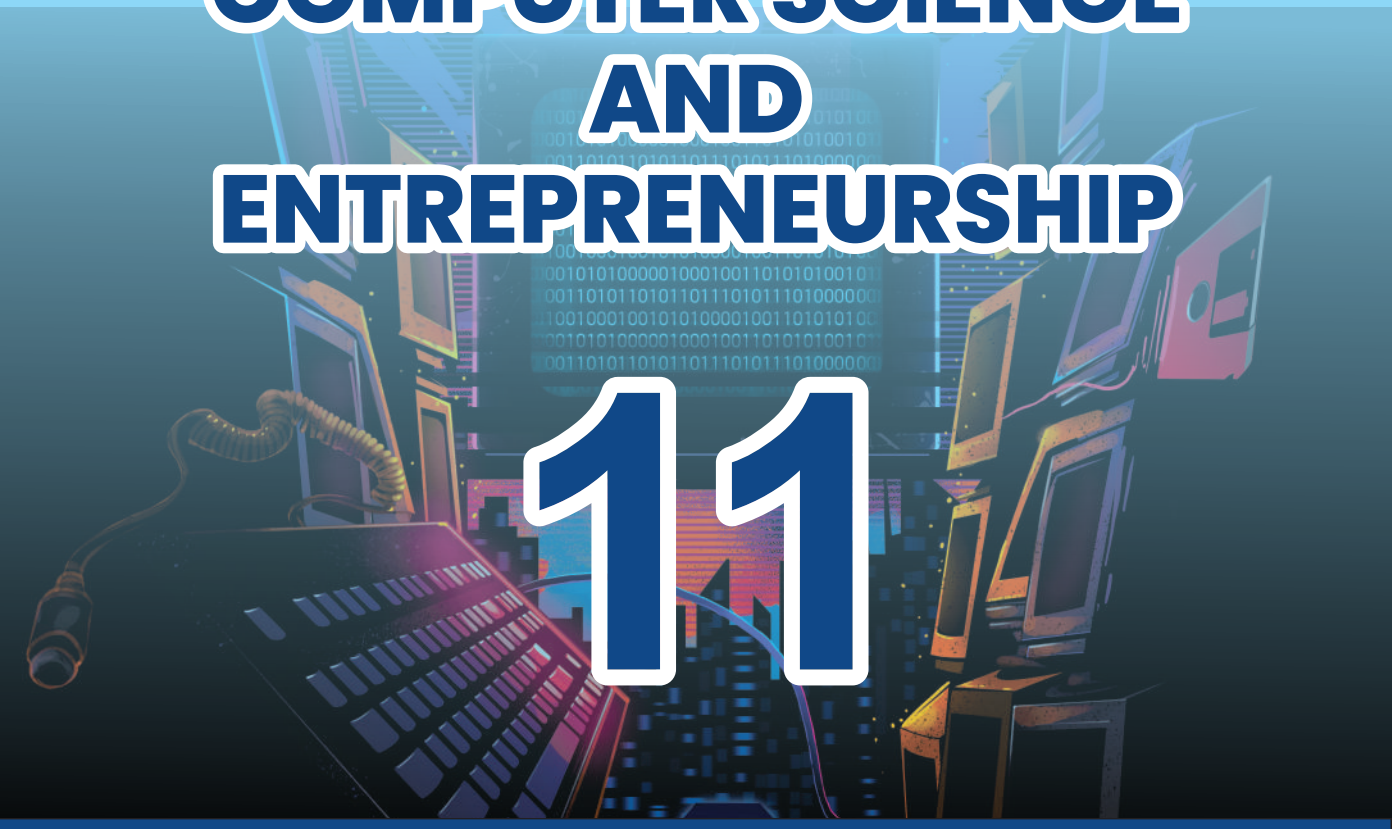
بِسۡمِ اللّٰہِ الرَّحۡمٰنِ الرَّحِیۡمِ

(In the Name of Allah, the Most Merciful, the Most Compassionate.)

COMPUTER SCIENCE AND ENTREPRENEURSHIP



11

PUNJAB EDUCATION, CURRICULUM, TRAINING AND ASSESSMENT AUTHORITY

Contents

Authors:

- Prof. Dr. Muhammad Atif Chattha
  Dean Faculty of Computer Science, Lahore Garrison University, D.H.A, Lahore.
- Prof. Dr. Syed Waqar ul Qounain Jaffry
  Chairman Department of IT, University of The Punjab,
  Allama Iqbal Campus, Lahore.

Experimental Edition

Reviewer:

- Dr. Arshad Ali
  Associate Professor, Department
  Head (Cyber Security), FAST School of Computing
  National University of Computing and Emerging Sciences, Lahore
- Dr. Mudasser Naseer
  Associate Professor(CS),
  Department of CS & IT,
  University of Lahore Defense Road, Lahore
- Dr. Abdul Sattar
  Assistant Professor, Lahore Garrison University, D.H.A Lahore.
- Mr. Saqib Ubaid
  Assistant Professor (Computer Science).
  Khawaja Fareed University of IT, Rahim Yar Khan
- Mr. Muhammad Fahim
  Associate Professor (Computer Science)
  Govt. Graduate College for boys, Gulberg, Lahore
- Mrs. Tabinda Muqaddas
  Assistant Professor, Head of Department (CS),
  Govt. Associate College for Women,
  Gulshan Ravi, Lahore.
- Muhammad Asif Majeed Khan
  SST(IT), Govt Model High School, Jampur
- Mr. Fahad Asif
  EST (CS),
  Govt. Lab Higher Secondary School,
  QAED Kasur.
- Prof. Mahmood Ahmad Chaudhry
  The Crescent College, Shadman, Lahore
- Mr. Jahanzaib Khan
  Asistant Director (Curriculum-Sciences), PECTAA

Director (Curriculum and Compliance)
Mr. Aamir Riaz

Deputy. Director (Compliance-Sciences)
Syed Saghir-Ul-Hassnain Tirmizi

Assistant Director (Curriculum-Sciences)
Mr. Jahanzaib Khan

Incharge Art Cell
Ms. Aisha Sadiq

Design & Layout
Ms. Minal Tariq
Ms. Sameira Ismail

Illustrator
Mr. Ayat Ullah

Composer
M. Azhar Shah

UNIT 1

Introduction to Software Development

Student Learning Outcomes

By the end of this chapter, students will be able to:

- Define software development and explain its importance.
- Understand and describe key software development terminology, including Software Development Life Cycle (SDLC), debugging, testing, and design patterns.
- Explain the stages of the SDLC and the objectives and activities involved in each stage.
- Differentiate between various software development methodologies such as the Waterfall model and Agile methodology.
- Plan a software project by setting timelines, estimating costs, and managing risks.
- Recognize and apply quality assurance techniques to ensure software standards.
- Utilize Unified Modeling Language (UML) diagrams to represent software systems.
- Identify and apply common software design patterns in software design.
- Employ debugging techniques and testing strategies to ensure software reliability.
- Understand and utilize various software development tools, including Integrated Development Environment (IDEs), compilers, and source code repositories.

Introduction

Software development is a systematic process that transforms user needs into software products. It involves a series of stages, from initial analysis through design, coding, testing, and deployment. Each stage has its own importance and requires specific skills and tools. Understanding the software development process is crucial for creating reliable, maintainable, and scalable software solutions. The chapter introduces the fundamental concepts of software development, including key terminology, the Software Development Life Cycle (SDLC), software development methodologies, project planning and management, quality assurance, and software design patterns.

1.1 Software Development

Software development is the process of creating computer programs designed to perform specific tasks. It involves writing code, testing it, and addressing any issues that arise.

1.2 Introduction to Software Development Life Cycle (SDLC)

Software Development Life Cycle (SDLC) is a framework that defines the processes used by organizations to build an application from its initial conception to its deployment and maintenance. The primary purpose of SDLC is to deliver high-quality software that meets customer expectations, reaches completion within time and cost estimates, and works efficiently.

1.2.1 Framework in Software Development

In software engineering, a framework is a standardized and reusable set of concepts, practices, and tools that provides a structured foundation for developing software applications. It offers predefined components and architectures that facilitate the implementation of specific software functionalities, allowing developers to focus on writing code specific to their application rather than reinventing common solutions. Frameworks promote efficiency, consistency, and code reusability, that improve the overall quality and maintainability of software systems.

Example: Imagine you want to create a website. Instead of writing all the code from scratch, you can use a framework like Django (for websites). Django comes with ready-made features like user login, database management, and page templates.

1.2.2 Stages involved in SDLC

The SDLC is an organized method for developing software that ensures it meets quality standards and functions properly. The SDLC consists of several steps as shown in Figure 1.1. Each step has distinct tasks and goals.

1.2.2.1 Requirement Gathering

In this initial phase, the goal is to understand and collect what the software needs to achieve. This involves talking to the people who will use the software, as well as other stakeholders, to find out their needs and expectations.

Figure 1.1: System development life cycle stages

Key activities in this phase include:

- **Interviews and Surveys:** Asking questions and collecting feedback from potential users to understand their needs and preferences.
- **Observations:** Watching how users interact with current systems to identify problems and opportunities for improvement.
- **Document Review:** Looking at existing documents, such as reports and user manuals, to gather additional information about the requirements.

Functional and Non-Functional Requirements

Requirements are generally categorized into two types, functional and non-functional requirements.

Functional Requirements

Functional requirements describe the specific behaviors or functions of a system. These requirements outline what the system should do and include tasks, services, and functionalities that the system must perform.

They define the interactions between the system and its users or other systems.

Example:

Some functional requirements for a Library Management System are:

- **User Registration:** The system should allow users (students and faculty) to register and create an account.
- **Book Borrowing:** The system should enable users to search for books and borrow them.
- **Inventory Management:** Librarians should be able to add, update, and remove books from the inventory.

Non-Functional Requirements

Non-functional requirements define the quality attributes, performance criteria, and constraints of the system. These requirements specify how the system performs a function rather than what the system should do.

Example:

Some non-functional requirements for a Library Management System are:

- **Performance:** The system should handle up to 1000 simultaneous users without performance degradation.
- **Reliability:** The system should be available 99.9% of the time, ensuring high availability and minimal downtime.
- **Security:** User data should be encrypted, and access should be controlled through secure authentication mechanisms.

Differentiating Functional and Non Functional Requirements:	
Functional Requirements	**Non-Functional Requirements**
Define specific behaviors or functions of the system	Define the quality attributes and constraints of the system
What the system should do	How the system should perform
Directly related to user interactions and system tasks	Related to system performance, usability, reliability, etc.

Table 1.1: Comparison between Functional and Non-Functional Requirements

1.2.2.2 Design

In the design phase, we plan out how the software will look and work. During this phase, we:

- **Create Diagrams:** To show how different parts of the software will connect and work together. For example, we draw a flowchart to map out the steps the program will take to complete a task.
- **Develop Models:** To represent the software's structure. This could include creating mockups of the user interface, showing what the program will look like, and how users will interact with it.
- **Plan the Architecture:** To decide the overall structure of the software, including how different components will interact. This helps ensure that the program is organized and functions smoothly.
- **Specify Requirements:** To define clearly what each part of the software needs to do, ensuring that all features are planned out and nothing is overlooked.

These steps help to ensure that the final software is well-organized, user-friendly, and meets the needs of its users.

Tidbits

Think of this phase like designing a new house. You need blueprints to show where the rooms and furniture will go before you start building.

1.2.2.3 Coding / Development

Based on the design specifications, which outline what the software should do and how it should look, programmers translate these specifications into a programming language.

1.2.2.4 Testing

Testing is the process of checking software to identify any bugs, errors, or issues. Think of it as a quality check to make sure everything works as expected. This includes:

- **Functionality Testing:** Ensuring all features of the software work according to the specifications.
- **Performance Testing:** Checking if the software performs well under different conditions, such as high traffic or heavy data.
- **Compatibility Testing:** Making sure the software works well on various devices and operating systems.

1.2.2.5 Deployment

Deployment is the process of making software available for users to access and use. This often involves several steps:

- **Installation:** The software is installed on the user's system or server. This may involve running an installation program that copies files and sets up necessary configurations.
- **Configuration:** The software is adjusted to fit the specific needs of the user or organization. It can include setting up user preferences, network settings, and database connections.
- **Testing in the Real-World:** After installation, the software is tested in its real-world environment to ensure it works correctly with other systems and meets user needs.

1.2.2.6 Maintenance

The final phase involves ongoing maintenance and updates. This ensures the software continues to function correctly and adapts to any changes in user needs or technology.

1.3 Software Development Methodologies

Software development methodologies are structured approaches to software development that guide the planning, creation, and management of software projects. They help ensure that the development process is systematic, efficient, and produces high-quality software.

1.3.1 Introduction to Software Process Models

Software process models are abstract representations of the processes involved in the SDLC. They provide a framework for planning, structuring, and controlling the development of software systems. The importance of software process models lies in their ability to provide:

- **Predictability:** By following a defined process, teams can predict outcomes and manage risks more effectively.
- **Efficiency:** Structured methodologies streamline the development process, reducing wasted effort.
- **Quality:** Adhering to a process model ensures that quality assurance practices are integrated throughout the SDLC.

1.3.1.1 Waterfall Model

The Waterfall Model is a straightforward approach to software development where each phase of the project must be completed before the next one begins. This model is linear and sequential, meaning that you move through each phase in order, without going back to previous phases once they are completed as shown in Figure 1.2. The main phases of the Waterfall Model are:

- **Requirements:** Gather and document what the software needs to do.
- **Design:** Plan how the software will be built and how it will look.
- **Implementation:** Write the actual code to create the software.
- **Testing:** Check for and fix any problems or bugs in the software.
- **Deployment:** Release the software for users to use.
- **Maintenance:** Make updates and fix any issues that come up after the software is in use.

Benefits and Limitations

- **Benefits:**
  1. **Simple and Easy to Understand:** The Waterfall Model is easy to follow because it has clear, distinct phases
  2. **Sequential Process:** Each phase is completed one at a time, which makes it easier to manage and track progress.
  3. **Suitable for Small Projects:** Works well for projects with clear, fixed requirements where changes are unlikely.
- **Limitations:**
  1. **Inflexibility:** Once a phase is completed, going back to make changes is difficult and costly.
  2. **Not Ideal for Complex Projects:** For projects with evolving requirements or complex designs, this model can be challenging to use effectively.
  3. **Risk and Uncertainty:** The model assumes that all requirements are known from the start, which can be risky if new needs or issues arise later in the process.

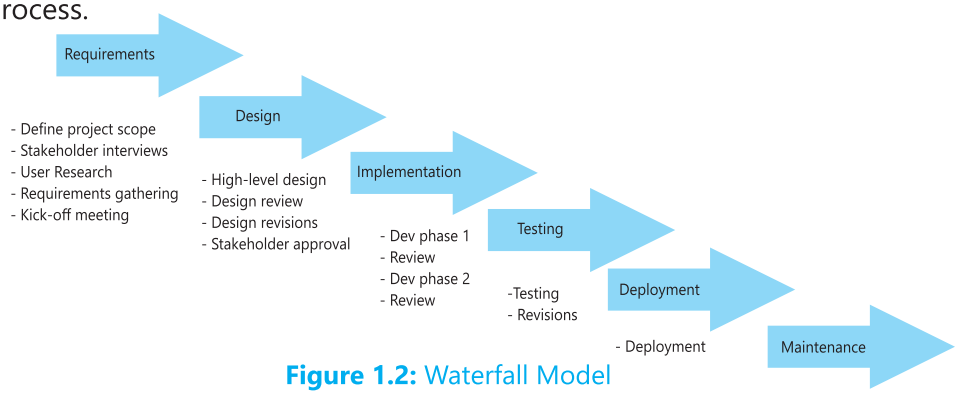


Figure 1.2: Waterfall Model

1.3.1.2 Agile Methodology

Agile Methodology is a flexible and adaptive approach to software development. Agile focuses on delivering small, functional parts of the software quickly and adapting to changes as the project progresses. The main idea is to work in short cycles, called iterations or sprints, which help teams deliver parts of the software rapidly and gather feedback early as shown in Figure 1.3. Agile methods include practices such as:

- **Continuous Integration:** Regularly merging code changes into a central repository to detect and fix issues early.
- **Test-Driven Development:** Writing tests before writing the code to ensure the software works as expected.
- **Pair Programming:** Two developers work together at one workstation, with one writing code and the other reviewing it in real-time.

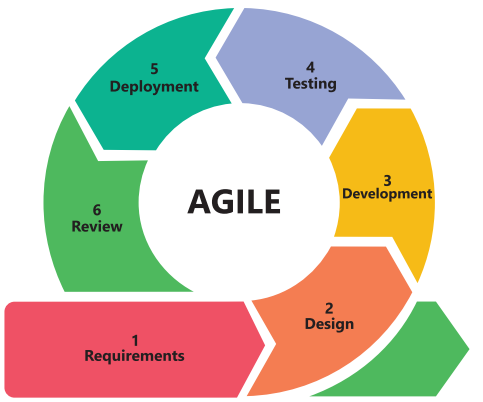


Figure 1.3: Agile Methodology

Benefits and Limitations

- **Benefits:**
  1. **High Flexibility:** Agile allows for changes in requirements even after development has started, making it easier to adapt to new needs or feedback.
  2. **Improved Customer Satisfaction:** Regular updates and frequent delivery of working software mean that customers can see progress and provide feedback more often.
- **Limitations:**
  1. **Scaling Challenges:** Managing large projects with many teams can be difficult, as it requires careful coordination and communication.
  2. **Stakeholder Involvement:** Agile requires active participation from all stakeholders, which can be challenging if some are unavailable or not fully engaged.
  3. **Less Predictable:** Since Agile projects evolve through feedback and changes, it can be harder to predict the exact timeline and scope of the final product.

1.4 Project Planning and Management

Planning a software project is like planning a trip. You need to know where you're going, how long it will take, and how much it will cost.

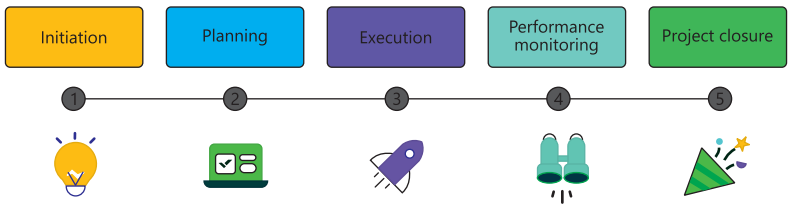


Figure 1.4: The 5 Phases of a Project Management Plan

1.4.1 Comprehensive Project Planning

Comprehensive project planning involves thinking about all the details of your project before you start. This includes understanding what needs to be done, who will do it, and how it will be done.

DO YOU KNOW? Big software companies are worth a lot of money, for example in 2023, Microsoft's worth was $ 2 Trillion. This shows how important software is in today's digital world.

1.4.2 Setting Project Timelines

Setting project timelines means deciding how long each part of the project will take. This helps keep the project on track and ensures it gets done on time.

1.4.3 Estimating Costs

Estimating the cost of a software project is a critical step in project planning and management. It involves predicting the total expenses required to complete the project successfully. Accurate cost estimation helps in budgeting, resource allocation, and setting realistic expectations.

