an estimate based on 100,000 simulations, which ensures the reliability of the results. Running the code multiple times ensures that the results are consistent and demonstrate the reliability of the simulation approach.

Let us run the simulation three times and compare the results with the theoretical expectation for the Coupon problem. The theoretical expected number of purchases required to collect all n coupons is given by: $H_n = n\left(\frac{1}{1}+\frac{1}{2}+\cdots+\frac{1}{n}\right)$, where H_n is the n-th harmonic number. Here are the results from three runs of the given simulation code for 10 coupons:

> **Run 1**: Expected number of jeans before getting a free one with 10 coupons: 29.2446
>
> **Run 2**: Expected number of jeans before getting a free one with 10 coupons: 29.3227
>
> **Run 3**: Expected number of jeans before getting a free one with 10 coupons: 29.3175

The theoretically expected number of jeans needed is approximately 29.29. The results from the three runs are all very close to this theoretical value.

- **Run 1**: 29.2446
- **Run 2**: 29.3227
- **Run 3**: 29.3175

This demonstrates that the simulation results are consistent with the theoretical expectation for the Coupon problem.

The approach we used to solve the Coupon problem is an instance of Monte Carlo simulation to estimate the expected value, making it suitable for scenarios where an analytical solution is complex or impractical. Do more simulations by adjusting **num_simulations** to balance between computation time and accuracy of the estimated value. More simulations generally provide a more precise estimation of the expected value.

13.5.4.2 Problem-2 (Hat Problem)

n people go to a party and drop off their hats to a hat-check person. When the party is over, a different hat-check person is on duty and returns the n hats randomly back to each person. What is the expected number of people who get back their hats?

To solve this problem, we can simulate the process of randomly distributing hats and count how many people get their own hats back. By running the simulation many times, we can calculate the expected number of people who receive their own hats. Let us start with an algorithmic solution in plain English.

Algorithmic Solution:

1. **Initialization**:
    - Set up variables to count the total number of correct matches across all simulations.

- Define the number of simulations to ensure statistical reliability.
- Define the number of people n.

2. **Simulate the Process**:
   - For each simulation:
     - Create a list of hats representing each person.
     - Shuffle the list to simulate random distribution.
     - Count how many people receive their own hat.
     - Add this count to the total number of correct matches.
3. **Calculate the Expected Value**:
   - Divide the total number of correct matches by the number of simulations to get the average.
4. **Output the Result**:
   - Print the expected number of people who get their own hats back.

Let us walk through an example

1. **Setup and Initialization Step**:
   - Suppose there are 5 people at the party.
   - Initialize **total_correct** to 0, which will keep track of the total number of people who receive their own hat across multiple simulations and **num_simulations** to 100,000.
2. **Simulate the Process**:
   - For each simulation:
     - Create a list of hats [1, 2, 3, 4, 5].
     - Shuffle the list, e.g., [3, 1, 5, 2, 4].
     - Initialize **correct** to 0.
     - Check each person:
       - Person 1 (hat 3) - not correct.
       - Person 2 (hat 1) - not correct.
       - Person 3 (hat 5) - not correct.
       - Person 4 (hat 2) - not correct.
       - Person 5 (hat 4) - not correct.
     - Add **correct** (which is 0 for this run) to **total_correct**.
3. **Repeat Many Times**:
   - Repeat the simulation 100,000 times, each time shuffling the hats and counting how many people get their own hats back.

- Sum the number of correct matches across all simulations.

4. **Calculate Average**:

    - Divide **total_correct** by 100,000 to get the average number of people who receive their own hat.

By following these steps, you can determine the expected number of people who will get their own hats back after the hats are randomly redistributed. In the case of this specific problem, the expected number will be approximately 1, meaning on average, one person will get their own hat back. This is due to the nature of permutations and the expectation of fixed points in a random permutation. This solution uses a Monte Carlo simulation approach to estimate the expected number of people who get their own hats back, which is a practical way to solve problems involving randomness and expectations.