Key Factors in Cost Estimation:

- **Development Team:** The cost depends on the number of developers, their expertise, and their hourly rates.
- **Technology Stack:** The choice of technology, programming languages, and tools can affect the cost. Some technologies require more resources or specialized knowledge.
- **Project Duration:** Longer projects generally incur higher costs due to prolonged resource engagement and potential changes in scope.
- **Risk Management:** Identifying potential risks and their mitigation strategies can add to the overall cost. Contingency funds are often included to address unforeseen issues.
- **Quality Assurance:** Costs associated with testing, bug fixing, and ensuring the

- software meets quality standards, are also part of the estimation.

1.4.4 Risk Assessment and Management

Risk assessment and management are crucial aspects of any software project. They involve identifying potential risks that could impact the project's success, analysing the likelihood and impact of these risks, and developing strategies to manage them.

Steps in Risk Assessment and Management:

1. **Identify Risks:** List all potential risks that could affect the project. These could be technical risks, such as technology changes; operational risks, like resource shortages; or external risks, such as market fluctuations.
2. **Analyze Risks:** Evaluate the likelihood of each risk occurring and its potential impact on the project.
3. **Develop Mitigation Strategies:** For each significant risk, develop a plan to reduce its likelihood or minimize its impact. This could involve adding buffers to the schedule, securing backup resources, or conducting additional testing.
4. **Monitor and Review:** Continuously monitor the project for new risks and review existing risks to adjust strategies as necessary.

1.4.5 Execution

This is the phase where actual development work happens. The team writes codes, creates designs, and builds the software based on the project plan, it requires team work, coordination and regular updates to stay on track.

1.4.6 Quality Assurance

Quality assurance ensures that a project meets set standards and works correctly. It involves methods such as testing, reviewing code, getting feedback from stakeholders, and regularly checking the project's progress.

1.5 Graphical Representation of Software Systems

Graphical representation of software systems involves using visual diagrams to depict various aspects of a software system's structure and behavior. This approach helps in simplifying complex systems, making it easier for developers and stakeholders to understand, communicate, and manage the system.

1.5.1 Introduction to UML

Unified Modeling Language (UML) is a standardized way to visualize the design of a software system. It helps developers understand how a system works and communicates.

1.5.2 Types of UML Diagrams

In this section, we will discuss four types of UML diagrams that are given below.

1.5.2.1 Use Case Diagrams

Use case diagrams provide a visual representation of the system's functionality from the

user's perspective, helping to identify the requirements and the interactions between the users and the system.

Definition and Purpose:

A use case is a description of a set of interactions between a user (actor) and a system to achieve a specific goal. Use cases are identified based on the functionalities that the system must support to meet the user's needs. Each use case represents a complete workflow from the user's perspective, detailing the steps involved in accomplishing a particular task.

Use Case Diagrams are used for several purposes:

1. **Capturing Functional Requirements:** They help in identifying and documenting the functional requirements of the system.
2. **Understanding User Interactions:** They illustrate how different users will interact with the system.
3. **Planning and Testing:** They aid in planning the development process and in designing test cases for validating system functionalities.

Identifying Use Cases:

The process of identifying use cases involves several steps:

1. **Identify Actors:** Determine the different types of users who will interact with the system. Actors can be human users or other systems.
2. **Define Goals:** For each actor, identify their goals or what they need to accomplish using the system.
3. **Outline Interactions:** Describe the interactions between the actors and the system to achieve these goals. Each interaction that results in a significant outcome is a potential use case.
4. **Validate Use Cases:** Review the identified use cases with stakeholders to ensure they accurately capture the required functionalities and interactions.

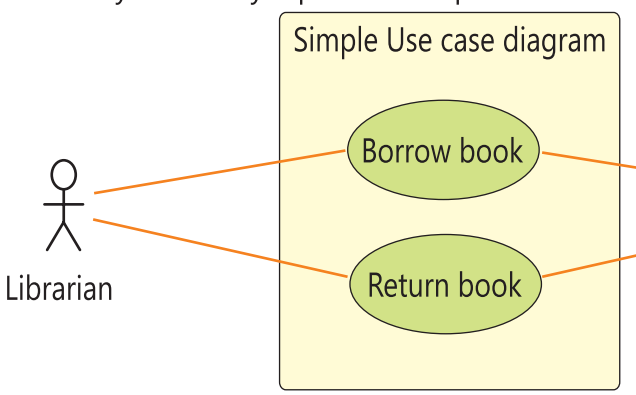


Figure 1.5: Example Use Case Diagram for a Library System

Class Activity

Statement: Imagine you are designing an online shopping platform. The platform allows customers to browse products, add items to their cart, and make purchases. Additionally, the platform includes features for administrators to manage product listings, process orders, and handle customer inquiries. There is also a feature for delivery personnel to update the status of deliveries.

In the above class activity, you can compare your findings with the following:

- **Actors:**
  - Customer
  - Administrator
  - Delivery Personnel
- **Use Cases:**
  - Browse Products
  - Add Items to Cart
  - Make Purchase
  - Manage Product Listings
  - Process Orders
  - Handle Customer Inquiries
  - Update Delivery Status

1.5.2.2 Class Diagram

A class diagram is like a map that shows how things are organized in a system.

Example:

In the example of organizing your room as shown in Figure 1.6:

- **Room:** Represents the overall space encompassing all other elements, analogous to the main structure in a class diagram.
- **Box:** Serves as a container within the room, akin to a class in a diagram.
- **Attributes:** Each box contains specific items, such as a 'ToyBox' holding toys or a 'BookBox' containing books.
- **Methods:** Boxes can perform actions like 'open' or 'close,' similar to methods in a class diagram that define what the box can do.
- **Specific Boxes:** Examples of specialized boxes include a 'ToyBox' for toys, a 'BookBox' for books, and a 'ClothesBox' for clothes, representing distinct instances of the general 'Box' class.

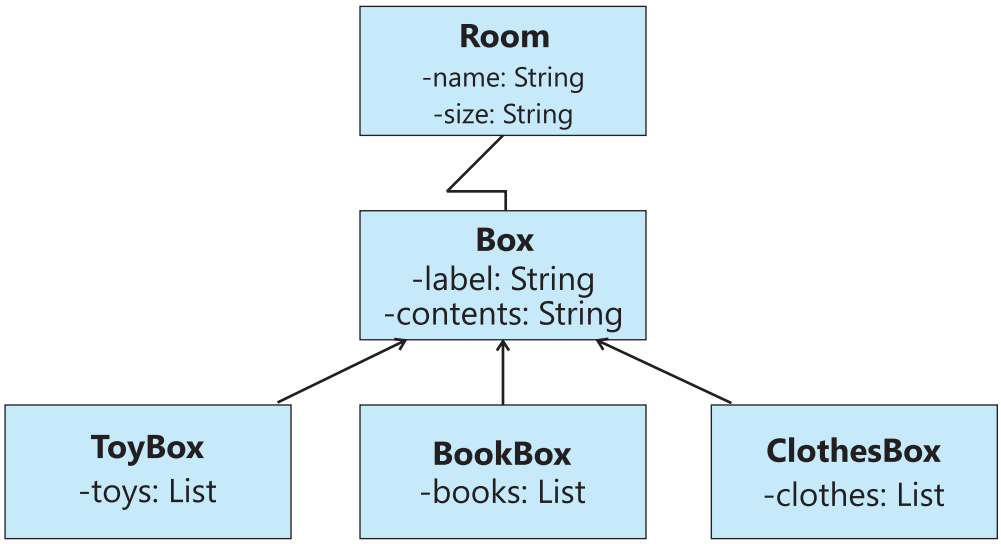


Figure 1.6: Class Diagram for Organizing Your Room

1.5.2.3 Sequence Diagrams

Sequence Diagrams show how objects in a system interact with each other in a particular sequence. They help in understanding the flow of messages between objects over time.

Interactions:

- **open():** User opens each box.
- **put toys/books/clothes inside:** User puts the respective items into the boxes.
- **close():** User closes each box.

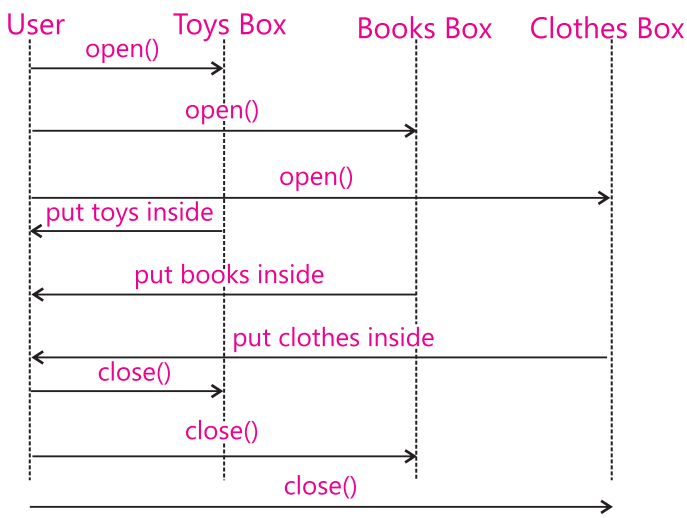


Figure 1.7: Sequence diagram of the user organizing items into labeled boxes

1.5.2.4 Activity Diagrams

Activity Diagrams illustrate the flow of activities or steps in a process. They are useful for modeling the logic of complex operations.

Example: In a restaurant management system, an activity diagram can represent the process from 'Order Placement' to 'Food Preparation' and finally to 'Order Delivery'.

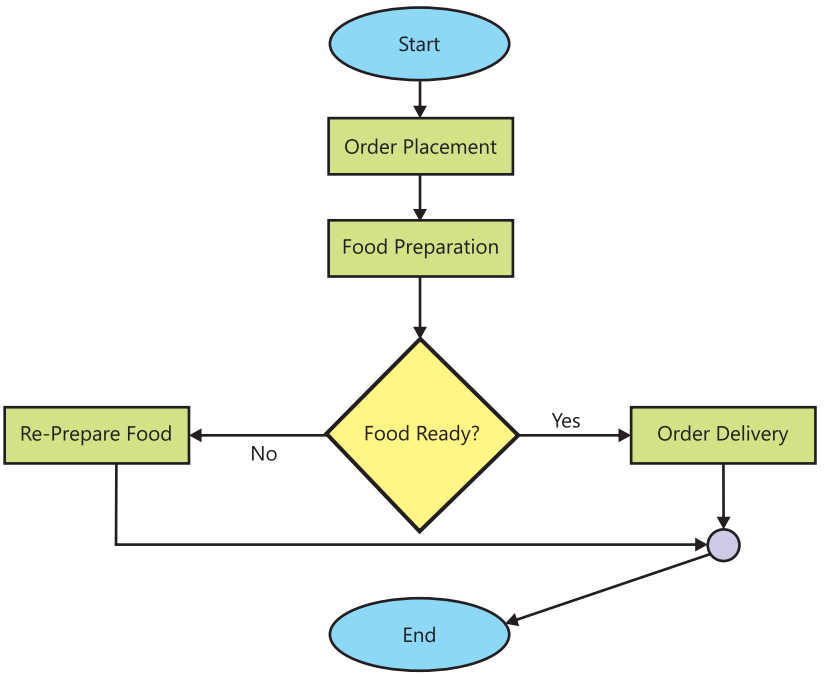


Figure 1.8: Activity Diagram with Decision and Connector Symbol

1.5.3 Using UML to Represent Software Systems

UML can be used in various stages of software development to improve understanding and communication. Here are some practical applications:

- **Planning:** Use UML diagrams to map out the system's requirements and design before writing any code.
- **Development:** Developers refer to UML diagrams to understand the structure and relationships within the system.
- **Communication:** UML diagrams help team members, including non-technical stakeholders, to understand how the system works.

1.6 Introduction to Design Patterns

Design pattern are common solutions to problem in software development, they act like templates to help make coding easier, faster and more consistent.

1.6.1 Commonly Used Design Patterns

Below are some of the most widely recognized design patterns:

1.6.1.1 Singleton Pattern

The Singleton Design Pattern is a way to make sure that a specific object or resource is created only once in a program and reused whenever needed.

1.6.1.2 Factory Pattern

The Factory Design Pattern is like having a special workshop that knows how to create different products, but you don't need to worry about the details of how those products are made. Instead, you just tell the factory what you need, and it gives you the finished product.

1.6.1.3 Observer Pattern

The Observer Design Pattern is like having a group of people who are interested in getting updates from one particular source. Whenever something important happens, the source automatically notifies all the interested people. It's a way to keep things in sync without everyone constantly checking for updates.

1.6.1.4 Strategy Pattern

The Strategy Design Pattern is like having a toolbox full of different tools, each designed for a specific job. When you face a problem, you can pick the right tool from the box based on the task at hand.

Class Activity

Identify a real-world scenario around you where you can apply one of these design patterns. Share your examples in the next class.

1.6.2 Applications of Design Patterns in Software Design

Design patterns are widely used in software development to solve common problems and create robust and maintainable code. They help in:

- Reducing code complexity by providing a clear structure.
- Enhancing code reusability by using proven solutions.
- Improving communication among developers by providing a common vocabulary.

Design patterns help create systems that are flexible, maintainable, and easy to understand.

Many popular software frameworks and libraries are built using design patterns. For example, the Model-View-Controller (MVC) pattern is used in web development frameworks like Ruby on Rails and Angular.

1.7 Software Debugging and Testing

Debugging and testing are important steps to make sure that software works correctly. They help find and fix errors so the software meets requirements and run as expected.

1.7.1 Debugging

Debugging is the process of finding and fixing bugs or errors in a software.Bugs are errors or mistakes in the software that cause it to behave unexpectedly. Identifying bugs involves observing the software's behavior and finding the source of the problem. Once identified, bugs requires making changes to the code to correct the error.

Tools and Best Practices

There are various tools and best practices for debugging, including:

- **Debuggers:** Software tools that help programmers find bugs by allowing them to step through code, inspect variables, and monitor program execution.
- **Print Statements:** Adding print statements in the code to display the values of variables at different points in the program.
- **Code Reviews:** Having other developers review your code to spot potential errors.

1.7.2 Testing

Testing is the process of evaluating the software to ensure it meets the requirements and works as expected. The testing process typically follows a hierarchy that begins with smaller components and gradually progresses to the entire system, including user acceptance. The main types of testing in this hierarchy are given below.

1.7.2.1 Unit Testing

Unit Testing is the first level of testing, where individual components or modules of the software are tested in isolation. Each "unit" is a small, testable part of the software, such as a function or method. The primary goal of unit testing is to verify that each component works correctly according to its design and performs as expected.

Class Activity

Try writing a unit test for a simple function in your favorite programming language.

1.7.2.2 Integration Testing

After unit testing, Integration Testing is performed to evaluate the interaction between different components or modules. While unit testing focuses on isolated units, integration testing ensures that these units work together correctly when combined.

This type of testing checks for interface errors, data flow between modules, and other integration-related issues.

1.7.2.3 System Testing

System Testing is a higher level of testing where the entire software system is tested as a whole. At this stage, the software is treated as a complete entity, and testers evaluate its overall functionality, performance, security, and compliance with specified requirements.

1.7.2.4 Acceptance Testing

Acceptance Testing is conducted to determine whether the software is ready for release. It is often performed by the end-users or clients to ensure that the software meets their expectations and requirements.

Acceptance testing is sometimes called User Acceptance Testing (UAT) because it is often done by the end-users of the software.

1.8 Software Development Tools

Software development tools are programs or applications that assist in various stages of software creation. They are used to write, edit, test, debug, and manage code, ensuring that software functions correctly and efficiently.

1.8.1 Language Editors

Language editors, also known as code editors, are tools that help developers write and edit code in different programming languages. Examples include:

- **Notepad++:** A simple yet powerful code editor.
- **VS Code:** A popular editor with many extensions.

1.8.2 Translators

Translators are tools that convert code written in one programming language into another language that the computer can understand. Translators convert high-level programming languages (like Python) into machine language (binary code) that computers can execute. It has two types:

- **Interpreters:** Translate code line-by-line (e.g., Python interpreter).
- **Compilers:** Translate the entire code at once (e.g., GCC for C/C++).

1.8.3 Debuggers

Debuggers are tools that help developers find and fix errors (bugs) in their code. The purpose of debuggers is to allow developers to test their code and identify where errors occur. Examples include:

GDB: GNU Debugger for C/C++.

Visual Studio Debugger: Integrated with Visual Studio IDE.

1.8.4 Integrated Development Environments (IDEs)

IDEs are comprehensive software suites that provide all the tools needed for software development in one place. IDE integrates various development tools like editors, compilers, debuggers, and version control systems to streamline the development process. An IDE offers a unified interface where developers can write, test, and debug their code efficiently. Examples include:

- **Visual Studio:** Popular for .NET and C++ development.
- **PyCharm:** Preferred for Python development.

1.8.5 Online and Offline Computing Platforms

These platforms provide environments where developers can write, run, and test their code.

- **Online Platforms:** Cloud-based platforms accessible via the internet (e.g., Repl.it, Gitpod).
- **Offline Platforms:** Local development environments on a computer (e.g., local installations of IDEs).

1.8.6 Source Code Repositories

Source code repositories are platforms where developers can store, manage, and track changes to their code. Repositories help in version control, allowing multiple developers to work on the same project without conflicts. Examples include:

- **GitHub:** Popular platform for open-source projects.
- **Bitbucket:** Used for both private and public repositories.

EXERCISE

Q.1: Multiple Choice Questions

1. Primary purpose of the Software Development Life Cycle (SDLC) is to:
   a) design websites
   b) deliver high-quality software within time and cost estimates
   c) manage database systems
   d) create hardware components
2. A type of requirement specifying system performance:
   a) Functional Requirements
   b) Non-Functional Requirements
   c) Technical Requirements
   d) Operational Requirements
3. Role of a framework in the context of SDLC is to:
   a) write code from scratch
   b) provide a structured foundation with predefined components and architectures
   c) manage hardware
   d) perform manual testing
4. Software development model involving short cycles or sprints:
   a) Waterfall Model
   b) Agile Methodology
   c) Lean Software Development
   d) Scrum
5. Crucial aspect of comprehensive project planning:
   a) Understanding the project scope and tasks
   b) Deciding the project's colour scheme
   c) Hiring a large development team
   d) Ignoring potential risks
6. Factor that does not influence cost estimation of a software project:
   a) Scope of the project
   b) Technology stack
   c) Number of meetings held
   d) Operational costs
7. The purpose of Use Case Diagrams is to:
   a) document the system's architecture
   b) identify and document the system's functional requirements
   c) illustrate the database schema
   d) define the system's user interface design

Short Questions

1. Differentiate between functional and non-functional requirements.
2. Explain why the testing phase is important in the Software Development Life Cycle (SDLC), and provide two reasons for its significance.
3. Illustrate the concept of continuous integration in Agile Methodology and

discuss its importance in software development.

4. Evaluate the main steps involved in risk assessment and management, and assess their importance in a software project.
5. Explain the purpose of a Use Case Diagram in software development.
6. Compare and contrast a Sequence Diagram with an Activity Diagram, highlighting the key differences.
7. Describe the Factory Pattern and explain how it differs from directly creating objects, with an example.

Long Questions

1. Design a flowchart for a user registration process in a software application. Outline its key steps.
2. Imagine you are managing a project to develop a simple mobile application. Describe how you would use the Agile Methodology to handle this project.
3. Consider an online banking system. Create a Use Case Diagram to show the interactions between customers, bank staff, and the system.
4. You are developing a food delivery application. Create a Sequence Diagram to show the process of placing an order, from the customer selecting items to the delivery of the order.
5. Discuss the importance of software development tools in the software development process.
   a) Explain the role of language editors, translators, and debuggers in creating and maintaining software.
   b) Provide examples of each tool and describe how they contribute to the efficiency and accuracy of software development.

UNIT 2

Python Programming

Student Learning Outcomes

By the end of this chapter, students will be able to:

- Understand basic programming concepts and set up a Python development environment.
- Write and interpret basic Python syntax and structure, including variables, data types, and input/output operations.
- Use various operators and expressions in Python, including arithmetic, comparison, and logical operators.
- Implement control structures such as decision-making statements and loops in Python.
- Work with Python modules, functions, and built-in data structures like lists.
- Apply modular programming techniques and object-oriented programming concepts in Python.
- Handle exceptions, perform file operations, and apply testing and debugging techniques in Python.

Introduction

Python is a popular and easy to learn programming language. In this unit, you will learn the basics, setup tools and explore key components. Later, we will learn advanced topics like file handling, debugging and data structure.

2.1 Introduction to Python Programming

Python is a versatile and applicable to various fields, including web development, data analysis, artificial intelligence, and more. Python's straightforward syntax and clear structure make it an excellent choice for beginners, allowing them to focus on learning programming concepts rather than dealing with complex syntax rules.

2.1.1 Understanding Basic Programming Concepts

Computer programming is the process of creating a set of instructions that tell a computer how to perform a task. These instructions are written in a programming language that the computer can understand and execute.

2.1.1.1 Programming Basics

Computer programming involves the following basic steps to write a program.

1. **Write Code:** Create a set of instructions in a programming language.

2. **Compile/Interpret:** Translate the code into a form that the computer can understand.
3. **Execute:** Run the code to perform the task.
4. **Output:** Display the results or perform actions based on the code.

2.1.1.2 Setting Up Python Development Environment

The development environment refers to the process of preparing a computer to write, run, and debug Python code effectively. This involves installing and configuring the necessary software, tools, and libraries. We can download and install Python from https://www.python.org/. When starting with Python programming, choosing a good Integrated Development Environment (IDE) can help make coding easier.

Tidbits

When installing Python, make sure to check the box that says "Add Python to PATH." This makes it easier to run Python from the command line. We can also use online services to write and run Python program.

2.2 Basic Python Syntax and Structure

The following Python program demonstrates the simplicity and readability of the language:

```
print("This is my first page")
```

In this example, the print function is utilized to output the message enclosed in double quotation marks. This illustrates Python's straightforward syntax, where function like print is used to perform actions such as displaying text.

Python Comments

Lines that are not executed by the Python interpreter. They are used to provide explanations or notes for the code. Single-line comments start with the # symbol while multi-line comments can be created using triple quotes (''') at the beginning and the end as shown below.

```
# This is a single-line comment
print ( "K2 is the second-highest mountain in the world " )
'''
This is a multi-line comment.
It can span multiple lines.
'''
print ("Edhi Foundation is the largest volunteer ambulance network." )
```

2.2.1 Variables, Data Types and Input / Output

2.2.1.1 Variable

A variable is a storage container in a computer's memory, that allows storage, retrieval and manipulation of data. The value of a variable can change throughout the execution of

a program:

```
age = 71
print( "Ahmad lived for", age, "years")
age =60
print ( "Iqbal lived for", age, "years")
```

2.2.1.2 Variable Naming Rules in Python

Variable names in Python must adhere to the following rules:

- The name must begin with a letter (a-z, A-Z) or an underscore (_).
- Subsequent characters can include letters, digits (0-9), or underscores (_).
- Variable names are case-sensitive, meaning age and Age are considered two different variables.
- Python's reserved keywords, such as for, while, if, etc., cannot be used as variable names.

Tidbits

Always use meaningful names for variables to make your code easier to understand. For example, use age instead of a.

2.2.1.3 Creating Different Types of Variables

In Python, you can create variables of different types to store various kinds of data. Here are some common types of variables:

- **Integer (int):** Stores whole numbers. Example: age = 17
- **Floating-point (float):** Stores decimal numbers. Example: price = 19.99
- **String (str):** Stores text. Example: name = "Ali"
- **Boolean (bool):** Stores True or False. Example: is_student = True

Tidbits

- It's a good practice to use lowercase letters for variable names and underscores to separate words in variable names (e.g., student_name).

2.2.1.4 Input and Output Operations

Input and output operations allow you to interact with the user. You can ask the user to enter data (input) and display information to the user (output).

- **Input:** Use the input () function to get user input. The input () function displays a message on the screen and waits for the user to type something and press Enter. The text entered by the user is then stored in a variable. For example:

```
name = input("Enter your name: ")
```

- **Output:** Use the print () function to display information on the screen. The print () function takes one or more arguments and displays them. For example:

```
print ("Hello , " + name + "!")
```

2.2.1.5 Handling Integer and Float Inputs

To handle numeric inputs, you typically use the int() or float() functions to convert input strings to integers or floating-point numbers, respectively.

Integer Inputs

```
# Example : Handling integer input
 user_age = int(input("Ent er your age: "))
 print("Your age is:",user_age)
```

Float Inputs

```
 # Example: Handling float input
user_height = float(input("Enter your height in meters: ") )
print("Your height is", user_height,"meter")
```

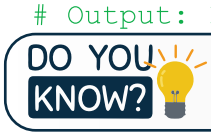
2.3 Operators and Expressions

Operators are symbols that perform operations on variables and values. An expression is a combination of variables, operators, and values that produces a result.

2.3.1 Arithmetic Operators

Arithmetic operators are used to perform basic mathematical operations such as addition, subtraction, multiplication, division, modulus, exponentiation, and floor division as shown in the following code.

```
# Define variables
a= 10, b= 3
# Perform all arithmetic operations
print(a, "+", b, "=", a + b) # Output: 10+3=13
print(a, "*", b, " = " , a * b) # Output: 10 * 3 = 30
print(a, "/", b, " = ", a / b)
  # Output: 10 / 3 = 3.3333333333333335
  print(a, "//", b, " = ", a // b)
  # Output: 10 // 3=3
  print(a, "%", b, " = ", a % b)
  # Output: 10 % 3 = 1
  print(a, "**", b, " = ", a ** b)
  # Output: 10**3= 1000
```



A tutorial on Python is available at
https: //docs.python.org/3/tutorial/

2.3.2 Comparison Operators

Comparison operators are used to compare two values or expressions. They determine the relational logic between them, such as equality, inequality, greater than, less than, and so on. These operators return a boolean value (True or False) based on the comparison result.

```
# Define variables
x, y = 10, 5
# Greater than
print (x, ">", y, "=", x > y) # Output: 10 > 5 = True
# Less than
print (x, "<", y, "=", x < y) # Output: 10 < 5 = False
# Equal to
print (x, "==", y, "=", x == y) # Output: 10 == 5 = False
# Not Equal to
print (x, "!=", y, "=", x!= y) # Output : 10 != 5 = True
# Greater than or equal to
print (x, ">=", y, "=", x >= y) # Output: 10 >= 5 = True
# Less than
print (x, "<=", y, "=", x <= y) # Output: 10 <= 5 = False
```

2.3.3 Assignment Operators

Assignment operators are used to assign values to variables. The most common assignment operator is the equal sign (=), which assigns the value on the right to the variable on the left. There are also compound assignment operators like +=, -=, *=, and /=, which combine arithmetic operations with assignment.

```
# Define initial values
a = 10
b = 5
# Assignment
assignment = a; print ("a = ", assignment) # Output: a = 10
# Addition assignment
a +=b; print ("a after addition =", a) # Output: a = 15
# Subtraction assignment
a -=b; print ("a after subtraction =", a) # Output: a = 5
# Multiplication assignment
a *=b; print ("a after multiplication =", a) # Output: a = 50
# Division assignment
a /=b; print ("a after division =", a) # Output: a = 2.0
# Modulus assignment
a %=b; print ("a after modulus division =", a) # Output: a = 2
# Exponentiation assignment
a **=b; print ("a after Exponentiation =", a) # Output: a = 100000
```

2.3.4 Logical Operators

Logical operators are used to combine multiple conditions or expressions in a program. The most common logical operators are *and, or* and *not*. They are used to perform

logical operations and return boolean values based on the evaluation of the expressions involved.

```
# Define variables
x = True
y = False
# Logical AND
logical_and = x and y
print(x, "and ", y, "=", logical_and) # Output: True and False =False
# Logical OR
logical_or = x or y
print(x, "or ", y, " =", logical_or) # Output: True and False = True
# Logical NOT
logical_not_x = not x
print(x, "not ",x, " =", logical_not) # Output: Not x = False
```

2.3.5 Expressions

An expression is a combination of variables, operators, and values that produces a result. For example, 3 + 4 is an expression that results in 7. More complex expressions can use parentheses ( ) to control the order of operations. For example:

```
result = (3 + 4) * 2      # result is 14
```

Class Activity

Write a program to calculate Body Mass Index (BMI). Ask the user for their weight and height, then compute and display their BMI and classification. The Body Mass Index (BMI) is calculated using the formula given below.

$$BMI = \frac{weight}{height}$$

where:

- weight is in kilograms (kg)
- height is in meters (m)

2.3.6 Operator Precedence in Python

Operator precedence determines the order in which operations are performed in an expression. In Python as well as in Mathematics, certain operators have higher precedence and are evaluated before others.

- **Parentheses '()':** Highest precedence. Operations inside parentheses are performed first. (3 + 2) * 4 evaluates to 20.
- **Exponentiation:** Performs power operations next. 2^3 evaluates to 8.

- **Multiplication '*', Division '/', and Modulus '%':** These operations come next. 4*3 evaluates to 12, 10/2 evaluates to 5.0 and 11%3 evaluates 2.
- **Addition '+' and Subtraction '-':** These have lower precedence compared to multiplication and division.
  5 + 2 evaluates to 7, and 10-4 evaluates to 6.

Class Activity

Compute the following expressions and compare results with your class fellows and class teacher.

1. 10 + 3*2 **2-5/5
2. (10 + 3) * (2 ** (2 - 1)) / 5

Using parentheses can help clarify complex expressions and ensure the operations are performed in the desired order.

2.4 Control Structures

In programming, we often need to control the flow of our program based on different conditions or repeat certain actions multiple times. There are two main types of control structures, Decision Making and Looping:

2.4.1 Decision Making

Decision making in programming allows the program to choose different actions based on conditions. Python provides a variety of conditional statements to implement decision making.

2.4.1.1 if Statement

The if statement allows us to make decisions based on conditions. If the condition is true, it runs a block of code.

```
#  Syntax of if statement if condition:
if condition:
#  code to run if the condition is true
```

Example: If the temperature is above 30 degrees, we print a message.

```
temperature = 35
if temperature > 30:
    print("It is a hot day")
```

2.4.1.2 if-else Statement

The if-else statement allows us to execute one block of code if a condition is true and another block if the condition is false.

```
# Syntax of if-else statement if condition:
if condition:
    # code to run if the condition is true else :
```

else:

```
    # code to run if the condition is false
```

Example:

```
temperature = 15
if temperature > 30:
        print ("It's a hot day")
else:
        print ("It's not a hot day")
```

2.4.1.3 Short Hand if-else Statement

Python also allows a short-hand if-else statement that can be written in a single line.

```
# Syntax of short hand if-else statement
```

action_if_true *if* condition *else* action_if_false

```
temperature = 15
m = "It's a hot day" if (temperature > 30)
else  "It's not a hot day"
print(m)
```

Class Activity

Write an if-else statement and a short-hand if-else statement to check if a number is even or odd and print the appropriate message.

2.4.1.3 if-elif-else Statement

The if-elif-else statement allows us to check multiple conditions and execute different blocks of code for each condition.

```
# Syntax of if-elif-else statement
if condition1:
    # code to run if condition1 is true
elif condition2:
    # code to run if condition2 is true
else:
    # code to run if none of the conditions are true
```

Example:

```
weather = "cloudy" # The output depends on the value stored in the variable weather"
if weather == "sunny":
    print("Wear sunglasses")
elif weather == "rainy":
    print("Take an umbrella")
else :
    print("Enjoy your day!")
```

Class Activity

Write an if-elif-else statement to check if a number is positive, negative, or zero.

2.4.2 Looping Constructs

Loops help us repeat actions, making our code more efficient and easier to read. There are two main types of loops in Python: while loops and for loops.

2.4.2.1 while Loop

A while loop runs as long as a condition is true. It checks the condition before each iteration and stops running when the condition is no longer true.

```
# Syntax of while loop while condition:
# code to run while the condition is true
```

Example: Add 1 to a number until it reaches 10.

```
number = 1
while number < 10:
    print(number)
    number += 1
```

Class Activity

Write a Python program that prints even and counts the odd numbers from 1 to 20 using a while loop.

2.4.2.2 for Loop

A for loop repeats a block of code a specific number of times. It is commonly used to iterate over a sequence (like a list, tuple, or string).

```
#Syntax of for loop
for variable in sequence:
#code to run for each element in the sequence
```

Example 1: Say "Hello" to each friend in a list of friends.

```
friends = ["Ahmad", "Ali", "Hassan"]
for friend in friends:
    print("Hello", friend)
```

Explanation: In this example, the code goes through each friend in the list and prints a greeting message for each one.

Class Activity

1. Write a for loop using range() to print the even numbers from 2 to 10.
2. Write a Python program that prints the first 10 multiples of 3 using a for loop and the range() function.

2.5 Python Modules and Built-in Data Structures

Python offers an extensive standard library that includes numerous built-in modules and data structures. A data structure refers to a particular format or method for organizing and storing data. For example, a list is a data structure that we have previously utilized. In this section, we will examine the utilization of functions, modules, and libraries within Python.

2.5.1 Functions and Modules

Functions and modules in Python are key to writing efficient and organized code. Functions allow you to encapsulate reusable blocks of code, while modules help you structure your program by grouping related functions together.

2.5.1.1 Defining and Invoking Functions

Functions are defined using the def keyword, followed by the function name and parentheses which may include parameters. The body of the function contains the code to be executed and must be indented.

def function_name (parameters):

code to be executed

Example: Define a function to greet a person.

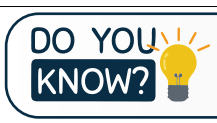
```
def greet(name):
    print("Hello", name)
# Function invoking means call the function by name and
perform the required task For example.
greet ('Ali')
```

2.5.1.2 Function Parameters and Return Values

Functions can take multiple parameters and return values.

Example: Define a function to add two numbers.

```
def add (a , b) :
    return a + b
```



You can call a function multiple times with different arguments to reuse the same code for different inputs.

2.5.1.3 Default Parameters

Functions can have default parameter values, which are used if no argument is provided during the function call.

Example: Define a function with a default parameter.

```
def greet(name = "Student") :
    return "Hello"+ name +"!"
print(greet( ))  # Output: Hello ,Student!
print(greet("Umer "))  # Output: Hello, Umer!
```

Class Activity

Define a function that takes a list of numbers and returns the maximum value.

2.5.2 Using Libraries and Modules

In Python, libraries and modules are like toolboxes, full of useful tools that help you solve different problems without having to build everything from scratch. In this section, we will explain how to import and use both standard and third-party libraries in your Python programs.

2.5.3 Importing and Using Libraries

Libraries are like pre-built toolkits that you can use without having to write all the code yourself.

Example: Import the random library to generate random numbers.

```
import random
# Generate a random number between 1 and 10
number = random.randint(1, 10)
print("The random number is:", number)
```

```
Import datetime
# Get the current date and time
current_time = datetime.datetime.now()
print("Current date and time:", current_time)
```

```
import statistics
# Calculate the mean of a list of numbers
data = [23, 45, 67, 89, 12, 44, 56]
mean_value = statistics.mean(data)
print("The mean value is:", mean_value)
```

2.5.3.1 Package Structure

To manage large projects, you can organize modules into packages. A package is simply a directory containing related modules. For example, if you're building an e-commerce platform, you could create a package named ecommerce with modules like products .py, customers .py, and orders.py.

Example: In ecommerce/products.py:

```
def List_products () :
        return ["Laptop","Mobile", "Tablet"]
```

In your main Script

```
from ecommerce import products
available_products = products.List_products()
print(available_products)
#    Output :
#    ['Laptop', 'Mobile', 'Tablet']
```

Explanation: In this case, ecommerce is the package, and products.py is the module. This structure helps you keep your code organized and manageable.

Tidbits

Organizing your modules into packages is like organizing books into sections of a library—it makes finding and maintaining your code much easier.

2.6 Built-in Data Structures

Python provides several built-in data structures that are essential for organizing and manipulating data efficiently. These include lists, tuples, and dictionaries, each offering unique features to handle various types of data and perform common operations.

2.6.1 Lists

In Python, a list is a versatile data structure that can hold a collection of items. You can create, access, and modify lists easily.

2.6.1.1 Creating, Accessing, and Modifying Lists

A list is created by placing items inside square brackets [ ], separated by commas. Lists can contain items of different types, such as numbers, strings, or even other lists.

Example: Create a list of your favorite fruits.

```
fruits = ["Mango", "Apple" "Banana"]
print(fruits)
# Output: ['Mango', 'Apple ', 'Banaṅ ]
```

2.6.1.2 Accessing List Items

You can access items in a list by referring to their index, starting from 0. Example: Access and print the second item from the list of fruits.

```
fruits = ["Mango", "Apple", "" Banana"
print(fruits [1])
# Output: Apple
```

Explanation: The code initializes a list 'fruits' containing 'Mango', 'Apple', and 'Banana', then prints the second item, 'Apple', using the index ' 1 '.

2.6.1.3 Modifying a List

You can modify list items by accessing them via their index and assigning a new value.

Example: Change the first item in the list to "Orange" and add a new fruit "Pineapple".

```
fruits = ["Mango", "Apple", "Banana"]
fruits [0] = "Orange"
fruits.append("Pineapple")
print(fruits)
# Output: ['Orange', 'Apple', 'Banana', 'Pineapple']
```

Explanation: The code modifies the first element of the 'fruits' list to 'Orange', appends 'Pineapple' at the end, and prints the updated list.

2.6.1.4 Methods and Operations on Lists

Python provides several built-in methods to work with lists. Here are a few useful ones:

- append (item) - Adds an item to the end of the list.
- remove (item) - Removes the first occurrence of an item from the list.
- sort () - Sorts the list in ascending order.
- reverse () - Reverses the order of the list.

Example: Add a new student to the list of students and then sort the list.

```
students = ["Ahmed", "Sara", "Ali"]
students.append("Hina")
students.sort ()
 print(students)
  # Output : ['Ahmed', 'Ali', 'Hina', 'Sara']
```

Explanation: The code creates a list of students, adds 'Hina' to it, and sorts the list alphabetically.