Here is a Python solution that implements this algorithm:

```
import random

def simulate_hat_problem(n, num_simulations):
    total_correct = 0

    for _ in range(num_simulations):
       hats = list(range(n))
       random.shuffle(hats)
       correct = sum(1 for i in range(n) if hats[i] == i)
       total_correct += correct



    expected_value = total_correct / num_simulations
    return expected_value

# Example usage
n = 10  # Number of people at the party
num_simulations = 100000  # Number of simulations to run
expected_hats_back = simulate_hat_problem(n, num_simulations)

print(f"The expected number of people who get their own hats
    back is approximately: {expected_hats_back}")
```

Explanation:

1. **Initialization**:

    - **total_correct** keeps track of the total number of people who get their own hats back across all simulations.

- **num_simulations** is the number of times we repeat the experiment to get a reliable average.

2. **Simulation**:
   - For each simulation, a list **hats** is created, representing the hats each person initially has.
   - The **random.shuffle(hats)** function randomly shuffles the list to simulate the random distribution of hats.
   - We then count how many people receive their own hat by checking if the index matches the value in the list.

3. **Calculate the Expected Value**:
   - The **total_correct** is divided by **num_simulations** to find the average number of people who get their own hats back.

4. **Output**:
   - The expected number is printed out, showing the average number of people who end up with their own hats.

Here are the results from three runs of the given simulation code:

Run 1: The expected number of people who get their own hats back is approximately: 1.00039

Run 2: The expected number of people who get their own hats back is approximately: 1.00051

Run 3: The expected number of people who get their own hats back is approximately: 0.99972

Across these three runs, the expected number of people who get their own hats back consistently revolves around 1. This aligns with the theoretical expectation that, on average, one person out of n will receive their own hat back in such a random distribution scenario. This outcome demonstrates the robustness of the Monte Carlo simulation approach for estimating expected values in problems involving randomness.

☞ The success of randomized approaches often comes from their ability to avoid worst-case patterns that deterministic algorithms might fall into.

The randomized approach plays a crucial role in modern problem-solving, leveraging probability and randomness to tackle challenges that defy straightforward deterministic solutions. By embracing uncertainty and probabilistic outcomes, randomized algorithms provide innovative solutions across various disciplines, highlighting their relevance and effectiveness in addressing complex, real-world problems. This chapter explores the theoretical foundations, practical applications, and considerations of randomized approaches, equipping readers with insights into their strategic deployment in diverse problem-solving scenarios.

13.6 Conclusion

Computational approaches to problem-solving include a diverse range of methodologies, each offering unique strategies to tackle complex challenges across various domains. This chapter has explored several fundamental techniques: brute force, divide-and-conquer, dynamic programming, greedy algorithms, and randomized approaches. Understanding these strategies equips you with a toolkit to address different problem classes effectively.

Brute Force: The brute-force approach is characterized by its simplicity and exhaustive computation. By systematically checking every possible solution, brute force ensures that the optimal solution is found. This method is well-suited for problems with relatively small solution spaces, such as cracking padlocks or guessing passwords. However, for larger problem spaces, brute force can become computationally prohibitive due to its high execution time and resource demands.

Divide-and-Conquer: This approach improves efficiency over brute force by breaking down complex problems into smaller, manageable sub-problems. The divide-and-conquer strategy, as illustrated by merge sort, recursively divides a problem into simpler components, solves each component, and then combines the results. While this method often leads to significant reductions in time complexity, it may incur additional overhead from recursive calls and the need for additional memory to manage sub-problems.

Dynamic Programming: Dynamic programming enhances efficiency by addressing overlapping sub-problems and storing intermediate results to avoid redundant computations. For instance, solving the Fibonacci series with dynamic programming involves memoizing previously computed results to reduce complexity. This approach trades off increased space complexity for reduced time complexity, making it particularly effective for optimization problems where solutions depend on previously computed results.

Greedy Algorithms: Greedy algorithms focus on making the locally optimal choice at each step, intending to achieve a global optimum. This method is beneficial for problems like scheduling tasks within a limited timeframe. Greedy algorithms are valued for their simplicity and efficiency, but they can sometimes fail to find the globally optimal solution due to their focus on immediate gains rather than long-term strategies.