2.6.1.5 List Operations

Lists also support various operations, such as slicing and concatenation.

Example: Slice a portion of the list and concatenate it with another list.

```
numbers = [1, 2, 3, 4, 5]
slice = numbers [1:4]  # Gets items from index 1 to 3
extra_numbers = [6, 7]
combined = slice + extra_numbers
print(combined)
# Output : [2, 3, 4, 6, 7]
```

Explanation: The code slices the 'numbers' list from index 1 to 3, combines it with 'extra_numbers', and prints the resulting list '[2, 3, 4, 6. 7]'.

Example: Sort a list of student names and remove a specific name.

```
student _names= ["Ahmed", "Sara", "Ali", "Hina"]
student_names.sort ()
student_names.remove("Sara")
print(student _names)
# Output: ['Ahmed ' , ' Ali', 'Hina ' ]
```

Explanation: The code sorts the list 'student_names' alphabetically, removes 'Sara' from the list, and then prints the updated list.

Class Activity

Imagine you are maintaining a list of your favorite books: ["To Kill a Mockingbird", "1984", "The Great Gatsby", "Pride and Prejudice"]. Perform the following tasks using Python:

1. Add a new book "Moby Dick" to the list.
2. Replace "1984" with "Brave New World".
3. Remove "The Great Gatsby" from the list.
4. Merge this list with another list of books: ["War and Peace", "Hamlet"].
5. Print the final list of books.

Tidbits

Use list methods like append() and remove () to efficiently manage and modify your lists. For larger projects, organizing data in lists helps keep your code clean and manageable.

2.6.2 Tuples

In Python, tuples are a type of data structure used to store an ordered collection of items, similar to lists, but with a key difference: tuples are immutable, meaning their values cannot be changed after creation.

Example

```
# Creating a tuple
my_tuple = (1, 2, 3, "Hello", 4.5)
# Accessing elements by index
print(my_tuple[0])  # Output: 1
print(my_tuple[3])  # Output: Hello
# Tuple length
print(len(my_tuple))  # Output: 5
```

2.6.3 Indexing and Slicing

Indexing and slicing are essential techniques in Python for accessing and manipulating sequences such as lists, tuples, and strings.

2.6.3.1 Indexing

Indexing allows you to access individual elements in a sequence. Python uses zero-based indexing, meaning the first element has an index of 0, the second element has an index of 1 and so on.

2.6.3.2 Slicing

Slicing allows you to access a subset of a sequence. The syntax for slicing is sequence [start: stop: step], where start is the starting index, stop is the ending index (not inclusive), and step is the step size.

2.6.3.3 Indexing and Slicing with Negative Indices

Negative indices count from the end of the sequence. For example, -1 refers to the last element, -2 refers to the second last element, and so on.

Example: Indexing and slicing with both positive and negative indices on a list.

```
# Create a list of fruits
fruits = ["Apple", "Banana", "Cherry", "Date", "Elderberry " ]
# Indexing
print("First fruit:", fruits [0])   # Positive index
print("Last fruit:", fruits [-1])   # Negative index
# Slicing with positive indices
print("Fruits from index 1 to 3:", fruits[1:4])
# Slicing with negative indices
print("Fruits from index -4 to -1:", fruits [-4 : - 1] )
```

Explanation: This code demonstrates list operations in Python: creating a list of fruits, accessing elements using positive and negative indexing, and slicing the list with both positive and negative indices.

Class Activity

Consider the following list, tuple, and string:

```
# List: [10, 20, 30, 40, 50, 60, 70, 80]
# Tuple: ("Math", "Science", "English", "History", "Geography")
# String: "Python Programming"
```

Perform the following operations:

1. Access and print the third element from each sequence (list, tuple, and string).
2. Slice and print elements from index 2 to 5 from the list and the tuple.
3. Slice and print characters from index 7 to the end of the string.
4. Use negative indexing to print the last two elements from the list and the tuple.
5. Use negative slicing to print characters from the second last to the last character of the string.

Write the Python code to perform these operations and print the results.

Tidbits

Indexing and slicing are powerful tools for working with sequences in Python. Practice these techniques to become more proficient in manipulating data and accessing specific parts of sequences.

2.7 Modular Programming in Python

Modular programming is a technique used to divide a program into smaller, manageable, and reusable pieces called modules. By breaking a program into modules, developers can work on different parts independently and reuse code efficiently. This approach simplifies managing complex programs and promotes code reuse.

The main Function

The main function in Python defines where the program should start. It's usually placed in a block that checks if the script is being run directly or imported as a module.

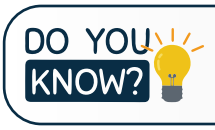
Example: Here's a simple example:

```
# main.py
def main ( ) :
    print("This is the main function.")
if __name__ == "__main__" :
   main()
```

Explanation: In this example, the main() function will only run if the script is executed directly, not when it's imported elsewhere. This setup is useful in larger projects that have multiple modules.

Tidbits

Using the main function with modules helps keep your code organized, making it easier to maintain. Always use the main function to define the starting point of your program, and use modules to separate different parts of your code.



Python's standard library is made up of hundreds of modules that you can be used to perform common tasks, like working with dates, generating random numbers, or reading files.

Class Activity

Create a Python module named calculator.py that includes two functions:

1. add (a, b) - This function should return the sum of two numbers.
2. subtract (a, b) - This function should return the difference between two numbers. Then, write a script named main.py that imports your calculator module and uses these functions to perform the following:
   1. Print the result of adding 15 and 8.
   2. Print the result of subtracting 10 from 25.

Make sure to run your main.py script and verify that the output is correct.

2.8 Object-Oriented Programming in Python

Object-Oriented Programming (OOP) is a way of designing and organizing code to make it easier to manage and understand.

2.8.1 Class and Objects

A class is like a template for creating things, and an object is an actual thing created from that template. Imagine you want to make a toy car. You first need a blueprint or a template that describes how the toy car should look and function. This template includes details like:

- Color
- Size
- Number of wheels
- Type of material

The template is not an actual toy car; it's just a plan and it represents a class. Using the template, you can create multiple toy cars. Each object is an instance of the class, meaning it follows the plan to have its own specific characteristics.

2.8.1.1 Defining Classes and Creating Objects

In programming, we use classes as concepts to define what an object should be like.

```
# Define a class called ToyCar
class ToyCar:
  # The _init_ method initializes the object with specific attributes
  def _ _init_ _(self, color, size, wheels):
      self.color = color       # Color of the toy car
      self.size = size         # Size of the toy car
      self.wheels = wheels     # Number of wheels in the toy car
  # Method to describe the toy car
  def describe(self):
    return f"This toy car is {self.color}, size {self.size}, and has {self.wheels}
wheels."
# Create objects of the ToyCar class
car1 = ToyCar("red", "small", 4)
car2 = ToyCar("blue", "large", 6)

# Print descriptions of the toy cars
print(car1.describe())
print(car2.describe())
```

Explanation:

Class Definition: The "ToyCar" class is like the template for making toy cars. It describes what attributes a toy car should have: color, size, and wheels.

Creating Objects: "carl" and "car2" are specific toy cars object created using the ToyCar template. Each has its own unique attributes.

Using Methods: The describe ( )method allows us to get a description of the toy car.

Self: self is a convention used in Object-Oriented Programming (OOP) to represent the instance of a class within its methods.

2.9 Advanced Python Concepts

Advanced Python concepts extend the foundational knowledge and empower programmers to handle more complex tasks effectively. This section covers key topics such as exception handling, which deals with managing errors gracefully, and file handling, which involves reading from and writing to files. Mastering these concepts is essential for developing robust and efficient Python applications.

2.9.1 Exception Handling

Exception handling is a mechanism to manage errors that occur during program execution. It allows a program to continue running or gracefully terminate if an error occurs, ensuring more robust and error-resilient code.

2.9.1.1 Try-Except Blocks

In Python, the try block lets you test a block of code for errors, and the except block lets you handle errors if occur.

Example:

```
Input a
try :
   result =10/a # This line creates error if the value of 'a' is 0
except ZeroDivisionError:
   print("You can't divide by zero!")
```

Explanation: In this example:

- The try block contains code that might cause an error.
- The except block catches the ZeroDivisionError and handles it by printing a message.

2.9.1.4 File Handling

File handling involves reading from and writing to files. It is essential for storing data persistently.

2.9.1.5 Opening, Reading, and Closing Files

To read a file, open it using the open() function, read its contents, and then close the file to free up resources.

```
# Open and read a
with open("example .txt", "r") as file:
content = file .read ()
print(content)
```

Explanation: In the above code:

- The with statement ensures that the file is properly closed after its suite finishes, even if an error occurs.
- The file is opened in read mode (r), read contents into content, and then printed.
- The file opened using 'with' is automatically closed.

2.9.1.6 Writing to Files

To write to a file, open it in write mode (w) and use the write () method. To append data, use append mode (a).

```
#  Writing to a file
with open("example.txt", "w") as file:
    file.write("As-Salaam-Alaikum, World!\n")
#  Appending to a file
with open("example.txt", "a") as file:
   file.write("Appending new line.\n")
```

Explanation: In the above code:

- The file is opened in write mode (w) to overwrite its contents and write new data.
- The file is opened in append mode (a) to add data without overwriting existing content.

2.10 Testing and Debugging in Python

In Python programming, testing and debugging are essential practices to ensure that your code works correctly and efficiently.

2.10.1 Testing

Testing is the process of running your code with various inputs to check if it behaves as expected. The goal is to find and fix any issues before the code is used in real-world applications.

2.10.1.1 Types of Testing

- Unit Testing: Tests individual parts of the code (like functions or classes) in isolation. Python's unittest module is commonly used for this.
- Integration Testing: Checks how different parts of the code work together.
- Functional Testing: Validates that the software behaves as expected from the user's perspective.
- Regression Testing: Ensures that new changes don't break existing functionality.

2.10.1.2 Debugging

Debugging is the process of finding and fixing errors (bugs) in your code. It involves identifying the root cause of problems and making the necessary changes.

2.10.1.3 Common Debugging Techniques

- Print Statements: Adding print statements to check the values of variables at different stages of the code.
- Debugging Tools: Using tools like pdb (Python Debugger) to step through the code, inspect variables, and understand the flow of execution.
- Error Messages: Reading and interpreting error messages to locate the source of the problem.

EXERCISE

Multiple Choice Questions

1. An action needed during Python installation to run from the command line easily:
   a) Uncheck "Add Python to PATH" b) Choose a different IDE
   c) Check "Add Python to PATH" d) Install only the IDE
2. A valid variable name in Python is:
   a) variable1 b) 1variable c) variable-name d) variable name
3. Output of the following piece of code is:

```
age = 25
print(" Age : ", age)
```

   a) Age: 25 b) 25 c) Age d) age
4. The operator used for exponentiation in Python is:
   a) * b) ** c) // d) /
5. A loop used to iterate over a collection such as lists is:
   a) while b) for c) do-while d) repeat
6. A range( )function used to generate a sequence of numbers:
   a) Generates a list of numbers
   b) Creates a sequence of numbers
   c) Calculates the sum of numbers
   d) Prints a range of numbers
7. A keyword used to define a function in Python:
   a) define b) function c) def d) func
8. The Output of the following code is:

```
temperature, humidity, wind_speed = 25, 60, 15
print("Hot and humid" if temperature > 30 and humidity > 50 else
"Warm and breezy" if temperature == 25 and wind_speed > 10 else
"Cool and dry" if temperature < 20 and humidity < 30 else
"Moderate")
```

   a) Hot b) Warm c) Cool d) Nothing
9. The operation used to combine two lists in Python:
   a) combine() b) concat() c) + d) merge()

Short Questions

1. Explain the purpose of using comments in Python code.
2. Describe the difference between integer and float data types in Python. Provide an example of each.
3. Define operator precedence and give an example of an expression where operator precedence affects the result.

4. How does the short-hand if-else statement differ from the regular if-else statement?
5. Explain the use of the range( ) function in a for loop.
6. Explain how default parameters work in Python functions.
7. Explain why modular programming is useful in Python.
8. Explain the difference between a class and an object in Python.

Long Questions

1. Evaluate the following Python expressions.
   (a) (18 / 3 +4 ** 2) - (2 * (7 - 3)) /(9 7, 4)
   (b) (25 + 3 * 4 ** 2 - 6) / (2 ** 3 + 1)- 7
   (c) (12 + 6 *(5-2)) ** 2 / ((4 ** 2 - 7) + 10)
   (d) 45 / (2 ** 2 + 3 * 4) + 8 * (7 - 3)
2. Translating the following mathematical expressions to Python syntax
3. (a) $5 \times (3 + 2^2)$
   $6-2x3$
   (b) $7 + 2^2$
4. Explain the concept of variables in Python.
5. Write a Python program that takes a number as input and checks whether it is positive, negative, or zero using an if-elif-else statement.
6. Write a Python program using a while loop that prints all the odd numbers between 1 and100. Also, count and print the total number of odd numbers.

UNIT 3

Algorithms and Problem Solving

Student Learning Outcomes

By the end of this chapter, students will be able to:

- Describe and categorize different types of computational problems.
- Explain the importance of algorithms in problem-solving.
- Apply the generate-and-test method to solve computational problems.
- Differentiate between solvable and unsolvable problems.
- Understand problem complexity and categorize problems into P, NP, NP-Hard, NP-Complete.
- Identify common computational problems like sorting and searching.
- Apply algorithm design techniques such as divide and conquer, greedy methods, and dynamic programming.
- Implement and compare algorithms such as Bubble Sort, Binary Search, BFS, and DFS.
- Evaluate algorithms in terms of efficiency and scalability.
- Develop algorithmic thinking to solve problems systematically.

Introduction

Understanding algorithms is essential not only for computer science but also for everyday problem-solving. We will start by learning what computational problems are and how to describe them clearly. Then, we will look at different types of algorithms and how they can help us solve various kinds of problems. We will also discuss how to measure the efficiency of algorithms to find the best solutions.

3.1 Understanding Computational Problems

A computational problem is a challenge that can be solved through a computational process, which involves using an algorithm, i.e., a set of step-by-step instructions that a computer can execute.

- **Input:** The data or information given to the algorithm at the beginning of the problem.
- **Process:** The steps or rules (i.e. the algorithm) that are applied to the input to generate the output.
- **Output:** The solution or result produced by the algorithm after processing the input.

3.1.1 Characterizing Computational Problems

To solve a problem computationally, we need to understand its characteristics. This involves identifying the inputs, the desired outputs, and the process needed to transform the inputs into outputs.

3.1.1.1 Classifying Computational Problems

Computational problems can be classified into different categories based on their characteristics and the methods required to solve them. Some common classifications include:

- **Decision Problems:** Problems where the output is a simple "yes" or "no".
- **Search Problems:** Problems where the task is to find a solution or an item that meets certain criteria.
- **Optimization Problems:** Problems where the goal is to find the best solution according to some criteria.
- **Counting Problems:** Problems where the objective is to count the number of ways certain conditions can be met.

3.1.1.2 Well-defined vs. ill-defined Problems

Problems can also be categorized based on how clearly they are defined:

- **Well-defined Problems:** These problems have clear goals, inputs, processes, and outputs. For instance, the problem of determining if a number is even, is a well-defined problem because it has a clear goal (determine if the number is even), clear input (a single integer), a clear process (check if the number is divisible by 2), and a clear output (even or odd) as shown in Figure 3.1.
- **Ill-defined Problems:** These problems lack clear definitions or may have ambiguous goals and requirements. For instance, consider a project aimed at "How to reduce poverty in Pakistan". This goal is vague and broad.

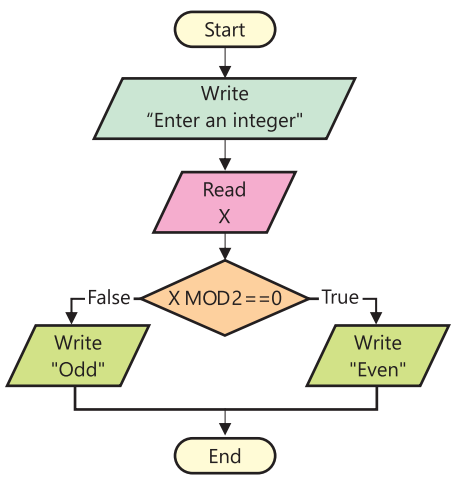


Figure 3.1: Finding an Even Number

3.2 Algorithms for Problem Solving

Algorithms are step-by-step procedures for solving problems, much like a recipe provides steps for cooking a dish. Understanding algorithms is essential because they provide the logic behind software operations, allowing us to solve complex problems, optimize performance, and ensure accuracy in various applications.

DO YOU KNOW?

The Google search engine uses a complex algorithm called PageRank to determine the relevance of web pages. This algorithm considers various factors, including the number of links to a page and the quality of those links, to rank pages in search results.

Tidbits

When learning about algorithms, try to relate them to real-life tasks you already know. This will help you understand how algorithms work and why they are important.

3.2.1 Generate-and-Test Method

This method works by generating potential solutions to a problem and then testing each one to determine if it meets the required conditions. The process continues until a satisfactory solution is found or all possible solutions have been exhausted.

The Generate-and-Test method is particularly useful in scenarios where:

- The problem space is small, making it feasible to generate and test all possible solutions.
- There is no clear strategy for finding a solution, and an exhaustive search is necessary.
- Heuristics or rules can be applied to reduce the number of generated solutions, making the process more efficient.

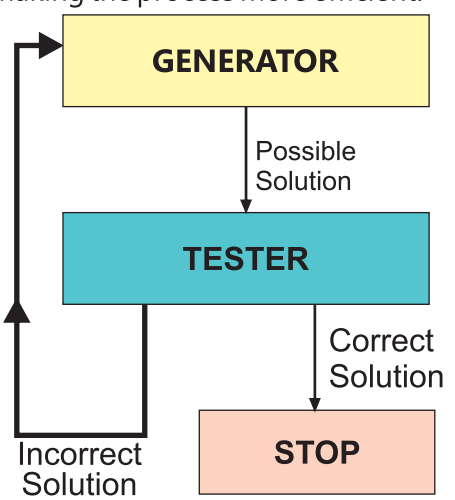


Figure 3.2: Flowchart of the Generate and Test Method

DO YOU KNOW?

The Generate-and-Test method is often used in AI applications, such as game playing and problem-solving, where the solution space is large, and the best approach is to try different possibilities until one works!

3.3 Problem Solvability and Complexity

Problem solvability and complexity helps us determine whether a problem can be solved using an algorithm and, if so, how efficiently it can be solved.

3.3.1 Solvable vs. Unsolvable Problems

In computer science, problems are classified as solvable or unsolvable based on whether there exists an algorithm that can provide a solution.

Solvable Problems: A problem is considered solvable if an algorithm can solve it within a finite amount of time. These problems have clearly defined inputs and outputs, and there is a step-by-step procedure to reach the solution.

Example: Calculating the greatest common divisor (GCD) of two integers is a solvable problem. The Euclidean algorithm provides a clear and finite method to determine the GCD, making it a classic example of a solvable problem.