Randomized Approaches: Randomized approaches introduce elements of chance into problem-solving, providing probabilistic solutions to problems that might otherwise be deterministic. Examples include scenarios like coupon collecting or hat-checking problems. These methods offer a way to explore solution spaces in a non-deterministic manner, which can be particularly useful when exact solutions are impractical or when dealing with complex stochastic phenomena.

In summary, each computational approach offers distinct advantages and limitations. Brute force guarantees optimal solutions but may be impractical for large problems. Divide-and-conquer and dynamic programming provide efficient solutions through systematic problem breakdown and memoization, respectively. Greedy algorithms offer quick solutions but may not always achieve the best outcome, while randomized approaches can explore complex solution spaces effectively. By understanding these methodologies, you can better select the appropriate strategies for solving diverse computational problems across various disciplines.

Here are some highly recommended books that provide a solid foundation for the related concepts that we discussed in this chapter. [1] Often referred to as CLRS, this classic textbook covers a wide array of algorithms and problem-solving techniques. Each approach is explained with rigorous theoretical insights, pseudocode, and practical applications. It is essential for anyone seeking to master algorithm design. [2] Skiena's manual is more application-focused, blending theory with practical examples. It's highly recommended for its problem catalog and real-world examples of each problem-solving strategy. [3] This book is an excellent introduction to algorithmic thinking and covers multiple paradigms in algorithm design. It focuses on visual and intuitive explanations, supported by thorough implementations. [5] This book emphasizes the design, analysis, and implementation of algorithms, with sections dedicated to each paradigm. It is known for a good balance between theory and implementation exercises. [4] This book takes a more theoretical approach to algorithms, explaining the fundamental principles behind each strategy. It is ideal for a deeper understanding of algorithmic efficiency and complexity. [6] This book focuses on the thought process behind solving computational problems with algorithms, providing exercises and real-world examples to explain dynamic programming and greedy approaches. [7] This book is the go-to resource for learning about randomized algorithms. It delves deep into the theory and analysis of randomness in computational problem-solving and is considered a benchmark reference for this approach. [8] This book offers a comprehensive introduction to the use of probability and randomness in the design and analysis of algorithms. It covers essential topics such as randomized algorithms, probabilistic analysis of algorithms, Markov chains, and more. The book is well-regarded for its clear explanations and practical examples, blending theoretical concepts with hands-on applications. It is ideal for readers interested in understanding how randomness can be harnessed to create efficient algorithms, especially in complex computational scenarios. It is often recommended for advanced undergraduate and graduate courses in computer science. [9] This book provides a practical approach to learning algorithms through Python. It focuses on teaching the fundamentals of algorithmic problem-solving while leveraging Python's simplicity and power. The book covers essential topics like sorting, searching, graph algorithms, and dynamic programming, with a special emphasis on implementation. Each algorithm is explained clearly and intuitively, making it accessible for beginners and useful for intermediate programmers who want to improve their algorithmic thinking. It is particularly recommended for those who want to deepen their understanding of algorithms using Python as a learning tool.

13.7 Exercises

1. Design and implement an algorithm to solve the subset sum problem using a brute-force approach.

2. Write a brute-force algorithm to solve the knapsack problem.

3. Use the brute-force approach to find the maximum element in an array by iterating through each element and keeping track of the largest value encountered. This approach requires examining each element one by one.

4. Use brute force to find the contiguous subarray with the maximum sum by examining all possible subarrays and calculating their sums. Compare the sums to identify the maximum.

5. Use the brute-force approach to generate all permutations of a given string by systematically swapping characters and exploring all possible arrangements. This involves recursively generating permutations and collecting them.

6. Use brute force to find all pairs of elements in an array that sum up to a specific target value by examining every possible pair of elements. This involves iterating through all element pairs and checking their sums.

7. Use brute force to find the longest common substring between two strings by checking all possible substrings of one string against all substrings of the other. This involves generating and comparing substrings from both strings.

8. Use brute force to find the element that appears most frequently in an array by counting the occurrences of each element and selecting the one with the highest count. This involves iterating through the array and maintaining a count for each element.

9. Use brute force to find all unique triplets in an array that add up to zero by checking every combination of three elements. This involves iterating through all possible triplets and verifying their sums.

10. Use brute force to find the longest palindromic substring in a given string by checking every possible substring and verifying if it is a palindrome.