Unsolvable Problems: On the other hand, a problem is unsolvable if no algorithm can be created that will provide a solution in all cases. These problems do not have a general procedure that can guarantee a solution for every possible input.

Example: The Halting Problem is a famous example of an unsolvable problem. It involves determining whether a given program will eventually halt (finish running) or continue to run forever. Alan Turing proved that no general algorithm can solve the Halting Problem for all possible program-input pairs, making it a fundamental example of an unsolvable problem.

Tidbits

When tackling complex problems, it's essential to first determine whether the problem is solvable. This saves time and resources by ensuring you are working on a problem that can be resolved using an algorithm.

3.3.2 Tractable vs. Intractable Problems

Once a problem is determined to be solvable, the next consideration is its computational complexity—how efficiently it can be solved. Problems are categorized as tractable or intractable based on the resources required (time and space) to solve them.

Tractable Problems: A problem is considered tractable if it can be solved in polynomial time, denoted as *P*. Polynomial time means that the time taken to solve the problem increases at a manageable rate (as a polynomial function) relative to the size of the input. Tractable problems are considered "efficiently solvable."

Example: Sorting a list of numbers using algorithms such as Merge Sort or Quick Sort is a tractable problem because these algorithms have a polynomial time complexity of $O(n \log n)$, where n is the number of elements in the list.

Intractable Problems: Intractable problems are those that require super-polynomial time to solve, often growing exponentially with the size of the input. These problems are impractical to solve for large inputs because the time required becomes unmanageable.

Example: The Traveling Salesman Problem (TSP), where the goal is to find the shortest possible route that visits a set of cities and returns to the origin, is an example of an intractable problem. The problem is NP-hard, meaning that as the number of cities increases, the number of possible routes grows factorially, making it infeasible to solve exactly for large instances.

3.3.3 Complexity Classes (P, NP, NP-hard, NP-complete)

Understanding the complexity of problems involves classifying them into different categories based on their solvability and the time required to solve them.

3.3.3.1 Class P

Class **P** refers to a category of problems that can be solved efficiently by a computer. In simpler terms, these are problems where a computer can find a solution quickly, even as the size of the problem grows:

problem grows.

Example: Let's consider a simple problem: sorting a list of numbers.

Suppose you have the following list:

[4, 1, 3, 2, 5]

The goal is to arrange these numbers in ascending order:

[1, 2, 3, 4, 5]

The time required to sort the list grows at a manageable rate as the list size increases. For example, going from 5 numbers to 10 numbers will increase the time, but it remains within a reasonable limit.

3.3.3.2 Class NP

Class **NP** refers to a category of problems for which, if a solution is given, it can be checked quickly by a computer. These are problems where verifying a proposed solution is easy, but finding that solution might be difficult and time-consuming.

Example:

Consider a common example of solving a Sudoku puzzle. In Sudoku, you fill a 9 x 9 grid with numbers so that each row, column, and 3 x 3 sub grid contains all digits from 1 to 9 exactly once as shown in Figure 3.3.

9	1	3				5		
6		7					2	4
	5			8			7	
	7	9						
		2		9			4	3
					4		9	
	4				1	9		
7		6			9			5
		1			6	4		7

Figure 3.3: A simple Sudoku Puzzle

3.3.3.3 Class NP-Hard

NP-hard problems are a class of problems that are at least as difficult as the hardest problems in Non-deterministic Polynomial time (NP). Solving an NP-hard problem is challenging, and no efficient algorithm is known for finding a solution.

Example:

A well-known example of an NP-hard problem is the Traveling Salesman Problem (TSP), we discussed in Section 3.3.2.

3.3.3.4 NP- Complete

NP-Complete problems form a special subset of NP problems. They are both in NP and as hard as the hardest problems in NP. This means that these problems are particularly challenging, and if you can solve one NP-Complete problem efficiently, you can solve all NP-problems efficiently.

Example: A classic example of an NP-Complete problem is the Knapsack Problem.

In the Knapsack Problem, you have a knapsack with a maximum weight capacity and a set of items, each with a weight and a value. The goal is to determine the most valuable combination of items to put in the knapsack without exceeding its weight capacity.

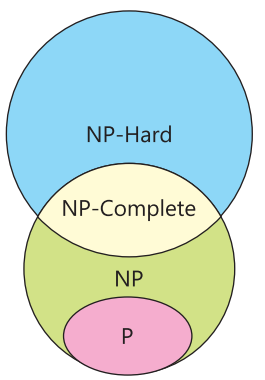


Figure 3.4 Venn diagram of the complexity classes P, NP, NP-hard, and NP-complete.

Figure 3.4 shows a Venn diagram illustrating the complexity classes P, NP, NP-hard, and NP-complete. It visually represents the relationships among these classes, highlighting how some problems can be solved efficiently, while others pose significant challenges in computational theory.

DO YOU KNOW?

The question of whether P equals NP is one of the most important unsolved problems in computer science. It has significant implications for cryptography, algorithm design, and the overall understanding of computational complexity.

3.4 Algorithm Analysis

Algorithm analysis is the process of determining the computational complexity of algorithms, which includes their time and space complexity. This analysis helps predict the algorithm's performance and is crucial for selecting the best algorithm for a particular task.

3.4.1 Time Complexity

Time complexity is a measure of how the running time of an algorithm increases as the size of the input data grows. It helps us understand how efficiently an algorithm performs when dealing with larger amounts of data.

Example: Consider the task of sorting a list of numbers. If the list contains only a few numbers, the task might be quick. However, as the volume of numbers increases, the time required to sort them also increases. Time complexity allows us to predict how this runtime scale with the input size.

3.4.1.1 Big *O* Notation

Big *O* notation is a mathematical way to describe the time complexity of an algorithm. It provides an upper bound on the time an algorithm will take to complete as the input size grows. This notation helps in comparing the efficiency of different algorithms by giving a clear picture of their performance.

How Big *O* Notation Works

Big *O* notation uses symbols to describe how the runtime of an algorithm changes with the size of the input. Here are some common examples:

- ***O*(1)-Constant Time:** The runtime remains the same regardless of the input size.
- ***O*(n)-Linear Time:** The runtime grows linearly with the input size. For example, suppose there are *n* students in a college, each with a unique student ID. If we need to find a specific student by searching through the list of all *n* students, the time it takes depends on the number of students, hence it is linear.
- ***O*(n^2)-Quadratic Time:** The runtime increases with the square of the input size. For instance, consider a scenario where *n* students in a college are participating in a programming competition, and we want to compare the performance of each pair of students to determine the best team. To do this, we must compare each student with every other student. The number of comparisons required is the sum of the first *n* - 1 integers, which can be approximated as *n(n* -1), a quadratic growth.
- ***O*(*log n*)-Logarithmic Time:** The runtime grows logarithmically, meaning it increases very slowly relative to the input size. Imagine you are thinking a number between 1 and 100, and . I want to guess it. I can only ask yes / no questions like", Is it greater than 50? ", " Is it less than 25?", "is it equal to 37?".

Every time I ask a question, I cut the range half this process (binary search) take logarithmic time.

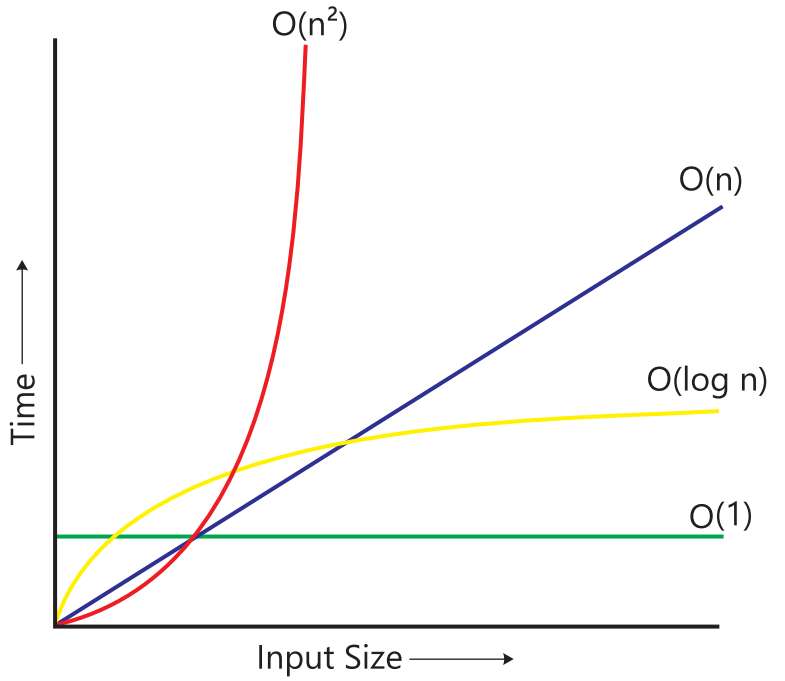


Figure 3.5: Growth of asymptotic notations

When comparing different time complexities, it's essential to understand how the time required for an algorithm grows as the size of the input *n* increases. Constant time, represented as *O*(1), remains unchanged regardless of the size of *n*. This means that no matter how large the input is, the time taken will be the same, as seen by the flat line in the graph.

3.4.2 Space Complexity

Space complexity measures how the amount of memory or space an algorithm uses changes as the size of the input data increases. It helps us understand how efficiently an algorithm uses memory when handling large datasets.

Example: if an algorithm needs to store a list of numbers, its space complexity tells us how much memory will be required as the volume of numbers increases.

3.5 Algorithm Design Techniques

Algorithm design is a critical aspect of problem-solving in computer science. It involves creating systematic methods to solve problems efficiently and effectively. There are several well-known algorithm design techniques that help in developing robust algorithms for a variety of computational problems.

3.5.1 Divide and Conquer

Divide and Conquer is a powerful algorithm design technique that works by breaking a large problem into smaller, more manageable parts. Each smaller part is solved

independently, and then their solutions are combined to solve the original problem, as shown in Figure 3.6. This approach is particularly effective for problems that can be divided into similar smaller problems, making it easier to find a solution step by step.

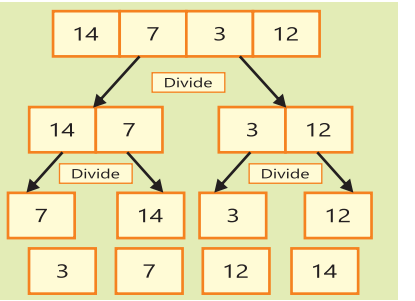


Figure 3.6: Merge Sort process

DO YOU KNOW?

Big O notation helps computer scientists understand the efficiency of an algorithm in the worst-case scenario, allowing them to predict how well it will perform as the size of the input data increases.

3.5.2 Greedy Algorithms

Greedy algorithms work by making a sequence of choices, each of which is locally optimal, with the hope that these choices will lead to a globally optimal solution. The greedy approach is often used when a problem has an optimal substructure, meaning that the optimal solution to the problem can be constructed from optimal solutions to its sub problems.

Example: A classic example of a greedy algorithm is the Coin Change problem. Suppose you have coins of different denominations and you want to make a specific amount with the fewest coins possible. The greedy algorithm would involve choosing the largest denomination coin that does not exceed the remaining amount, then subtracting that value and repeating the process until the desired amount is achieved.

Tidbits

Greedy algorithms are often faster and easier to implement than other techniques, but they don't always guarantee the optimal solution for every problem. Always analyze the problem to ensure that a greedy approach is appropriate.

3.5.3 Dynamic Programming

Dynamic Programming (DP) is an optimization technique used to solve problems by breaking them down into simpler subproblems and storing the results of these subproblems to avoid redundant calculations. DP is particularly useful for problems with overlapping subproblems and optimal substructure.

Example: The Fibonacci sequence is a well-known example where DP can be applied. Instead of recalculating Fibonacci numbers repeatedly, DP stores the results of each Fibonacci number as it is computed, allowing the algorithm to retrieve these values directly when needed, significantly reducing the number of calculations.

3.5.4 Backtracking

Backtracking is a method used in solving problems where you build up a solution step by step. If you find that a particular path doesn't lead to a solution, you simply go back and try a different path. It's like trying out different routes on a map and turning back if you find that you're going the wrong way. This method is often used for problems where you need to look at all possible options, like puzzles or problems that involve different combinations.

3.6 Commonly Used Algorithms

Algorithms are essential tools in computer science and are applied to a wide range of problems, from sorting data to searching for information in large datasets. Some algorithms are foundational, serving as building blocks for more complex operations. This section explores some of the most commonly used algorithms, including sorting, searching, and graph traversal algorithms.

3.6.1 Sorting Algorithms

Sorting algorithms are used to arrange data in a particular order, such as ascending or descending. Sorting is a fundamental operation that often serves as a prerequisite for other tasks like searching and data analysis.

3.6.1.1 Bubble Sort

Bubble Sort is one of the simplest sorting algorithms. It works by repeatedly stepping through the list, comparing adjacent elements, and swapping them if they are in the wrong order. This process is repeated until the list is sorted.

Process:

- Start from the beginning of the list.
- Compare each pair of adjacent elements.
- Swap them if they are in the wrong order.
- Continue the process until no more swaps are needed.

Example: Consider the list [5,3,8,4,2]. Bubble Sort will first compare 5 and 3, swap them, then move to the next pair (5 and 8), and so on. After several passes through the list, the algorithm will sort the list as [2,3,4,5,8].

Complexity: The time complexity of Bubble Sort is $O(n^2)$, making it inefficient for large datasets. However, it is easy to understand and implement, making it useful for educational purposes and small datasets.

Tidbits

While Bubble Sort is easy to implement, consider using more efficient sorting algorithms like Quick Sort or Merge Sort for larger datasets to save time and resources.

3.6.1.2 Selection Sort

Selection Sort is another simple sorting algorithm. It works by selecting the smallest (or largest, depending on the desired order) element from the unsorted part of the list and swapping it with the first element of the unsorted part. This process is repeated for the remaining unsorted portion of the list.

Process:

- Find the minimum element in the unsorted part of the list.
- Swap it with the first unsorted element.
- Move the boundary of the sorted and unsorted sections by one element.
- Repeat the process for the remaining elements.

Example: For the list [29,10,14,37,13], Selection Sort will first find the smallest element, 10, and swap it with 29. The list becomes [10,29,14,37,13]. The process continues until the list is fully sorted.

Complexity: The time complexity of Selection Sort is $O(n^2)$. Like Bubble Sort, it is not efficient for large datasets but is straightforward to implement.

3.6.2 Search Algorithms

Search algorithms are designed to find specific elements or a set of elements within a dataset. They are critical for tasks such as information retrieval, database queries, and decision-making processes.

3.6.2.1 Linear Search:

A linear search is a straightforward method for finding an item in a list. You check each item one by one until you find what you're looking for. Here's how it works,

1. **Start at the Beginning:** Look at the first item in the list.
2. **Check Each Item:** Compare the item you are looking for with the current item.
3. **Move to the Next:** If they don't match, move to the next item in the list.
4. **Repeat:** Continue this process until you find the item or reach the end of the list.

Example: Suppose you have a list of city names: [Karachi, Lahore, Islamabad, Faisalabad] And you want to find out if *Islamabad* is in the list.

1. Start with Karachi. Since Karachi isn't Islamabad, move to the next city.
2. Next is Lahore. Lahore is not Islamabad, so move to the next city.
3. Now you have Islamabad. This is the city you're looking for!

In this case, you've found Islamabad in the list. If Islamabad weren't in the list, you would check all the cities one by one and then conclude that it's not there. This method is called a linear search because you check each item in a straight line, from start to finish.

3.6.2.2 Binary Search

Binary Search is an efficient algorithm for finding an item in a sorted list. It works by repeatedly dividing the search interval in half and discarding the half where the item cannot be, until the item is found or the interval is empty.

Process:

- Start with the middle element of the sorted list.
- If the middle element is the target, return its position.
- If the target is smaller than the middle element, repeat the search on the left half.
- If the target is larger, repeat the search on the right half.

Example: Suppose you have a sorted list [1,3,5,7,9,11,13] and you are searching for a number.

- Binary Search will start at the middle element (7) and find the target immediately.

Complexity: The time complexity of Binary Search is *O*(log *n*), making it much faster than linear search algorithms, especially for large datasets. Figure 3.6 illustrates the binary search process, showing how the search interval is halved at each step, making the search more efficient.

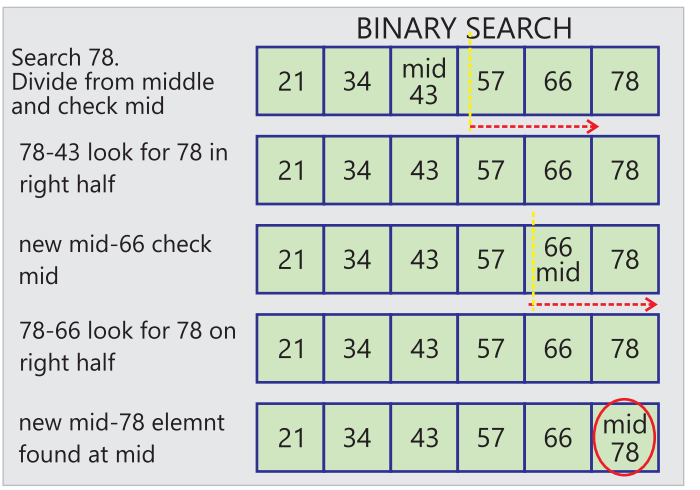


Figure 3.6: Binary Search Process

Binary Search is only effective on sorted lists. If your data isn't sorted, consider using a sorting algorithm like Merge Sort before applying Binary Search!

3.6.3 Graph Algorithms

Graph algorithms are used to explore and analyze graphs, which are data structures made up of nodes (vertices) connected by edges. These algorithms are essential for network analysis, route planning, and social network analysis.

3.6.3.1 Breadth-First Search (BFS)

Breadth-First Search (BFS) is a graph traversal algorithm that explores all the nodes of a graph level by level, starting from a given node (often called the root). It uses a queue to keep track of the nodes that need to be explored.

Process:

- Start from the root node and enqueue it.
- Dequeue a node, process it, and enqueue all its unvisited neighbors.
- Repeat the process until the queue is empty.

Example: In a social network graph, where each node represents a person and edges represent friendships, BFS can be used to find the shortest path between two people (e.g., finding the degree of separation between two users).

Complexity: The time complexity of BFS is $O(V + E)$, where V is the number of vertices and E is the number of edges. This makes it efficient for exploring large graphs.

3.6.3.2 Depth-First Search (DFS)

Depth-First Search (DFS) is another graph traversal algorithm that explores as far down a branch as possible before backtracking to explore other branches. It uses a stack to manage the nodes to be explored.

Process:

- Start from the root node and push it onto the stack.
- Pop a node, process it, and push all its unvisited neighbors onto the stack.
- Repeat the process until the stack is empty.

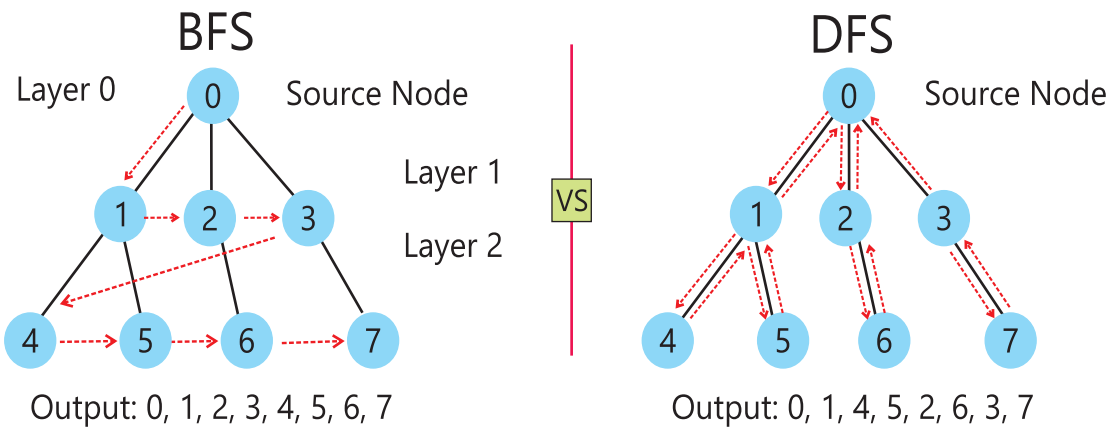


Figure 3.7: Comparison of BFS and DFS

Example: DFS can be used in solving puzzles like mazes, where the algorithm explores one possible path to the end, and if it hits a dead end, it backtracks and tries another path.

Complexity: The time complexity of DFS is O (V+E), similar to BFS. However, DFS is more memory-efficient for deep graphs, while BFS is more suited for shallow graphs.

EXERCISE

Multiple Choice Questions

1. The characteristic of a well-defined problem is:
   a) Ambiguous goals and unclear requirements
   b) Vague processes and inputs
   c) Clear goals, inputs, processes, and outputs
   d) Undefined solutions
2. Complexity class representing problems solvable efficiently by a deterministic algorithm:
   a) NP b) NP-hard
   c) NP-complete d) P
3. The statement that applies to unsolvable problems:
   a) They can be solved in polynomial time
   b) They cannot be solved by any algorithm
   c) They are always in NP class
   d) They require exponential time to solve
4. The meaning of NP in computational complexity is:
   a) Non-deterministic Polynomial time
   b) Negative Polynomial time
   c) Non-trivial Polynomial time
   d) Numerical Polynomial time
5. Search algorithm more efficient for large datasets:
   a) Bubble Sort b) Merge Sort
   c) Selection Sort d) Quick Sort
6. A scenario where Dynamic Programming proves most useful:
   a) Problems without overlapping subproblems
   b) Problems solved by making local choices
   c) Problems with overlapping subproblems and optimal substructure
   d) Problems divided into independent subproblems
7. An algorithm that sorts data by stepping through the list and swapping adjacent elements if needed is:
   a) Selection Sort b) Quick Sort
   c) Bubble Sort d) Merge Sort
8. Time complexity of Depth-First Search (DFS) in a graph is:
   a) O(n log n) b) O(V) c) O(V + E) d) O(n)

9. Best description of time complexity:
   a) Amount of memory an algorithm needs
   b) Time taken as a function of input size
   c) Efficiency as input size grows
   d) Upper bound of space requirements

10. An algorithm with a time complexity of O(n log n):
   a) Bubble Sort b) Binary Search c) Merge Sort d) Insertion Sort

Short Questions

1. Differentiate between well-defined and ill-defined problems within the realm of computational problem-solving.
2. Outline the main steps involved in the Generate-and-Test method.
3. Compare tractable and intractable problems in the context of computational complexity.
4. Summarize the key idea behind Greedy Algorithms.
5. Discuss the advantages of using Dynamic Programming.
6. Compare the advantages of Breadth-First Search (BFS) with Depth-First Search (DFS) in graph traversal.
7. Explain the importance of breaking down a problem into smaller components in algorithmic thinking.
8. Identify the key factors used to evaluate the performance of an algorithm.

Long Questions

1. Provide a detailed explanation of why the Halting Problem is considered unsolvable and its implications in computer science.
2. Discuss the characteristics of search problems and compare the efficiency of Linear Search and Binary Search algorithm.
3. Discuss the nature of optimization problems and provide examples of their applications in real-world scenarios.
4. Explain the process and time complexity of the Bubble Sort algorithm. Compare it with another sorting algorithm of your choice in terms of efficiency.
5. Discuss the differences between time complexity and space complexity. How do they impact the choice of an algorithm for a specific problem?

UNIT 4

Computational Structures

Student Learning Outcomes

By the end of this chapter, students will be able to:

- Define and explain the purpose of primitive computational structures, including lists, stacks, queues, trees, and graphs.
- Identify and describe the characteristics and properties of different computational structures.
- Perform basic operations such as insertion, deletion, traversal, and searching on various computational structures.
- Understand and implement the LIFO (Last-In, First-Out) and FIFO (First-In, First-Out) principles in stacks and queues, respectively.
- Compare and contrast different types of trees and graphs, and apply appropriate operations to them.
- Analyze and choose the most suitable computational structure based on problem requirements, data organization, and performance considerations.
- Apply computational structures in real-world scenarios, including data organization, task scheduling, and network modeling.
- Combine different computational structures to solve complex problems and enhance functionality.

Introduction

In this chapter, we will explore key computational structures, such as lists, stacks, queues, trees, and graphs, which are fundamental in programming. We will examine their properties, operations, and how to implement them efficiently. Additionally, we will discuss selecting the appropriate structure based on specific problem requirements and demonstrate their application in real-world scenarios.

4.1 Primitive Computational Structures

There are following commonly used computational software:

4.1.1 Lists

A list is a data structure used to store multiple pieces of data in a specific sequence. Each piece of data, known as an element, is positioned at a particular index within the list, facilitating easy access and management.

4.1.1.1 List Creation

In Python, Lists are created using square brackets '[]', with each item separated by a comma.

```
# Create a list of items
items= [ "Decorations" ,  "Snacks" ,  "cold drinks", "Plates", "Balloon"]
# Print the list
print(items)
```

In the above code:
"Items" is the name of the list. The items inside the list are "Decorations", "Snacks", "Cold drinks", "Plates", and "Balloons". Each item is enclosed in quotes (for text) and separated by a comma.

4.1.1.2 List Properties

List has following properties:

1. **Dynamic Size** A list in Python can change its size. You can add new items to the list or remove items without any problem. The list will automatically adjust to fit the changes.
2. **Index-Based Access** Every item in a list has a position, called an index. The first item has an index of 0, the second item has an index of 1, and so on. You can use these indexes to get specific items from the list.
3. **Ordered Collection** The order in which you add items to a list is preserved. This means that if you add an item first, it will stay in that position unless you change it.

4.1.1.3 List Operations:

Some common operations of a list are:

1. **Insertion:** Adding a new item to your list is like adding a new task to your to-do list. You can insert an item at different positions in the list. You can insert an item at any position in the list using the 'insert( )' function.

```
party_list = ["Buy drinks", "Buy decorations", "Buy snacks", "Buy cold drinks "]
party_list.insert (0 , "Invite friends") # add Invite friends at start
print(party_list)
# Output: ["Buy drinks, "Buy decorations", "Buy snacks", "Buy cold drinks "]
```

2. **Deletion:** Removing an item from your list is like crossing off a task you've completed. You can remove items in various ways
   a. **Removing by Value:** Use the 'remove()' function to delete the first occurrence of a specific item.

```
party_list = ["Invite friends ","Buy decorations", "Buy snacks", "Buy cold drinks"]
party_list.remove ("Buy snacks") # Removes 'Buy snacks ' from the list
print(party_list)
# Output: ['Invite friends', 'Buy decorations', 'Buy cold drinks ']
```

b. Removing by Index: Use the 'pop( )' function to remove an item at a specific index.

```
party_list = ["Invite friends ", "Buy decorations", "Buy cold drinks " ]
party_list.pop (0) # Removes the item at index 0
print(party_list)
# Output: ['Buy decorations', 'Buy cold drinks']
```

3. Searching: Finding an item in a list is similar to looking for a specific task in your to-do list. You can search for an item using different functions: Use the 'in' keyword to check if an item exists in the list.

```
party_list = ["Invite friends", "Buy decorations", "Buy cold drinks"]
if "Buy cold drinks" in party_list:
    print("Buy cold drinks is on the list.")  # Prints if 'Buy cold drinks' is found
else:
    print("Buy cold drinks is not on the list.")
# Output: Buy cold drinks is on the list.
```

4.1.1.4 Applications of Lists

- **Data Storage and Manipulation:** Lists are commonly used to store and manage collections of data, such as records, entries, or values. They allow for easy insertion, deletion, and access to elements.
- **Stack and Queue Implementations:** Lists can be used to implement stack (LIFO) and queue (FIFO) data structures, which are fundamental for various algorithms and tasks in computing.

4.1.2 Stacks

A stack is a simple data structure where you can only add or remove items from one end, known as the "top". Both insertion and deletion of elements occur at this top end. A stack operates on the Last-In, First-Out (LIFO) principle, meaning that the most recently added element is the first one to be removed.

Figure 4.1: Stack of Books

4.1.2.1 Stack Operations

There are two basic operations in a stack:

- **Push Operation:** Push means adding an item to the top of the stack.
- **Pop Operation:** Pop means removing the item from the top of the stack.

```
#    Create an empty stack of books
stack_of_books = []
print("Initial stack:", stack_of_books)          # Empty stack
#    Add books to the stack (push operation)
print("\n Adding books to the stack (push operation):")
stack_of_books.append('Book A')
print("Stack after pushing 'Book A':", stack_of_books)
stack_of_books.append('Book B')
print("Stack after pushing 'Book B':", stack_of_books)
# Remove the top book from the stack (pop operation)
print("\nDeletion of top book (pop operation):")
top_book = stack_of_books.pop()
print("Removed book:", top_book)
print("Stack after popping the top book:", stack_of_books)
```

The code creates an empty stack to hold books. It then adds books ("Book A" and "Book B") one by one to the top of the stack. Finally, it removes the top book "Book B" from the stack, showing how the last book added is the first one taken off.

4.1.3 Queues

A queue is like a line in front of a bank or a ticket counter. The first person to get in line is the first person to be served. In a computer, a queue works the same way. It keeps track of things so that the first item added is the first one to be taken out. Just like in a bank line, you add things to the back and remove them from the front, following the FIFO (First-In, First-Out) principle, as shown in Figure 4.2.

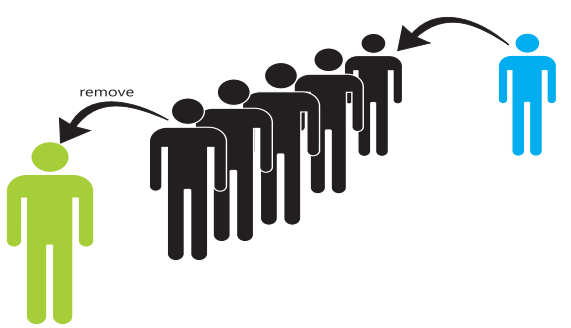


Figure 4.2: Queue of persons in front of the bank

4.1.3.1 Queue Operations

Queues support two primary operations:

- **Enqueue (Add an Item):** This is like adding a person to the end of the line. In a queue, you add items to the back.
- **Dequeue (Remove an Item):** This is like serving the person at the front of the line. In a queue, you take items out from the front. Additional operations might include checking if the queue is empty, retrieving the element at the front without removing it, and determining the size of the queue.

```
# Built-in module to implement queues in Python
from queue import Queue
# Create a new queue
q = Queue ()
# Add people to the queue (Enqueue)
q. put (" Ahmed") # Adds Ahmed to the end of the queue
q.put ("Fatima")  # Adds Fatima to the end of the queue
# View the person at the front of the queue (Peek)
front_person = q. queue [0] # Looks at the person at the front without
removing them
print(front_person) #Remove a person from the front of the queue(Dequeue)
removed_person = q.get ( ) # Removes and returns the person at the front
of the queue
print(removed_person)
# Add another person to the queue (Enqueue)
q.put("Sara") # Adds Sara to the end of the queue
# View the updated queue
updated_queue = list(q.queue)
print(updated_queue)
```

The code manages a line of people using a queue. It adds people to the end of the line, checks who is at the front without removing them, and then serves (removes) the person at the front. Finally, it adds another person to the end and shows the updated line.

4.1.4 Trees

A tree data structure organizes information in a way that spreads out from a main point called the root node. In a tree, each piece of information, called a node, can connect to other pieces, which are also nodes, forming a branching structure. This branching

structure is different from a list, where items are organized one after the other in a straight line.

Example: In a family tree, the oldest ancestors represent the root node, serving as the starting point of the hierarchy. Each individual in the tree may have descendants, forming subsequent levels of the hierarchy, as illustrated in Figure 4.3. This hierarchical structure is not suitable for storage in a linear format, such as a list, due to the complex parent-child relationships. Therefore, a tree data structure is employed to efficiently store and access such hierarchical data, enabling clear representation and retrieval of information.

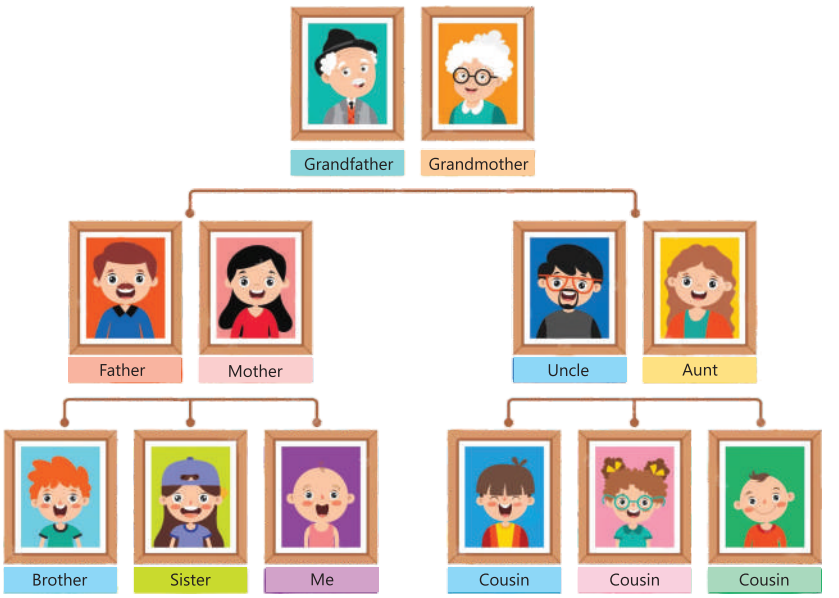


Figure 4.3: Family Tree

4.1.4.1 Properties of Trees

1. **Root Node:** The root is the very first or top node in a tree, like the main folder in a computer where all other folders and files are contained.
2. **Edges and Nodes:** Nodes are the individual elements in the tree, and they are connected by lines called edges. A node without any child nodes is called a leaf, similar to a file in a folder that doesn't contain any other files.
3. **Height:** The height of a tree is the longest path from the root node down to the farthest leaf. It tells us how deep or tall the tree is.
4. **Balanced Trees:** A tree is considered balanced if the branches on the left and right sides are nearly the same height.

4.1.4.2 Applications of Trees

1. **File Systems:** Pre-order tree traversal is useful for creating backups of file systems. By visiting the root first and then recursively backing up each directory, it ensures that directories are backed up before their contents.
2. **File System Deletion:** In file systems, Post-order traversal ensures that files and directories are deleted in the correct order; by first deleting all sub directories and files before deleting the parent directory.
3. **Hierarchical Data Representation:** Trees are used in representing data with a clear hierarchical relationship, such as organisational charts and family trees.
4. **Decision Making:** Trees, such as decision trees, are used in algorithms to make decisions based on various conditions and outcomes.

4.1.5 Introduction to Graphs

A Graph is a data structure that consists of a set of vertices (or nodes) connected by edges. Graphs are used to represent networks of connections, where each connection is a relationship between two vertices. These vertices can represent anything, like cities, people, or even abstract concepts, and the edges represent the relationships or pathways between them.

Imagine you are mapping out all the cities in Pakistan and the roads that connect them. Each city is a vertex, and each road between two cities is an edge. Unlike a tree, a graph does not have a single ''root" and does not follow a hierarchical structure. In a graph, any two vertices can be connected, creating a complex web of relationships.

Example: In a social network, each person can be connected to many others, forming a graph. There is no single starting point, and people (vertices) can have multiple connections (edges) that do not follow a strict parent-child relationship like in a tree.

Difference from a Tree: While both graphs and trees are used to represent relationships between objects, a tree is a special kind of graph with some important differences:

- A Tree is hierarchical, meaning it has a single root node from which all other nodes branch out. In contrast, a Graph does not necessarily have a hierarchy or a root.
- In a Tree, there is exactly one path between any two nodes, ensuring no cycles (loops). However, in a Graph, there can be multiple paths between nodes, and cycles are allowed.
- Trees are often used to represent structured data like family trees or organizational charts. Graphs are more flexible and can represent a broader range of connections, such as networks, web links, or transport systems.

4.1.5.1 Characteristics of Graphs

Graphs have several defining features that help us understand and use them effectively:

4.1.5.2 Properties of Graphs

Graphs also have specific details that describe their structure:

- **Degree:** This is the number of edges connected to a vertex. For instance, if a city is connected to three other cities, the degree of that city's vertex is 3.
- **Weight:** In some graphs, edges have weights that represent values like distances or costs. For example, if a road between two cities is 50 kilometres long, its edge might have a weight of 50.
- **Direction:** Edges can be either directed or undirected. Directed edges have a one-way connection, meaning a road from city A to city B does not necessarily have a return road from B to A. Undirected edges represent a two-way connection.

4.1.5.3 Types of Graphs

Graphs can be classified into several types based on their structure and properties. The main types of graphs are directed, undirected, and weighted. Each type has its own characteristics, which can be better understood through simple examples.

- **Directed Graphs:** In a directed graph, edges have a direction, which means they go from one vertex to another in a specific way as shown in Figure 4.4.

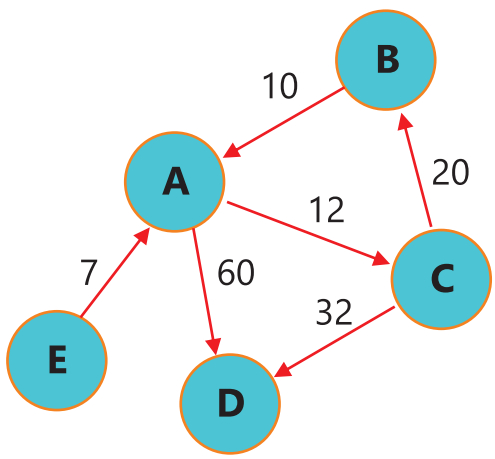


Figure 4.4: Directed Weighted Graph

Example: Consider a graph shown in Figure 4.4. If you want to travel from city A to city B, you can only go in the direction permitted by the city's sign. If there's no one-way street going from city A to city B, you cannot travel directly from city A to B.

- **Undirected Graphs:** In an undirected graph, edges do not have a direction. This means that if there is a connection between two vertices, you can travel in both directions.
  Example: Consider a graph shown in Figure 4.5, if Person A is friends with Person B, then Person B is also friends with Person A. There is no restriction on the direction of the friendship, so you can move freely between friends.

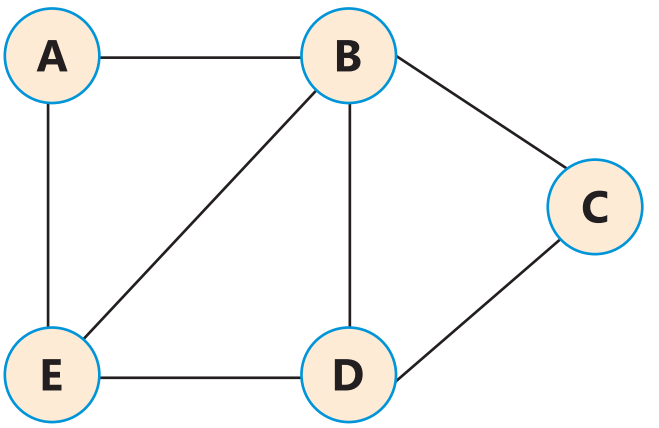


Figure 4.5: Undirected Graph

- **Weighted Graphs:** In a weighted graph, each edge has a weight or cost associated with it. This weight represents the distance, time, or cost required to travel from one vertex to another as shown in Figure 4.4.
  Example: Imagine a map of a city where each road has a different distance or travel time. If you want to travel from one landmark to another, the map provides the distance or travel time for each road. This information helps you determine the shortest or quickest route between landmarks.

EXERCISE

Multiple Choice Questions

1. The function used to add an item at the end of a list in Python:

   a) insert( ) b) append( ) c) remove( ) d) pop( )

2. The purpose of the in keyword used with a Python list:

   a) Adds an item to the list b) Removes an item from the list

   c) Checks if an item exists in the list d) Returns the length of the list

3. An operation that removes an item from the top of the stack:

   a) Push b) Pop c) Peek d) Add

4. The operation used to add an item to a queue:

   a) Dequeue b) Peek

   c) Enqueue d) Remove

5. True statement about the height of a tree:

   a) Number of edges from the root to the deepest node

   b) Number of nodes from the root to the deepest node

   c) Number of children of the root node

   d) Always equal to the number of nodes in the tree

6. A scenario where a graph data structure is most suitable:

   a) Managing a to-do list

   b) Modeling a line of customers in a store

   c) Representing connections in a social network

   d) All of the above

Short Questions

1. Explain how the ' insert( ) ' function works in python lists. Provide an example.

2. Explain the potential issues which could arise when two variables reference the same list in a program? Provide an example.
3. Define a stack and explain the Last-In, First-Out (LIFO) principle.
4. Differentiate between the Enqueue and Dequeue operations of queue.
5. Name two basic operations performed on stack
6. What is difference between enqueue ( ) and dequeue ( ).

Long Questions

1. Discuss the dynamic size property of lists in Python. How does this property make lists more flexible?
2. Explain the operations on stack with real-life example and Python code.
3. Write, a simple program to implement a queue (insertion and deletion).
4. Define Tree and explain its properties
5. What is a graph? Explain differences between directed and undirected graphs.

UNIT 5

Data Analytics

Student Learning Outcomes

By the end of this chapter, students will be able to:

- Understand the role and importance of model building and their real world applications
- Build basic statistical models for real-world problems and evaluate their performance
- Understand and explain the principles of experimental design in data science
- Explain the types, uses and methods of data visualizations,
- Understand the benefits of visualizing data through descriptive statistics
- Create and interpret data visualization using data visualization software such as MS Excel, Google Sheets, Python, Tableau, and Matplotlib.

Introduction

Data analytics is the process of examining data to find useful information, patterns and trends to support decision-making.

5.1 Basic Statistical Concepts

Statistics is a branch of mathematics that helps us understand and analyze data. By using statistics, we can summarize large sets of information in a simple way, making it easier to draw conclusions. By using statistics large sets of information can be summarized in simple way making it easier to analyze and draw conclusions.

5.1.1 Measures of Central Tendency

Measures of central tendency help us identify the "center" or typical value in a dataset. There are three main measures of central tendency: mean, median, and mode. These measures give us a sense of the average or most common values of a dataset.

Mean

The mean is the average of all the numbers in a dataset. To calculate the mean, we add all the numbers together and then divide the sum by the total number of values.

Example: Imagine 5 students scored 50, 60, 70, 80, and 90 in a test. The mean score is calculated by adding all the scores and then dividing by the number of students:

$$\text{Mean} = \frac{50+40+70+80+90}{5} = 70$$

Median

The median is the middle value in a dataset when the numbers are arranged in order. If there is an odd number of values, the median is the exact middle number. If there is an even number of values, the median is the average of the two middle numbers.

Example: Using the same test scores: 50, 60, 70, 80, and 90. When we arrange these scores in ascending order (which they already are), the middle value is 70. Therefore, the median score is 70. Example with Even Number of values: If the scores were 50, 60, 70, and 80, we would take the average of the two middle scores (60 and 70):

Median = (60+70)/2= 65, so the median is 65.

The median helps us understand the middle point of the data.

Mode

The mode is the number that appears most often in a data set. There can be more than one mode if multiple numbers appear with the same highest frequency. The mode helps us identify the most frequent or common value in the data.

Example: If 5 students scored 50, 60, 70, 70, and 90, the number 70 appears twice, while all other numbers appear only once. Therefore, the mode is 70.

Example with Multiple Modes: If the scores were 50, 60, 70, 70, 60, and 90, both 60 and 70 appear twice. So, there are two modes: 60 and 70.

5.1.2 Measures of Dispersion

Measures of dispersion tell us how spread out or scattered the data is. Two common measures of dispersion are variance and standard deviation. These help us understand whether the data points are close to the average (mean) or spread far from it.

5.1.2.1 Variance

The variance shows how much the numbers in a data set differ from the mean. A higher variance means that the numbers are more spread out, while a lower variance means that the numbers are closer to the mean. To calculate variance, we use the following mathematical formula.

$$\text{Variance } (\sigma^2) = \frac{1}{N}\sum_{i=1}^{N}(x_1 - \mu)^2$$

Where: x_i represents each individual value in the data set, μ is the mean of the data set, and N is the total number of values in the data set.

Example: for the following two classes find out which class is more spread out by calculating their variance?

Class A: 50, 52, 55, 57, 60

Class B: 30, 45, 55, 75, 90

Steps are involved in variance calculations:

Step 1: Variance for Class A

Given Score = 50,52,55,57,60

Step 1.1: Compute the Mean (μ)

$$\mu = \frac{50+52+55+57+60}{5} = \frac{274}{5}$$
$$= 54.8$$

Step 1.2: Compute Each Squared Deviation $(x_i - \mu)^2$

x_i	$x_i - \mu$	$(x_i - \mu)^2$
50	50−54.8=−4.8	23.04
52	52−54.8=−2.8	7.84
55	55−54.8=0.2	.04
57	57−54.8=2.2	4.84
60	60−54.8=5.2	27.04

Step 1.3: Compute Variance

$$\text{Variance } (\sigma^2) = \frac{23.04+7.84+0.04+4.84+27.04}{5}$$
$$\sigma^2 = \quad 62.8/5 = 12.56$$

Step 2: Variance for Class B

Given Scores: 30,45,55,75,90

$$\mu = \frac{30+45+55+75+90}{5} = \frac{295}{5} = 59$$

x_i	$x_i - \mu$	$(x_i - \mu)^2$
30	30−59 = −29	841
45	45− 59 = −14	196
55	55 – 59 = – 4	16
75	75 – 59 = 16	256
90	90 – 59 = 31	961

Step 2.3: Compute Variance

$$\sigma^2 = \frac{841+196+16+256+961}{5}$$
$$= 2270/5 = 454$$

- **Variance of Class A**: **12.56**
- **Variance of Class B**: **454**

This confirms that Class B has a much higher variance, meaning the scores are more spread out compared to Class A.

5.1.2.2 Standard Deviation

It is similar to variance but provides a more practical and interpretable value because it is in the same unit as the original data. The standard deviation tells us how spread out the numbers are in relation to the mean. The standard deviation is simply the square root of the variance. To calculate standard deviation, we use the following mathematical formula.

$$\text{Standard Deviation} = \sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_1 - \mu)^2}$$

Where: x represents each individual value in the data set, μ is the mean of the dataset, and N is the total number of values in the data set.

Calculating Standard Deviation:

Class A:

$$\text{Standard Deviation} = \sqrt{12.56} = 3.55$$

Class B:

$$\text{Standard Deviation} = \sqrt{456} = 21.26$$

The standard deviation for Class A is approximately 3.55, while for Class B, it is about 21.26. This means that Class A's scores are closely packed around the mean, whereas Class B's scores are more widely scattered. The standard deviation helps us easily understand how much variation exists in a dataset.

5.1.3 Introduction to Probability

Probability is the study of how likely an event is to happen. It helps us make predictions based on known information.

Example: Consider flipping a coin. There are two possible outcomes: heads or tails. Since both outcomes are equally likely, the probability of getting heads is 50% (or 1/2), and the probability of getting tails is also 50%.

We can express this mathematically as:

$$\textit{Probability} = \frac{\textit{Number of favorable outcomes}}{\textit{Total number of outcomes}}$$

In the case of the coin flip:

Probability of heads = (1 favorable outcome, 2 total outcomes) = 1/2

Probability is not just for coin flips. It is used in many areas, such as predicting the weather, making business decisions, or even predicting outcomes in sports like cricket.

For illustration, you can use the following sample data: 3, 5, 8, 8, 10, 12, 15, 15, 16, 18. But if you are interested in collecting real data from your classmates, you are welcome to do so.

Tidbits

Statistics can help us understand patterns in data, leading to better decision-making in various fields, from healthcare to marketing!

Class Activity

Instructions: You will analyze a small dataset, calculate measures of central tendency (mean, median, mode), and measures of dispersion (variance and standard deviation).

1. Collect Data: Imagine you are surveying your classmates about the number of hours they spend on homework in a week. Gather data from 10 classmates. Record the number of hours (use reasonable values , e.g., between 0 to 20 hours).
2. Calculate Measures of Central Tendency:
   - Mean: Calculate the average number of hours spent on homework.
   - Median: Determine the middle value when the hours are arranged in order.
   - Mode: Identify which number appears most frequently in your data.
3. Calculate Measures of Dispersion:
   - Variance: Use the appropriate formula to calculate variance based on your data.
   - Standard Deviation: Calculate the standard deviation using the variance obtained.
4. Reflect: Write a brief reflection (3-4 sentences) on what these calculations reveal about your classmates' study habits.

Statistics is used in many everyday activities, such as predicting weather patterns or analyzing sports performance.

5.2 Data Collection and Preparation

In order to carry out any research or analysis, data collection and preparation are crucial steps. The quality and relevance of the data directly impact the results and insights drawn from the study. This section discusses various methods of data collection and how the collected data is prepared for further analysis.

5.2.1 Data Collection Methods

Data collection refers to the process of gathering relevant information for a particular purpose. Depending on the nature of the research, different methods can be used for data collection. These methods include surveys, observations, and experiments, each having its own strengths and appropriate contexts. Choosing the right method depends on the research objective and the type of data required.

5.2.1.1 Surveys

Surveys are a commonly used method for collecting large amounts of data in a structured way. They involve asking a predefined set of questions to a sample group. Surveys can be conducted using various means such as online forms, telephone calls, or

face-to-face interviews.

Example: A small local grocery store in Islamabad wants to know customer preferences regarding which products they would like to see more frequently. The store creates a short survey consisting of five questions as shown below and distributes it to 50 customers over the weekend. The collected responses are then analyzed to stock products that align with customer demand, helping improve business operations.

Customer Preference Survey

1. Which product categories do you buy most often? (e.g., fruits, vegetables, dairy)
2. Are there any products you would like to see more often?
3. How often do you shop at this grocery store? (e.g., daily, weekly, monthly)
4. What influences your purchasing decisions the most? (e.g., price, quality, availability)
5. Any additional comments or suggestions?

5.2.1.2 Observations

Observation involves collecting data by watching or monitoring subjects in their natural environment. This method is useful when researchers want to gather data on behaviors or phenomena without interference.

Example: A restaurant is interested in knowing which tables are most frequently chosen by customers during lunchtime. A staff member observes the seating choices over a period of one week. Based on these observation, the restaurant arranges its seating to optimize the customer comfort and traffic flow, which helps in improving customer satisfaction and service efficiency.

5.2.1.3 Experiments

Experiments involve manipulating one or more variables to determine their effect on another variable. This method is particularly useful in scientific and engineering fields where controlled environments are necessary for accurate measurement.

Example: A school teacher wants to test whether providing students with printed notes helps improve their performance in exams. The teacher conducts an experiment with two groups of students, one receiving printed notes and the other relying solely on lectures. After one month, both groups took the same test, and the teacher compared the results to see if printed notes had a positive impact on academic performance.

5.2.2 Data Preparation

Once data has been collected, it is important to prepare it for analysis. This includes cleaning the data to remove errors or inconsistencies, organizing it in a meaningful way, and converting it into a format suitable for analysis. In cases where data is missing or incorrect, researchers may need to employ techniques such as interpolation or statistical adjustments to ensure the accuracy and reliability of the results..

Example: If survey responses contain incomplete information, missing values can be

estimated based on the available data.
Proper data preparation ensures that the analysis leads to reliable and valid results.

5.2.3 Data Cleaning and Transformation

Data cleaning and transformation are important steps to prepare data for analysis. Raw data often has errors, missing values, or may be in an incorrect format. To ensure accurate results in analysis, it is important to fix these issues before moving forward.

5.2.3.1 Data Cleaning

Data cleaning means correcting or removing any problems in the data. These problems can include incorrect entries, missing values, or duplicate results. If these errors are not fixed, the results of the analysis will be misleading.

Example: Imagine a school collecting data on student scores. Some students may have entered their names incorrectly, or a few scores may be missing from the records. In this case, data cleaning would involve correcting any wrong names and including in the missing grades to complete the dataset. Table 5.1 and 5.2, illustrates the data cleaning process for student scores in a school. It shows examples of common issues such as incorrect names, missing scores and duplicate entries.

Original Data (with Errors)

Name	Scores	Class	Section
Ali	84	10	A
Alie	90	10	A
Sara		10	A

Table 5.1: Original Data with errors

Cleaned Data (After Data Cleaning)

Name	Grade	Class	Section
Ali	84	10	A
Ali	90	10	A
Sara	87	10	A

Table 5.2: Cleaned data after removing errors

5.2.3.2 Data Transformation

Once the data is clean, it often needs to be transformed it into a format that is easier to work with. This transformation may include converting data into different formats, creating new columns, or organizing data in a different way. These changes help make the data more suitable for analysis or modeling.

Example: After cleaning the student grade records, it may be necessary to transform this data for better analysis. For instance, instead of displaying grades for each individual student, the data might be aggregated or summarized to show class-level statistics such

as average scores or grade distribution.

5.2.3.3 Handling Missing Data

Sometimes, data is incomplete or has missing values. There are different techniques to handle missing data. One option is to remove the rows with missing values if they are very few. Another option is to fill in the missing values with an average or with data from similar cases. The choice depends on the type of data and the amount of missing information.

Example: In the dataset of student grades, if Sara's grade is missing, this creates a challenge in assessing her performance. To address this issue, several strategies can be employed:

1. **Imputation:** One common method is to estimate the missing value using existing data. **Example:** The school can calculate the average score of all students in Sara's class. If the average score is 87, the school may assign this value to Sara's record temporarily. This approach allows the school to maintain a complete dataset while making a reasonable assumption about Sara's performance.
2. **Flagging:** The school can also keep track of Sara's missing score by adding a note in the dataset. This method indicates that Sara's score is not available, making analysts aware of the incomplete data. This approach ensures transparency while allowing the analysis to proceed without filling in the gap.
3. **Removal:** If the number of missing entries is small, the school might choose to exclude Sara's record from specific analyses. This decision is acceptable if it does not significantly impact the overall understanding of student performance. However, it risks losing valuable information about Sara.

5.3 Building Statistical Models

In this section, we will explore the basic building blocks of statistical models, including different types of models, how they are developed, and how to evaluate their performance. We'll also look at real-world examples to make the concepts easy to understand.

5.3.1 Introduction to Statistical Modeling

Statistical modeling is a introduced to analyse data to make sense of the real-world and to predict what will happen in the future. Think of it like this: if you want to know how much money you'll spend on groceries next month, you can look at your past expenses. By analyzing that data, you can create a model to help you estimate your future grocery expenses.

5.3.1.1 Model Development

Building a statistical model involves several steps. Let's break them down:

- **Step 1: Define the Problem**

  First, we need to understand the problem. Example: If we are trying to predict grocery expenses, we need to identify the factors that influence them (e.g., family size, location, or income).

- **Step 2: Collect Data**
  Next, we gather data related to the problem. In our example, we will collect data on past spending habits, number of family members, and any other factors that may affect grocery costs.
- **Step 3: Choose an Algorithm**
  Based on the problem and the data, we choose an appropriate algorithm. Algorithms are methods that help us build models. Some popular algorithms are linear regression and logistic regression, which we will later discuss in this section.
- **Step 4: Train the Model**
  The model is then trained using the collected data. This means the model learns from the data to make predictions.
- **Step 5: Evaluate the Model**
  Finally, we test the model to see how well it works by using new or unseen data. This step is very important to ensure the model makes accurate predictions.

5.3.1.2 Linear Regression

Linear regression is a widely used statistical model that helps understand the relationship between two variables. It is often used to predict one variable based on another. Let's go through a practical example to explain how it works.

Example: Imagine you run a small fruit stall in your town, and you want to predict how much money you will make each day based on the number of customers who visit your stall. The number of customers is the independent variable (the cause), and the money you earn is the dependent variable (the effect). We will use linear regression to understand this relationship and help you forecast future earnings.

- **Step 1: Collecting Data**
  To build a linear regression model, we need historical data. Let's assume you've recorded the number of customers and your daily earnings for the past 5 days:

Number of Customers	Daily Earnings (Rs.)
10	500
15	700
20	900
25	1, 100
30	1, 300

Table 5.3: Customer's data

Here, the number of customers is our independent variable (X), and the daily earnings are the dependent variable (Y).

- **Step 2: Understanding the Linear Regression Formula**
  The formula for simple linear regression is:

$$Y = \beta_0 + \beta_1 \ x + \epsilon$$

Where:

- Y is the dependent variable (in our case, daily earnings),
- X is the independent variable (the number of customers),
- β_0 is the intercept, which is the value of Y when $X = 0$,
- β_1 is the slope of the line, which shows how much Y changes with each unit change in X,
- ε is the error term, which accounts for the difference between the predicted and actual values.

- **Step 3: Building the Linear Regression Model**

  When building a linear regression model, our goal is to find the best line that explains how two things are related in this case, the number of customers and daily earnings. Here's how we get the values for the slope (40) and intercept (300):

  Understanding the Slope ($\beta_1 = 40$)

  The slope shows how much extra money we make for every new customer. Let's use our data to figure it out:

If you notice, for every 5 extra customers, earnings go up by 200 rupees. So, for each new customer:

$\beta_1 = 200/5 = 40$

This means every new customer adds 40 rupees to our earnings.

Understanding the Intercept (β_0):

The intercept (β_0) represents the earnings when no customers visit. To find this value, we look at where the line crosses the vertical axis when the number of customers is zero. In simpler terms, it tells us what the base earnings are, even if no one shows up.

Now, to find the intercept, we need to consider how much we earn when there are no customers.

We can use the equation:

Earnings = $\beta_0 + \beta_1 \times$ Customers

If we take any data point, say when there are 10 customers, the earnings are 500 rupees.

Substituting these values into the equation:

$500 = \beta_0 + (40 \times 10)$

$500 = \beta_0 + 400$

Solving this gives:

$\beta_0 = 500 - 400 = 100$

This means that, based on the data, if no customers show up, you'd still expect to make 100 rupees, maybe from regular customers or other fixed earnings. So, the intercept value of 100 rupees represents the minimum amount you'd make on a day with zero customers.

- **Final Equation:**

Earnings = 100 + 40 x Customers

This equation means:

– You'll always earn 100 rupees, even if no one comes.

– Each new customer adds 40 Rs to your total.

- **Step 4: Interpreting the Model**

  Once the model is built, we can use it to predict future earnings. For example, if you expect 22 customers tomorrow, the predicted earnings would be:

  Earnings = 100 + (40 x 22) = 100 + 880 = 980 Rs

  This means that with 22 customers, you can expect to earn around 980 rupees.

- **Step 5: Testing the Model**

  After building the model, it's important to test it using new data. Let's say on the 6th day, 28 customers visited your stall, and you earned 1,250 Rs. Using the model, we can predict the earnings for 28 customers:

  Predicted Earnings = 100 + (40 x 28) = 100 + 1, 120 = 1,220 Rs.

  However, you actually earned 1,250 Rs. The difference between the predicted and actual earnings is called the error:

  Error = 1,220 - 1,250 = -30 Rs.

  While the prediction was close, it is not perfect, showing that real-world data often has some variation.

Tidbits

To improve your statistical model, consider these suggestions:

1. Use more data points for better accuracy.
2. Include relevant factors such as like family size or special events that may affect spending's.
3. Regularly update your model with new data to keep it relevant.
4. Test your predictions against actual spending to refine your approach.

5.3.1.3 Logistic Regression

Logistic regression is a powerful tool used when we want to predict an outcome that can be categorized as "yes" or "no".

Example: Let's say we want to determine whether a student will pass or fail an exam based on the number of hours they study. Instead of predicting a specific score, logistic regression helps us find the probability of passing.

Understanding Logistic Regression

Logistic regression is different from linear regression because it does not predict exact numbers. Instead, it provides a probability value between 0 and 1. This means it tells us how likely something is to happen.

5.3.1.4 Clustering Techniques

Clustering is a way of grouping similar things together based on their characteristics. Imagine you have a group of students in your class, and you want to divide them into groups based on their performance in different subjects like math, science, and English. Clustering helps us do that by creating groups of students who perform similarly.

Example: Clustering of Students by Performance

Let's say we have data for five students, showing their scores in math and English:

Student	Math Score	English Score
Basim	85	70
Umer	90	65
Anie	50	80
Tallat	40	85
Maliha	60	60

Table 5.4: Data for Clustering Techniques

We can use clustering to group these students based on their performance in these two subjects.

K-means Clustering

K-means clustering is one of the simplest and most popular techniques to group data. In K-means, we need to decide how many groups (clusters) we want. For this example, let's say we want to divide the students into two clusters: one for students who are strong in math and one for students who are strong in English. The algorithm will group students with similar performance in math and English together by calculating the distance between their scores and finding patterns. It will assign students like Basim and Umer (who are good at math) to one group and students like Anie and Tallat (who are good at English) to another group.

5.3.2 Evaluating and Interpreting Models

Once a model is built, it is important to check how well it performs and to understand the results it provides. This is called model evaluation.

5.3.2.1 Performance Metrics

Performance metrics help us measure how well a model is doing. Some common metrics include:

- **Error Metrics**

  Error metrics measure how much the model's predictions differ from the actual values. In our grocery example, if the model predicts a monthly grocery bill of

8,000 rupees but the actual bill is 10,000 rupees, the difference is the error.

- **Accuracy Metrics**

  Accuracy metrics tell us how many of the model's predictions were correct. For example, if a model predicts whether a student will pass or fail an exam, accuracy measures the percentage of correct predications made by the model.

5.3.2.2 Interpreting Outputs

Interpreting a model's output means understanding what the results reveals.

Drawing Conclusions from Insights

For example, if our linear regression model shows that number of hours studied strongly affects exam scores, we can conclude that students should study more hours to improve their scores.

5.3.2.3 Ethical Considerations

When building models, it is important to consider the ethical implications, such as fairness and privacy.

Fairness and Bias

A model should be fair and unbiased. For example, if a model is used to decide who gets a loan, it should not unfairly favor one group of people over another.

Data Privacy

When using personal data to build models, it is important to respect privacy. For example, if a company is using customer data to build models, they should ensure the data is secure and not shared without permission.

Tidbits

Always visualize your data before building models and test your results on new data for better accuracy.

5.4 Introduction to Data Visualization

Data visualization is the process of representing data in a visual format, such as graphs or charts. It helps us to quickly identify patterns, trends, and insights from the data.

5.4.1 Types of Visualizations

Data visualization is a powerful way to understand complex information. Different types of visualizations serve various purposes, making it easier to interpret and analyze data. Below are some common types of visualizations explained in detail:

5.4.1.1 Bar Charts

Bar charts are ideal for comparing different categories. Each bar represents a category, and the

height (or length) of the bar indicates the value associated with that category.

Example: Imagine you want to compare the sales figures for different products in a store. A bar chart can visually represent this data.

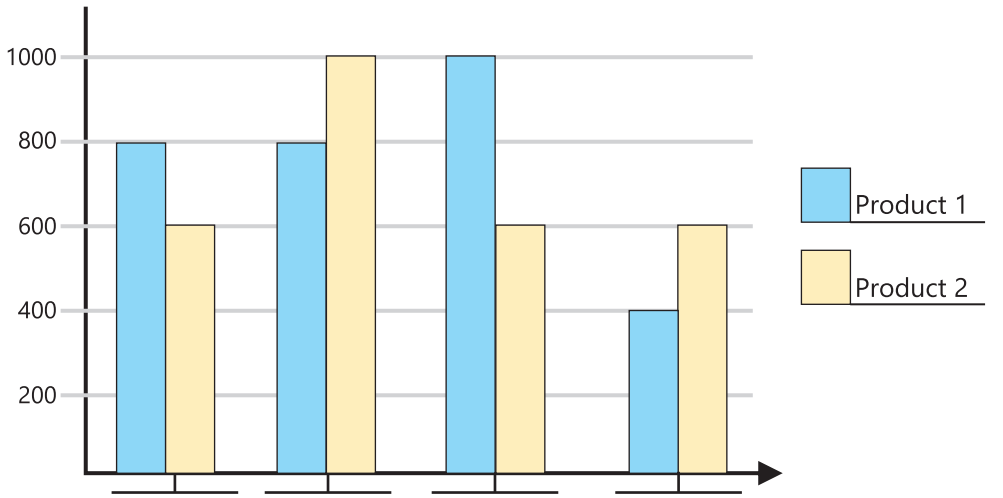


Figure 5.1: A bar chart showing sales figures of different products

5.4.1.2 Line Graphs

Line graphs are used to show trends over time. They plot data points and connect them with a line, making it easy to observe changes.

Example: If you track the temperature over a week, a line graph will show how the temperature rises and falls each day.

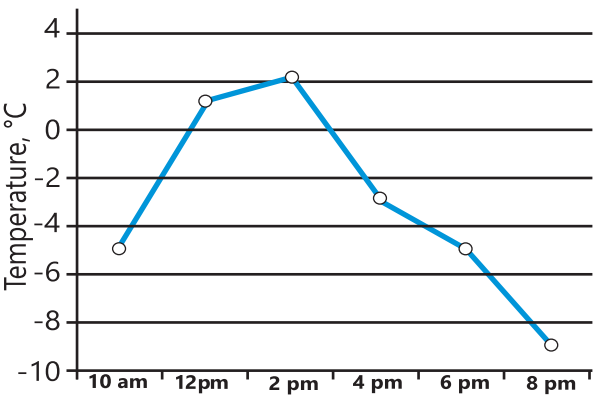


Figure 5.2: A line graph showing variation of temperature over time

5.4.1.3 Histograms

Histograms are used to show the distribution of a dataset. They group data into bins or intervals, allowing you to see how frequently values occur within those ranges.

Example: If you want to analyze how students performed in math exam, a histogram can show the distribution of scores.

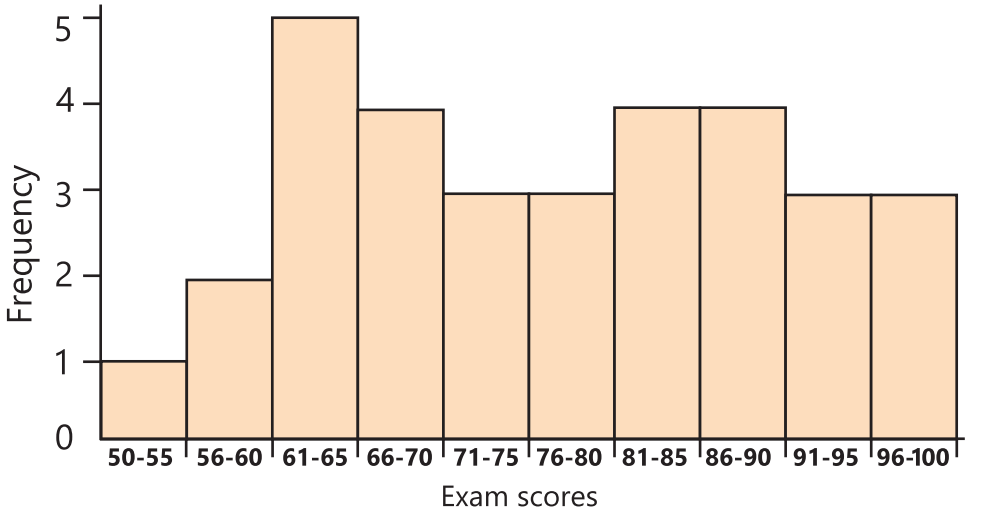


Figure 5.3: Example of a Histogram showing the distribution of exam scores

5.4.1.4 Scatterplots

Scatterplots are used to display the relationships between two variables. Each point on the graph represents an observation, and the position indicates values for both variables.

Example: A scatterplot can be used to explore the relationship between the number of hours studied and exam scores. (see Figure 5.4)

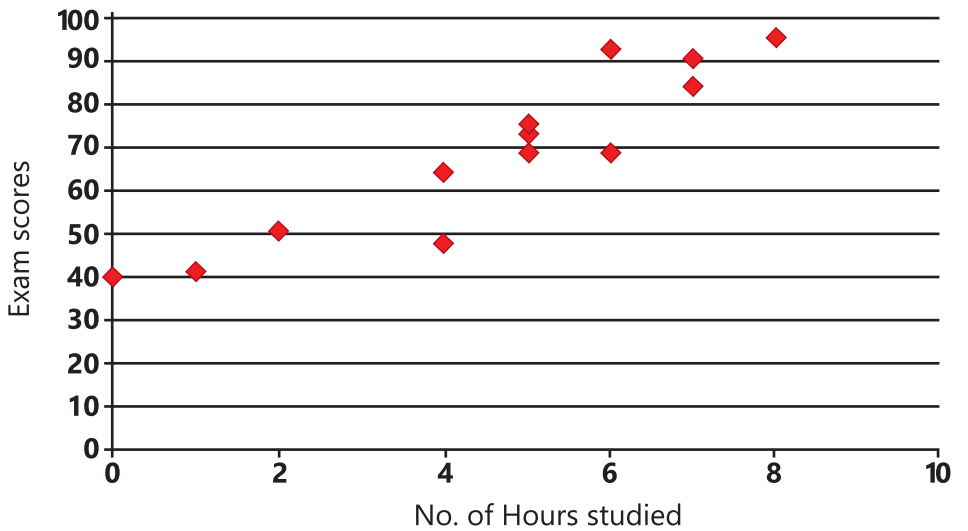


Figure 5.4: Scatterplot showing the relationship between hours studied and exam scores

5.4.1.5 Boxplots

Boxplots, or whisker plots, summarize data distribution by displaying the median, quartiles, and potential outliers. They provide a visual summary of data variability and spread.

Example: A boxplot can be used to compare the exam scores of different classes to see which class performed better overall.

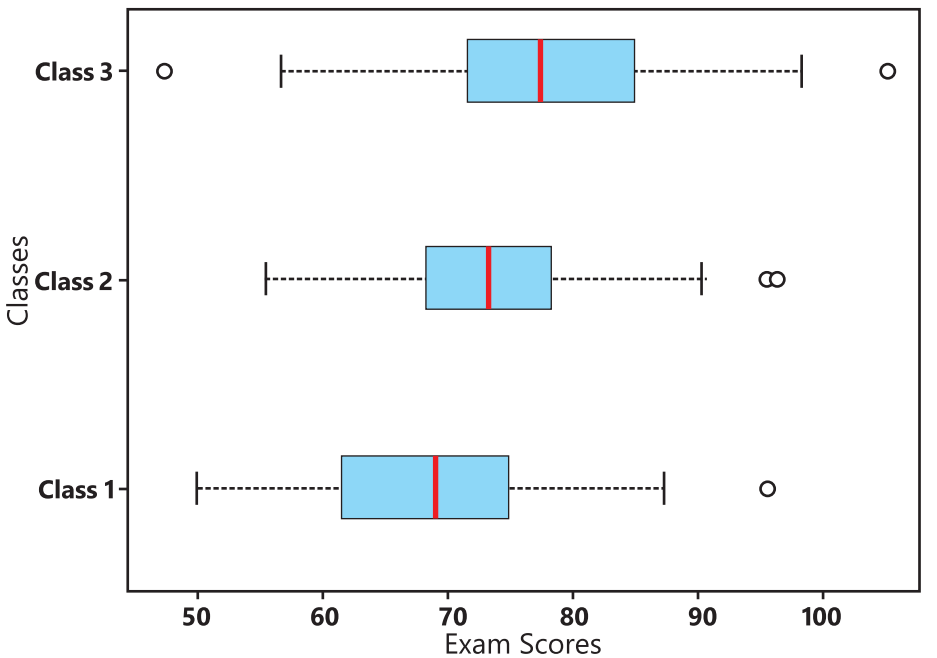


Figure 5.5: A Boxplot, showing class scores performance of three classes

5.5 Tools for Data Visualization

As we discussed in the above section visualization data helps us make sense of large amounts of information by turning numbers into easy-to-understand charts and graphs. In this section, we will discuss tools that can be used to create these visualizations and guide you through how to create and interpret them step by step.

There are many tools available for creating data visualizations, but some of the easiest to use are ones you may already be familiar with, such as such as Microsoft Excel and Google Sheets. These tools are widely accessible and provide straightforward methods for creating to create charts, graphs, and other visual representations of data.

5.5.1 Using Excel and Google Sheets for Visualization

Excel and Google Sheets: These tools allow you to easily enter your data and then generate a variety of visualizations such as bar charts, line graphs, and scatterplots.

Example: Let's say you run a small business in your local area, and you want to track how many products you sell each month. You can enter the data for each month in Excel or Google Sheets, and with just a few clicks, you can create a bar chart to see which month had the most sales.

5.2.2 Creating and Interpreting Visualizations

Step-by-Step Guide: Here's a simple guide to creating a visualization in Excel or Google Sheets.

1. **Enter Your Data:** Start by entering your data into the spreadsheet. Example: In one column, you could have the months (January, February, etc.), and in another column, the sales figures for each month.
2. **Select the Data:** Highlight the data you want to visualize by clicking and dragging your mouse over the cells.
3. **Choose a Chart Type:** Click on the "Insert" tab and select the type of chart you want to create (bar chart, line graph, etc.).
4. **Customize the Chart:** You can add labels to your chart to make it clearer, such as labeling the x-axis with the months and the y-axis with the sales figures. This makes the chart easier to interpret.
5. **Understanding Statistical Representations:**
   When you create a visualization, it's important to understand what the chart is telling you.

EXERCISE

Multiple Choice Questions

1. An example of a basic statistical model:
   a) Linear Regression b) Neural Networks
   c) Decision Trees d) Support Vector Machines
2. The activity involved in experimental design in data science:
   a) Creating visualizations
   b) Collecting and analyzing data systematically
   c) Writing code for machine learning
   d) Building databases
3. A commonly used tool for creating data visualizations:
   a) MS Excel b) Python (Matplotlib)
   c) Tableau d) All of the above
4. The meaning of the slope in a linear regression model:
   a) The intercept of the model
   b) The change in the dependent variable for a unit change in the independent variable
   c) The error term
   d) The mean of the data
5. An example of a real-world application of statistical models:
   a) Predicting house prices
   b) Creating social media posts
   c) Designing websites
   d) Writing essays
6. Option not considered a benefit of data visualization:
   a) Identifying trends and patterns
   b) Communicating insights effectively
   c) Making data more complex
   d) Summarizing large datasets
7. A primary goal of K-Means Clustering:
   a) To classify data into predefined categories
   b) To group data into clusters based on similarity
   c) To predict continuous outcomes
   d) To reduce the dimensionality of data

8. The meaning of "K" in K-Means Clustering:

a) Number of features in the dataset

b) Number of clusters to be formed

c) Number of iterations required for convergence

d) Number of data points in the dataset

Short Questions

1. What is the importance of building statistical models in real-world applications?
2. Name one basic statistical model used for predicting outcomes and explain its purpose.
3. List two types of data visualizations and describe when you would use each.
4. How does visualizing data help in understanding descriptive statistics?

Long Questions

1. Explain the role and importance of statistical models in solving real-world problems.
2. Describe the steps involved in building a basic statistical model (e.g., linear regression). Include details on data collection, model training, and evaluation.
3. Discuss the types of data visualizations and their uses.
4. Explain data collection methods.
5. Discuss the concept of measure of tendency with example.

ANSWER

Unit 1	
1	b
2	b
3	b
4	b
5	a
6	c
7	b

Unit 2	
1	c
2	a
3	a
4	b
5	b
6	d
7	c
8	b
9	c

Unit 3	
1	c
2	d
3	b
4	a
5	d
6	c
7	c
8	c
9	b
10	c

Unit 4	
1	b
2	c
3	b
4	c
5	a
6	c

Unit 5	
1	a
2	b
3	d
4	b
5	a
6	b
7	b

Unit 6	
1	b
2	c
3	c
4	b
5	b
6	b
7	d
8	d
9	a

Unit 7	
1	a
2	b
3	c
4	d
5	c
6	b
7	b
8	b
9	a

Unit 8	
1	c
2	c
3	c
4	b
5	c
6	b
7	b
8	b
9	d
10	c

Unit 9	
1	b
2	b
3	c
4	b
5	b
6	b
7	b
8	b
9	b
10	b